CAPITULO 3

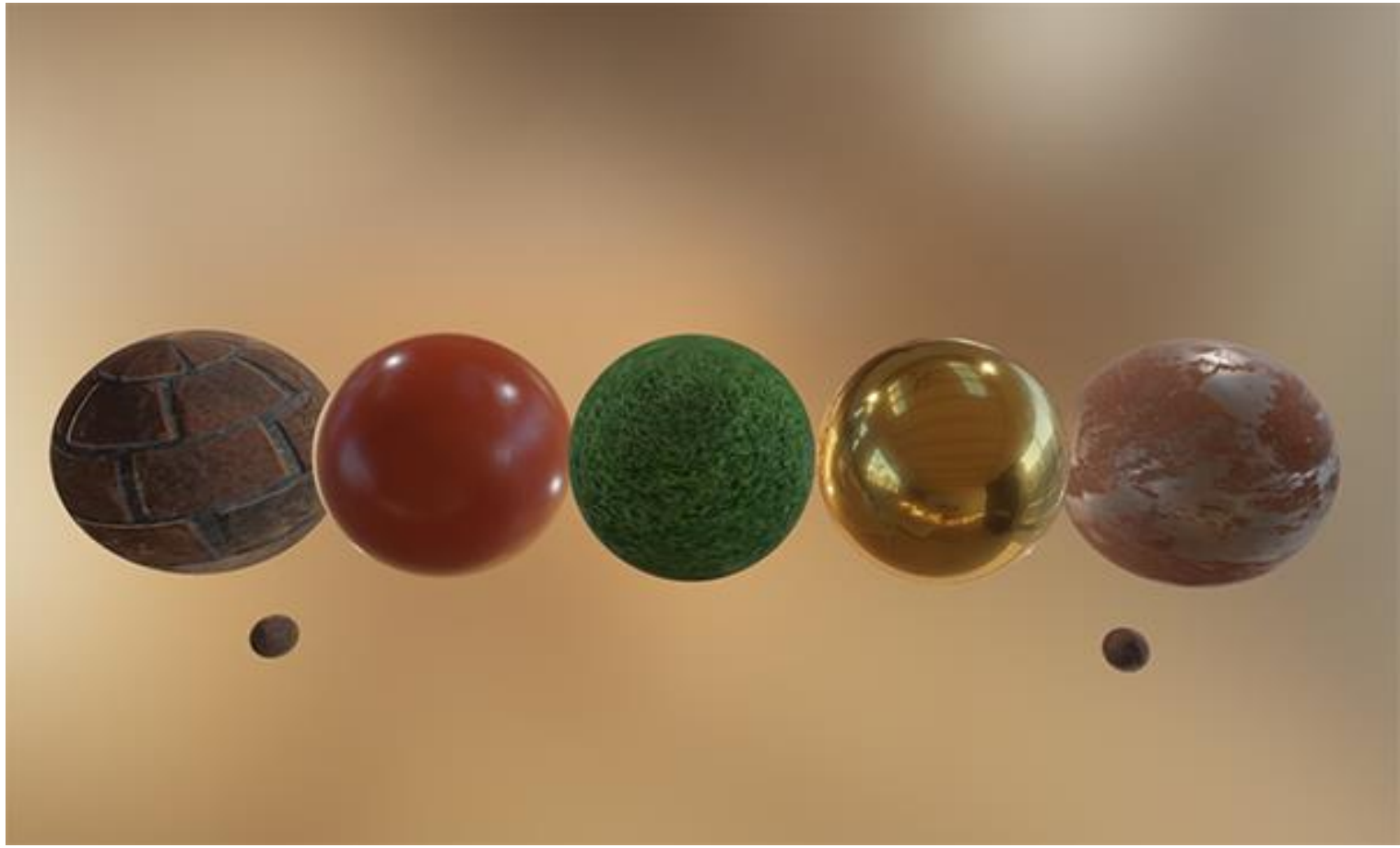
PROPIEDADES Y RENDERING

3.1 Color, Luz, sombreado, materiales y textura
3.3 Fuentes de luz
3.4 Rendering

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**.

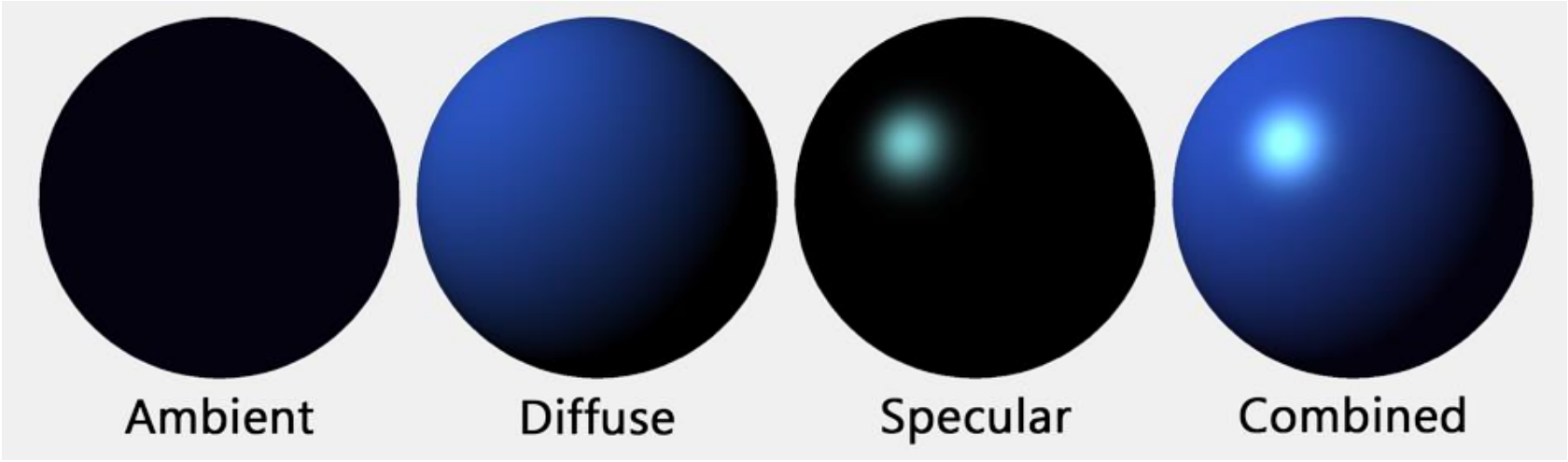


Materials

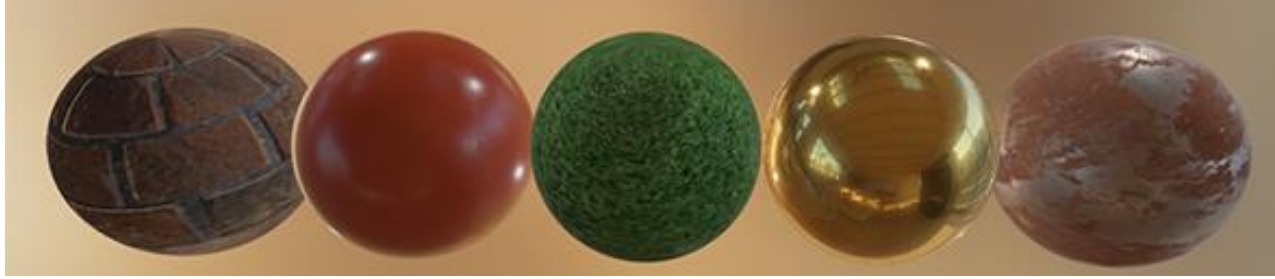
Previously, we defined an **object and light color** to define the visual output of the object, **combined with an ambient and specular intensity component**.

- 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;
```



Materials



In the **fragment shader** we create a **struct to store the material properties of the surface**.

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

uniform Material material;
```

- 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.
- 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**.

Materials

With these 4 components that define an object's material we can **simulate many real-world materials**. The present Table shows a list of material properties that simulate real materials found in the outside world.

