CSCE 441 - Computer Graphics

Programming Assignment 4
Deadline: Nov. 6th (11:59 pm)

1 Goal

The goal of this assignment is to write shading codes in GLSL.

2 Starter Code

The starter code can be downloaded from here.

3 Task 1

Download the code and run it. You should be able to see a red bunny as shown in Fig. 1. Make sure you write your name in the appropriate place in the code, so it shows up at the top of the window. Here is a brief explanation of the starter code:

- There are three folders in the package. "obj" contains the "bunny.obj" file that has the geometry information (vertex position, normal, etc.) for a bunny object. "shaders" contains the vertex and fragment shader programs. You'll be mainly modifying these files to implement different shading methods. Finally, "src" contains the source files. `Program` is a class for loading, compiling, and linking the shader programs as well as sending data to them. Moreover, "tiny_obj_loader.h" is a simple header file for loading the obj files. You will use the `Program` class and "tiny_obj_loader.h" as is, but have to modify the main file. In summary, you'll be modifying the following functions:
  - "main.cpp": You need to modify this function to pass appropriate data to the program shaders and set up the materials and lighting.
  - "shader.vert": This is the vertex shader and you'll be filling it out to implement a specific shader.
  - "shader.frag": This is the fragment shader and you'll be filling it out to implement a specific shader.
- Now let's take a look at "main.cpp"
  - There are several global variables defined at the top of this file. Specifically, `program` is responsible for processing the shader programs. `posBuff`, `norBuff`, and `texBuff` store the vertex position, normal, and texture coordinates, respectively. `materials` and `lights` are structures for storing the material parameters and lighting information.
  - The structure of `main` function in "main.cpp" is similar to the one in all the previous assignments.
  - The `Init` function
    - The first few functions are called to initializes the window and events.
    - `LoadModel` function reads the obj file and saves the position, normal, and texture coordinates of each vertex of the geometry in the `posBuff`, `norBuff`, and `texBuff` vectors. We do not use the `texBuff` data in this assignment.
    - The last four lines are responsible for setting up the shader programs.
    - `program.SetShadersFileName` sets the address of the vertex and fragment shader files so we can load and compile them next.

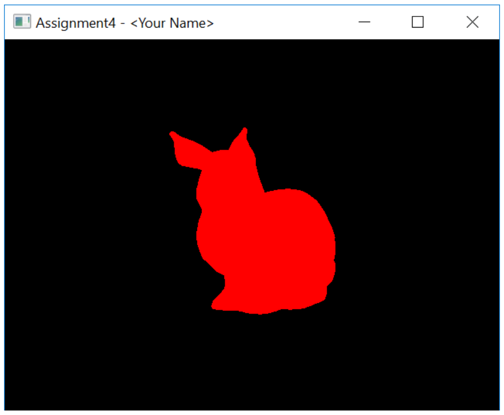

Figure 1: Running the skeleton code should produce a red bunny as shown here.

    * `program.Init` loads, compiles, and links, the shader files, so they are ready to be executed.
    * `program.SendAttributeData` sends the vertex attributes to the GPU. Attributes are defined at each vertex and will be directly passed as the input to the vertex shader program. `program.SendAttributeData(posBuff, ``vPositionModel'')` sends the position of each vertex which is stored in `posBuff` to the vertex program under the name of `vPositionModel`. If you look at the vertex program ("shader.vert"), you see a variable called `vPositionModel` which contains the position of each vertex. Here, we send the position and normal of the vertices to the vertex program.
  - The `Display` function
    * The first few lines of this function set the projection, view, and model matrices. These are the matrices used to project the vertices from the object space (the space that the positions in `posBuff` are given in) to the normalize device coordinate.
    * `program.Bind` activates the shader programs so they can be used for drawing. Note that you can have an array of `program` variables each set up with a different vertex and shader files, e.g., `program[0]` set up with "shader0.vert" and "shader0.frag" and `program[1]` set up with "shader1.vert" and "shader1.frag". This set up can be done in the `Init` function. Then in the `Display` function, you can just bind the particular shader program that you would like to be used for drawing, e.g., `program[1].Bind()`. In fact in this assignment you have to write three different shaders and be able to cycle through them using a key.
    * `program.SendUniformData` sends uniform data to the shader programs (both vertex and fragment). These are the variables that are the same for all the vertices. Here, we send the model, view, and projection matrices to the vertex program, so we can use them to transform the vertices to normalize device coordinate in the vertex program. Variables `model`, `view`, and `projection` in "shader.vert" are basically these 4×4 matrices.
    * `glDrawArrays` basically performs the graphics pipeline including the shader programs to display the triangles on the screen.
    * Finally, `program.Unbind` deactivates the shader.

