



Learn OpenGL - Graphics Programming

Learn modern OpenGL graphics programming in a step-by-step fashion.

Joey de Vries





Lighting

12	Colors	108
13	Basic Lighting	113
14	Materials	124
15	Lighting Maps	129
16	Light Casters	135
17	Multiple lights	146
18	Review	152

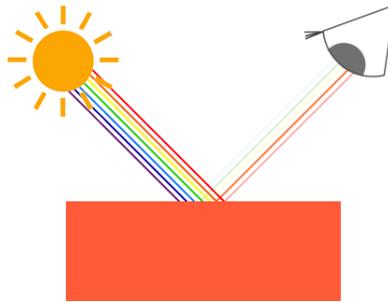
12. Colors

We briefly used and manipulated colors in the previous chapters, but never defined them properly. Here we'll discuss what colors are and start building the scene for the upcoming Lighting chapters.

In the real world, colors can take any known color value with each object having its own color(s). In the digital world we need to map the (infinite) real colors to (limited) digital values and therefore not all real-world colors can be represented digitally. Colors are digitally represented using a red, green and blue component commonly abbreviated as RGB. Using different combinations of just those 3 values, within a range of $[0, 1]$, we can represent almost any color there is. For example, to get a *coral* color, we define a color vector as:

```
glm::vec3 coral(1.0f, 0.5f, 0.31f);
```

The color of an object we see in real life is not the color it actually has, but is the color **reflected** from the object. The colors that aren't absorbed (rejected) by the object is the color we perceive of it. As an example, the light of the sun is perceived as a white light that is the combined sum of many different colors (as you can see in the image). If we would shine this white light on a blue toy, it would absorb all the white color's sub-colors except the blue color. Since the toy does not absorb the blue color part, it is reflected. This reflected light enters our eye, making it look like the toy has a blue color. The following image shows this for a coral colored toy where it reflects several colors with varying intensity:



You can see that the white sunlight is a collection of all the visible colors and the object absorbs a large portion of those colors. It only reflects those colors that represent the object's color and the combination of those is what we perceive (in this case a coral color).

Technically it's a bit more complicated, but we'll get to that in the PBR chapters.

These rules of color reflection apply directly in graphics-land. When we define a light source in OpenGL we want to give this light source a color. In the previous paragraph we had a white color so we'll give the light source a white color as well. If we would then multiply the light source's color with an object's color value, the resulting color would be the reflected color of the object (and thus its perceived color). Let's revisit our toy (this time with a coral value) and see how we would calculate its perceived color in graphics-land. We get the resulting color vector by doing a component-wise multiplication between the light and object color vectors:

```
glm::vec3 lightColor(1.0f, 1.0f, 1.0f);  
glm::vec3 toyColor(1.0f, 0.5f, 0.31f);  
glm::vec3 result = lightColor * toyColor; // = (1.0f, 0.5f, 0.31f);
```

We can see that the toy's color *absorbs* a large portion of the white light, but reflects several red, green and blue values based on its own color value. This is a representation of how colors would work in real life. We can thus define an object's color as *the amount of each color component it reflects from a light source*. Now what would happen if we used a green light?

```
glm::vec3 lightColor(0.0f, 1.0f, 0.0f);
glm::vec3 toyColor(1.0f, 0.5f, 0.31f);
glm::vec3 result = lightColor * toyColor; // = (0.0f, 0.5f, 0.0f);
```

As we can see, the toy has no red and blue light to absorb and/or reflect. The toy also absorbs half of the light's green value, but also reflects half of the light's green value. The toy's color we perceive would then be a dark-greenish color. We can see that if we use a green light, only the green color components can be reflected and thus perceived; no red and blue colors are perceived. As a result the coral object suddenly becomes a dark-greenish object. Let's try one more example with a dark olive-green light:

```
glm::vec3 lightColor(0.33f, 0.42f, 0.18f);
glm::vec3 toyColor(1.0f, 0.5f, 0.31f);
glm::vec3 result = lightColor * toyColor; // = (0.33f, 0.21f, 0.06f);
```

As you can see, we can get interesting colors from objects using different light colors. It's not hard to get creative with colors.

But enough about colors, let's start building a scene where we can experiment in.

12.1 A lighting scene

In the upcoming chapters we'll be creating interesting visuals by simulating real-world lighting making extensive use of colors. Since now we'll be using light sources we want to display them as visual objects in the scene and add at least one object to simulate the lighting from.

The first thing we need is an object to cast the light on and we'll use the infamous container cube from the previous chapters. We'll also be needing a light object to show where the light source is located in the 3D scene. For simplicity's sake we'll represent the light source with a cube as well.

So, filling a vertex buffer object, setting vertex attribute pointers and all that jazz should be familiar for you by now so we won't walk you through those steps. If you still have no idea what's going on with those I suggest you review the previous chapters, and work through the exercises if possible, before continuing.

So, the first thing we'll need is a vertex shader to draw the container. The vertex positions of the container remain the same (although we won't be needing texture coordinates this time) so the code should be nothing new. We'll be using a stripped down version of the vertex shader from the last chapters:

```
#version 330 core
layout (location = 0) in vec3 aPos;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
}
```

Make sure to update the vertex data and attribute pointers to match the new vertex shader (if you want, you can actually keep the texture data and attribute pointers active; we're just not using them right now).