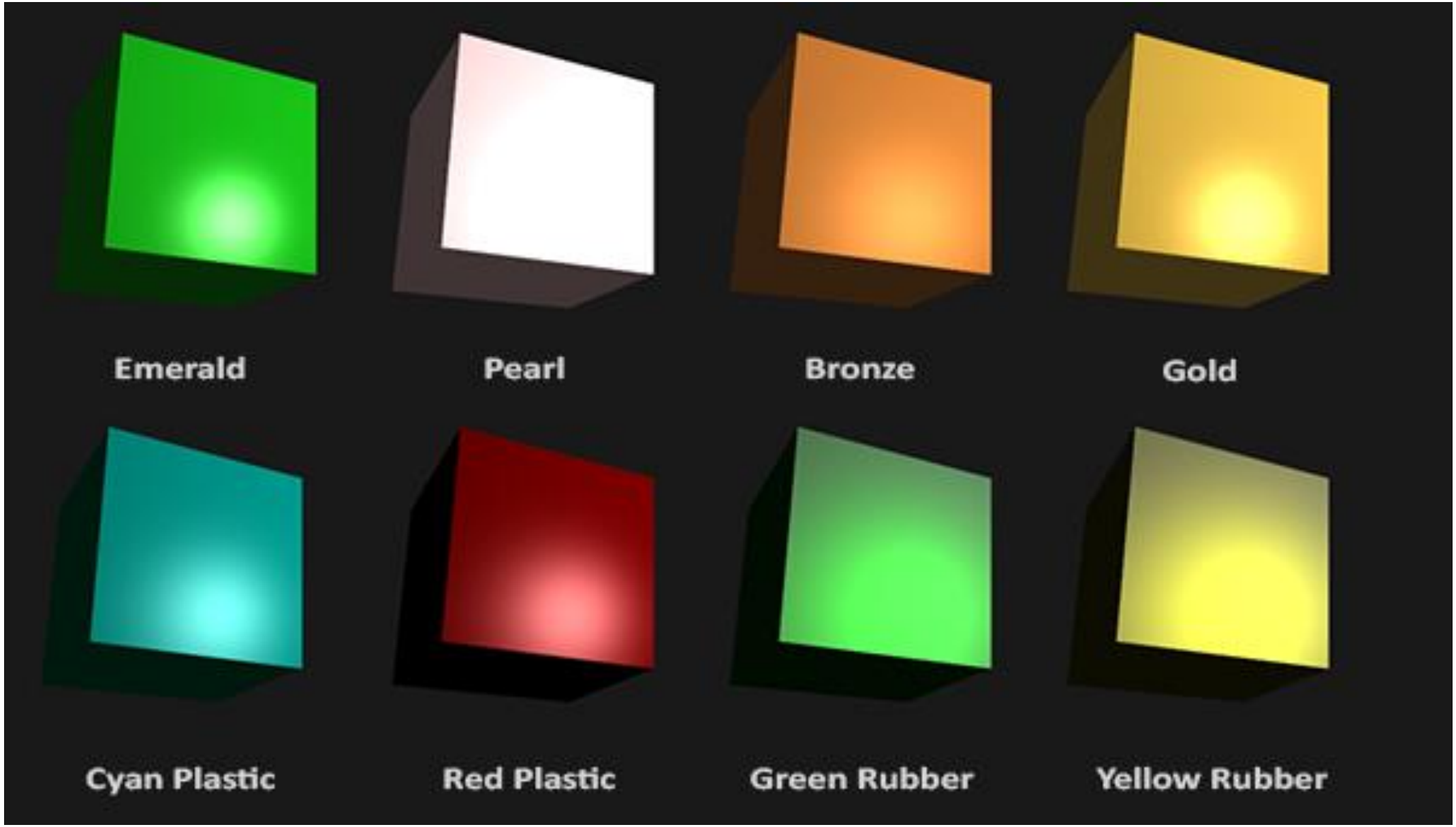
The numbers

Name	Ambient			Diffuse			Specular			Shininess
emerald	0.0215	0.1745	0.0215	0.07568	0.61424	0.07568	0.633	0.727811	0.633	0.6
jade	0.135	0.2225	0.1575	0.54	0.89	0.63	0.316228	0.316228	0.316228	0.1
obsidian	0.05375	0.05	0.06625	0.18275	0.17	0.22525	0.332741	0.328634	0.346435	0.3
pearl	0.25	0.20725	0.20725	1	0.829	0.829	0.296648	0.296648	0.296648	0.088
ruby	0.1745	0.01175	0.01175	0.61424	0.04136	0.04136	0.727811	0.626959	0.626959	0.6
turquoise	0.1	0.18725	0.1745	0.396	0.74151	0.69102	0.297254	0.30829	0.306678	0.1
brass	0.329412	0.223529	0.027451	0.780392	0.568627	0.113725	0.992157	0.941176	0.807843	0.21794872
bronze	0.2125	0.1275	0.054	0.714	0.4284	0.18144	0.393548	0.271906	0.166721	0.2
chrome	0.25	0.25	0.25	0.4	0.4	0.4	0.774597	0.774597	0.774597	0.6

Table: http://devernay.free.fr/cours/opengl/materials.html

Materials

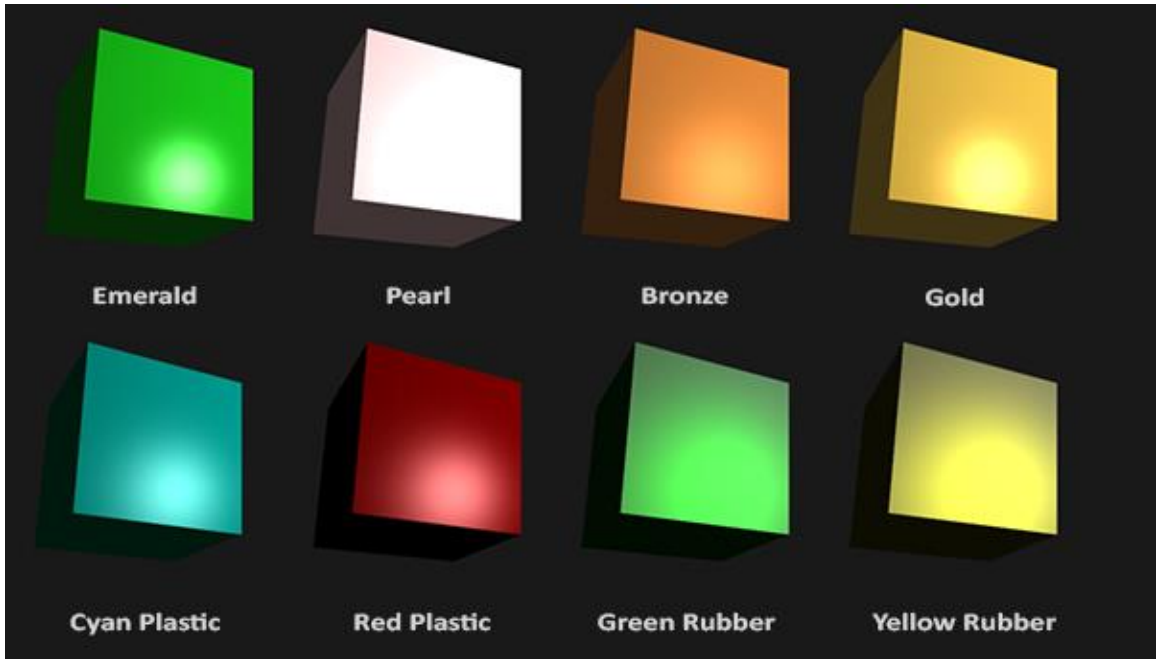
With these 4 components that define an object's material we can **simulate many real-world materials**.



Materials

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.

