CAPITULO 3

PROPIEDADES Y RENDERING

3.1 Color, Luz, sombreado, materiales y textura
3.4 Rendering

Prop. & Rend. - Colors

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);
```

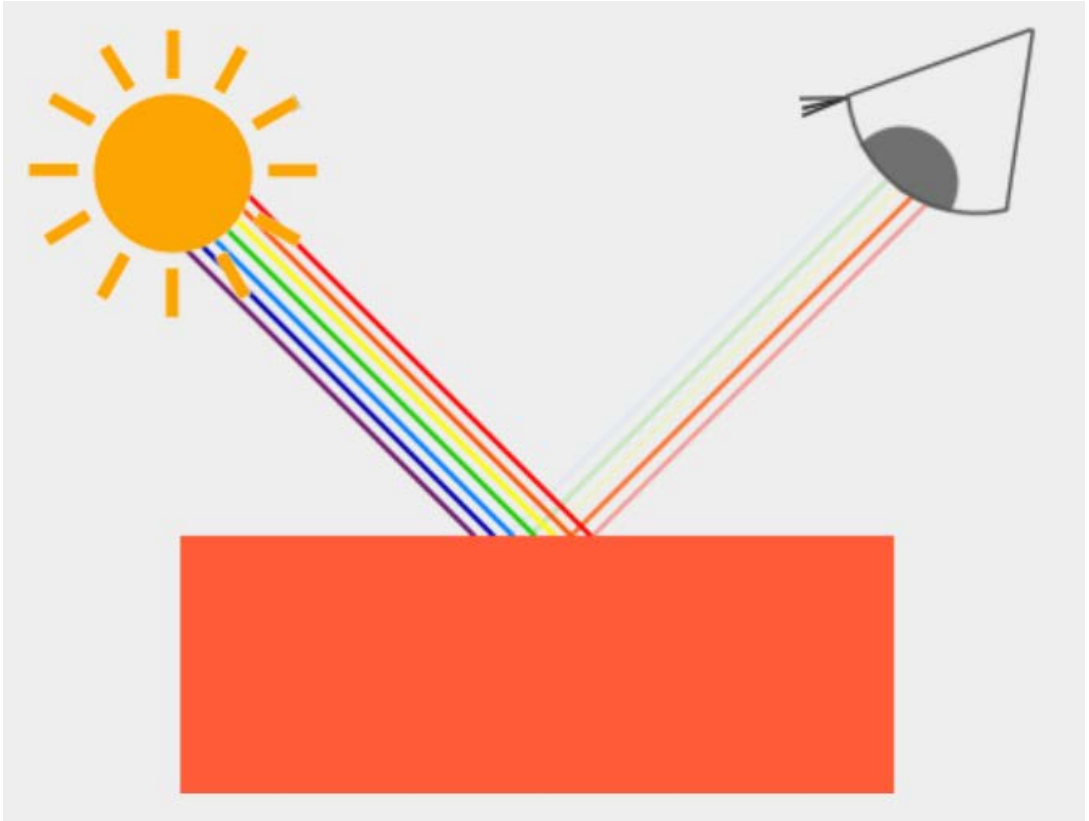
Prop. & Rend. - Colors

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:



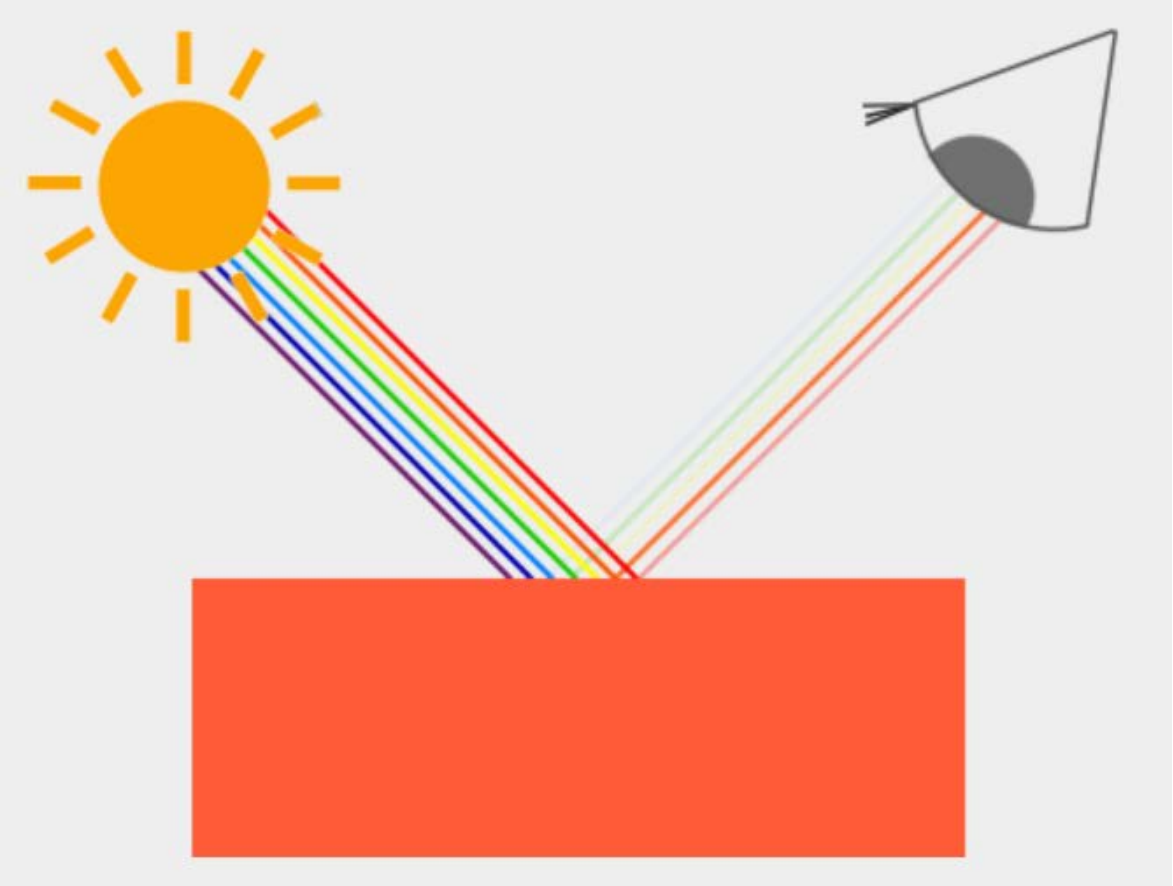
You can see that the white **sunlight is a collection of all the visible colors** and the **object absorbs a large portion of those colors**. It only reflects those colors that represent the **object's color and the combination of those is what we perceive** (in this case a coral color).

