Politecnico di Torino

Laboratorio 09

OpenGL

Part IV – Lighting and Material

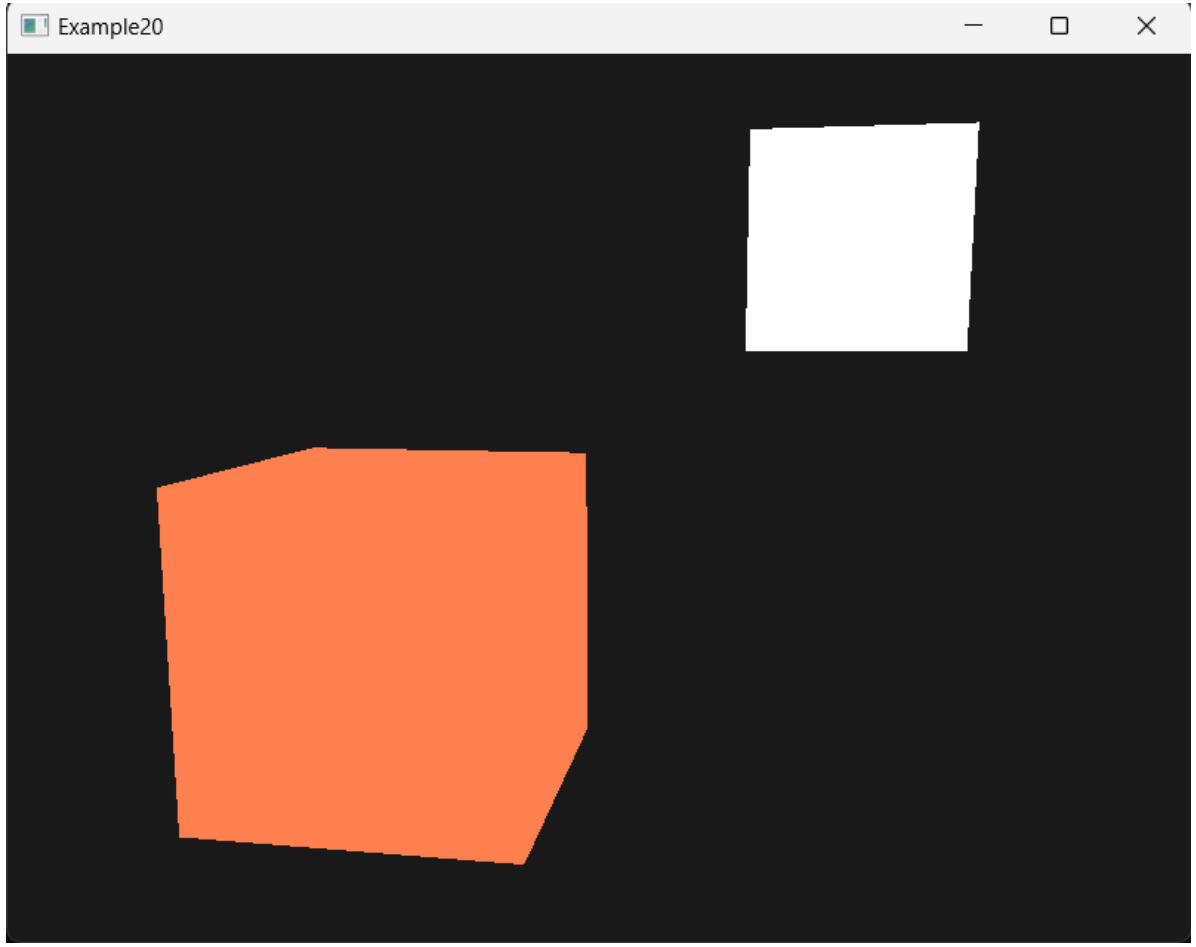
Outline

- Lighting
- Material

A lighting scene

Example20

This example shows a basic set up for a lighting scene.



A lighting scene

Defining objects

In this example, the light source is represented as a visual object in the scene. Therefore, there is the need to add two different objects (i.e., cubes): an object to cast the light on (i.e., the object that receive the lighting) and an object that casts the light in the scene (i.e., light source). For this purpose, two cubes are used. Cubes are defined by using the vertex data of the previous example without texture coordinate.

A lighting scene

Defining objects

The vertex shaders for the cube that receive the lighting is defined as follow:

```
#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);
}
```

A lighting scene

Defining objects

A new VAO is specified to render the light source. Actually, the light source could be rendered with the same VAO and a few light position transformations on the model matrix (as done in the previous examples). However, in the upcoming examples, there will be the need to change the vertex data quite often.

```
unsigned int lightCubeVAO;
glGenVertexArrays(1, &lightCubeVAO);
glBindVertexArray(lightCubeVAO);

glBindBuffer(GL_ARRAY_BUFFER, VBO);

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

A lighting scene

Defining objects

The fragment shader of the object in the scene accepts both an object color and a light color from a uniform variable. The shader multiplies the light's color with the object's (reflected) color to compute the output color.

