Computer graphics - lab exercise 5 - Shaders and lighting

Start this exercise with the code in ***Control3DBase.zip***. *It would be most straightforward to start by implementing at least parts of* **lab exercises 3 and 4** *in the base program before starting this one. There are walk through videos online that can help setting up the shaders so they work and get a graphics program to a good point to start this exercise*.
It's best to start with a program where you have some *real-time* control over the camera, just to be able to view the changes that are being made here. Otherwise it's just important to have some objects in the world, that the lights will affect.

NOTE - DO THIS! There are some file paths in the file Shaders.py. Please change them so that the relative paths are always correct. To do this add:
import sys
at the top of the file. Then change
`"simple3D.vert"`
to
`sys.path[0] + "/simple3D.vert"`
Note the slash - it is important!
Also do this for simple3D.frag
Now the paths will always be relative to the main python script that you run, not relative to the folder that is open in Visual Studio Code.

Also do this for paths to any other files that you use in your assignments.

On Canvas, under Modules, there is extra material about lighting. It can help for certain parts of this exercise to study some of the slides (and/or slide lecture videos) on lighting calculations and light models.

Efficiency/optimization hint for drawing:
Call **glVertexAttribPointer** *(***shader.set_position_attribute** *and* **shader.set_normal_attribute***) as seldom as possible. Sending the array to the shader is much slower than actually drawing the vertices in the array. Try to make a separate operation that sets the arrays, so that you can call that once and then call the draw operation (that then only calls glDrawArrays) as many times as you need, always drawing the same object, but not re-sending the arrays to the graphics hardware.*

Adding lights and materials to the shader

Now open the *vertex shader* file itself (*simple3D.vert*). Read through the code and understand what it does.

Remove or comment out the lines that calculate ***v_color*** (and its *light factors*). In the current version this uses hard coded values without much in the way of a clear explanation of what they stand for. We will do our first lighting in *global coordinates*, after multiplying the *position* and the *normal* with the *model matrix*, but before multiplying the *position* with the *view matrix*.
Now remove the variable definition for ***u_color*** and instead add ***u_light_position*** (vec4 position), ***u_light_diffuse*** and ***u_material_diffuse*** (both vec4 colors).

Now go ahead and program the *diffuse lighting* into the *shader*. First make the vector ***s*** that's the difference between the *light position* and the *vertex position* (it's called ***position*** and is the result from multiplying with the *model matrix*). Then make the intensity scalar ***lambert*** that's the dot product between the *normal* and the vector ***s***, divided by their lengths. This is the *cosine* of the *angle* between the vectors and gives us the *diffuse intensity*. Finally multiply ***lambert*** with the *light_diffuse* and the *material_diffuse* colors to get the final color. It's OK to multiply vectors, even though it's not proper linear algebra. This multiplication is done *component wise*. When multiplying with *matrices* the multiplication is done in the normal way. So this will give you the color:

```
v_color = lambert * u_light_diffuse * u_material_diffuse;
```

Make sure the *shader* still finishes the calculations to the end to get the final value for ***gl_Position***, but you've already used the *normal* now, so you won't have to multiply that through the *view matrix*. The *position* needs to go all the way though.

In the *Shader* class remove references to ***u_color*** and ***colorLoc*** and make variables for ***lightPosLoc***, ***lightDifLoc*** and ***matDifLoc*** instead. Get the *uniform* locations for those in the constructor.
You can also add functions to set the *light position*, *light diffuse* and *material diffuse* as well as removing the setter for *color*.

Now, in your game class, change all the references to ***setColor*** to ***setMaterialDiffuse***, and add calls to ***setLightPosition*** and ***setLightDiffuse*** somewhere towards the top of the display function.

Experiment a bit with light positions and colors, and try to get the light to move around.

Add more types of lighting

Add a variable for the camera position to the shader, as well as *specular colors* for the *light* and *material* and a *shininess* for the material. Then, next to the *diffuse lighting* add calculations for *specular lighting*. Remember the vector ***v*** is the difference between the *camera position* and the *vertex position*. Then ***h = s + v*** and finally *phong* is the dot product of ***h*** and the *normal* divided by their lengths. That intensity to the *power of shininess* (***pow(phong, shininess)***) is then multiplied with the *specular colors*, the same way *lambert* was multiplied with the *diffuse*.
Add the *specular* color to the *diffuse* color and you have your final color.

