CSc 471 - Fall 2019
Fundamentals of Computer Rendering
Assignment 1
University of Victoria

Due: September 22, 2019 at 23:55. **Late assignments will not be accepted.**

1 Overview

One of the main goals in real-time computer graphics is to accurately represent the way lighting behaves in the real world while still retaining an acceptable level of performance. In most industries, the standard for real-time performance (measured in frames per second, or *FPS*) is 60FPS. As you might expect, attempting to accurately compute lighting equations is incredibly difficult and expensive. In order to maintain performance, approximations are employed to give a result that is close enough to the real result without hindering the framerate. Two of the simplest shading models are Gouraud and Phong shading.

The idea behind Phong shading is to combine three lighting components to produce a single, more accurate model. These components are: ambient, diffuse, and specular lighting. Here's a brief overview of how to compute each one (note that for the purposes of this assignment it is not important to understand how these equations are derived, but rather how they are implemented):

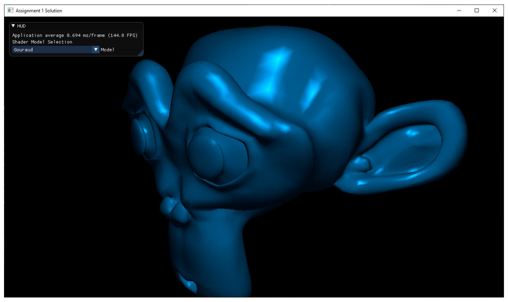
- **Ambient:** Take the colour of the light c_l and multiply it by a factor known as the *ambient factor* k_a. Then multiply this result with the colour of the object c_s. The equation is then:

$$k_a \cdot c_l \cdot c_s$$

- **Diffuse:** Take the normal to the surface $\mathbf{N}$ and the vector that goes from the position on the surface to the light $\mathbf{L}$. This vector is also known as the *incident light.* The colour is then computed as the product of the colour of the light by the cosine of the angle between $\mathbf{N}$ and $\mathbf{L}$. This result is then multiplied by a *diffuse constant* k_d. The resulting equation is:

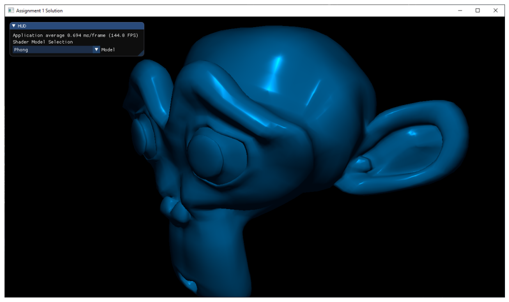
$$k_d \cdot c_l \cdot \cos(\theta) = k_d \cdot c_l \cdot \max(0, \mathbf{N} \cdot \mathbf{L})$$

- **Specular:** Take the vector that results from reflecting $\mathbf{L}$ on $\mathbf{N}$. This is the reflectance vector $\mathbf{R}$. Now take the vector that goes from the surface to the camera, or the view direction $\mathbf{V}$. The colour is computed as the colour of the light multiplied by the cosine of the angle between



```
Application average 0.694 ms/frame (144.0 FPS)
Shader Model Selection
Gouraud ▼ Model
```

(a) Gouraud Shading



```
Application average 0.694 ms/frame (144.0 FPS)
Shader Model Selection
Phong ▼ Model
```

(b) Phong Shading

Figure 1: Sample output

$\mathbf{V}$ and $\mathbf{R}$ elevated to the power of a *shininess coefficient* n. This result is then multiplied by a *specular coefficient* k_s. The resulting equation is:

$$k_s \cdot c_l \cdot \cos(\phi)^n = k_s \cdot c_l \cdot \max(0, \mathbf{R} \cdot \mathbf{V})^n$$

The final colour in the Phong shading model is then computed by adding these terms together and multiplying by both the surface colour and the *emission coefficient* k_e. The final equation is then:

$$c = k_e \cdot (c_{ambient} + c_{diffuse} + c_{specular}) \cdot c_s$$

These equations apply to both Phong and Gouraud shading. The main difference between the two is that Gouraud is performed *per vertex* whereas Phong shading is interpolated across the surface and computed *per fragment*.

Your task is to write a single C++ program that performs the following tasks:

- Load a provided OBJ file,
- Render the provided mesh using both Gouraud and Phong shading.