- 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**

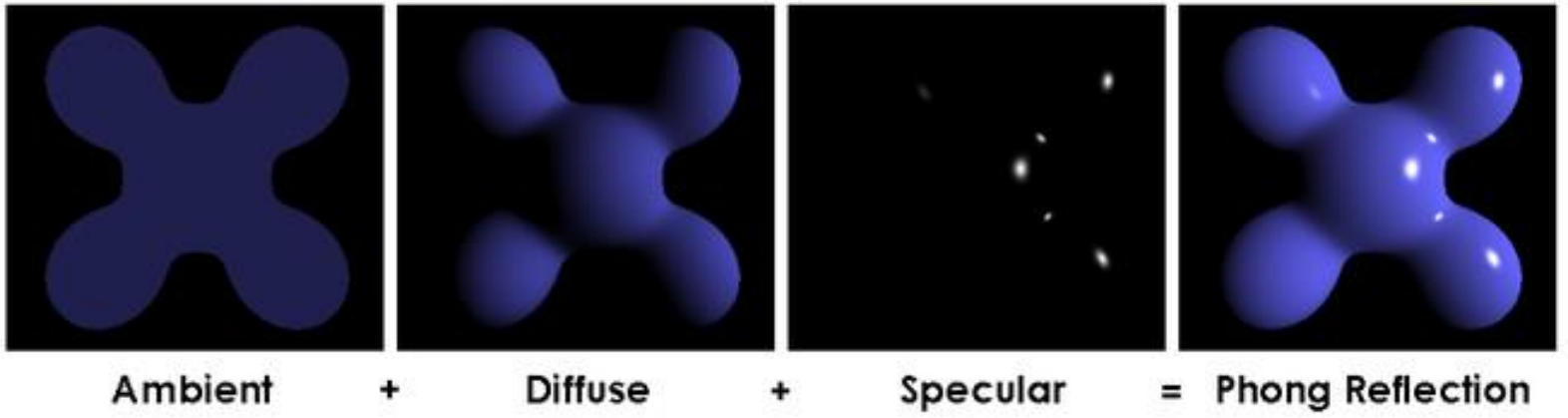


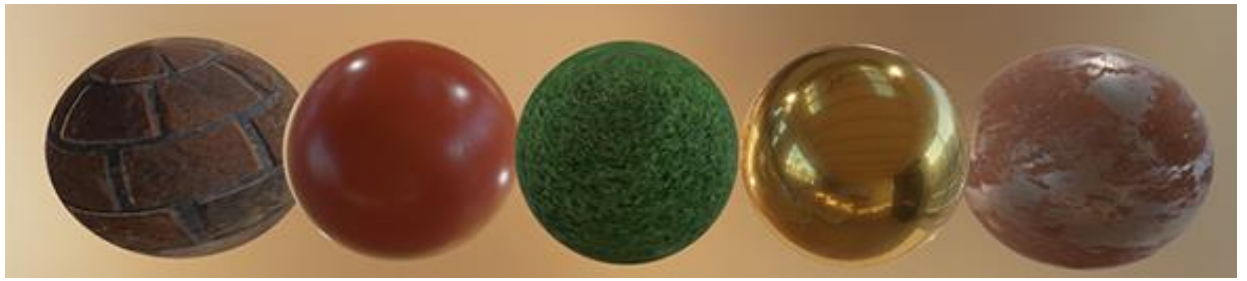
Materials – Setting materials

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);
}
```





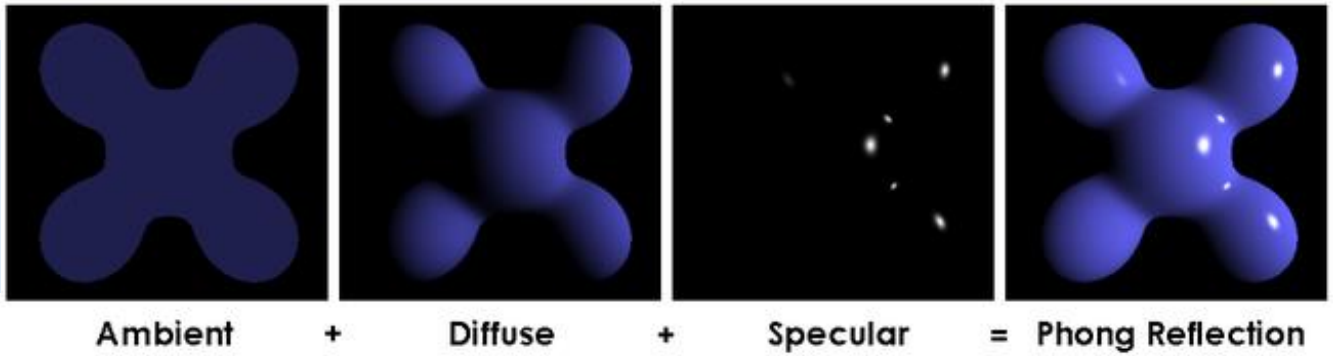
Materials – Setting materials

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:

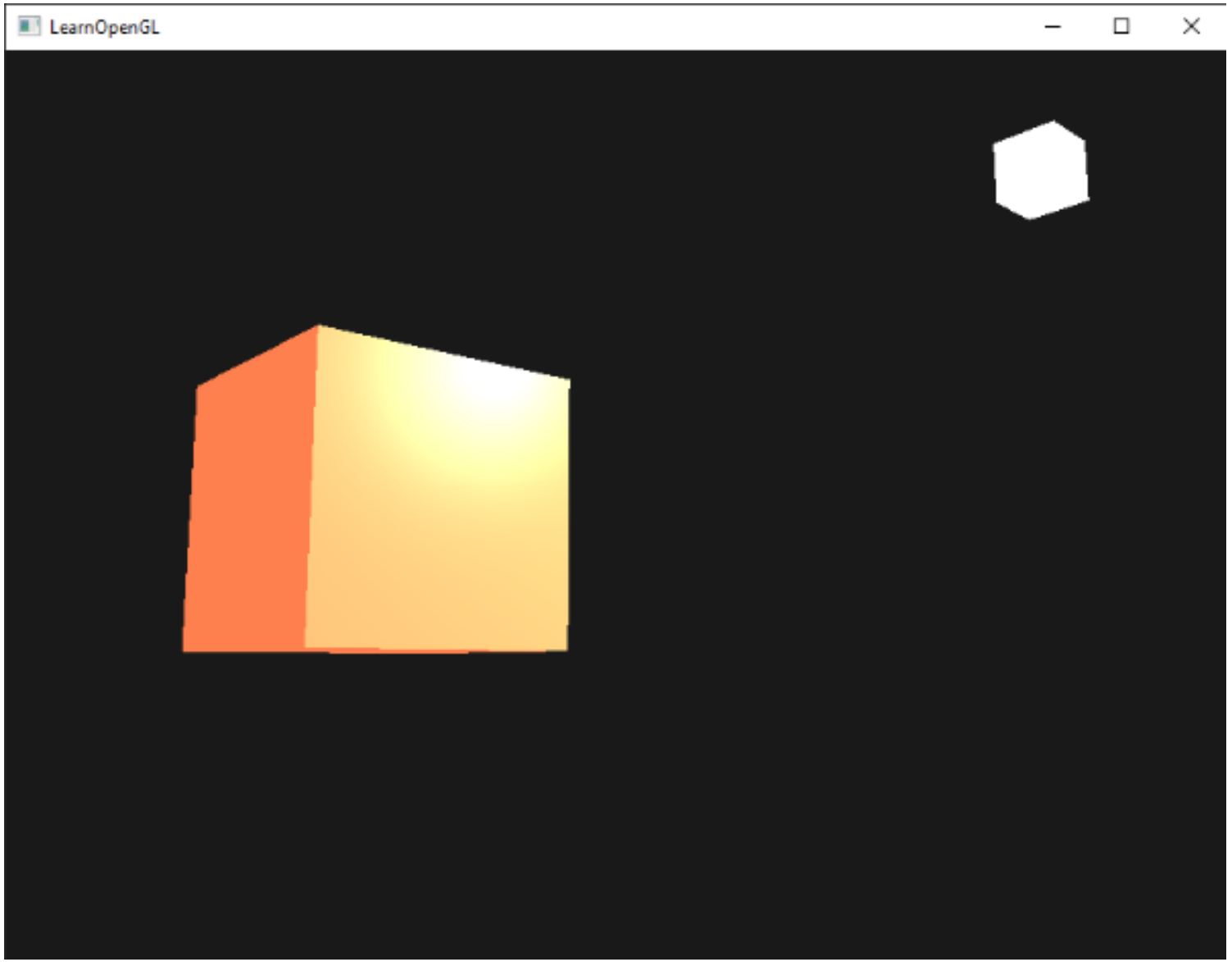
```
lightingShader.setVec3("material.ambient", 1.0f, 0.5f, 0.31f);
lightingShader.setVec3("material.diffuse", 1.0f, 0.5f, 0.31f);
lightingShader.setVec3("material.specular", 0.5f, 0.5f, 0.5f);
lightingShader.setFloat("material.shininess", 32.0f);
```



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.

Materials – Setting materials

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



The object is way too bright.

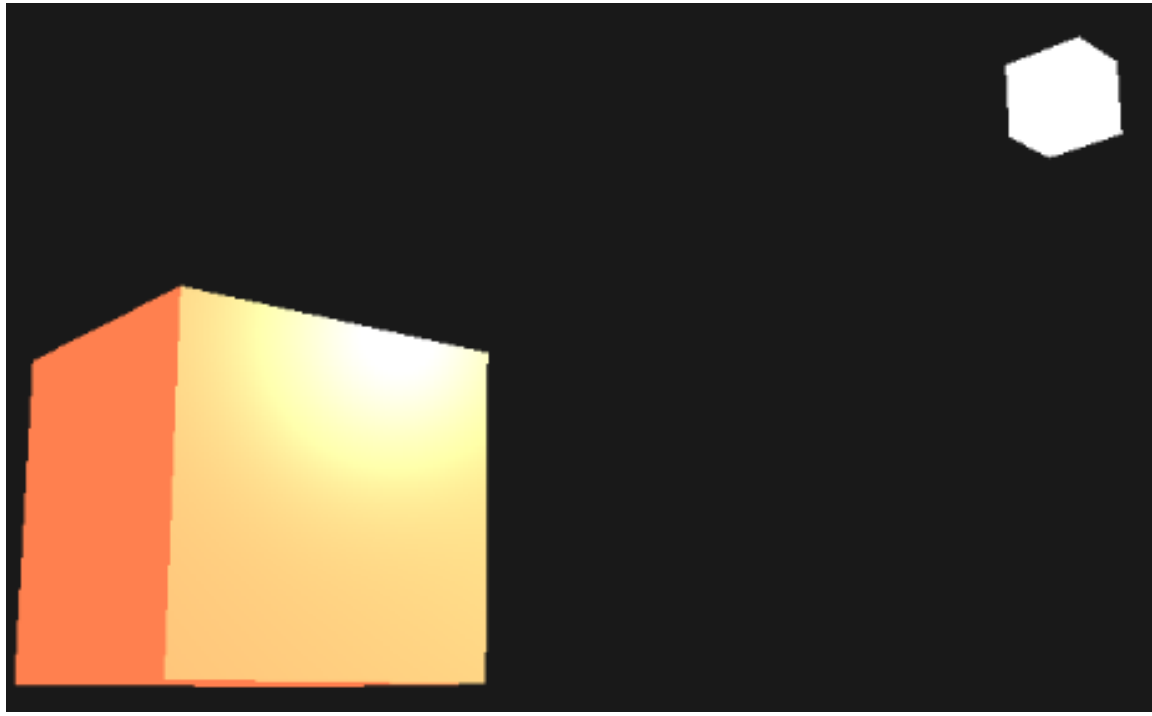
Materials – Light properties

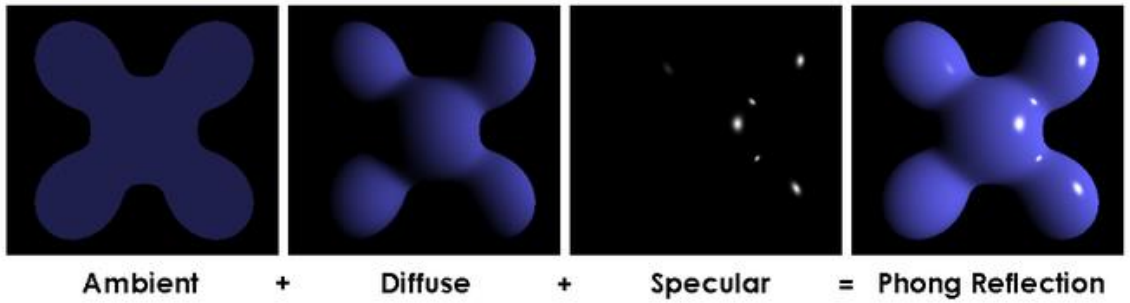
The reason for the object being too bright is that the ambient, diffuse and specular colors are **reflected with full force from any light source**.

- 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);
```



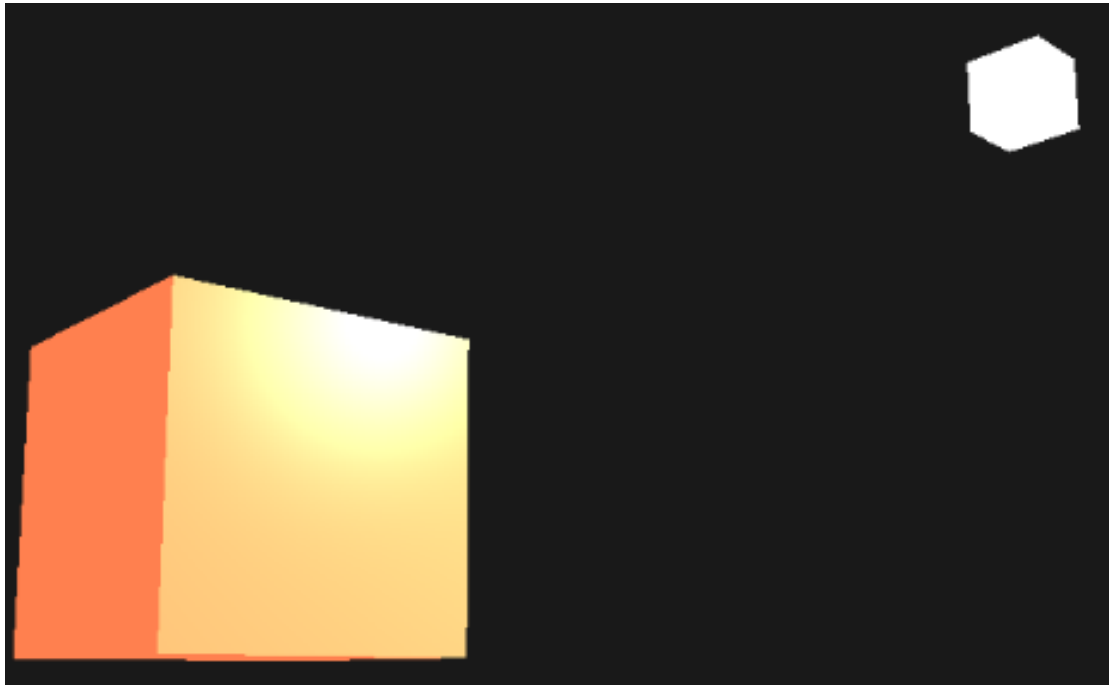


Materials – Light properties

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.