Because we're also going to render a light source cube, we want to generate a new VAO specifically for the light source. We could render the light source with the same VAO and then do a few light position transformations on the `model` matrix, but in the upcoming chapters we'll be changing the vertex data and attribute pointers of the container object quite often and we don't want these changes to propagate to the light source object (we only care about the light cube's vertex positions), so we'll create a new VAO:

```
unsigned int lightVAO;
glGenVertexArrays(1, &lightVAO);
glBindVertexArray(lightVAO);
// we only need to bind to the VBO, the container's VBO's data
// already contains the data.
glBindBuffer(GL_ARRAY_BUFFER, VBO);
// set the vertex attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float),
                    (void*)0);
glEnableVertexAttribArray(0);
```

The code should be relatively straightforward. Now that we created both the container and the light source cube there is one thing left to define and that is the fragment shader for both the container and the light source:

```
#version 330 core
out vec4 FragColor;

uniform vec3 objectColor;
uniform vec3 lightColor;

void main()
{
    FragColor = vec4(lightColor * objectColor, 1.0);
}
```

The fragment shader accepts both an object color and a light color from a uniform variable. Here we multiply the light's color with the object's (reflected) color like we discussed at the beginning of this chapter. Again, this shader should be easy to understand. Let's set the object's color to the last section's coral color with a white light:

```
// don't forget to use the corresponding shader program first
lightingShader.use();
lightingShader.setVec3("objectColor", 1.0f, 0.5f, 0.31f);
lightingShader.setVec3("lightColor", 1.0f, 1.0f, 1.0f);
```

One thing left to note is that when we start to update these *lighting shaders* in the next chapters, the light source cube would also be affected and this is not what we want. We don't want the light source object's color to be affected the lighting calculations, but rather keep the light source isolated from the rest. We want the light source to have a constant bright color, unaffected by other color changes (this makes it look like the light source cube really is the source of the light).

To accomplish this we need to create a second set of shaders that we'll use to draw the light source cube, thus being safe from any changes to the lighting shaders. The vertex shader is the same

as the lighting vertex shader so you can simply copy the source code over. The fragment shader of the light source cube ensures the cube's color remains bright by defining a constant white color on the lamp:

```
#version 330 core
out vec4 FragColor;

void main()
{
    FragColor = vec4(1.0); // set all 4 vector values to 1.0
}
```

When we want to render, we want to render the container object (or possibly many other objects) using the lighting shader we just defined, and when we want to draw the light source we use the light source's shaders. During the Lighting chapters we'll gradually be updating the lighting shaders to slowly achieve more realistic results.

The main purpose of the light source cube is to show where the light comes from. We usually define a light source's position somewhere in the scene, but this is simply a position that has no visual meaning. To show where the light source actually is we render a cube at the same location of the light source. We render this cube with the light source cube shader to make sure the cube always stays white, regardless of the light conditions of the scene.

So let's declare a global `vec3` variable that represents the light source's location in world-space coordinates:

```
glm::vec3 lightPos(1.2f, 1.0f, 2.0f);
```

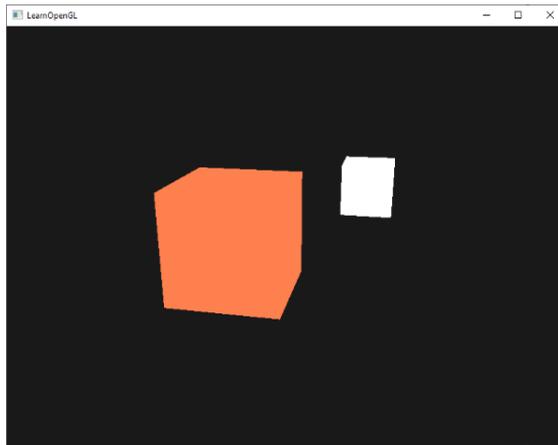
We then translate the light source cube to the light source's position and scale it down before rendering it:

```
model = glm::mat4(1.0f);
model = glm::translate(model, lightPos);
model = glm::scale(model, glm::vec3(0.2f));
```

The resulting render code for the light source cube should then look something like this:

```
lightCubeShader.use();
// set the model, view and projection matrix uniforms
[...]
// draw the light cube object
glBindVertexArray(lightCubeVAO);
glDrawArrays(GL_TRIANGLES, 0, 36);
```

Injecting all the code fragments at their appropriate locations would then result in a clean OpenGL application properly configured for experimenting with lighting. If everything compiles it should look like this:

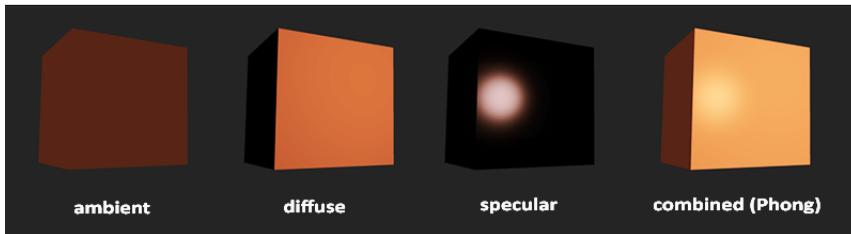


Not really much to look at right now, but I'll promise it'll get more interesting in the upcoming chapters.

If you have difficulties finding out where all the code snippets fit together in the application as a whole, check the source code at `/src/2.lighting/1.colors/` and carefully work your way through the code/comments.

13. Basic Lighting

Lighting in the real world is extremely complicated and depends on way too many factors, something we can't afford to calculate on the limited processing power we have. Lighting in OpenGL is therefore based on approximations of reality using simplified models that are much easier to process and look relatively similar. These lighting models are based on the physics of light as we understand it. One of those models is called the **Phong lighting model**. The major building blocks of the Phong lighting model consist of 3 components: ambient, diffuse and specular lighting. Below you can see what these lighting components look like on their own and combined:



- **Ambient lighting:** even when it is dark there is usually still some light somewhere in the world (the moon, a distant light) so objects are almost never completely dark. To simulate this we use an ambient lighting constant that always gives the object some color.
- **Diffuse lighting:** simulates the directional impact a light object has on an object. This is the most visually significant component of the lighting model. The more a part of an object faces the light source, the brighter it becomes.
- **Specular lighting:** simulates the bright spot of a light that appears on shiny objects. Specular highlights are more inclined to the color of the light than the color of the object.

To create visually interesting scenes we want to at least simulate these 3 lighting components. We'll start with the simplest one: *ambient lighting*.