```
#version 330 core
out vec4 FragColor;

uniform vec3 objectColor;
uniform vec3 lightColor;

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

A lighting scene

Defining objects

The uniforms are set as follow:

```
lightingShader.use();
lightingShader.setVec3("objectColor", 1.0f, 0.5f, 0.31f);
lightingShader.setVec3("lightColor", 1.0f, 1.0f, 1.0f);
```

A lighting scene

Defining objects

A second set of shaders has to be created for the light source. The vertex shader is the same of the object. 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);
}
```

A lighting scene

Defining objects

The main purpose of the light source cube is to show where the light comes from. Usually, a light source's position is defined somewhere in the scene, but this is simply a position that has no visual meaning. To this aim, a global `vec3` variable is used to represents the light source's location in world-space coordinates:

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

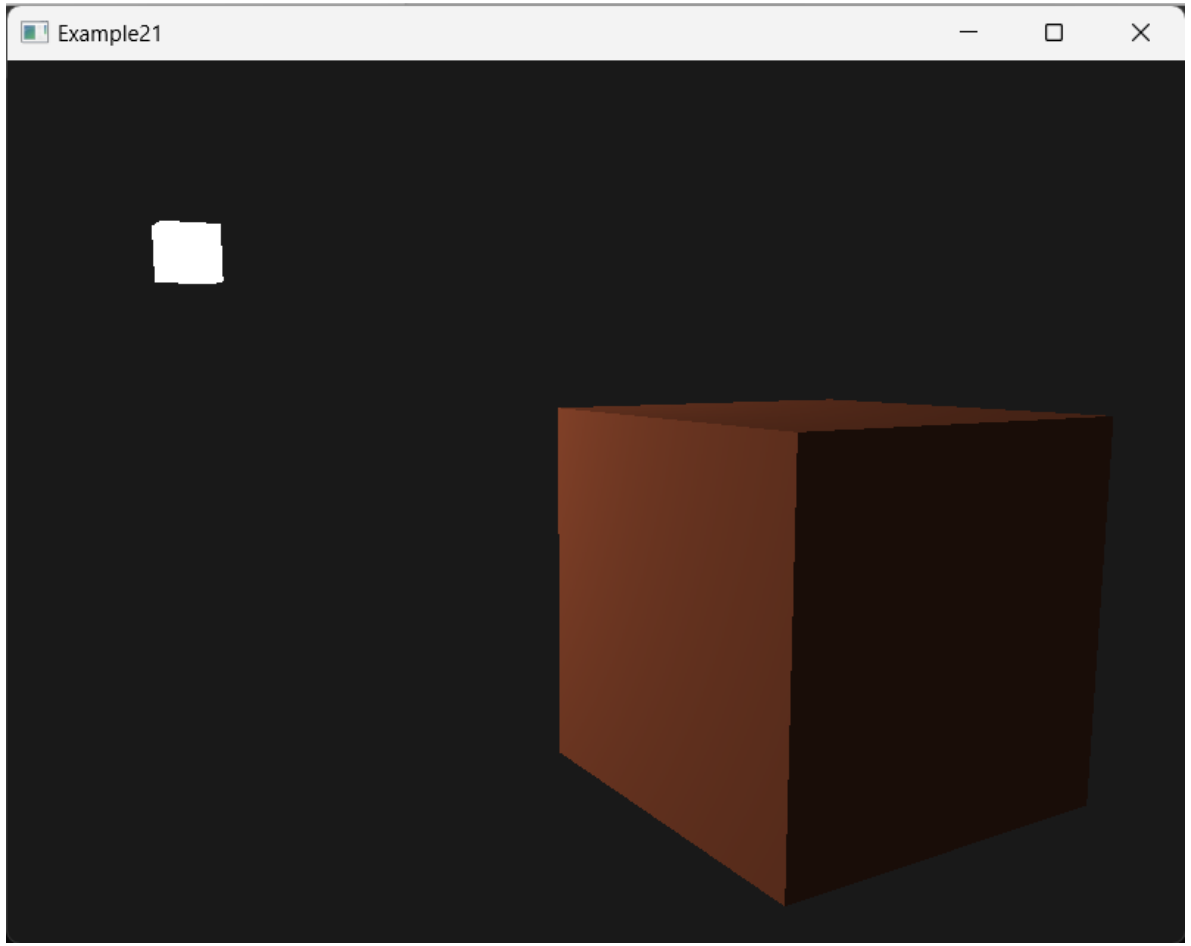
The light source cube is then translated to the light source's position and scale it down before rendering it