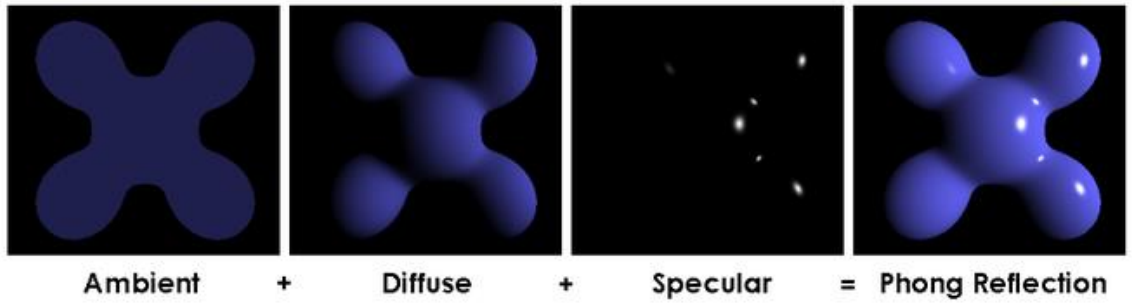
```
vec3 ambient  = vec3(1.0) * material.ambient;
vec3 diffuse  = vec3(1.0) * (diff * material.diffuse);
vec3 specular = vec3(1.0) * (spec * material.specular);
```



- 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;
```



Materials – Light properties

We can influence the diffuse and specular intensity of the light source in the same way. We'll want to create something similar to the **material struct for the light properties**:

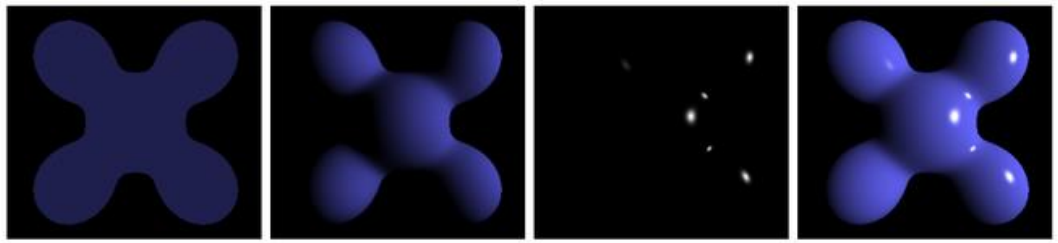
A light source has a different intensity for its ambient, diffuse and specular components.

```
struct Light {
    vec3 position;
     vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};
uniform Light light;
```

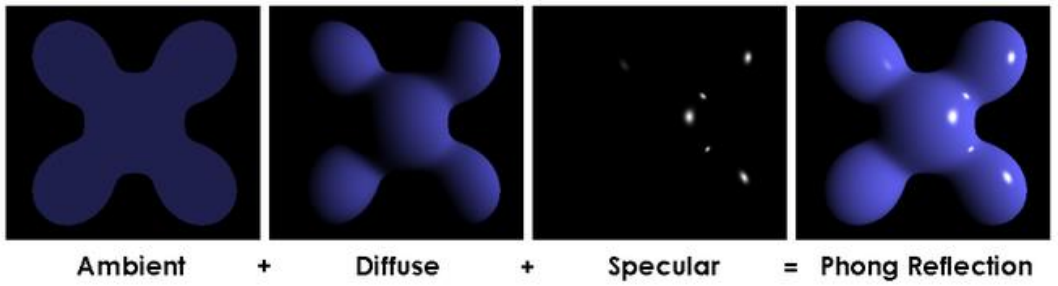
- The ambient light is usually set to a low intensity because we don't want the ambient color to be too dominant.
- The diffuse component of a light source is usually set to the exact color we'd like a light to have; often a bright white color.
- The specular component is usually kept at vec3(1.0) shining 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);
```



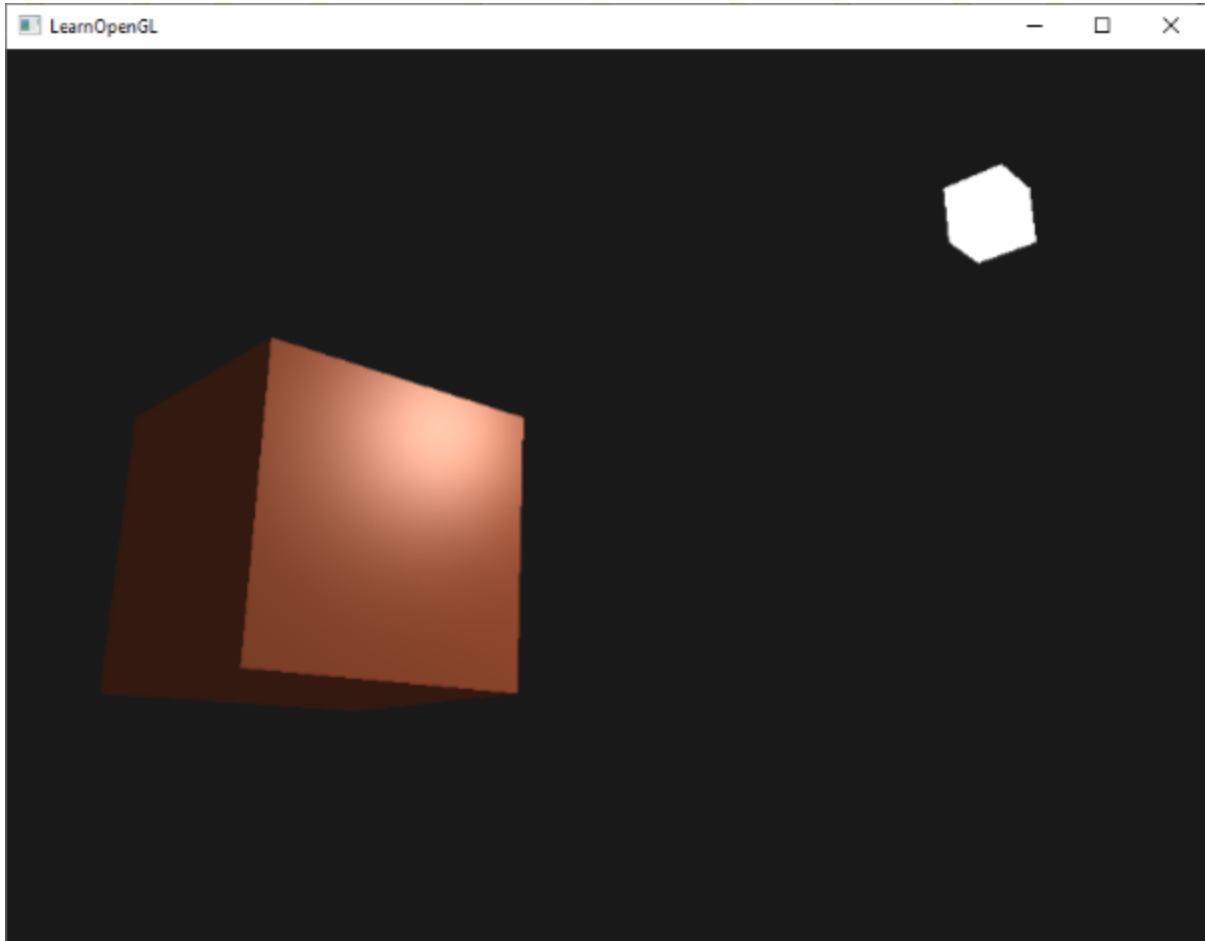
Materials – Light properties



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); // darken diffuse light a bit
lightingShader.setVec3("light.specular", 1.0f, 1.0f, 1.0f);
```

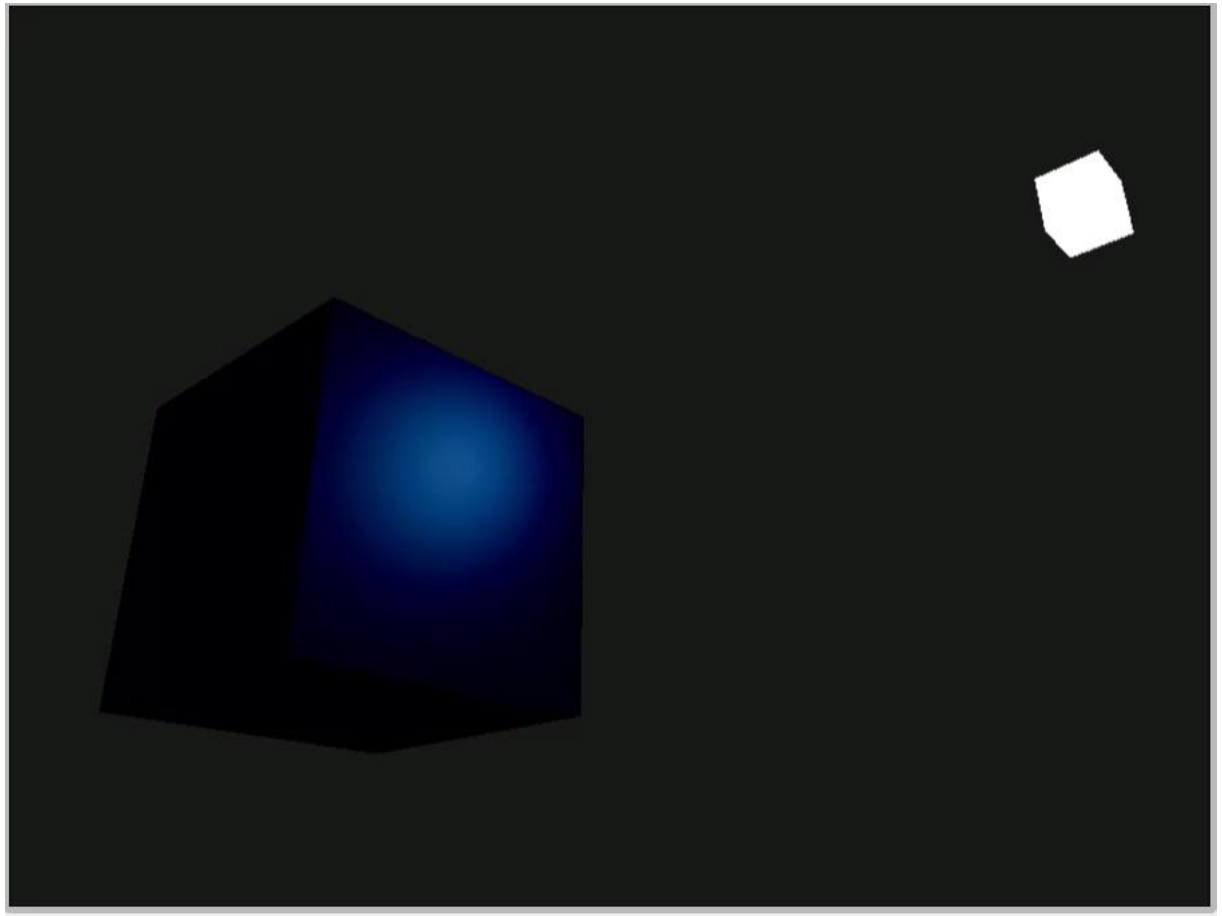
- Now that we modulated how the light influences the object's material, we get a visual output that looks much like the output from the previous chapter.
- This time however we got full control over the lighting and the material of the object:



Materials – Different light colors

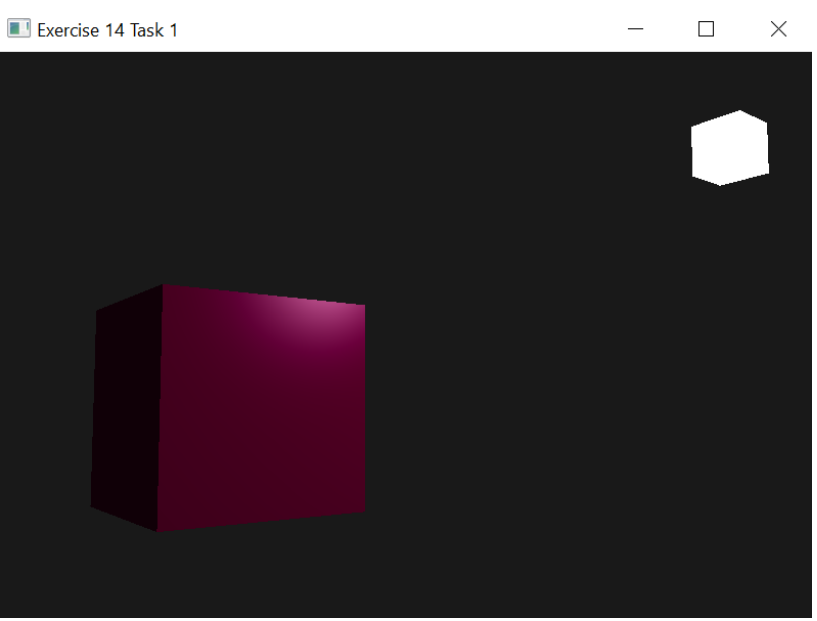
We can easily change the light's colors over time by changing the light's ambient and diffuse colors via sin and **glfwGetTime**:

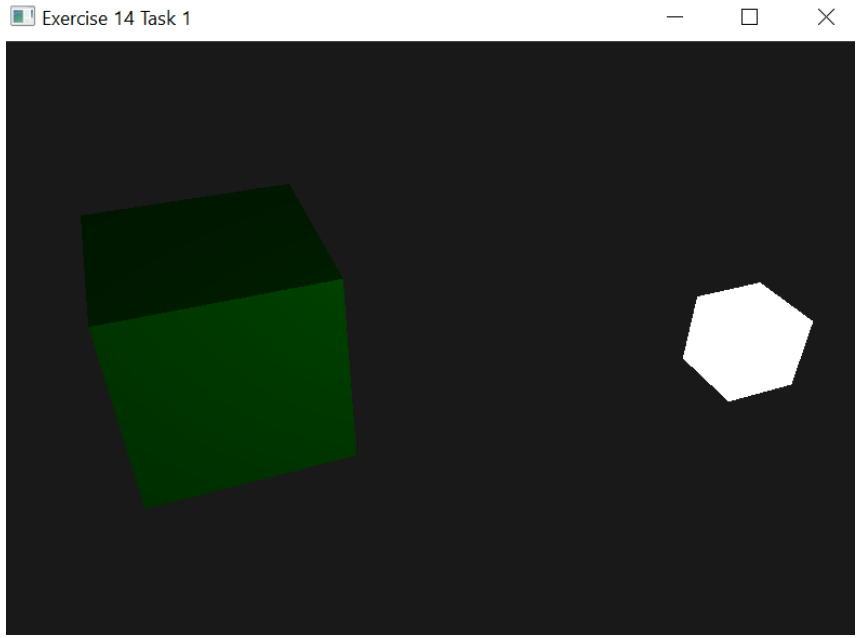
```
glm::vec3 lightColor;
lightColor.x = sin(glfwGetTime() * 2.0f);
lightColor.y = sin(glfwGetTime() * 0.7f);
lightColor.z = sin(glfwGetTime() * 1.3f);

