DigiPen Institute of Technology

CS300

Assignment 2: Implementing Phong Illumination Model

Due date:

- Week 10, Friday 23:59:00
- For late submission, please refer the course outline document.
- `Note: A sample program is provided (Windows executable) for your reference on Moodle. Please feel free to experiment with the program by loading different OBJ files and checking how they appear under various shaders.`

Description

Add the following functionality to your implementation of the first programming assignment:

Shaders (30% - 10% per shader)

- Write vertex and fragment shaders to implement Phong illumination model with support for point, spot and directional light source type. We will implement the model using three different shader programs.

Shader Program 1 - Phong Lighting: This shader implements the vertex-based lighting model. This is the implementation of the OpenGL compatibility pipeline (version 2.1 or older). Lighting computations are done in the vertex shader, and only the final color is interpolated to the fragments. **The only "`out`" variable from the vertex shader should be the final color of the fragment.**

Shader Program 2 - Phong Shading: This is the implementation of the Phong model where the lighting computations are implemented in the fragment shader

- In vertex shader:
  - Transform the geometry vertices and normal from the model space to the projection/clip space.
  - Save **view space** vertex and normal, and pass them as output to fragment shaders.
  - Calculate other "`out`" variables for the lighting calculation and pass them to the fragment shader.

- In fragment shader:
  - Setup the uniform blocks for the application to pass in lighting properties and other settings.
  - Use the values of the appropriate "`in`" variables in the Phong equation, if applicable. Note that these "`in`" variables MUST match the "`out`" variables from the vertex shader.
  - Implement the Phong Model equations explained in class.

Shader Program 3 - Blinn Shading: The variation to the Phong Shading (shader 2 above) where we do not have to calculate the reflection vector (expensive part of the computation). Instead, we use the half-vector between the light vector (L) and view vector (V), and combine it with the surface normal for the specular computation. **The lighting calculations are done in the fragment shader.**

Requirements for implementing the shaders:

- Shader code MUST be loaded from external file, i.e. you must NOT embed the shader code within your 'C/C++' code as strings.
- Your application should provide the ability to load the shaders on-the-fly. This is perhaps the most important debugging tool you can provide yourself. **Hint: Implement a "reload shaders" button to provide this functionality.**
- The shaders must compile and link without errors or warnings. **Failure to do so will result in a zero grade for the assignment.**
- If there is an error, display the error message to console/GUI. The error message MUST specify which shader file(s) is (/are) causing the problem.
- Helpful convention: Use following extensions to the shader files – Vertex shader (*.vert), Fragment shader (*.frag), Common GLSL code (*.glsl).

Scene & Light Setup (20%)

- Your application must support variable number of light sources. The user can specify the number of active lights at run-time (in the range [1, 16].)
- In Assignment 1, we arranged the light sources in a circle around the object. Now, you should rotate the light spheres about the center of this circular path. (Think planets revolving around the Sun.)
- Color the light sphere with the diffuse color of the light source. **Do not apply the Phong shaders to the light-spheres. Write separate shaders for the light sources that only use the diffuse component of the Phong model.**
  - Remember, the light spheres must only display the diffuse color

- Your application must support all three types of light sources
  - *Point lights*
    - The position of the point light is the position of the sphere.
  - *Directional lights*
    - The direction of the light source is along a vector that points to the center of the object. This vector will be constantly updated as the lights rotate in orbit.
  - *Spotlights*
    - The direction of the spotlight will also be pointed towards the center of the object.
    - The user can change the inner and outer angle values, and spotlight falloff value from GUI.
- Implement the following lighting scenarios and provide capability to toggle between them using suitable menu actions:
  - Scenario 1: All lights have the same color parameters and type (position/directional/spot).
  - Scenario 2: All lights have different color parameters and same type (position/directional/spot).
  - **Scenario 3: Implement a third scenario that mixes the lights and their types. For example, you can have half of your active lights positional and the other half spotlights. Arrange them in a creative combination to get cool lighting effects.**
- Use `Uniform blocks` to send the light information to the shaders
- Add a plane (quad) under the model. The quad should be aligned to the XZ plane (horizontal), with min-max range [-5, 5], and situated below the object

Material Setup (10%)

- For the material values, use the following textures:
  - Diffuse: metal_roof_diff_512x512.ppm [RGB]
  - Specular: metal_roof_spec_512x512.ppm [RGB]
  - Ambient: No texture, user specified value [RGB]
  - Emissive: No texture, user specified value [RGB]

Refer http://netpbm.sourceforge.net/doc/ppm.html for details about reading the PPM format. You may use other libraries like SOIL (https://www.lonesock.net/soil.html) to load the images into OpenGL textures.