```
lightCubeShader.use();
lightCubeShader.setMat4("projection", projection);
lightCubeShader.setMat4("view", view);
model = glm::mat4(1.0f);
model = glm::translate(model, lightPos);
model = glm::scale(model, glm::vec3(0.2f));
lightCubeShader.setMat4("model", model);
```

Basic Lighting

Example21

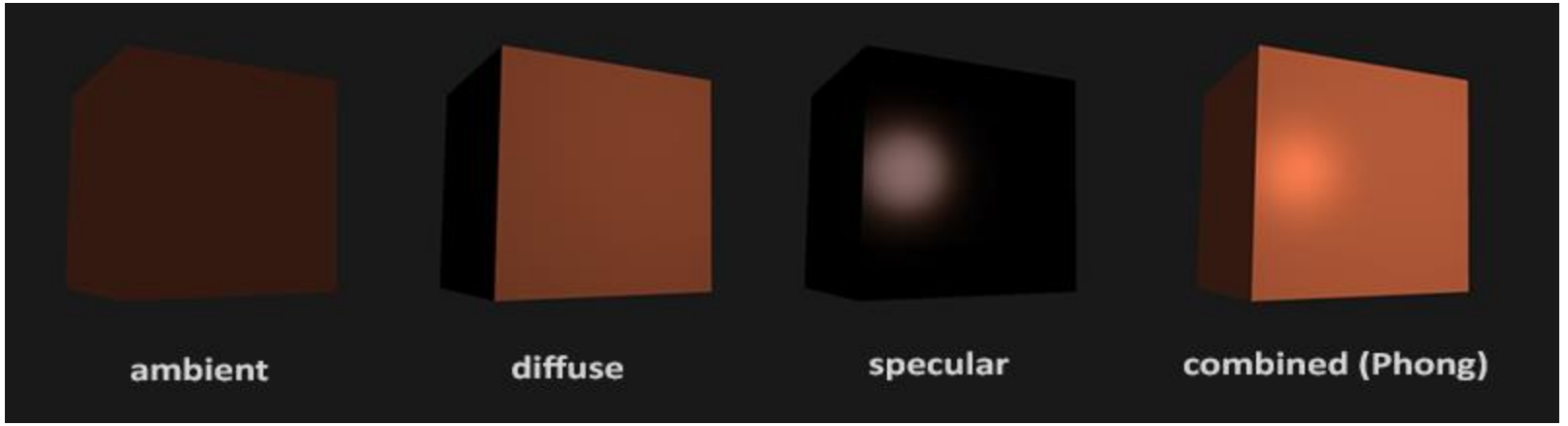
This example shows how to implement lighting model, thus imitating real lighting in OpenGL.



Basic Lighting

Lighting model

Lighting in OpenGL is 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. 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.



To create visually interesting scenes, these 3 lighting components should be simulated at least.

Basic Lighting

Ambient lighting

Light usually does not come from a single light source, but from many light sources scattered all around the scene. 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**. A very simplistic model of global illumination is named **ambient lighting**.

Basic Lighting

Ambient lighting

Adding ambient lighting to the scene is relatively easy. The light's color is first multiplied with a small constant ambient factor. Then the result is multiplied with the object's color to obtain the fragment's color in the cube object's shader.

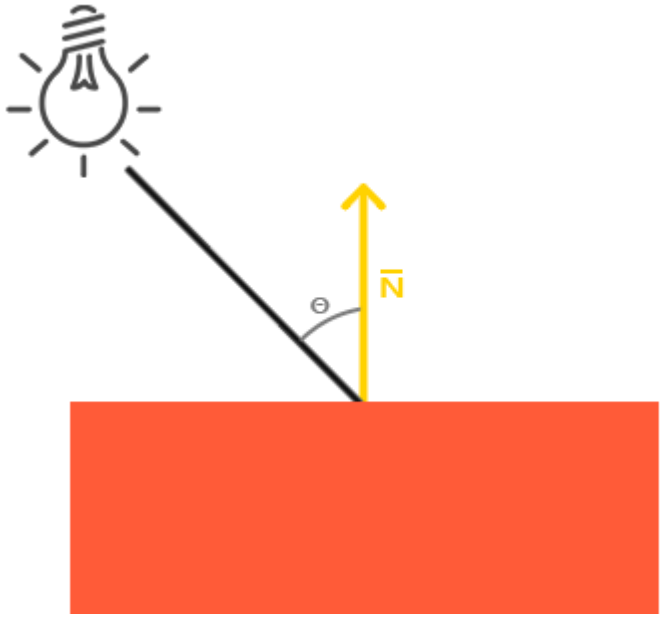
```
void main()
{
    // ambient
    float ambientStrength = 0.1;
    vec3 ambient = ambientStrength * lightColor;

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

Basic Lighting

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. The image helps to understand the diffuse lighting.

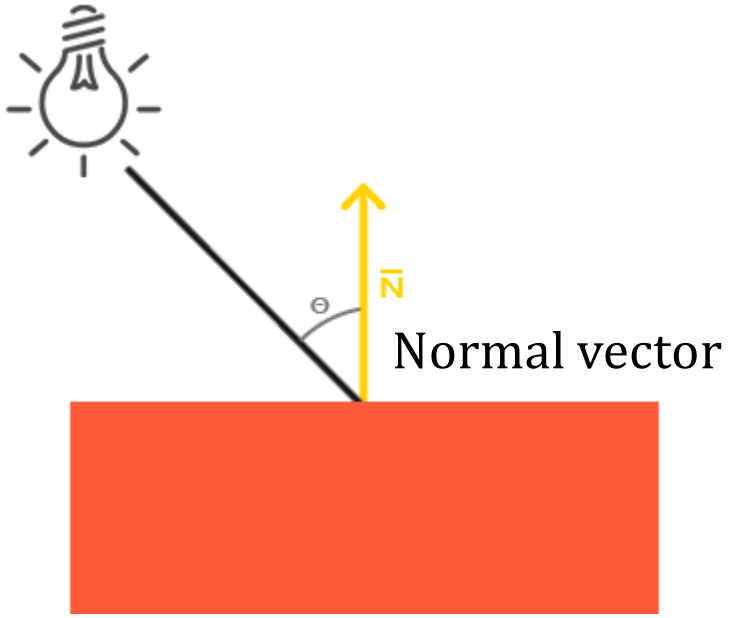
A light source, on the left, is casting a light ray targeting a single fragment of the object’s color. If the light ray is perpendicular to the object's surface the light has the greatest impact. Therefore, it is needed to measure at what angle the light ray touches the fragment.



Basic Lighting

Normal vectors

The normal vector is used to measure the angle between the light ray and the fragment. This is a vector perpendicular to the fragment's surface (the yellow vector). The angle between the two vectors can be calculated with the dot product. The resulting dot product returns a scalar that can be used to calculate the light's impact on the fragment's color, resulting in differently lit fragments based on their orientation towards the light.



Basic Lighting

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), a normal vector is retrieved by using its surrounding vertices to figure out the surface of the vertex. Generally normal vectors are computed by using the cross product, but since a 3D cube is not a complicated shape it is possible to manually add them to the vertex data.