Now experiment with values. Try to have strong white colors first for the *specular* values and a *shininess* of 10 is OK to start. Try higher and lower values though to get a feel for what it does.
Now experiment. Try adding more lights and see if adding *ambience* helps define the objects.
To add *ambience* simply add an *ambience* color for the *material* and an *ambience* factor for the *light model*. Then multiply the two together and add that to the combined color. No intensity calculations needed.

Look at the slides with the lighting lecture for reference.

See if you can make the lighting ***per pixel*** instead of ***per vertex***. Then you will need *varying variables* for the vectors ***s***, ***v*** and *normal* (or ***s***, ***h*** and the *normal*). You also won‘t need a varying variable for the color. Instead you calculate the color in the fragment shader (***simple3D.frag***) using the vectors that you define as *varying*. Then put the color you get from the calculations into ***gl_FragColor***.

If you want to render some things with a flat color instead of lighting calculations you can either put if statements into your shader and uniform values for switches. You can also build another shader with other shader files and another shader class, and then switch between them using ***glUseProgram***.

Light model example 1

A full light model for a fixed number of lights needs the following variables in the shader:

```
vec4 eyePosition;
vec4 globalAmbient;
```

//these variables for each light in the light model

```
vec4 light1Position;
vec4 light1Diffuse;
vec4 light1Specular;
vec4 light1Ambient;
vec4 material_diffuse;  //light that scatters
vec4 materialSpecular;  //light that reflects
vec4 materialAmbient;  //light that fills the environment
vec4 materialEmission;  //self-illumination of material
float materialShininess;  //shininess of light reflection
```

In the shader you will need to calculate the following:

```
vec4 v = eyePosition - position;  //this can be calculated once, not per light

//calculate the following for each light
vec4 s = light_position - position;
vec4 h = s + v;

float lambert = max(0.0, dot(normal, s) / (length(normal) * length(s)));
float phong = max(0.0, dot(normal, h) / length(normal) * length(h)));

vec4 ambientColor = lightAmbient * materialAmbient;
vec4 diffuseColor = light_diffuse * material_diffuse * lambert;
vec4 specularColor = lightSpecular * materialSpecular * pow(phong, materialShininess);
vec4 light1CalculatedColor = ambientColor + diffuseColor + specularColor;
//imagine we also calculate light2CalculatedColor & light3CalculatedColor, etc.
```

Finally we add up the entire color for the vertex (or fragment):

```
v_color = globalAmbient * materialAmbient + materialEmission + light1CalculatedColor;
    // + light2CalculatedColor + light3CalculatedColor + ... etc.
```

Light model example 2

A simpler light model that is often nicer to work with combines some elements:

```
Vec4 eyePosition;
vec4 globalAmbient;

//these variables for each light in the light model
vec4 light1Position;
vec4 light1Color; //used for both Diffuse and Specular
vec4 material_diffuse; //used for both Diffuse and Ambient
vec4 materialSpecular;
float materialShininess;
```

Then the calculations:

```
vec4 v = eyePosition – position;  //this can be calculated once, not per light

//calculate the following for each light
vec4 s = light_position – position;
vec4 h = s + v;
float lambert = max(0.0, dot(normal, s) / (length(normal) * length(s)));
float phong = max(0.0, dot(normal, h) / length(normal) * length(h)));
vec4 diffuseColor = lightColor * material_diffuse * lambert;
vec4 specularColor = lightColor * materialSpecular * pow(phong, materialShininess);
vec4 light1CalculatedColor = diffuseColor + specularColor;

//imagine we also calculate light2CalculatedColor & light3CalculatedColor, etc.
```

Finally we add up the entire color for the vertex (or fragment):

```
v_color = globalAmbient * material_diffuse + light1CalculatedColor;
// + light2CalculatedColor + light3CalculatedColor + ... etc.
```

In this light model we don‘t keep track of as many variables so it gets less confusing when editing your scene. We also skip the emission, which means that if you want an entirely flat color on something you will have to either use another shader or have a switch that skips lighting calculations entirely and uses a color variable instead (v_color = u_color).
We also skip ambience per light and only work with the global ambience, making it simpler to work with.
Use on of these models or something in between, or mix, skip and combine values as you see fit and works for your project. Always make sure your shader is using any variable that you define and that your program is sending some data into every variable so that you get predictable results.