# Let there be light and Blinn-Phong reflection model

Due: Sunday, May 19, 2019 (11:59 AM, Noon)

## 1 Overview

This assignment is designed for you to understand Lighting and Shader programming. In this homework, you will create an animated scene with the theme of *cycle*, like day-night, seasons, etc. Starting with our skeleton code, you are asked to create a scene containing at least 2 types of lights and at least 2 types of materials.

First requirement of this homework is to import at least one external model other than the one included in the given skeleton. You may find appropriate models in public 3D model databases such as Free3D. The engine we provide only supports models in a standard format. So, please choose the models carefully. If you need more information on supported formats, see Appendix A.

Second requirement of this homework is to design a data structure supporting point light, a directional light, and a spotlight. You should implement at least 2 types of light but spotlight is mandatory. Modify `struct Light` to hold necessary properties of all cases in one struct. You may find `Engine::Transform` in the structure given for the skeleton code. If so, please read Appendix B. You should also modify `DiffuseFragmentShader.glsl` to match the structure. Then, modify `class DiffuseMaterial` to send `struct Light` to the fragment shader. Finally, modify `DiffuseFragmentShader.glsl` to support lighting of all type of lights you choose to implement. It is just for verifying that your structure works before you move on to the next step. So, it is sufficient to implement diffuse reflection in `DiffuseFragmentShader.glsl` for each light you implement. Note that your point light should be attenuated as it becomes distant.

Next is to implement Toon shading and Phong shading based on Blinn-Phong reflection model. You need to fill the functions on `ToonMaterial.cpp` and `PhongMaterial.cpp` like in `DiffuseMaterial.cpp`. Then, you should create `.glsl` files and implement both vertex and fragment shaders for Toon and Phong shading. Both materials should support lights you implemented for `DiffuseMaterial`.

Now, you are ready to design your own *cycle*-themed animated scene. You do not have to contain all materials simultaneously in a scene. However, you need to use each materials at least once, for example, by switching materials with user inputs. This is also same for the lights. If the scene changes by user inputs, explain in report how you implemented the change. Again, you may be unfamilar with newly added `Engine::Transform`. If that is the case, please read Appendix B. In addition, the skeleton code contains a sample animated scene using `Animation`, a aimple keyframe based animation tool. It is completely optional to utilize it or not, but it will be helpful for you to express the scene you want.

Your scene must contain a smooth animation of lights matching with the theme, such as orbiting sunlight and moonlight. Only the transformation of lights and materials will be graded, so you don't need to focus on the movement of your object so much. You can compose multiple scenes to express *cycles*. For example, you can make 2 scenes of summer and winter, switched by keyboard inputs. In this case, you don't have to implement smooth transitions between each scene — proper light animations are sufficient for full score.

To earn creativity points, it is recommended for you to create your own shader or modify toon/phong shader to look more interesting. Implement your creative shader in `class MyMaterial`. You can give a bunny some stylish look with tweaking the fragment shader. Or, you can make a bunny wiggle by changing the vertex shader. Your material should change its look under lights, though it does not need to strictly follow Blinn-Phong reflection model. Write what you wanted to express by `MyMaterial` and how you achieved it in your report clearly.

*Please keep the due date!* You also need to write a comment in your code and write up 2-3 pages (10pt, 1.5 space) reports explaining what and how you have done to meet the specifications given below. **Do not just copy your code to your report. Please explain 'how' your implementation satisfies the specification**.

## 2 Specification & Grading

| Specification | Max pts |
|---|---|
| 1. Using one or more external 3D model(s) other than given ones | 5 |
| 2. Designing a structure for lights | 30 |
| 2.1 Spotlight (20) | |
| 2.2 Choose one - point light or directional light (10) | |
| 3. Shader programming | 30 |
| 3.1 Phong shading based on Blinn-Phong reflection model (15) | |
| 3.2 Toon shading based on Blinn-Phong reflection model (15) | |
| 4. Creating scenes | 10 |
| 4.1 Scenes themed with *cycle* (5) | |
| 4.2 Smooth animations of lights in each scene (5) | |
| 5. Creativity | 15 |
| 6. Document | 10 |
| Total | 100 |

Table 1: Specification and Grading

## A More on 3D model formats

3D models can be stored in various formats. There are more than 50 different formats to deal with 3D models. Fortunately, Open Asset Import Library or Assimp, the library you installed for the Lab 3, can support most of these formats. So, you will have no problem importing models in popular formats, like .obj, .ply, .fbx, .stl, .3ds, and .blend, for this homework. Some formats like .obj and .ply can even be edited via text editors. But, this library is so good, you do not need to know the details about the formats — it just works! However, there are some caveats you need to aware before importing and using the models. In this section, we wrote some problems you may encounter, and how to solve it.

**Z-up coordinates** You may find that some formats like .3ds or .blend are rotated -90 degrees about x axis, even though their orientations are unchanged. This is because these formats uses Z-up system, where z axis is the vertical axis. Because our Engine uses Y-up system, meaning y axis is the vertical axis, meshes from Z-up system will shown to be rotated -90 degrees about x axis. It can be solved by simply rotate the render object 90 degrees about x axis, transforming Z-up coordinates back to Y-up coordinates.

**Inconsistant scales** Each model may have different scale units. Some models might appear so tiny because their vertex coordinates are in meters. Other models may have coordinates in small units like millimeters, so their coordinates reach up to thousands. If that is the case, they are not even rendered because most vertices are out of the view frustum. Therefore, you should properly scale your render object to find a perfect scale for your scene.

**Resetting base transformation** It might be frustrating to compute transformation of an object whose base orientation is not an identity matrix because it was transformed from Z-up coordinates back to Y-up. You can solve this inconvinience with a simple trick. The trick is to put the object under the children of an empty transform whose orientation is identity matrix. So, you can rotate the object by transforming the parent without confusion.

**Composite models** Some 3D models may be organized with multiple meshes, like your snowmans in Homework 2. You may see the prompt saying that the file you loaded has 2 or more different meshes during Lab 3. Most formats store the structure of these meshes, by recording relative position and rotation of each parts. However, in our `Engine::ModelLoader` those informations are not considered, and it only loads meshes of each part. Therefore, `Engine` does not support loading hierarchical objects from external models. It means that you need to assemble manually, by moving parts into proper position and setting a correct hierarchy.

## B More on Engine::Transform

In previous labs and homeworks, `Engine::Camera` and `Engine::RenderObject` contained similar position and orientation information separately, thus there was no common method to handle transformation of both the camera and objects. Also, object hierarchy was implemented only for `Engine::RenderObject`, resulting the camera out of the hierarchical structure. Some of you might find it uncomfortable and restricting during previous labs and homeworks. And now, you need to make an animated scene with yet another object — lights.

To give you a unified object transformation methods, we created `Engine::Transform` for this and future labs and homeworks. It is similar to what `Engine::RenderObject` had, in a sense of its interfaces. So, you will be able to easily use it. All objects — the camera, render objects, and lights — has `Engine::Transform` as a member. You can move, rotate, or scale any objects through the shared method of `Engine::Transform`.

Another benefit is that you can now put any object in a hierarchy, so the camera or lights can be a part of a render object or vice versa. This will enable more flexible animations for this and future homeworks.

For more detailed usage of `Engine::Transform`, why don't you read the skeleton code that contains a sample scene? We tried to provide a helpful use case for you who may feel unfamilar to the 3D animations. But if you still feel missing, feel free to ask TAs.