CS 770/870
P4: Lighting Redux v.2

March 31, 2017

Due: Friday, 4/14 (by midnight 23:59:59)
Late penalties: Sat/Sun -3, Mon -7, Tue -15

This assignment focuses on the Phong lighting model. We'll keep our existing multi-object, multi-scene framework along with a small expansion of previous keyboard interaction functionality to help demonstrate the lighting effects. You probably will build on your P2 with lighting components taken from P1 if you have any.

1. Assignment overview

The original *BoxDemo* code implements a light in the box's object coordinate system without specular reflection; some of you were able to get some light functionality from scene and/or world lights. You'll have a head start on this assignment, but there are also many new features that will be needed:

1. Add specular reflection to the basic lighting code in BoxDemo.
2. Add light distance attenuation with programmably variable constants, c_1, c_2, and c_3.
3. Support multiple scene and world lights; you may have an implementation maximum for each (at least 3).
4. Implement both **directional** and **point** light sources.
5. Define a displayable object to represent each kind of light: a "line" for directional, a small box for point.
6. Implement your lighting model code in the fragment shader.
7. All lighting information should be defined in the cpu and downloaded to the shaders as uniform variables.
8. Provide some basic keyboard interaction features to help test and demonstrate lighting effects.
9. Your scenes should include both Box and Cylinder objects. The curved surfaces of cylinders will be more effective at showing some lighting features (especially specular reflection) than boxes.
10. You should create a *Material* class to encapsulate light reflection characteristics to be assigned to a *Shape3D* object. This class should have fields for all of the shading parameters: k_a, k_d, k_s factors. In principle, the sum of the 3 factors should be around 1, but most surfaces are not shiny, so often we let the sum of k_a and k_d be close to one and let shiny surfaces go over one. Your goal is to come up with scenes that show all those features working; use whatever factors you need to show that.
11. Each *Scene* should have a global ambient light factor I_a that is an rgb triple that can be included in the light calculation. The k_a factor in the materials property associated with the object is multiplied by this light component.

2. Lighting

You should have a Light class and it is quite appropriate to have child classes for point and directional lights. Most of the data needed by the different children is shared as is some of the functionality. The light object itself should be responsible for displaying itself, so it might make sense to have Light extend Shape3D.

A point light source can be represented as a small box colored with the light color. A directional light can be represented as a long thin box in the direction parallel to the light direction; this is easily done with a Box object centered at the origin with large x size and very short y and z sizes and a color that matches the light's color. You can then rotate it so it matches the light direction. Light objects should not themselves be subject to lighting and it should be possible to turn their rendering on and off. If a light is off, it's image should not be displayed, but it should also be possible to "turn off the images of all lights" option.

With multiple lights it is very easy to generate "raw" color values where one or more of the color parameters exceeds the supposed maximum value of 1. Clamping individual values to 1 is not helpful since it usually changes the hue of the colors. It is better to do a "color normalization", which is to find the maximum component and then divide all components by that value, so the computed color remains the same, but its overall intensity is reduced. It is also possible to do a normalization that allows the maximum value of a component to be a little higher than 1, such as 1.1 or 1.2.

In addition to direction or position, the Light object itself should have fields for the color of the light and the c_1, c_2, and c_3 needed for light attenuation. This allows you to decide for each light whether it is "weak" enough to have light attenuation independent from the overall intensity of its color specification.

3. Keyboard interaction (most are already in P2)

1. Provide the polygon line/fill option, but use 'L' as a **toggle** (first 'L' shows lines, next shows filled).
2. Use '<' and '>' (and ',' and '.') to change scenes.
3. The functions of **l**eft, **r**ight, **u**p, **d**own, zoom **i**n, zoom **o**ut, 'p', 'q' and esc keys from should be retained.
4. Use the '0' key to turn the pre-defined object coordinate light source on and off.
5. Use the '1' key to turn **all** scene lights on and off; if the light is off, so is its image.
6. Use the '2' key to turn **all** world lights on and off; if the light is off, so is its image.
7. Use the 'h' key as a **toggle** to hide/show the light images of all the lights that are currently on.
8. Use the 'n' key to toggle on/off the "light normalization" option.
9. Use the 'g' key to toggle on/off whether the global ambient light is to be included in the light calculation.
10. Use the 7, 8, 9 keys to **lower** the values of the c_1, c_2, and c_3 light attenuation constants.
11. Use the &, *, (keys to **raise** the values of the c_1, c_2, and c_3 light attenuation constants.

4. Tentative point allocation

10	Box and Cylinder objects in scenes
15	Directional light in scene and world
15	Point light in scene and world
10	Displayable light objects
10	Specular reflection
10	Distance attenuation
20	Multiple lights of each type, color normalization
10	Interaction features (lights on/off, light images on/off, light normalization on/off, scene view changes, etc.)

All point components include elements identifying how well you **showed** that you implemented the functionality with your scenes **and** their titles.

Points are earned by correctly implementing features robustly. Points are deducted for bugs, incorrect implementation, poor style, poor commenting, poor design decisions, non-functioning make file, etc.

5. Notes

1. Each vertex normal, vertex location, viewer position, and light location or direction have to be mapped to the same coordinate system before applying the lighting model. Because you have to implement the lighting in shader code, you need to map all parameters to normalized projection coordinates.
2. Your code should be written to support an arbitrary number of lights although you may have a maximum number specified by a constant in your program. Shaders don't support dynamic length arrays.
3. Be sure to have some/most scenes use different colored lights; it's one of the best ways of showing that your lighting actually works.
4. Even if your Light extends Shape3D, it's probably best to have Light **contain** a Box object for representing the light rather than having Light extend Box. This is because you don't want the application to be using public Shape3D methods of Box to modify arbitrarily the Light's graphical representation.
5. Your lighting code has to be sure to check the results of the different dot product calculations that are computing essentially the cosines of angles. In all cases negative cosine calculations indicate special cases: if dot(V,R) < 0, the viewer is looking the wrong direction to see any specular reflection. If dot(L,N) < 0, the light is on the wrong side of the object surface.
6. You might want to define a default Material object that could be allocated by the Shape3D constructor in order to reduce the complexity of *SceneManager*.
7. Normal vectors require special processing in order to be correctly mapped in the NPCS. Information will be provided.
8. The original class notes on the lighting model had a misleading/incorrect minor flaw. It was incorrect to suggest that the C_s term should normally be the same as C_d and C_a. In fact, it almost never is. Shininess means that the material is reflect the color of the light, not the color of the material. It is rare for the C_s is to be anything except (1,1,1). I've revised the notes on the web to throw out C_s entirely.