- Now let's look at the "shader.vert" and "shader.frag"

- "shader.vert"
    - At the top, we define the two attributes `vPositionModel` and `vNormalModel` containing the position and normal of each vertex, respetively. These two variables are of type `vec3` meaning that they have three elements. The data for these two variables are privided through calling `program.SendAttributeData` in the `Init` function of "main.cpp".
    - Next we define three uniform 4×4 matrices (`mat4`) to serve as the model, view, and projection matrices. Uniform variables are constant for all the vertices (do not change from one vertex to another). These matrices are passed to the vertex shader through calling `program.SendUniformData` in the `Display` function of "main.cpp".
    - The next few lines of codes define a structure for holding the information about the light sources.
    - The line `uniform lightStruct lights[NUM_LIGHTS]` creates an array of the previously defined light structure, called `lights`. Note that since the light sources are the same for all the vertices, they should be defined as uniform variables. Moreover, please note that currently no value is passed to these variables. You need to define these light sources in the "main.cpp" by indicating the position and color of each light source and then pass them to the vertex program by calling `program.SendUniformData` with appropriate arguments in the `Display` function.
    - Next, we have four uniform variables (three `vec3` and one `float`) which contain information about the materials. These are the parameters required to calculate the color of each pixel or vertex based on the Phong shading model. Again the value for these variables need to be passed by calling `program.SendUniformData` with appropriate arguments in the `Display` function.
    - In the next line, we define a `varying` variable of type `vec3`, called `color`. Similar to attributes, varying variables are defined per vertex. These are the variables that are defined in the vertex program and are passed in the fragment program. This is in contrast to attributes which are passed from CPU to the vertex program.
    - The next few lines are the `main` function of the vertex program. We first multiply the model, view, and projection matrices to transform the vertices stored in `vPositionModel` to normalized device coordinate. The output of this process is stored in `gl_Position` which is a pre-defined output of the vertex shader. Note that, we add 1 to the end of each vertex to take them to homogeneous coordinate. We then define the color of each vertex to be red (`vec3(1.0f, 0.0f, 0.0f)`).
- "shader.frag"
    - We first define the varying variable `color`. This is the variable that is passed from the vertex program to the fragment program.
    - In the `main` function, we set `gl_FragColor` which is a pre-defined output of the fragment shader to be equal to `color`. Note that, `gl_FragColor` has four elements corresponding to red, gree, blue, and alpha. Alpha defines the transparency of the color. Alpha equal to 1 means the object is opaque which is why we add 1 to the end of the `color` variable before passing it as `gl_FragColor`.

4 Task 2

In this part, you will be implementing Phong shading model using Gouraud approach. Phong shading model uses the following equation to calculate the color of each point:

$$I = k_a + \sum_{i=1}^{k} C_i \left[k_d \max(0, L_i \cdot N) + k_s \max(0, R_i \cdot E)^s \right]. \tag{1}$$

Note that this equation is slightly different from the one in the slides. In the slides, the ambient term is defined as $k_a A$, but here we assume the intensity of the ambient illumination A is equal to (1, 1, 1). Use the following material and light sources to implement the shading.

- Material 1
  - k_a = (0.2, 0.2, 0.2)
  - k_d = (0.8, 0.7, 0.7)
  - k_s = (1.0, 1.0, 1.0)
  - s = 10.0
- Light
  - light 1
    - position = (0.0, 0.0, 3.0) in world coordinate
    - color = C_1 = (0.5, 0.5, 0.5)
  - light 2
    - position = (0.0, 3.0, 0.0) in world coordinate
    - color = C_2 = (0.2, 0.2, 0.2)

You need to first set the material and light information in the structure arrays provided at the top of "main.cpp". Note that `materials` is an array of size 3, but for this task you only set the first element, i.e., `materials[0]`. You need to set the other two materials and be able to cycle through them in the next task.