The program will use a *single* light source for the scene. The position and colour of the light source (along with the colour of the mesh) is up to you. The values for the constants k_e, k_a, k_d, k_s is also up to you. Figure 1 shows some samples of what your submission should produce. The images were generated with the following values:

Name	**Value**
Light position	$(10, 10, 10)$
Light colour	$(1, 1, 1)$
Material colour	$(0, 0.46, 0.69)$
k_a	0.1
k_d	0.5
k_s	0.9
k_e	1.5
n	32

2 Specification

For full marks, your submissions **must** have the behaviour specified below. Failure to implement this functionality will result in marks being deducted.

- Load the provided OBJ file. This can be done at program startup as shown in the labs.
- Accurately render the mesh without any artifacts.
- Implement Gouraud shading utilizing the appropriate vertex and fragment shaders.
- Implement Phong shading using the appropriate vertex and fragment shaders.
- Utilize C++ and the framework provided.

You will be provided with an assignment bundle that will contain the following:

1. A full CMake setup for the assignment including code to generate the headers like the one seen in the lab,
2. an OBJ file for Suzanne,
3. a blank `main.cpp` file,
4. a blank `assignment.hpp` header, and
5. a blank `UniformLocations.glsl` file.

Your submission must contain the following:

1. One `main.cpp` containing all the appropriate C++ code to perform the specified tasks.
2. One `assignment.hpp` containing the definitions of any auxiliary data structures, functions, etc that you may require. Note that you may leave this file blank.
3. Any vertex and fragment shaders that you require, along with the `UniformLocations.glsl` file (you may leave this blank if you wish). Note that these *must* observe the file extension convention specified in the labs.
4. In the submission comments, document the controls for your program. In particular, describe how to switch between the two shading models. For suggestions, see Recommendations. If you choose to implement your own camera system, you should also explain how to control it.

Please note that the full assignment bundle does not need to be provided in your submission, only the individual files listed above.

3 Evaluation

Your code must compile and run correctly in ECS 354. This assignment is worth 8% of your final grade and will be marked out of 10.

4 Recommendations

Implementing the Shading Models

For both Phong and Gouraud, your shaders will need access to the following data:

- The position of the vertices,
- the vertex normals,
- the position of the light,
- the position of the camera, and
- the components of the MVP matrix.

Note that vertex position and normals can be simply sent down through the attribute locations in the shaders as shown in the labs. The rest of the data can be sent in one of two ways:

- Uniform variables: these were not covered in the labs as they are a more inefficient way of sending data to the GPU. That being said, for the purposes of this assignment, they don't make that much difference. Uniform variables are slightly easier to specify in shaders, but are more involved to update in C++. Take a look at the OpenGL documentation for how to accomplish this.
- Uniform buffers: this is the way that was shown in the labs. Read the documentation on the specifications for `std140` so you can figure out how to organize your data in the buffer. It is recommended (and in general good practice) to split the data into 2 buffers. The first will hold the matrices (similar to the labs), while the second buffer will contain the remaining data. Note that any 3D vector gets allocated as a 4D vector in `std140`, so plan accordingly.

If you use *any* transform matrices in your model matrix, remember that the normal must then be transformed by the inverse of the transpose of the model matrix. For computing the vector $\mathbf{R}$, you may look at the documentation on the `reflect` function from GLSL.

Switching Between Shaders

The assignment requires you to implement two separate shading models and have the ability to switch between them in some way. There are two ways of switching the models:

- Create two shader programs: one for Gouraud and one for Phong. You can then switch which shader program is used in C++.
- Implement both models in a single shader program. The shaders can then have a uniform variable that switches which model is used.

Note that in terms of performance, these two are roughly equivalent. Now, in terms of how to handle the switching itself, there are a few options:

- Atlas provides a wrapper for a UI library to create simple interfaces. The modifications to the code are very simple, since it just involves creating two new structs, tweaking the window callbacks to handle them, and initializing the state. You can take a look at the examples provided in Atlas, the documentation, or ask the instructor.
- You may use the keyboard callback to switch the shader (be it the program or changing the value of a uniform) by pressing specific keys. For example, you could have Gouraud be the `G` key and Phong be `P`. You can handle this inside the keyboard callback and the codes for the individual keys can be seen in the documentation for GLFW (or if you are using Visual Studio, you can use the auto-completion feature to see the codes there. All of them begin with `GLFW_KEY_`).
- You can use the live-reloading feature of the shaders to switch out the model on-the-fly by changing the value of a constant variable.