13.1 Ambient lighting

Light usually does not come from a single light source, but from many light sources scattered all around us, even when they're not immediately visible. One of the properties of light is that it can scatter and bounce in many directions, reaching spots that aren't directly visible; light can thus *reflect* on other surfaces and have an indirect impact on the lighting of an object. Algorithms that take this into consideration are called **global illumination** algorithms, but these are complicated and expensive to calculate.

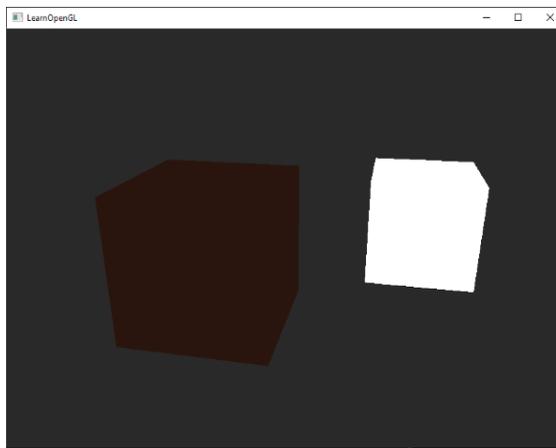
Since we're not big fans of complicated and expensive algorithms we'll start by using a very simplistic model of global illumination, namely **ambient lighting**. As you've seen in the previous section we use a small constant (light) color that we add to the final resulting color of the object's fragments, thus making it look like there is always some scattered light even when there's not a direct light source.

Adding ambient lighting to the scene is really easy. We take the light's color, multiply it with a small constant ambient factor, multiply this with the object's color, and use that as the fragment's color in the cube object's shader:

```
void main()
{
    float ambientStrength = 0.1;
    vec3 ambient = ambientStrength * lightColor;

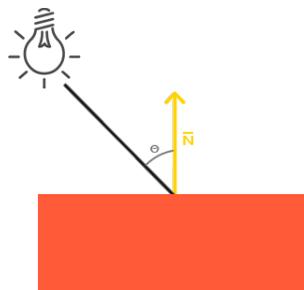
    vec3 result = ambient * objectColor;
    FragColor = vec4(result, 1.0);
}
```

If you'd now run the program, you'll notice that the first stage of lighting is now successfully applied to the object. The object is quite dark, but not completely since ambient lighting is applied (note that the light cube is unaffected because we use a different shader). It should look something like this:



13.2 Diffuse lighting

Ambient lighting by itself doesn't produce the most interesting results, but diffuse lighting however will start to give a significant visual impact on the object. Diffuse lighting gives the object more brightness the closer its fragments are aligned to the light rays from a light source. To give you a better understanding of diffuse lighting take a look at the following image:



To the left we find a light source with a light ray targeted at a single fragment of our object. We need to measure at what angle the light ray touches the fragment. If the light ray is perpendicular to the object's surface the light has the greatest impact. To measure the angle between the light ray and the fragment we use something called a **normal vector**, that is a vector perpendicular to the fragment's surface (here depicted as a yellow arrow); we'll get to that later. The angle between the

two vectors can then easily be calculated with the dot product.

You may remember from the *Transformations* chapter that, the lower the angle between two unit vectors, the more the dot product is inclined towards a value of 1. When the angle between both vectors is 90 degrees, the dot product becomes 0. The same applies to θ : the larger θ becomes, the less of an impact the light should have on the fragment's color.

Note that to get (only) the cosine of the angle between both vectors we will work with *unit vectors* (vectors of length 1) so we need to make sure all the vectors are normalized, otherwise the dot product returns more than just the cosine (see the *Transformations* chapters).

The resulting dot product thus returns a scalar that we can use to calculate the light's impact on the fragment's color, resulting in differently lit fragments based on their orientation towards the light.

So, what do we need to calculate diffuse lighting:

- Normal vector: a vector that is perpendicular to the vertex' surface.
- The directed light ray: a direction vector that is the difference vector between the light's position and the fragment's position. To calculate this light ray we need the light's position vector and the fragment's position vector.