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);
```

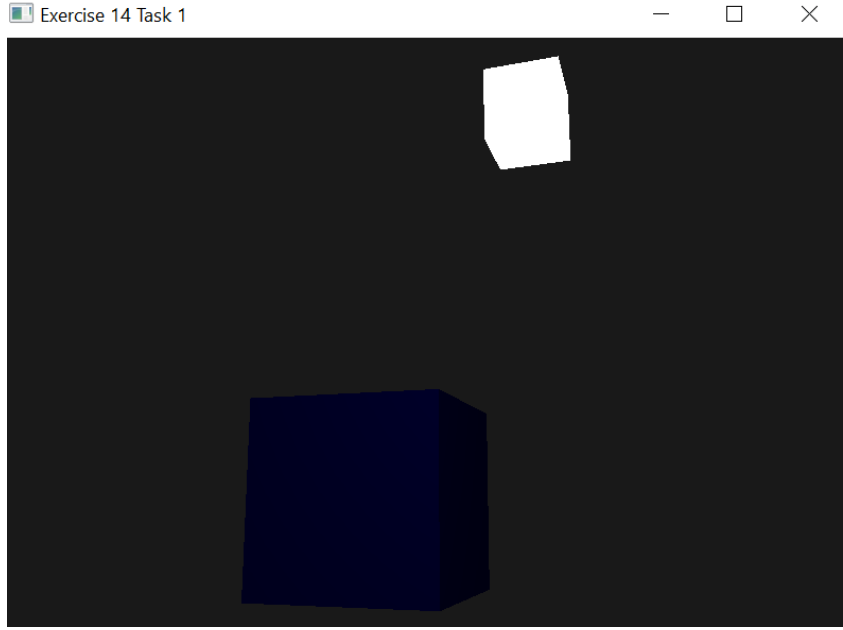


Materials

Exercise 14 Task 1: Change the light's colors over time using the materials properties.







Try and experiment with several lighting and material values and see how they affect the visual output.

Materials

Exercise 14 Task 1: Change the light's colors over time using the materials properties.

Use different materials properties: http://devernay.free.fr/cours/opengl/materials.html

For example: cyan plastic container

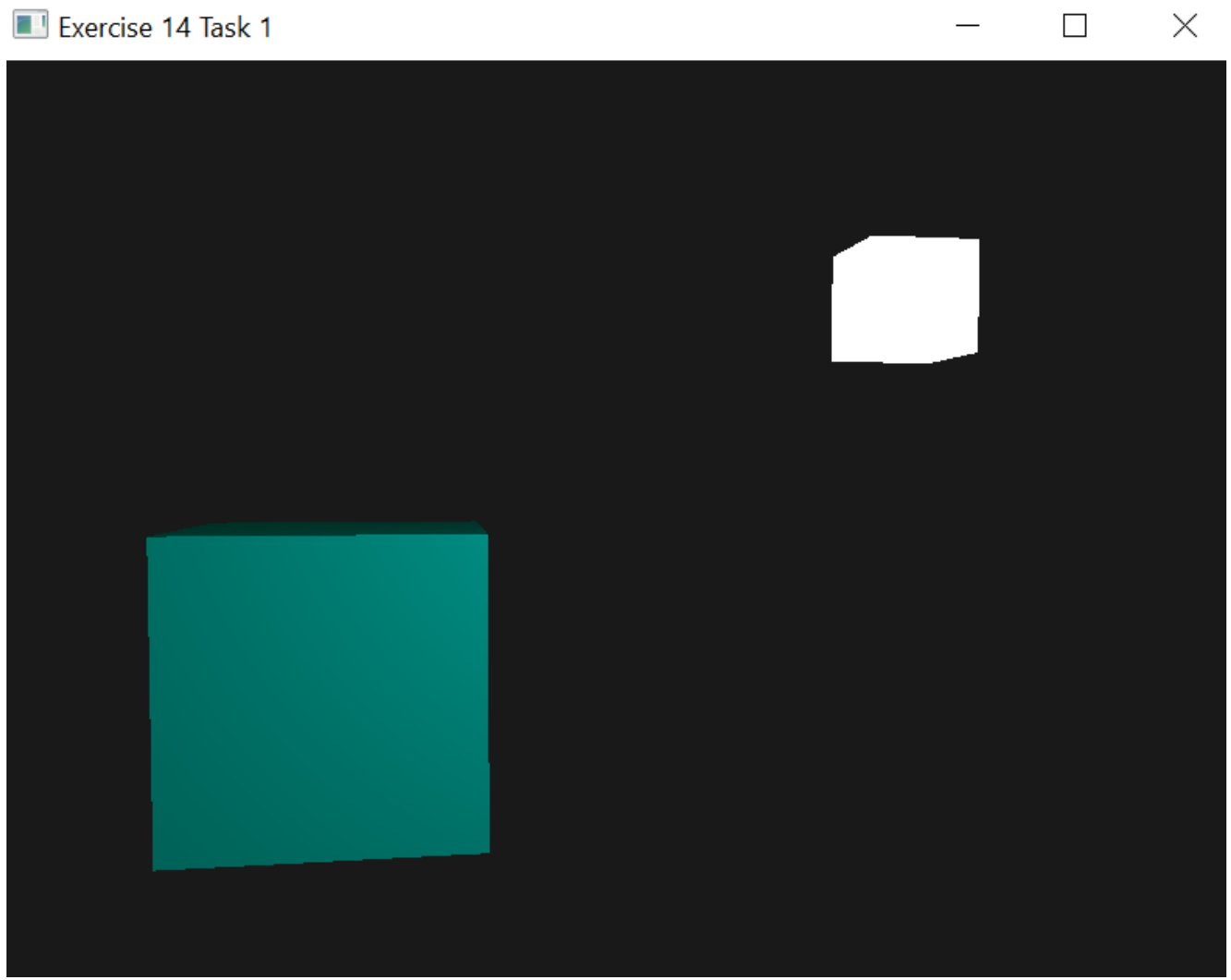
```
 // light properties
// note that all light colors are set at full intensity