Once you set material and light sources properly, you need to pass them to the vertex shader by calling `program.SendUniformData` with appropriate arguments in the `Display` function. Note that, the position and color of the light sources are defined with a structure in the shader program. You won't be able to pass a structure from CPU to GPU, so you should just pass the individual properties for each light source. For example, you can pass the position of the first light source as:

```
program.SendUniformData(lights[0].position, ``lights[0].position'')
```

Here, the first argument is the name of the variable on CPU, and the second argument is the name of the variable on GPU inside quotations.

The position and color of light sources along with the material properties (k_a, k_d, k_s, s) can then be used to implement Eq. 1 in the vertex program. I in this equation is basically the color of the vertex. So you need to set `color` (the varying variable in the vertex program) to I. This color is then interpolated in the graphics hardware to obtain the color of each pixel (fragment). The varying variable `color` in the fragment shader is the interpolated color and thus setting `gl_FragColor` to this `color` variable in the fragment shader results in outputting the shaded pixels. Since the shading is done in the vertex program (per each vertex) and the color of each pixel is obtained by interpolating the color at the three vertices, this shader is the implementation of the Gouraud approach. Note that, the interpolation is done on the graphics hardware in the processes between vertex and fragment shaders and you do not need to implement it.

Note that, in order to implement Eq. 1, in addition to the materials and light properties, you need to compute L_i, N, R_i, and E. As discussed in the class, you can compute them either in the world space or

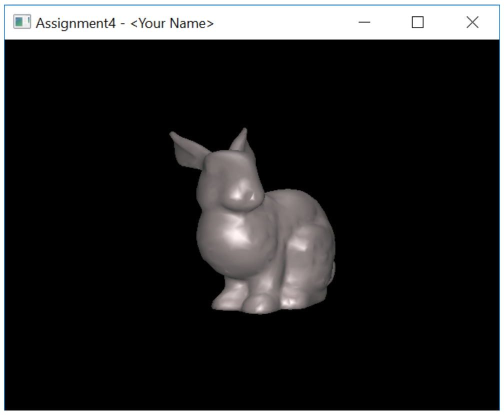

Figure 2: The outcome of task 2.

camera space. Let's say you choose to compute them in the world space. In this case, the position of the vertices and normals (`vPositionModel` and `vNormalModel`) are in the object space. So you have to first transform them properly using the model transformation to the world space (you have to be careful about normal transformation, as discussed in the class). Since position of light sources and eye is given in the world space, they do not require any transformations.

If you implement this task correctly, you should be able to see a bunny shown in Fig. 2

5 Task 3

Here, you'll be creating two more materials and add keyboard hooks to be able to cycle through these materials with the m/M keys (m moves forward, while M moves backward). The two additional materials are as follows:

- Material 2
  - k_a = (0.0, 0.2, 0.2)
  - k_d = (0.5, 0.7, 0.2)
  - k_s = (0.1, 1.0, 0.1)
  - s = 100.0
- Material 3
  - k_a = (0.2, 0.2, 0.2)
  - k_d = (0.1, 0.3, 0.9)
  - k_s = (0.1, 0.1, 0.1)
  - s = 1.0

The rendered bunny using these two materials and Gouraud approach is shown in Fig. 3

6 Task 4

Here, you'll be implementing two more shaders.

- **Phong approach for shading –** As discussed in the class, the Phong approach performs per-pixel shading, as opposed to per-vertex shading in Gouraud approach. This is very similar to the what you implemented for the Gouraud approach. Here, instead of calculating Eq. 1 in the vertex shader, you

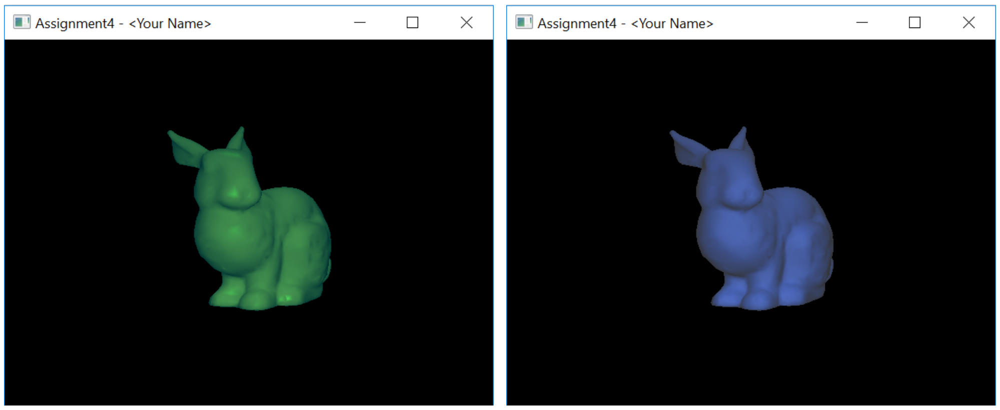

Figure 3: The rendered bunny using material 2 (left) and 3 (right).