13.3 Normal vectors

A normal vector is a (unit) vector that is perpendicular to the surface of a vertex. Since a vertex by itself has no surface (it's just a single point in space) we retrieve a normal vector by using its surrounding vertices to figure out the surface of the vertex. We can use a little trick to calculate the normal vectors for all the cube's vertices by using the cross product, but since a 3D cube is not a complicated shape we can simply manually add them to the vertex data. We'll list the updated vertex data array next. Try to visualize that the normals are indeed vectors perpendicular to each plane's surface (a cube consists of 6 planes).

```
float vertices[] = {
    // positions          // normals          // texture coords
    -0.5f, -0.5f, -0.5f, 0.0f, 0.0f, -1.0f, 0.0f, 0.0f,
     0.5f, -0.5f, -0.5f, 0.0f, 0.0f, -1.0f, 1.0f, 0.0f,
     0.5f, 0.5f, -0.5f, 0.0f, 0.0f, -1.0f, 1.0f, 1.0f,
     0.5f, 0.5f, -0.5f, 0.0f, 0.0f, -1.0f, 1.0f, 1.0f,
    -0.5f, 0.5f, -0.5f, 0.0f, 0.0f, -1.0f, 0.0f, 1.0f,
    -0.5f, -0.5f, -0.5f, 0.0f, 0.0f, -1.0f, 0.0f, 0.0f,

    -0.5f, -0.5f, 0.5f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f,
     0.5f, -0.5f, 0.5f, 0.0f, 0.0f, 1.0f, 1.0f, 0.0f,
     0.5f, 0.5f, 0.5f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f,
     0.5f, 0.5f, 0.5f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f,
    -0.5f, 0.5f, 0.5f, 0.0f, 0.0f, 1.0f, 0.0f, 1.0f,
    -0.5f, -0.5f, 0.5f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f,

    -0.5f, 0.5f, 0.5f, -1.0f, 0.0f, 0.0f, 1.0f, 0.0f,
    -0.5f, 0.5f, -0.5f, -1.0f, 0.0f, 0.0f, 1.0f, 1.0f,
    -0.5f, -0.5f, -0.5f, -1.0f, 0.0f, 0.0f, 0.0f, 1.0f,
    -0.5f, -0.5f, -0.5f, -1.0f, 0.0f, 0.0f, 0.0f, 1.0f,
    -0.5f, -0.5f, 0.5f, -1.0f, 0.0f, 0.0f, 0.0f, 0.0f,
    -0.5f, 0.5f, 0.5f, -1.0f, 0.0f, 0.0f, 1.0f, 0.0f,
```

```

    0.5f,  0.5f,  0.5f,  1.0f,  0.0f,  0.0f,  1.0f,  0.0f,
    0.5f,  0.5f, -0.5f,  1.0f,  0.0f,  0.0f,  1.0f,  1.0f,
    0.5f, -0.5f, -0.5f,  1.0f,  0.0f,  0.0f,  0.0f,  1.0f,
    0.5f, -0.5f,  0.5f,  1.0f,  0.0f,  0.0f,  0.0f,  1.0f,
    0.5f,  0.5f,  0.5f,  1.0f,  0.0f,  0.0f,  1.0f,  0.0f,

    -0.5f, -0.5f, -0.5f,  0.0f, -1.0f,  0.0f,  0.0f,  1.0f,
    0.5f, -0.5f, -0.5f,  0.0f, -1.0f,  0.0f,  1.0f,  1.0f,
    0.5f, -0.5f,  0.5f,  0.0f, -1.0f,  0.0f,  1.0f,  0.0f,
    0.5f, -0.5f,  0.5f,  0.0f, -1.0f,  0.0f,  1.0f,  0.0f,
    -0.5f, -0.5f,  0.5f,  0.0f, -1.0f,  0.0f,  0.0f,  1.0f,
    -0.5f, -0.5f, -0.5f,  0.0f, -1.0f,  0.0f,  0.0f,  1.0f,

    -0.5f,  0.5f, -0.5f,  0.0f,  1.0f,  0.0f,  0.0f,  1.0f,
    0.5f,  0.5f, -0.5f,  0.0f,  1.0f,  0.0f,  1.0f,  1.0f,
    0.5f,  0.5f,  0.5f,  0.0f,  1.0f,  0.0f,  1.0f,  0.0f,
    0.5f,  0.5f,  0.5f,  0.0f,  1.0f,  0.0f,  1.0f,  0.0f,
    -0.5f,  0.5f,  0.5f,  0.0f,  1.0f,  0.0f,  0.0f,  1.0f,
    -0.5f,  0.5f, -0.5f,  0.0f,  1.0f,  0.0f,  0.0f,  1.0f
};

```

Since we added extra data to the vertex array we should update the cube's vertex shader:

```

#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
...

```

Now that we added a normal vector to each of the vertices and updated the vertex shader we should update the vertex attribute pointers as well. Note that the light source's cube uses the same vertex array for its vertex data, but the lamp shader has no use of the newly added normal vectors. We don't have to update the lamp's shaders or attribute configurations, but we have to at least modify the vertex attribute pointers to reflect the new vertex array's size:

```

glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float),
                    (void*)0);
glEnableVertexAttribArray(0);

```

We only want to use the first 3 floats of each vertex and ignore the last 5 floats so we only need to update the *stride* parameter to 8 times the size of a `float` and we're done.

It may look inefficient using vertex data that is not completely used by the lamp shader, but the vertex data is already stored in the GPU's memory from the container object so we don't have to store new data into the GPU's memory. This actually makes it more efficient compared to allocating a new VBO specifically for the lamp.

All the lighting calculations are done in the fragment shader so we need to forward the normal vectors from the vertex shader to the fragment shader. Let's do that:

```

out vec3 Normal;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    Normal = aNormal;
}

```

What's left to do is declare the corresponding input variable in the fragment shader:

```
in vec3 Normal;
```

13.4 Calculating the diffuse color

We now have the normal vector for each vertex, but we still need the light's position vector and the fragment's position vector. Since the light's position is a single static variable we can declare it as a uniform in the fragment shader:

```
uniform vec3 lightPos;
```

And then update the uniform in the render loop (or outside since it doesn't change per frame). We use the `lightPos` vector declared in the previous chapter as the location of the diffuse light source:

```
lightingShader.setVec3("lightPos", lightPos);
```

Then the last thing we need is the actual fragment's position. We're going to do all the lighting calculations in world space so we want a vertex position that is in world space first. We can accomplish this by multiplying the vertex position attribute with the model matrix only (not the view and projection matrix) to transform it to world space coordinates. This can easily be accomplished in the vertex shader so let's declare an output variable and calculate its world space coordinates:

```

out vec3 FragPos;
out vec3 Normal;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    FragPos = vec3(model * vec4(aPos, 1.0));
    Normal = aNormal;
}

```

And lastly add the corresponding input variable to the fragment shader:

```
in vec3 FragPos;
```

This `in` variable will be interpolated from the 3 world position vectors of the triangle to form the `FragPos` vector that is the per-fragment world position. Now that all the required variables are set we can start the lighting calculations.

The first thing we need to calculate is the direction vector between the light source and the fragment's position. From the previous section we know that the light's direction vector is the difference vector between the light's position vector and the fragment's position vector. As you may

remember from the *Transformations* chapter we can easily calculate this difference by subtracting both vectors from each other. We also want to make sure all the relevant vectors end up as unit vectors so we normalize both the normal and the resulting direction vector:

```
vec3 norm = normalize(Normal);  
vec3 lightDir = normalize(lightPos - FragPos);
```

When calculating lighting we usually do not care about the magnitude of a vector or their position; we only care about their direction. Because we only care about their direction almost all the calculations are done with unit vectors since it simplifies most calculations (like the dot product). So when doing lighting calculations, make sure you always normalize the relevant vectors to ensure they're actual unit vectors. Forgetting to normalize a vector is a popular mistake.