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

// material properties
      lightingShader.setVec3("material.ambient", 0.0f, 0.1f, 0.06f);
      lightingShader.setVec3("material.diffuse", 0.0f, 0.50980392f, 0.50980392f);
      lightingShader.setVec3("material.specular", 0.50196078f, 0.50196078f, 0.50196078f);
      lightingShader.setFloat("material.shininess", 32.0f);
```

Materials

Exercise 14 Task 1: Change the light's colors over time using the materials properties.

Use different materials properties: http://devernay.free.fr/cours/opengl/materials.html

For example: cyan plastic container



Lighting Maps

Previously, we defined a material for an entire object as a whole. **Objects** in the real world however usually **do not consist of a single material, but of several materials**.

Think of a car:

- Its exterior consists of a shiny fabric, it has **windows that partly reflect the surrounding environment**, its tires are all but shiny so they don't have specular highlights and it has **rims that are super shiny**.
- The car also has **diffuse and ambient colors that are not the same for the entire object**; a car displays many **different ambient/diffuse colors**.
- All by all, such an object has **different material properties for each of its different parts**.

We need to extend the system by introducing **diffuse and specular maps**. These allow us **to influence the diffuse** (and indirectly the ambient component since they should be the same anyways) **and the specular component of an objec**t with much more precision.

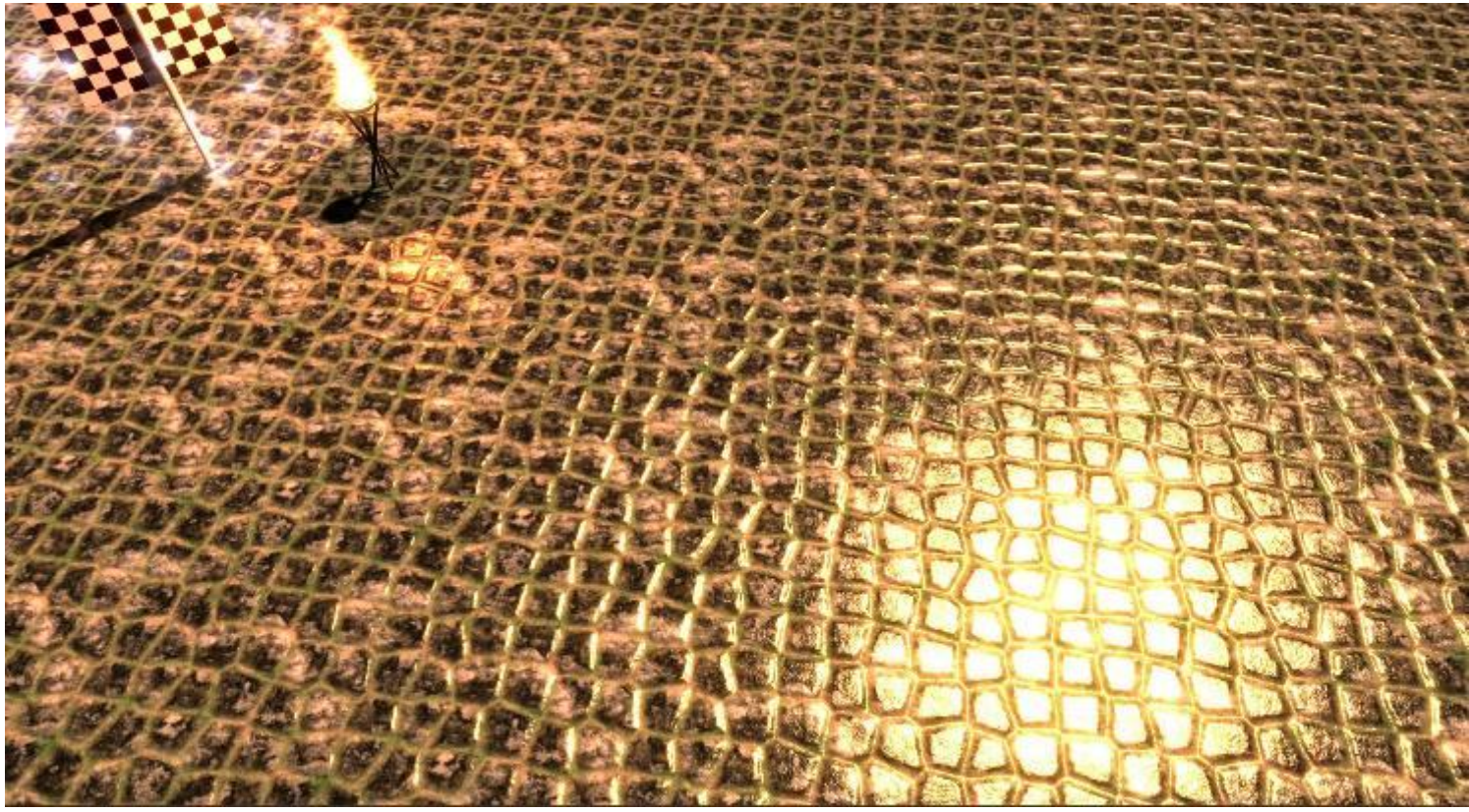
Lighting Maps – Diffuse Maps

A way to set the **diffuse colors of an object for each individual fragment**. Some sort of system where we can retrieve a color value based on the fragment's position on the object?

This should probably all sound familiar, and we've been using such a system for a while now → **Textures**

Textures →using an image wrapped around an object that we can index for unique color values per fragment.

In lit scenes this is usually called a **diffuse map** since a **texture image represents all of the object's diffuse colors**.



Using a **diffuse map** in shaders is exactly like using **textures**

Lighting Maps – Diffuse Maps

We **store the texture as a sampler2D inside the Material struct**.

- We replace the earlier defined vec3 diffuse color vector with the **diffuse map**.
- We also **remove the ambient material color vector** since **the ambient color is equal to the diffuse color** anyways now that we control ambient with the light.

So there's no need to store it separately:

```
struct Material {
    sampler2D diffuse;
    vec3      specular;
    float     shininess;
};
...
in vec2 TexCoords;
```

If you're a bit stubborn and still want to set the ambient colors to a different value (other than the diffuse value) **you can keep the ambient vec3, but then the ambient colors would still remain the same for the entire object**. To get different ambient values for each fragment you'd have to use another texture for ambient values alone.

Keep in mind that **sampler2D is a so called opaque type** which means **we can't instantiate these types, but only define them as uniforms.** If the struct would be instantiated other than as a uniform (like a function parameter) GLSL could throw strange errors; the same thus applies to any struct holding such opaque types

Lighting Maps – Diffuse Maps

Note that we are going **to need texture coordinates** again in the fragment shader, so we add an **extra input variable**. Then we simply sample from the texture to retrieve the fragment's diffuse color value:

```
vec3 diffuse = light.diffuse * diff * vec3(texture(material.diffuse, TexCoords));
```

Also, don't forget to set the ambient material's color equal to the diffuse material's color as well:

```
vec3 ambient = light.ambient * vec3(texture(material.diffuse, TexCoords));
```

And that's all it takes to use a diffuse map. As you can see it is nothing new, but it does provide a dramatic increase in visual quality.

To get it working we do need to update the vertex data with texture coordinates, transfer them as vertex attributes to the fragment shader, load the texture, and bind the texture to the appropriate texture unit.

Lighting Maps – Diffuse Maps

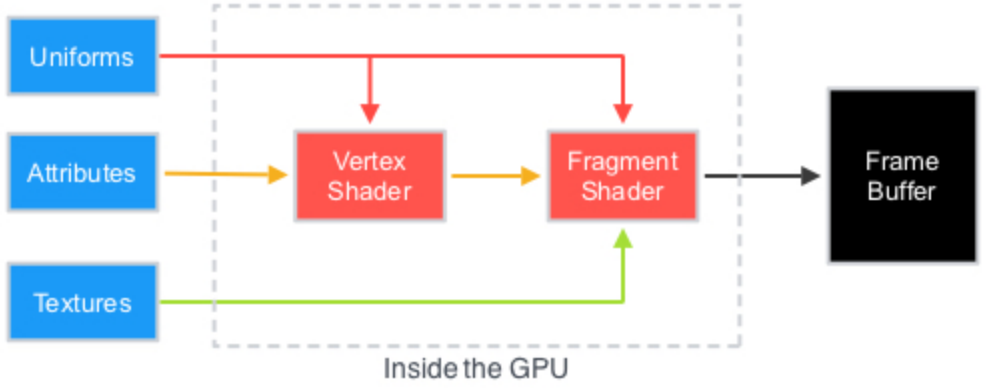
The vertex data should be updated:

```
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,
...
```

The vertex data now includes **vertex positions**, **normal vectors**, **and texture coordinates** for each of the cube's vertices.

Let's **update the vertex shader** to accept texture coordinates as a vertex attribute and forward them to the fragment shader:

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoords;
...
out vec2 TexCoords;

void main()
{
    ...
    TexCoords = aTexCoords;
}
```



Be sure to update the vertex attribute pointers of both VAOs to match the new vertex data and load the container image as a texture.

Lighting Maps – Diffuse Maps

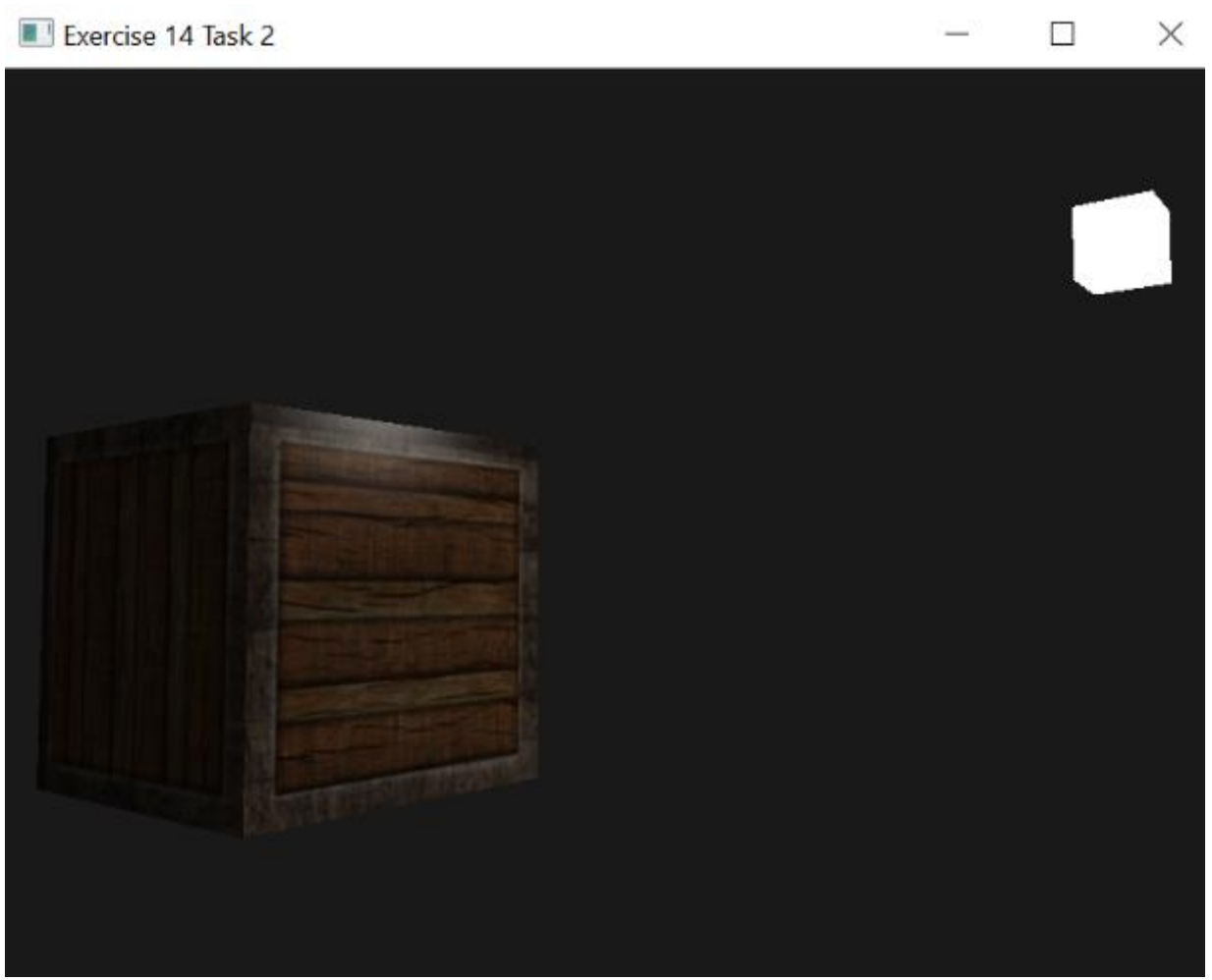
Before rendering the cube, we want to **assign the right texture unit** to the **material.diffuse** uniform sampler and bind the container texture to this texture unit:

```
lightingShader.setInt("material.diffuse", 0);
...
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, diffuseMap);
```