- The textures may have data in the range [0,255]. You should convert this data to the range [0,1] for use in the shaders as material reflectance coefficients.

- Use the following mapping from texture colors to material coefficients
  - Diffuse (`Kd`), Specular (`Ks`): All three components from the texture color - RGB
  - Ambient (`Ka`): Use a RGB vector value (specified by user through the GUI)
  - Shininess (`ns`): Use square of **Ks.r** as the shininess value. The brighter the intensity of the texture value, the higher its shininess will be.
- You will generate the texture coordinates for the models "by hand" using the formulae for spherical, cylindrical and planar (6-sided) texture mapping. There are two ways this can be done:
  - CPU calculation
    - Calculate the UV values on the CPU and pass them to the shaders as vertex attributes. This is how you would import a model with pre-defined UV values as exported from Maya (or other related software).
  - GPU Calculation
    - Calculate the UV values on the GPU. This requires implementing the projector functions for each type of mapping using GLSL.
  - Implement both functions and compare and contrast them (in your README file)
- Texture Entity: For Assignment 2 we will use the following as our Texture Entities:
  - The vertex position (normalized to the bounding box), and
  - The vertex normal (normalized and converted to range [0,1])

  User should be able to toggle between the two.

Light(s) properties (20%)

- Define the following variables in your 'C/C++' source and pass them into the shader as **uniform** / **uniform block** variables:
  - Global
    - The number of active lights
    - Distance attenuation coefficients – we will use common values for all light sources
    - Atmospheric (fog) color
    - Global ambient color
  - Per-light values
    - Light type (point, spot or directional)
    - Ambient, diffuse and specular colors (can change per light)
    - Position for point and spot

- Direction for spot and directional
- Inner and outer angle for spot light
- Spot falloff

GUI features to be implemented (10%):

- Select specific lights to be used in the scene by changing the number of active lights.
- Select the light types. Default light type should be a point light.
- Toggle to pause/start the light 'rotation'.
- **Toggle between the three scenarios. This will definitely be used by the TA when grading to quickly evaluate any issues with your code. Definitely implement this to make the grading process efficient.**

Note

1. Make sure all information is in correct "space."
2. **DEBUG tip: To check values of certain variable(s) in the fragment shader, you can set your fragment color to that variable so that you can visualize the variable value. You might need to do some transformation to bring the variables value to a [0, 1] range first.**
3. DO NOT use GLSL's 'reflect' function for your specular calculation, i.e. calculate the reflection vector using the formula discussed in class.

Misc. Functionality (10%)

- Check that your code compiles and executes cleanly (without errors or warnings).
- Check that your shaders compile and execute on DigiPen machines.
- Check that all features from previous assignment have been implemented.
- Check that your README is filled with all the information required for this assignment.
- **If above requirements are not met, then you will receive an automatic zero for the entire assignment**.

Assignment Submission Guideline

Please refer to the syllabus for assignment submission guideline. Failure to the submission guidelines correctly might cause you to lose points.

Suggested Weekly Breakdown of the Assignment

This is perhaps the most expansive programming assignment for this class as it introduces and requires implementation of a variety of methods. It is **required** that you work in weekly chunks and demo these to the instructor / TA.

Week	Task
1	- Add the plane under the object - Implement Phong Lighting & Phong Shading for one light source. - Setup correct vertex attributes and uniform variables - User should be able to change light properties using a GUI
2	- Add multiple lights to the above shaders - Calculate UVs on the CPU and pass them to the GPU as vertex attributes - Implement the first scenario (all lights of same type and parameters) - Implement second scenario (all lights different types but same color properties)
3	- Implement Blinn Shading - Calculate UVs on the GPU and use them directly. Allow the user to toggle between the two. - Implement third scenario (all lights and properties are randomly decided)
4	- Complete implementation for multiple lights - UI functionality to change values of parameters - Debug, document, repeat!