Next we need to calculate the diffuse impact of the light on the current fragment by taking the dot product between the `norm` and `lightDir` vectors. The resulting value is then multiplied with the light's color to get the diffuse component, resulting in a darker diffuse component the greater the angle between both vectors:

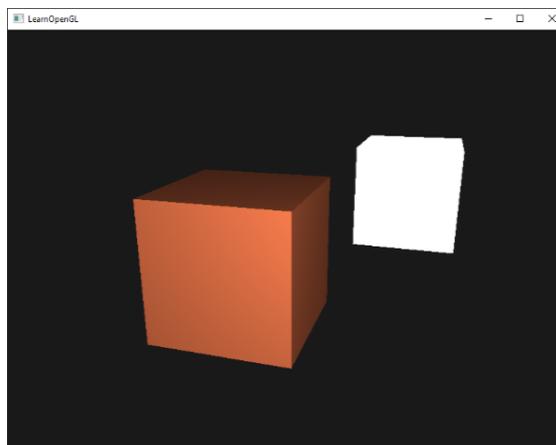
```
float diff = max(dot(norm, lightDir), 0.0);  
vec3 diffuse = diff * lightColor;
```

If the angle between both vectors is greater than 90 degrees then the result of the dot product will actually become negative and we end up with a negative diffuse component. For that reason we use the `max` function that returns the highest of both its parameters to make sure the diffuse component (and thus the colors) never become negative. Lighting for negative colors is not really defined so it's best to stay away from that, unless you're one of those eccentric artists.

Now that we have both an ambient and a diffuse component we add both colors to each other and then multiply the result with the color of the object to get the resulting fragment's output color:

```
vec3 result = (ambient + diffuse) * objectColor;  
FragColor = vec4(result, 1.0);
```

If your application (and shaders) compiled successfully you should see something like this:



You can see that with diffuse lighting the cube starts to look like an actual cube again. Try visualizing the normal vectors in your head and move the camera around the cube to see that the larger the angle between the normal vector and the light's direction vector, the darker the fragment becomes.

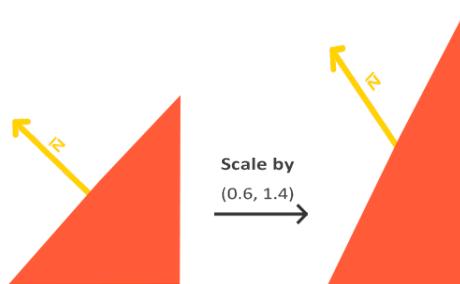
Feel free to compare your source code with the complete source code at `/src/2.lighting/2.1.basic_lighting_diffuse/` if you're stuck.

13.5 One last thing

in the previous section we passed the normal vector directly from the vertex shader to the fragment shader. However, the calculations in the fragment shader are all done in world space, so shouldn't we transform the normal vectors to world space coordinates as well? Basically yes, but it's not as simple as simply multiplying it with a model matrix.

First of all, normal vectors are only direction vectors and do not represent a specific position in space. Second, normal vectors do not have a homogeneous coordinate (the w component of a vertex position). This means that translations should not have any effect on the normal vectors. So if we want to multiply the normal vectors with a model matrix we want to remove the translation part of the matrix by taking the upper-left 3×3 matrix of the model matrix (note that we could also set the w component of a normal vector to 0 and multiply with the 4×4 matrix).

Second, if the model matrix would perform a non-uniform scale, the vertices would be changed in such a way that the normal vector is not perpendicular to the surface anymore. The following image shows the effect such a model matrix (with non-uniform scaling) has on a normal vector:



Whenever we apply a non-uniform scale (note: a uniform scale only changes the normal's magnitude, not its direction, which is easily fixed by normalizing it) the normal vectors are not perpendicular to the corresponding surface anymore which distorts the lighting.

The trick of fixing this behavior is to use a different model matrix specifically tailored for normal vectors. This matrix is called the **normal matrix** and uses a few linear algebraic operations to remove the effect of wrongly scaling the normal vectors. If you want to know how this matrix is calculated, I suggest the normal matrix article¹ from LightHouse3D.

The normal matrix is defined as 'the transpose of the inverse of the upper-left 3×3 part of the model matrix'. Phew, that's a mouthful and if you don't really understand what that means, don't worry; we haven't discussed inverse and transpose matrices yet. Note that most resources define the normal matrix as derived from the model-view matrix, but since we're working in world space (and not in view space) we will derive it from the model matrix.

¹www.lighthouse3d.com/tutorials/glsl-tutorial/the-normal-matrix/

In the vertex shader we can generate the normal matrix by using the `inverse` and `transpose` functions in the vertex shader that work on any matrix type. Note that we cast the matrix to a `3x3` matrix to ensure it loses its translation properties and that it can multiply with the `vec3` normal vector:

```
Normal = mat3(transpose(inverse(model))) * aNormal;
```

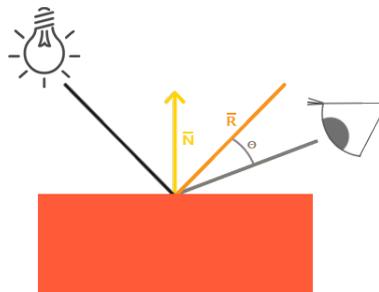
Inverting matrices is a costly operation for shaders, so wherever possible try to avoid doing inverse operations since they have to be done on each vertex of your scene. For learning purposes this is fine, but for an efficient application you'll likely want to calculate the normal matrix on the CPU and send it to the shaders via a uniform before drawing (just like the model matrix).

In the diffuse lighting section the lighting was fine because we didn't do any scaling on the object, so there was not really a need to use a normal matrix and we could've just multiplied the normals with the model matrix. If you are doing a non-uniform scale however, it is essential that you multiply your normal vectors with the normal matrix.

13.6 Specular Lighting

If you're not exhausted already by all the lighting talk we can start finishing the Phong lighting model by adding specular highlights.