Basic Lighting

Normal vectors

The updated vertex data array are represented alongside. The normals can be regarded as vectors perpendicular to each plane's surface (a cube consists of 6 planes). Since extra data are added to the vertex array, the cube's vertex shader should be updated:

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

...
```

```
float vertices[] = {
     -0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,
      0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,
      0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f,
      0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f,
     -0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f,
     -0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,

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

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

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

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

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

Basic Lighting

Normal vectors

A normal vector is added to each of the vertices. Updating the vertex shader means that the vertex attribute pointers should be updated 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. There is no need to update the lamp's shaders or attribute configurations, only the vertex attribute pointers are modified to reflect the new vertex array's size:

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
```

Basic Lighting

Normal vectors

All the lighting calculations are done in the fragment shader, it is needed to forward the normal vectors from the vertex shader to the fragment shader.

```
...
out vec3 Normal;

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

In the fragment shader the corresponding input variable is declared

```
in vec3 Normal;
```

Basic Lighting

Calculating the diffuse color

Now the normal vector for each vertex is passed to the fragment shader. However, to calculate the diffuse color the light's position vector and the fragment's position vector are needed. Since the light's position is a single static variable, it can be declared as a uniform in the fragment shader.

```
uniform vec3 lightPos;
```

The uniform can be updated in the render loop (or outside if the position of the lights do not change):

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

Basic Lighting

Calculating the diffuse color

For what it concerns the actual fragment's position, it should be notice that all the lighting calculations are done in world space. For this reason, it is needed to transform vertex position to world space coordinates. This can be accomplished by multiplying the vertex position attribute with the model matrix only (not the view and projection matrix). This operation is made in the vertex shader and passed to the fragment shader with a variable.

```
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;
}
```

The corresponding input variable is added to the fragment shader

```
in vec3 FragPos;
```

Basic Lighting

Calculating the diffuse color

To start the lighting calculations, the first thing to do is calculating the direction vector between the light source and the fragment's position. The light's direction vector is the difference vector between the light's position vector and the fragment's position vector. This difference can be computed by subtracting both vectors from each other. To make sure that all the relevant vectors end up as unit vectors, a normalization is applied to both the normal and the resulting direction vector.

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

Basic Lighting

Calculating the diffuse color

Next the diffuse impact of the light on the current fragment is calculated 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, ending up with a negative diffuse component. For that reason, the max function is used that returns the highest of both its parameters to make sure the diffuse component (and thus the colors) never become negative.

Basic Lighting

Calculating the diffuse color

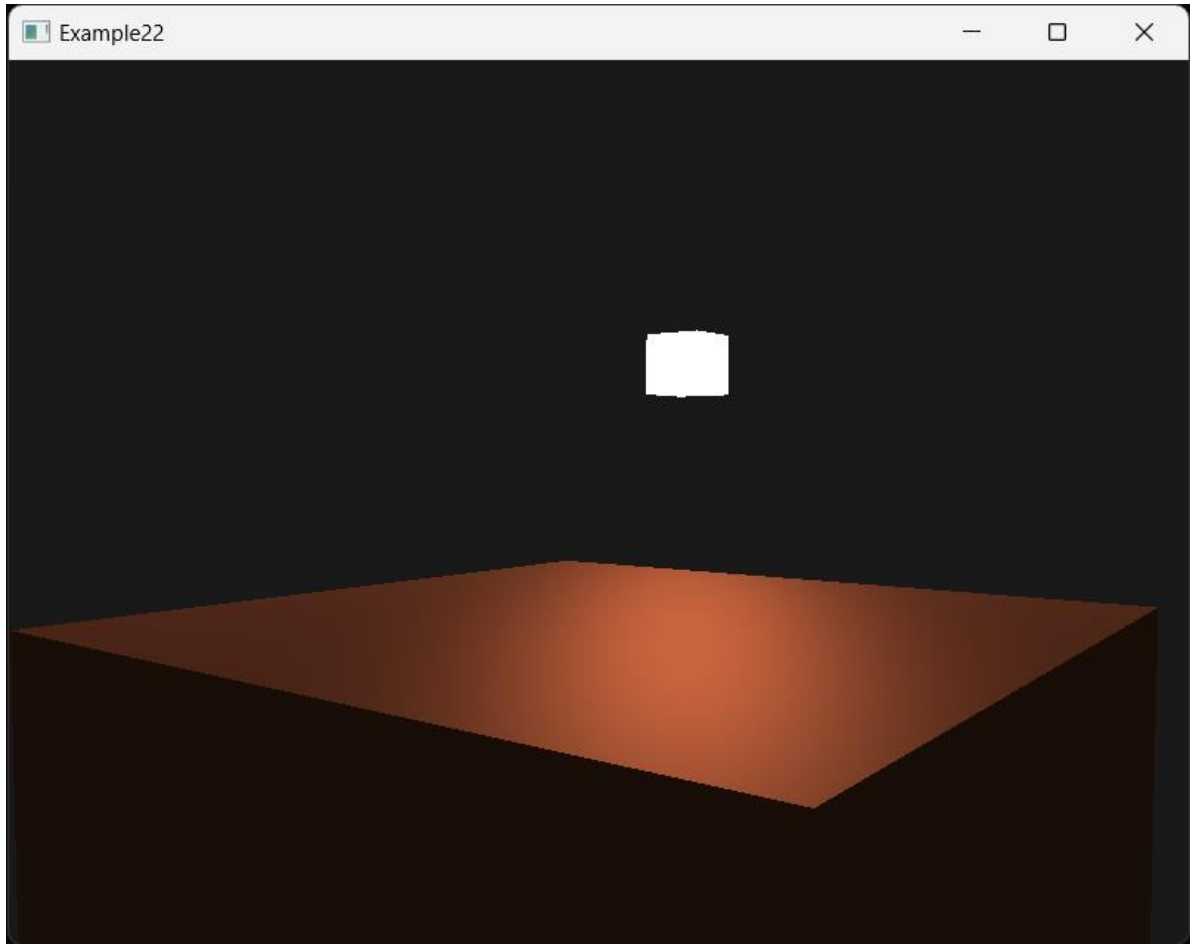
Now that both ambient and diffuse components are computed, they can be added to each other and then the result is multiplied with the color of the object to get the resulting fragment's output color:

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

Specular Lighting

Example22

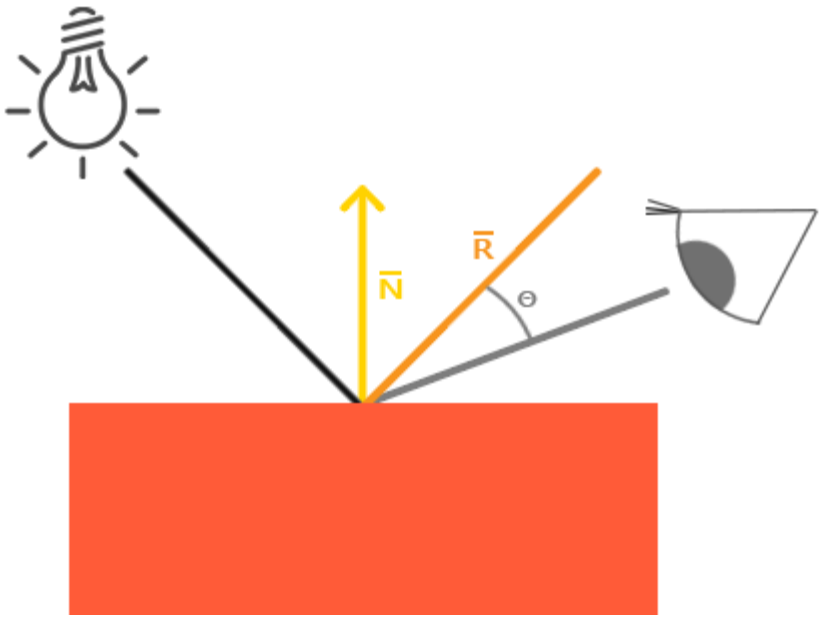
This example shows how to complete the implementation of the Phong lighting model by adding specular highlights.



Specular Lighting

Computing specular lighting

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. The effect is depicted in the image.

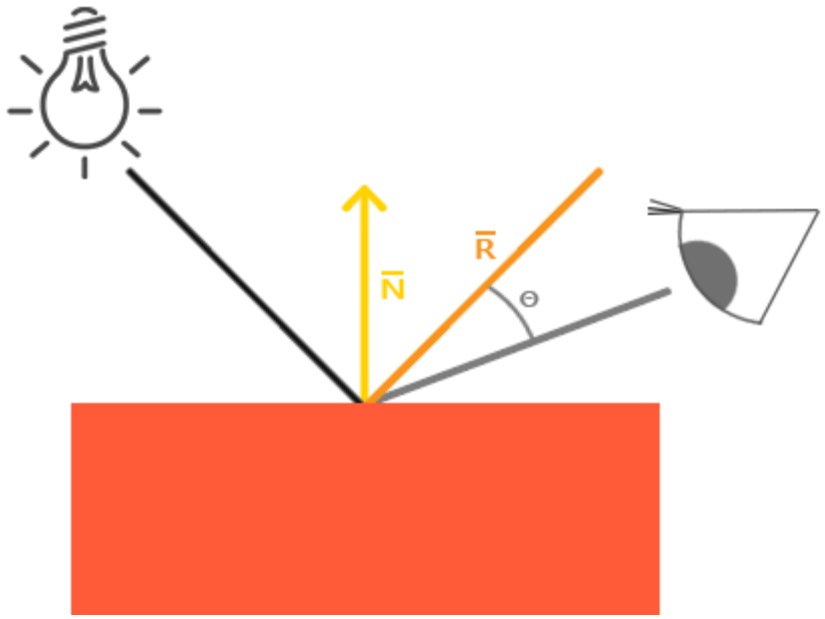


Specular Lighting

Computing specular lighting

A reflection vector is calculated by reflecting the light direction around the normal vector. Then the angular distance between this reflection vector and the view direction is computed. The closer the angle between them, the greater the impact of the specular light.

The view vector is calculated using the viewer's world space position and the fragment's position. Once the specular's intensity is calculated, it is multiplied with the light color and the result added to the ambient and diffuse components.



Specular Lighting

Computing specular lighting

To get the world space coordinates of the viewer, it is possible to consider the position vector of the camera object. To this aim, a uniform is added to the fragment shader.

```
uniform vec3 viewPos;
```

The camera position vector is passed to the shader:

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

Specular Lighting

Computing specular lighting

To compute the specular intensity:

```
float specularStrength = 0.5;
vec3 viewDir = normalize(viewPos - FragPos);
vec3 reflectDir = reflect(-lightDir, norm);
float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
vec3 specular = specularStrength * spec * lightColor;
```

First a strength factor is defined to give the specular highlight a medium-bright color:

Specular Lighting

Computing specular lighting

To compute the specular intensity:

```
float specularStrength = 0.5;
vec3 viewDir = normalize(viewPos - FragPos);
vec3 reflectDir = reflect(-lightDir, norm);
float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
vec3 specular = specularStrength * spec * lightColor;
```

Next the view direction vector and the corresponding reflect vector along the normal axis are calculated. Note that the `lightDir` vector is negated as 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 used earlier to calculate the `lightDir` vector). The second argument expects a normal vector.