Now using a diffuse map, we get an enormous boost in detail again and this time the container really starts to shine (quite literally).

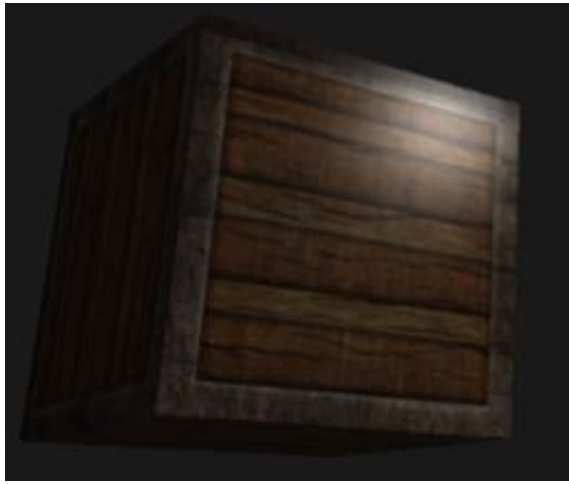
Lighting Maps – Diffuse Maps

Exercise 14 Task 2: Implement a container using a diffuse map. Use “container2.png” as a texture file.



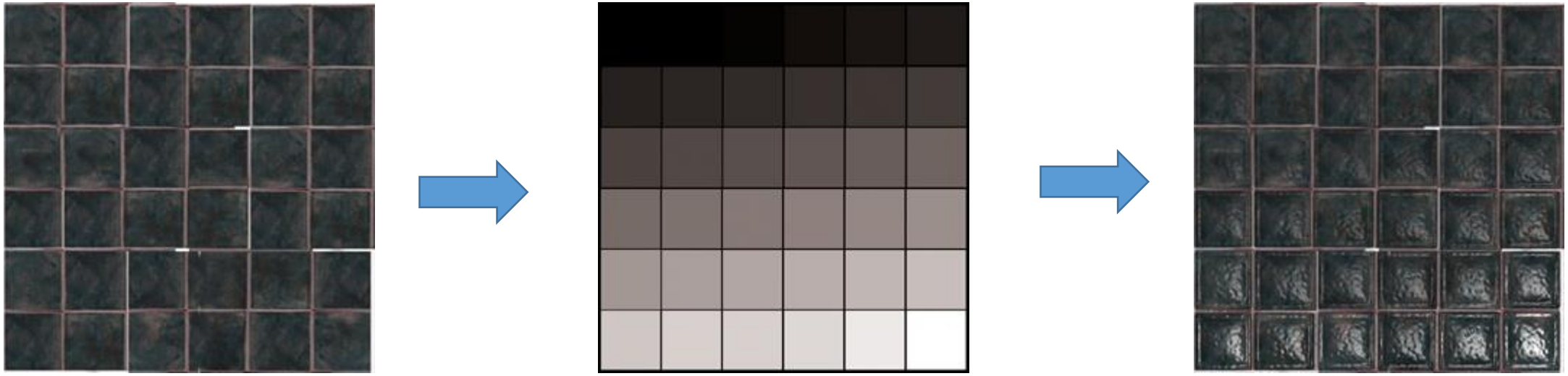
Lighting Maps – Specular Maps

You probably noticed that the **specular highlight looks a bit odd** since the object is a container that **mostly consists of wood and** wood doesn't have specular highlights like that.



- We can fix this by setting the **specular material of the object to vec3(0.0)** but that would mean that the **steel borders** of the container would **stop showing specular highlights** as well and steel should show specular highlights.

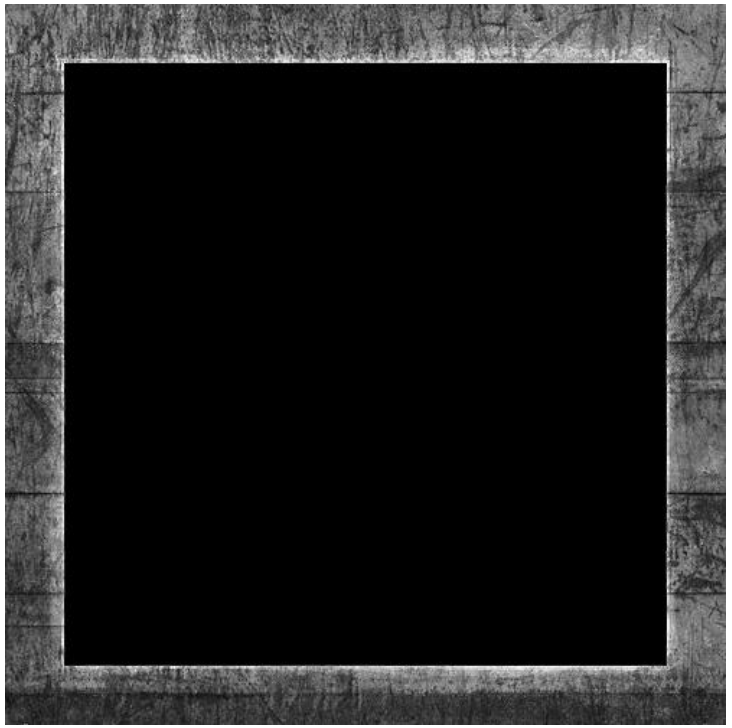
We would like to **control** what parts of the object should show a **specular highlight with varying intensity**.



Lighting Maps – Specular Maps

Specular Map → **use a texture map just for specular highlights**. This means we need to **generate a black and white** (or colors if you feel like it) **texture** that defines the **specular intensities** of each part of the object..

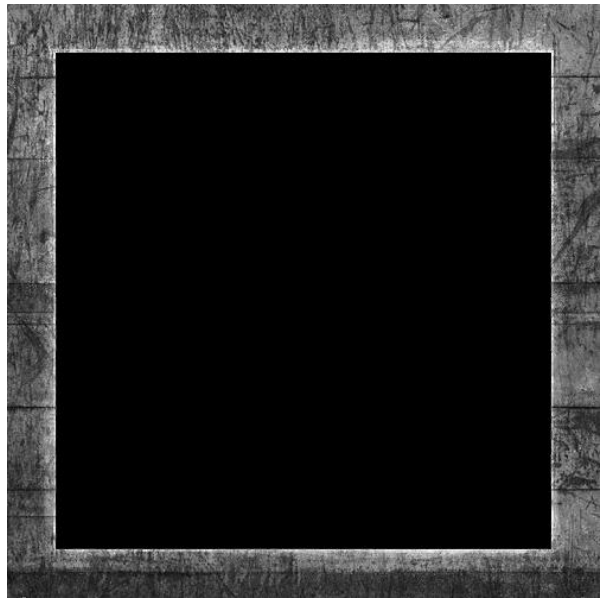
- The **intensity** of the specular highlight comes from the **brightness** **of each pixel in the image**.
- Each pixel of the specular map can be displayed as a color vector where **black** represents the color vector **vec3(0.0)** and **gray** the color vector **vec3(0.5)** for example.
- In the **fragment shader** we then sample the corresponding color value and **multiply this value with the light's specular intensity**.
- The more '**white**' a pixel is, the **higher** the result of the **multiplication** and thus the **brighter** the specular component of an object becomes.



Lighting Maps – Specular Maps

Because the container mostly consists of **wood**, and wood as a material should **have no specular highlights**, the entire wooden section of the diffuse texture was converted to black: **black sections do not have any specular highlight**.

- The **steel border** of the container has **varying specular intensities** with the steel itself being relatively susceptible to specular highlights while the cracks are not.

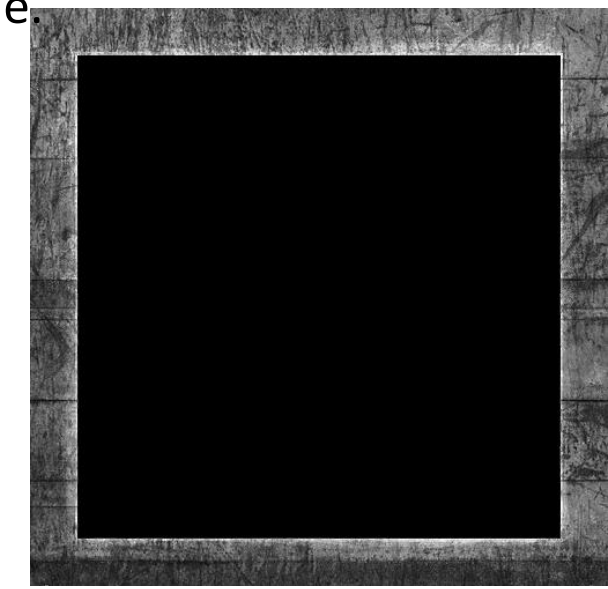


Technically **wood also has specular highlights** although with a **much lower shininess value** (more light scattering) and less impact, but for learning purposes we can just pretend wood doesn't have any reaction to specular light.

Using tools like **Photoshop or Gimp** it is relatively easy to **transform a diffuse texture to a specular image** like this by cutting out some parts, **transforming it to black and white** and **increasing the brightness/contrast**.