Similar to diffuse lighting, specular lighting is based on the light's direction vector and the object's normal vectors, but this time it is also based on the view direction e.g. from what direction the player is looking at the fragment. Specular lighting is based on the reflective properties of surfaces. If we think of the object's surface as a mirror, the specular lighting is the strongest wherever we would see the light reflected on the surface. You can see this effect in the following image:



We calculate a reflection vector by reflecting the light direction around the normal vector. Then we calculate the angular distance between this reflection vector and the view direction. The closer the angle between them, the greater the impact of the specular light. The resulting effect is that we see a bit of a highlight when we're looking at the light's direction reflected via the surface.

The view vector is the one extra variable we need for specular lighting which we can calculate using the viewer's world space position and the fragment's position. Then we calculate the specular's intensity, multiply this with the light color and add this to the ambient and diffuse components.

We chose to do the lighting calculations in world space, but most people tend to prefer doing lighting in view space. An advantage of view space is that the viewer's position is always at $(0, 0, 0)$ so you already got the position of the viewer for free. However, I find calculating lighting in world space more intuitive for learning purposes. If you still want to calculate lighting in view space you want to transform all the relevant vectors with the view matrix as well (don't forget to change the normal matrix too).

To get the world space coordinates of the viewer we simply take the position vector of the camera object (which is the viewer of course). So let's add another uniform to the fragment shader and pass the camera position vector to the shader:

```
uniform vec3 viewPos;
```

```
lightingShader.setVec3("viewPos", camera.Position);
```

Now that we have all the required variables we can calculate the specular intensity. First we define a specular intensity value to give the specular highlight a medium-bright color so that it doesn't have too much of an impact:

```
float specularStrength = 0.5;
```

If we would set this to `1.0f` we'd get a really bright specular component which is a bit too much for a coral cube. In the next chapter we'll talk about properly setting all these lighting intensities and how they affect the objects. Next we calculate the view direction vector and the corresponding reflect vector along the normal axis:

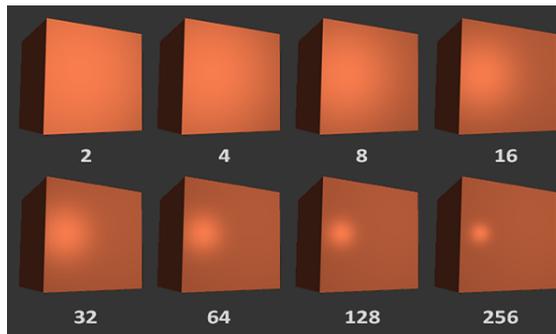
```
vec3 viewDir = normalize(viewPos - FragPos);  
vec3 reflectDir = reflect(-lightDir, norm);
```

Note that we negate the `lightDir` vector. The `reflect` function expects the first vector to point **from** the light source towards the fragment's position, but the `lightDir` vector is currently pointing the other way around: from the fragment **towards** the light source (this depends on the order of subtraction earlier on when we calculated the `lightDir` vector). To make sure we get the correct `reflect` vector we reverse its direction by negating the `lightDir` vector first. The second argument expects a normal vector so we supply the normalized `norm` vector.

Then what's left to do is to actually calculate the specular component. This is accomplished with the following formula:

```
float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);  
vec3 specular = specularStrength * spec * lightColor;
```

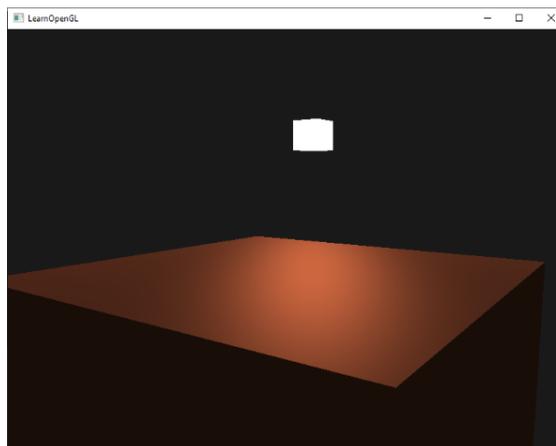
We first calculate the dot product between the view direction and the reflect direction (and make sure it's not negative) and then raise it to the power of 32. This 32 value is the `shininess` value of the highlight. The higher the shininess value of an object, the more it properly reflects the light instead of scattering it all around and thus the smaller the highlight becomes. Below you can see an image that shows the visual impact of different shininess values:



We don't want the specular component to be too distracting so we keep the exponent at 32. The only thing left to do is to add it to the ambient and diffuse components and multiply the combined result with the object's color:

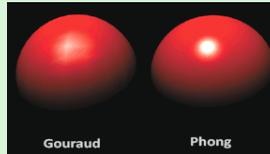
```
vec3 result = (ambient + diffuse + specular) * objectColor;  
FragColor = vec4(result, 1.0);
```

We now calculated all the lighting components of the Phong lighting model. Based on your point of view you should see something like this:



You can find the complete source code of the application at `/src/2.lighting/2.2.basic_lighting_specular/`.

In the earlier days of lighting shaders, developers used to implement the Phong lighting model in the vertex shader. The advantage of doing lighting in the vertex shader is that it is a lot more efficient since there are generally a lot less vertices compared to fragments, so the (expensive) lighting calculations are done less frequently. However, the resulting color value in the vertex shader is the resulting lighting color of that vertex only and the color values of the surrounding fragments are then the result of interpolated lighting colors. The result was that the lighting was not very realistic unless large amounts of vertices were used:



When the Phong lighting model is implemented in the vertex shader it is called **Gouraud shading** instead of **Phong shading**. Note that due to the interpolation the lighting looks somewhat off. The Phong shading gives much smoother lighting results.

By now you should be starting to see just how powerful shaders are. With little information shaders are able to calculate how lighting affects the fragment's colors for all our objects. In the next chapters we'll be delving much deeper into what we can do with the lighting model.