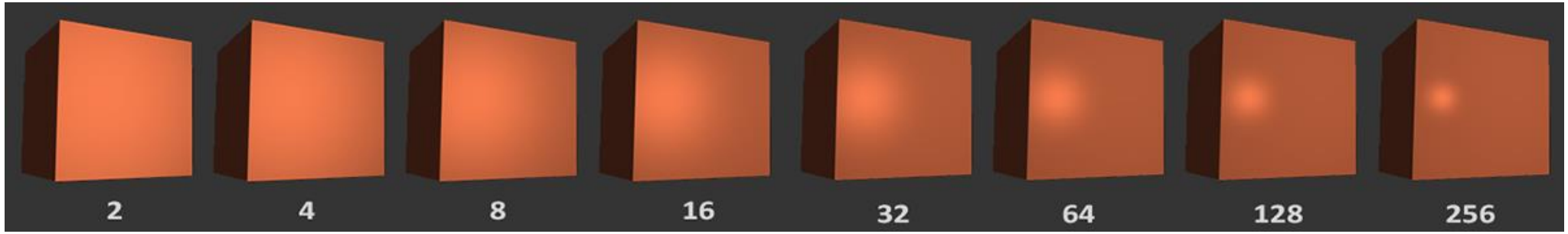
Specular Lighting

Computing specular lighting

To compute the specular intensity:

```
float specularStrength = 0.5;
vec3 viewDir = normalize(viewPos - FragPos);
vec3 reflectDir = reflect(-lightDir, norm);
float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
vec3 specular = specularStrength * spec * lightColor;
```

Finally, the specular component is computed by calculating first the dot product between the view direction and the reflect direction and then raising it to the power of 32. This 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.