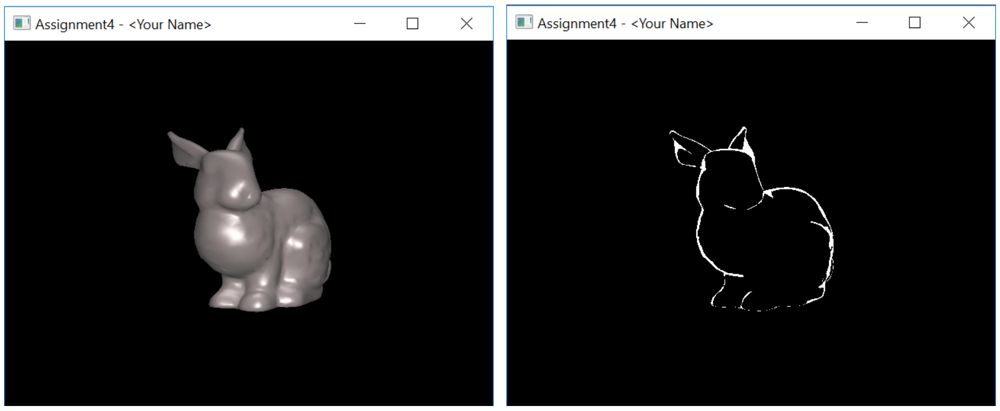

Figure 4: The rendered bunny using Phong approach (material 1) (left) and silhouette shader (right).

have to implement it in the fragment shader and directly set the final color I to `gl_FragColor`. Of course in this case, you need to pass the materials and lighting information as well as other variables to the fragment shader to be able to do the computations. The outcome of this shader for material 1 is shown in Fig. 4 (left).

- **Silhouette shader** To implement a silhouette shader, you need to color every fragment black, except for the ones that the angle between the normal and eye is close to 90°. For this, you need to compute the dot product of the normal and eye vector and threshold the results. Make sure the threshold is chosen properly so you get similar result to Fig. 4 (right). Note that this shader doesn't use the light or material information.

Pressing '1', '2', and '3' should switch to Gouraud, Phong, and Silhouette shaders, respectively.

For this, you need to implement each shader approach in a set of different shader files, e.g., Gouraud in "shader1.vert" and "shader1.frag", Phong in "shader2.vert" and "shader2.frag", and silhouette in "shader3.vert" and "shader3.frag". You should also create an array of `program`. Then in the `Init` function you assign a particular shader to each program element (e.g., program[0] for Gouraud – "shader1.vert" and "shader.frag") and load and compile all of these shaders and pass appropriate attributes to them. In the `Display` code, you should then bind the correct shader program based on the keyboard input.

7 Task 5

Provide the ability to move the light sources. The user should be able to cycle through the two light sources with l/L (l move forward and L move backward) and move the selected light source in x, y, and z axis using keys x/X, y/Y, and z/Z, respectively; Positive direction for lower case letters and negative direction for upper case letters.

8 Deliverables

Please follow the instruction below or you may lose some points:

- You should include a README file that includes the parts that you were not able to implement, any extra part that you have implemented, or anything else that is notable.
- Your submission should contain folders “src” and ’shaders’ as well as “CMakeLists.txt” file. You should not include the “build” or “obj” folder.
- Zip up the whole package and call it “Firstname_Lastname.zip”. Note that the zip file should extract into a folder named “Firstname_Lastname”. So you should first put your package in a folder called “Firstname_Lastname” and then zip it up.

9 Ruberic

Total credit: [100 points]

[30 points] - Implementing the Gouraud approach
[30 points] - Implementing the Phong approach
[15 points] - Implementing the silhouette shader
[10 points] - Ability to cycle through multiple materials with the keyboard
[15 points] - Ability to move the light sources with the keyboard

Extra credit: [10 points]

[10 points] - Implement a spotlight