13.7 Exercises

- Right now the light source is a boring static light source that doesn't move. Try to move the light source around the scene over time using either `sin` or `cos`. Watching the lighting change over time gives you a good understanding of Phong's lighting model. Solution: /src/2.lighting/2.3.basic_lighting_exercise1/.
- Play around with different ambient, diffuse and specular strengths and see how they impact the result. Also experiment with the shininess factor. Try to comprehend why certain values have a certain visual output.
- Do Phong shading in view space instead of world space. Solution: /src/2.lighting/2.4.basic_lighting_exercise2/.
- Implement Gouraud shading instead of Phong shading. If you did things right the lighting should look a bit off as you can see at: learnopengl.com/img/lighting/basic_lighting_exercise3.png (especially the specular highlights) with the cube object. Try to reason why it looks so weird. Solution: /src/2.lighting/2.5.basic_lighting_exercise3/.

14. Materials

In the real world, each object has a different reaction to light. Steel objects are often shinier than a clay vase for example and a wooden container doesn't react the same to light as a steel container. Some objects reflect the light without much scattering resulting in small specular highlights and others scatter a lot giving the highlight a larger radius. If we want to simulate several types of objects in OpenGL we have to define **material** properties specific to each surface.

In the previous chapter we defined an object and light color to define the visual output of the object, combined with an ambient and specular intensity component. When describing a surface we can define a material color for each of the 3 lighting components: ambient, diffuse and specular lighting. By specifying a color for each of the components we have fine-grained control over the color output of the surface. Now add a shininess component to those 3 colors and we have all the material properties we need:

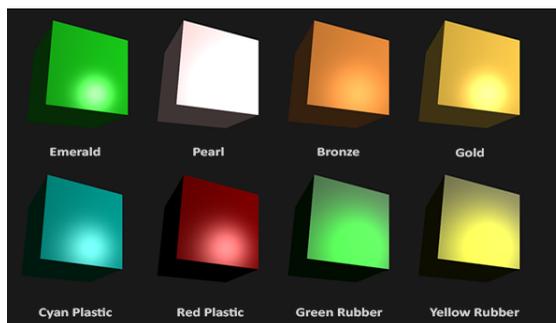
```
#version 330 core
struct Material {
    vec3  ambient;
    vec3  diffuse;
    vec3  specular;
    float shininess;
};

uniform Material material;
```

In the fragment shader we create a `struct` to store the material properties of the surface. We can also store them as individual uniform values, but storing them as a struct keeps it more organized. We first define the layout of the struct and then simply declare a uniform variable with the newly created struct as its type.

As you can see, we define a color vector for each of the Phong lighting's components. The `ambient` material vector defines what color the surface reflects under ambient lighting; this is usually the same as the surface's color. The `diffuse` material vector defines the color of the surface under diffuse lighting. The diffuse color is (just like ambient lighting) set to the desired surface's color. The `specular` material vector sets the color of the specular highlight on the surface (or possibly even reflect a surface-specific color). Lastly, the `shininess` impacts the scattering/radius of the specular highlight.

With these 4 components that define an object's material we can simulate many real-world materials. A table as found at devernavy.free.fr/cours/opengl/materials.html shows a list of material properties that simulate real materials found in the outside world. The following image shows the effect several of these real world material values have on our cube:



As you can see, by correctly specifying the material properties of a surface it seems to change

the perception we have of the object. The effects are clearly noticeable, but for the more realistic results we'll need to replace the cube with something more complicated. In the later *Model Loading* chapters we'll discuss more complicated shapes.

Figuring out the right material settings for an object is a difficult feat that mostly requires experimentation and a lot of experience. It's not that uncommon to completely destroy the visual quality of an object by a misplaced material.

Let's try implementing such a material system in the shaders.

14.1 Setting materials

We created a uniform material struct in the fragment shader so next we want to change the lighting calculations to comply with the new material properties. Since all the material variables are stored in a struct we can access them from the `material` uniform:

```
void main()
{
    // ambient
    vec3 ambient = lightColor * material.ambient;

    // diffuse
    vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(lightPos - FragPos);
    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = lightColor * (diff * material.diffuse);

    // specular
    vec3 viewDir = normalize(viewPos - FragPos);
    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0),
                    material.shininess);
    vec3 specular = lightColor * (spec * material.specular);

    vec3 result = ambient + diffuse + specular;
    FragColor = vec4(result, 1.0);
}
```

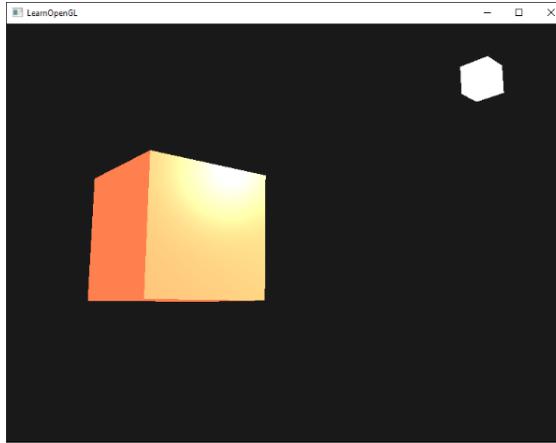
As you can see we now access all of the material struct's properties wherever we need them and this time calculate the resulting output color with the help of the material's colors. Each of the object's material attributes are multiplied with their respective lighting components.

We can set the material of the object in the application by setting the appropriate uniforms. A struct in GLSL however is not special in any regard when setting uniforms; a struct only really acts as a namespace of uniform variables. If we want to fill the struct we will have to set the individual uniforms, but prefixed with the struct's name:

```
lightingShader.setVec3("material.ambient", 1.0f, 0.5f, 0.31f);
lightingShader.setVec3("material.diffuse", 1.0f, 0.5f, 0.31f);
lightingShader.setVec3("material.specular", 0.5f, 0.5f, 0.5f);
lightingShader.setFloat("material.shininess", 32.0f);
```

We set the ambient and diffuse component to the color we'd like the object to have and set the specular component of the object to a medium-bright color; we don't want the specular component to be too strong. We also keep the shininess at 32.