Specular Lighting

Computing specular lighting

The last operation envisages to add the specular component 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);
```

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. To simulate several types of objects in OpenGL it is necessary to define material properties specific to each surface.

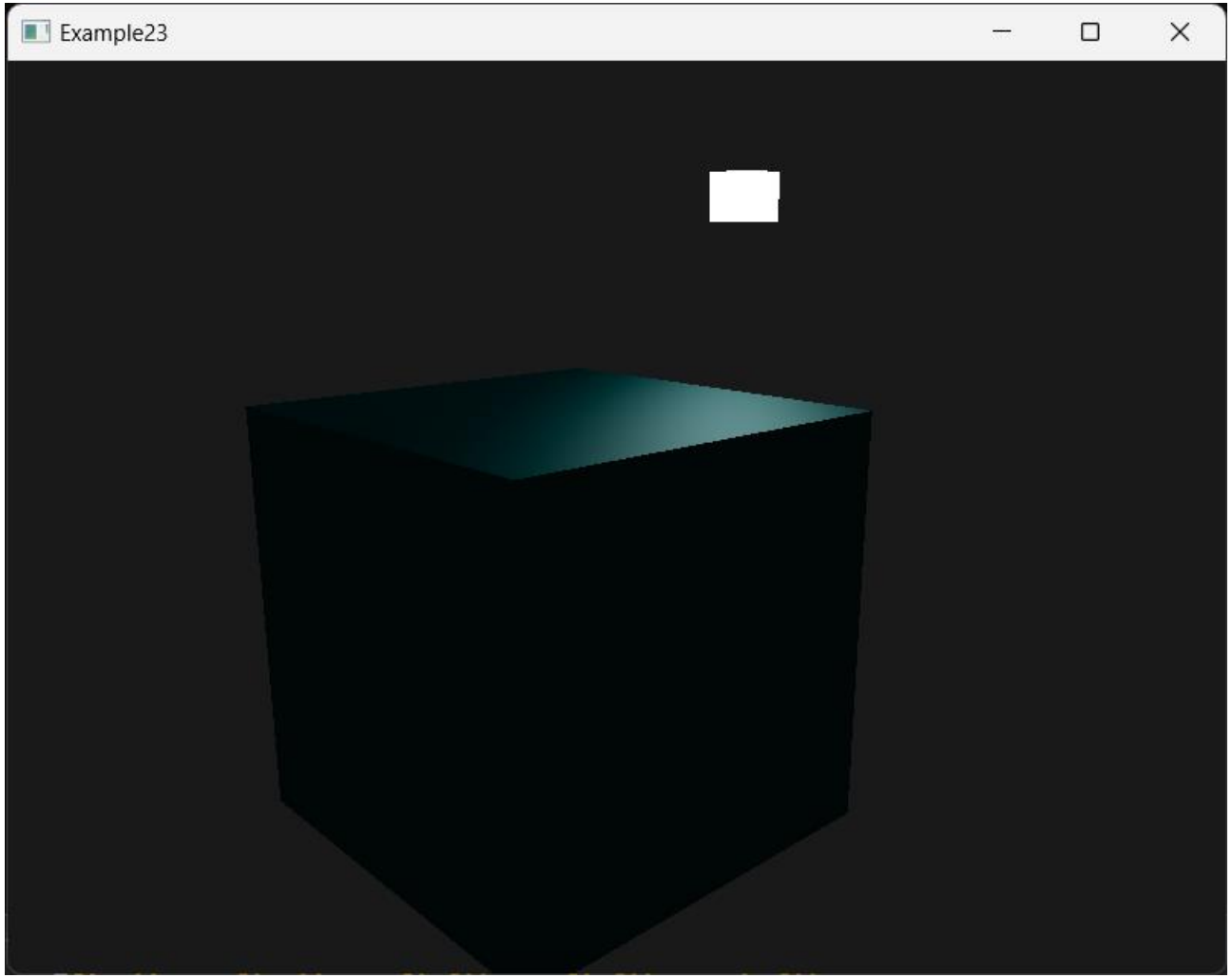
Materials

In the previous examples, an object and light color are defined to describe the visual output of the object, combined with an ambient and specular intensity component. When describing a surface, it is possible to 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 it is possible to have a fine-grained control over the color output of the surface.

Materials

Example23

This example shows how to define a material color for each of the 3 lighting components: ambient, diffuse and specular lighting.



Materials

Material components

In the fragment shader a `struct Material` is created to store the material properties of the surface. A shininess component is added to the ambient, diffuse, and specular component to have all the material properties needed.

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

Materials

Material components

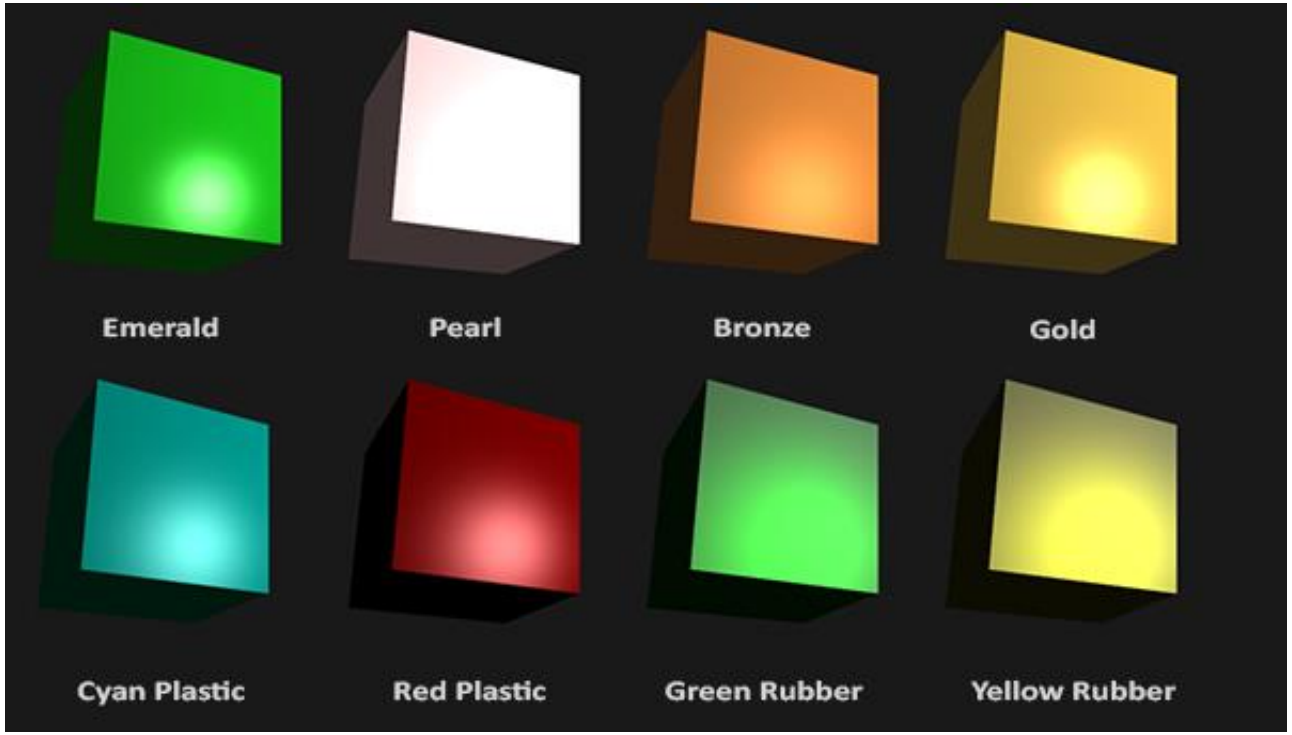
```
#version 330 core
struct Material {
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float shininess;
};
uniform Material material;
```

As can be seen in, a `vec3` (color vector) is defined 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.

Materials

Material components

With these 4 components it is possible to simulate many real-world materials. The table report at http://devernay.free.fr/cours/opengl/materials.html shows a list of material properties that simulate real materials. The image shows the effect several of these real-world material values have:



Materials

Setting materials

A uniform material struct was created in the fragment shader, hence the lighting calculations has to be updated to comply with the new material properties. Since all the material variables are stored in a struct, it is possible to access them from the material uniform.

Materials

Setting materials