Lighting Maps – Sampling Specular Maps

A **specular map** is just **like any other texture** so the **code is similar** to the diffuse map code.



- Make sure to properly load the image and generate a texture object.
- Since we're using another **texture sampler** in the same fragment shader, we have to use **a different texture unit for the specular map** so let's bind it to the appropriate texture unit before rendering

```
lightingShader.setInt("material.specular", 1);
...
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, specularMap);
```

Then update the material properties of the fragment shader to accept a sampler2D as its specular component instead of a vec3:

```
struct Material {
    sampler2D diffuse;
    sampler2D specular;
    float     shininess;
};
```

Lighting Maps – Sampling Specular Maps

And lastly we want to sample the specular map to retrieve the fragment's corresponding specular intensity:

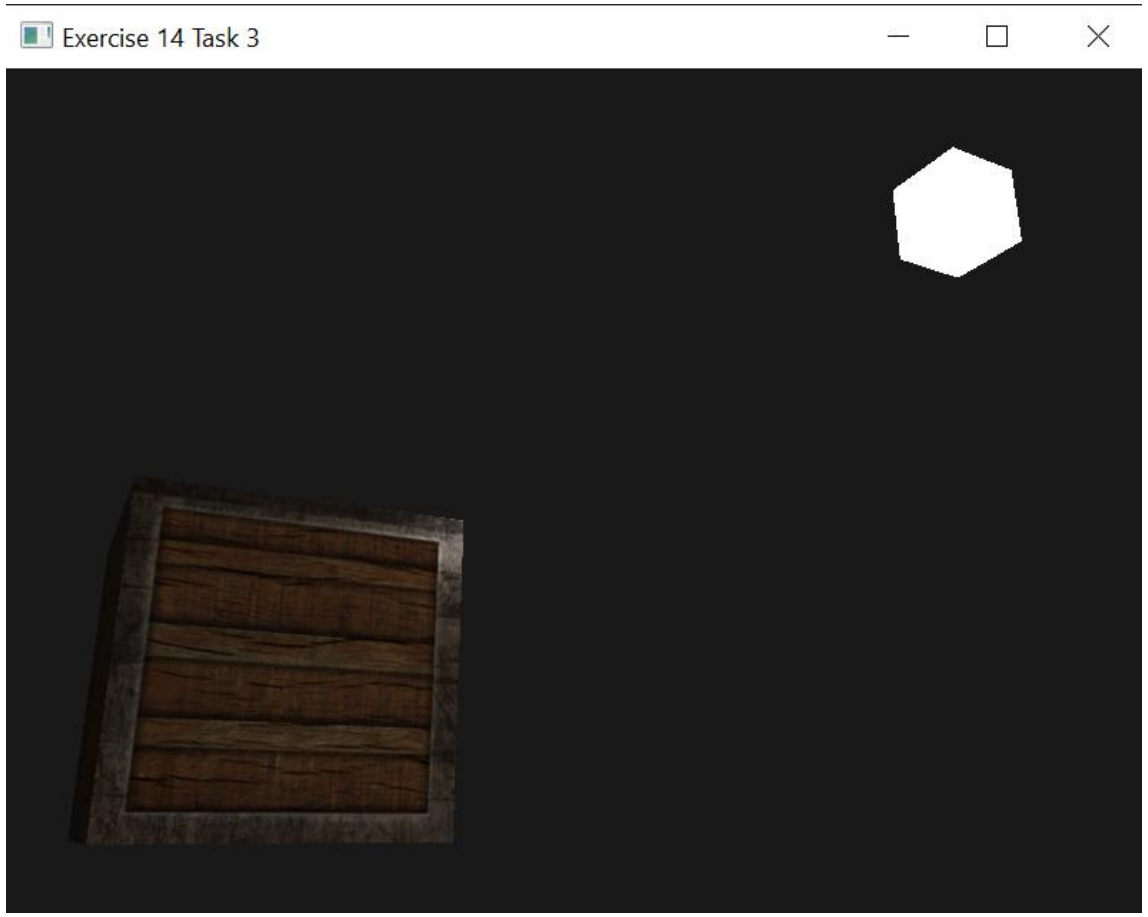
```
vec3 ambient  = light.ambient  * vec3(texture(material.diffuse, TexCoords));
vec3 diffuse  = light.diffuse  * diff * vec3(texture(material.diffuse, TexCoords));
vec3 specular = light.specular * spec * vec3(texture(material.specular, TexCoords));
FragColor = vec4(ambient + diffuse + specular, 1.0);
```

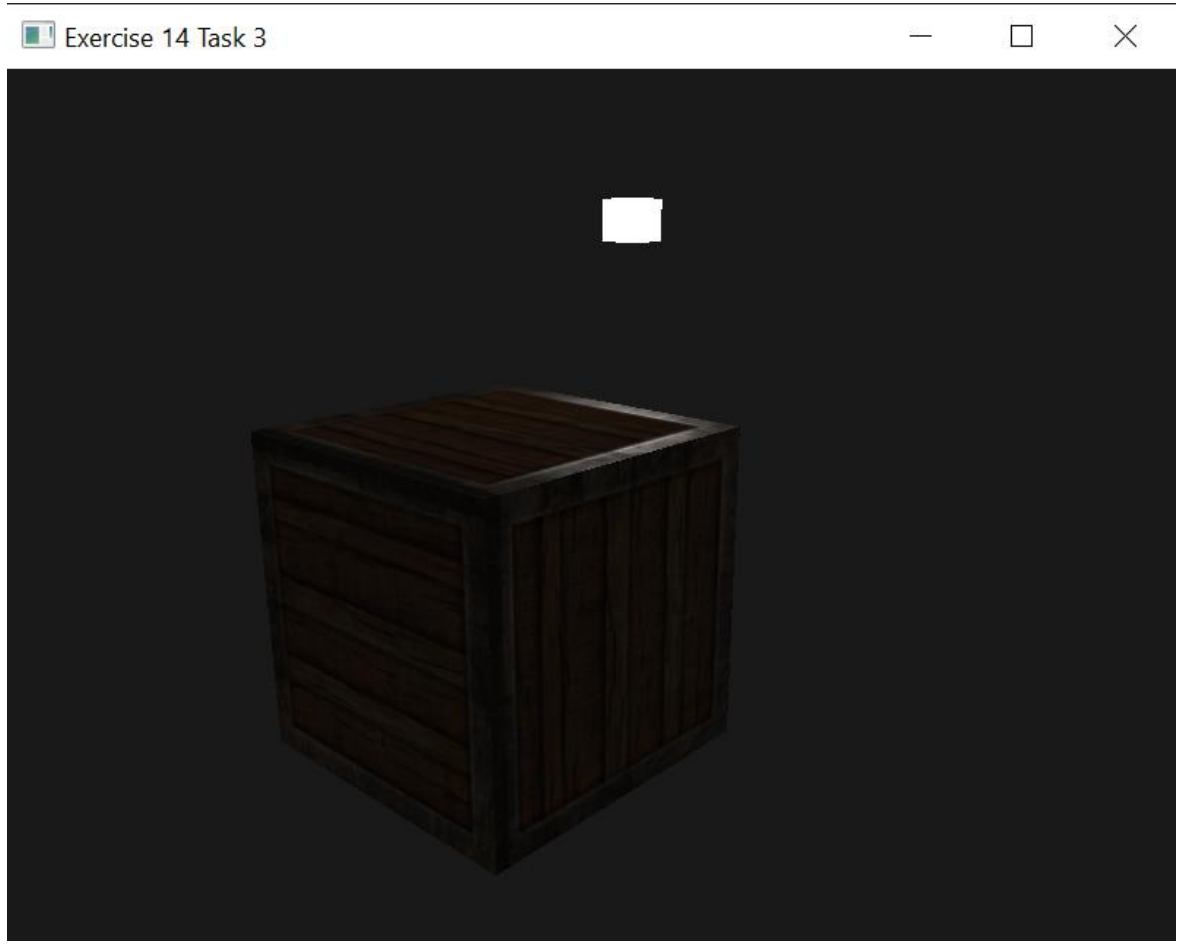
By using a specular map we can **specify with enormous detail what parts of an object have shiny properties** and we can even **control the corresponding intensity**. Specular maps give us an added layer of control over lighting on top of the diffuse map.

If you don't want to be too mainstream you could also use actual colors in the specular map to not only set the specular intensity of each fragment, but also the **color of the specular highlight**. Realistically however, the **color of the specular highlight is mostly determined by the light source** itself so it wouldn't generate realistic visuals (that's why the images are usually black and white: we only care about the intensity).

Lighting Maps – Sampling Specular Maps

Exercise 14 Task 3: Improve the light's colors using the materials properties and specular maps.





Lighting Maps – [Optional] - Emission map

Emission map is a texture that stores **emission values per fragment**.

- Emission values are colors that an object may emit as if it **contains a light source itself**
- This way an object can **glow regardless of the light conditions**.
- Emission maps are often what you see when **objects in a game glow**.

Lighting Maps – Emission map

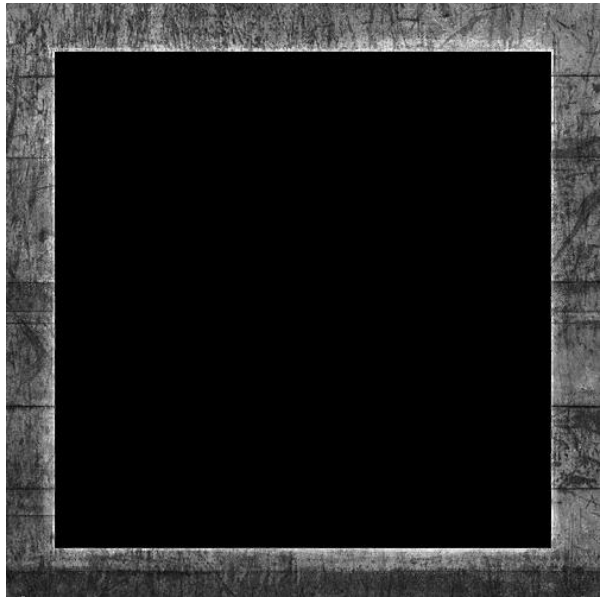
Emission map is a texture that stores **emission values per fragment**.

- Emission values are colors that an object may emit as if it **contains a light source itself**
- This way an object can **glow regardless of the light conditions**.
- Emission maps are often what you see when **objects in a game glow**.

Lighting Maps – Emission map

Exercise 14 Task 4: Improve the light's colors using the materials properties, specular and emission maps.

Specular Map



Emission Map

Lighting Maps – Emission map

Exercise 14 Task 4: Improve the light's colors using the materials properties, specular and emission maps.