We can now easily influence the object's material from the application. Running the program gives you something like this:



It doesn't really look right though?

14.2 Light properties

The object is way too bright. The reason for the object being too bright is that the ambient, diffuse and specular colors are reflected with full force from any light source. Light sources also have different intensities for their ambient, diffuse and specular components respectively. In the previous chapter we solved this by varying the ambient and specular intensities with a strength value. We want to do something similar, but this time by specifying intensity vectors for each of the lighting components. If we'd visualize `lightColor` as `vec3(1.0)` the code would look like this:

```
vec3 ambient = vec3(1.0) * material.ambient;
vec3 diffuse = vec3(1.0) * (diff * material.diffuse);
vec3 specular = vec3(1.0) * (spec * material.specular);
```

So each material property of the object is returned with full intensity for each of the light's components. These `vec3(1.0)` values can be influenced individually as well for each light source and this is usually what we want. Right now the ambient component of the object is fully influencing the color of the cube. The ambient component shouldn't really have such a big impact on the final color so we can restrict the ambient color by setting the light's ambient intensity to a lower value:

```
vec3 ambient = vec3(0.1) * material.ambient;
```

We can influence the diffuse and specular intensity of the light source in the same way. This is closely similar to what we did in the previous chapter; you could say we already created some light properties to influence each lighting component individually. We'll want to create something similar to the material struct for the light properties:

```

struct Light {
    vec3 position;

    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};
uniform Light light;

```

A light source has a different intensity for its `ambient`, `diffuse` and `specular` components. The ambient light is usually set to a low intensity because we don't want the ambient color to be too dominant. The diffuse component of a light source is usually set to the exact color we'd like a light to have; often a bright white color. The specular component is usually kept at `vec3(1.0)` shining at full intensity. Note that we also added the light's position vector to the struct.

Just like with the material uniform we need to update the fragment shader:

```

vec3 ambient = light.ambient * material.ambient;
vec3 diffuse = light.diffuse * (diff * material.diffuse);
vec3 specular = light.specular * (spec * material.specular);

```

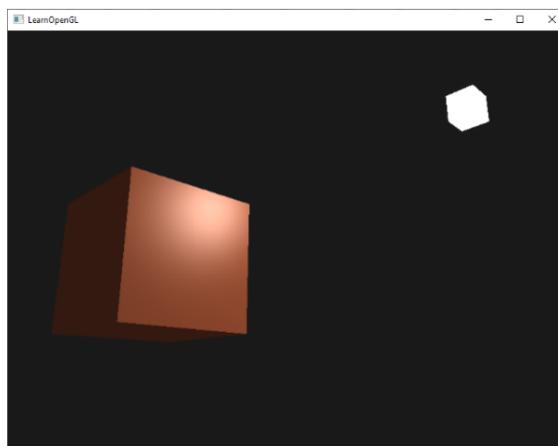
We then want to set the light intensities in the application:

```

lightingShader.setVec3("light.ambient", 0.2f, 0.2f, 0.2f);
lightingShader.setVec3("light.diffuse", 0.5f, 0.5f, 0.5f); // darkened
lightingShader.setVec3("light.specular", 1.0f, 1.0f, 1.0f);

```

Now that we modulated how the light influences the object's material we get a visual output that looks much like the output from the previous chapter. This time however we got full control over the lighting and the material of the object:

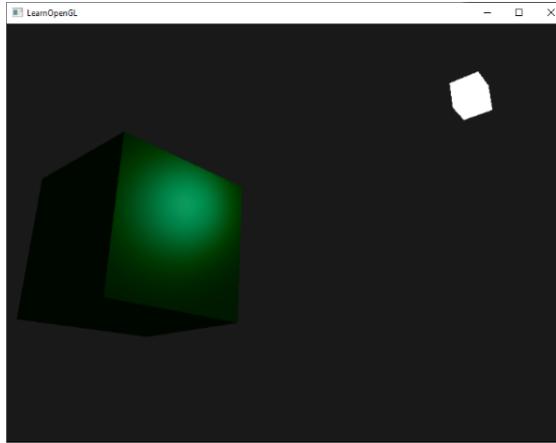


Changing the visual aspects of objects is relatively easy right now. Let's spice things up a bit!

14.3 Different light colors

So far we used light colors to only vary the intensity of their individual components by choosing colors that range from white to gray to black, not affecting the actual colors of the object (only its intensity). Since we now have easy access to the light's properties we can change their colors over

time to get some really interesting effects. Since everything is already set up in the fragment shader, changing the light's colors is easy and immediately creates some funky effects:



As you can see, a different light color greatly influences the object's color output. Since the light color directly influences what colors the object can reflect (as you may remember from the *Colors* chapter) it has a significant impact on the visual output.

We can easily change the light's colors over time by changing the light's ambient and diffuse colors via `sin` and `glfwGetTime`:

```
glm::vec3 lightColor;
lightColor.x = sin(glfwGetTime() * 2.0f);
lightColor.y = sin(glfwGetTime() * 0.7f);
lightColor.z = sin(glfwGetTime() * 1.3f);

glm::vec3 diffuseColor = lightColor * glm::vec3(0.5f);
glm::vec3 ambientColor = diffuseColor * glm::vec3(0.2f);

lightingShader.setVec3("light.ambient", ambientColor);
lightingShader.setVec3("light.diffuse", diffuseColor);
```

Try and experiment with several lighting and material values and see how they affect the visual output. You can find the source code of the application at `/src/2.lighting/3.1.materials/`.

14.4 Exercises

- Can you make it so that changing the light color changes the color of the light's cube object?
- Can you simulate some of the real-world objects by defining their respective materials like we've seen at the start of this chapter? Note that the linked table's ambient values are not the same as the diffuse values; they didn't take light intensities into account. To correctly set their values you'd have to set all the light intensities to `vec3(1.0)` to get the same output. Solution: `/src/2.lighting/3.2.materials_exercise1/` of cyan plastic container.