```
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);
}
```

Materials

Setting materials

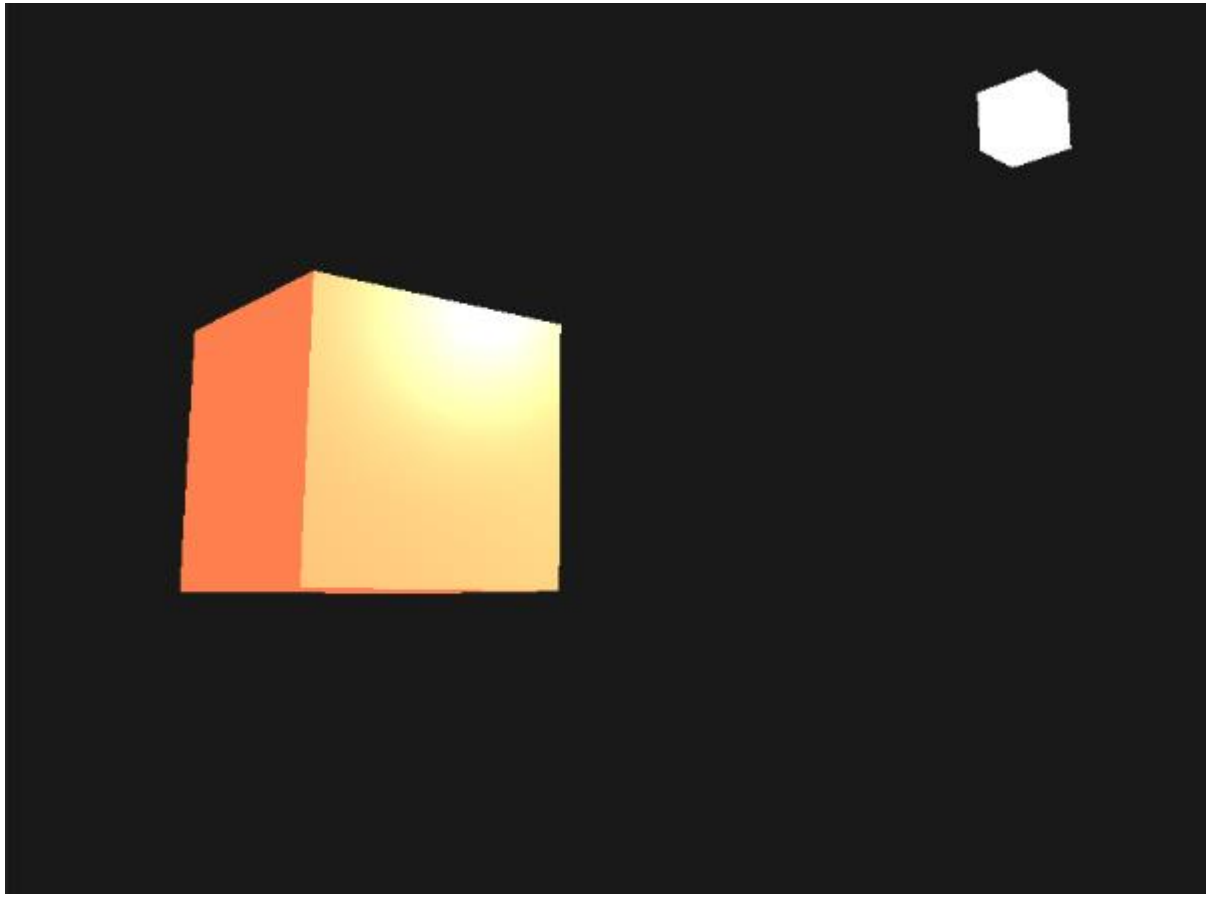
The material of the object can be defined in the application by setting the appropriate uniforms.

```
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);
```

Materials

Light properties

Launching the application with this configurations, it can be noticed that the object appears too bright. The reason for this issue 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 example, the issue was solved by varying the ambient and specular intensities with a strength value.



Materials

Light properties

Similarly, it is possible to specify intensity vectors for each of the lighting components. If the `lightColor` is visualized as `vec3(1.0)`, the code looks like:

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

Each material property of the object is returned with full intensity for each of the light's components. However, these `vec3(1.0)` values should be influenced individually as well for each light source.

Materials

Light properties

Right now, the ambient component of the object is fully influencing the color of the cube. To restrict its impact, it is possible to set the light's ambient intensity to a lower value:

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

The diffuse and specular intensity of the light source can be influenced in the same way. Similar to the material struct introduced in the previous example, it is possible to create a new struct for the light properties:

```
struct Light {
    vec3 position;

    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

uniform Light light;
```

Materials

Light properties

A light source has a different intensity for its ambient, diffuse and specular components. The ambient light is usually set to a low intensity to make the ambient color not too dominant. The diffuse component of a light source is usually set to the exact color of the light to have; often a bright white color. The specular component is usually kept at `vec3(1.0)` shining at full intensity.

Materials

Light properties

The fragment shader should be updated:

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

The light intensities are ser in the application:

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

Materials

Light properties

So far, the light colors are used only to 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 the access to the light's properties is easier than the previous example, it is easy to change the colors over time. A different light color greatly influences the object's color output, since the light color directly influences what colors the object can reflect

Materials

Light properties

Light's colors can be altered over time by changing the light's ambient and diffuse colors via `sin` and `glfwGetTime` function:

```
glm::vec3 lightColor;
lightColor.x = static_cast<float>(sin(glfwGetTime() * 2.0));
lightColor.y = static_cast<float>(sin(glfwGetTime() * 0.7));
lightColor.z = static_cast<float>(sin(glfwGetTime() * 1.3));

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);
lightingShader.setVec3("light.specular", 1.0f, 1.0f, 1.0f);
```

Advanced Lighting

More resources on lighting are available at:

- Lighting maps: https://learnopengl.com/Lighting/Lighting-maps
- Light casters: https://learnopengl.com/Lighting/Light-casters
- Multiple lights: https://learnopengl.com/Lighting/Multiple-lights