Prop. & Rend. - Colors

- 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**).

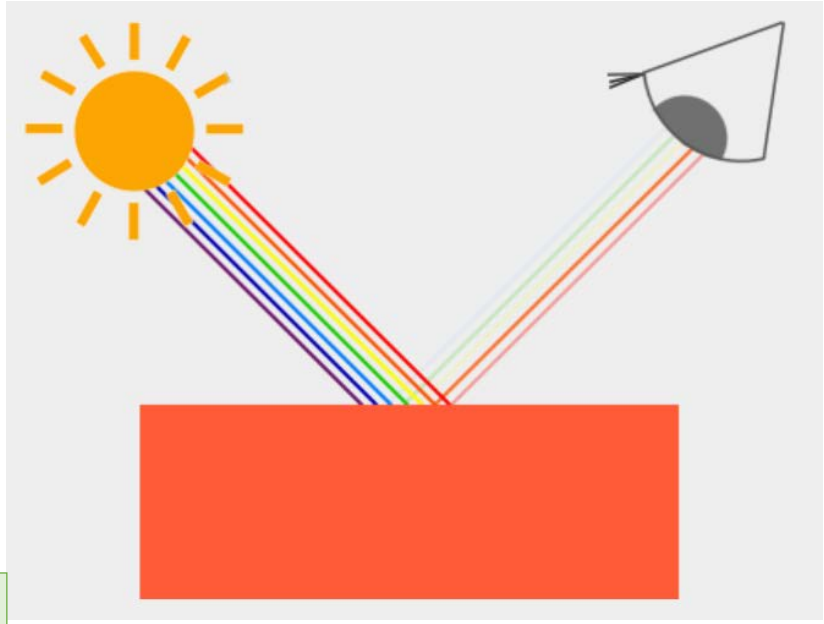
- We get the resulting color vector by doing a component-wise multiplication between the light and object color vectors:

```
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 thus define an **object's color** as the amount of each color component it reflects from a light source.

Prop. & Rend. - Colors



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);
```

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**.

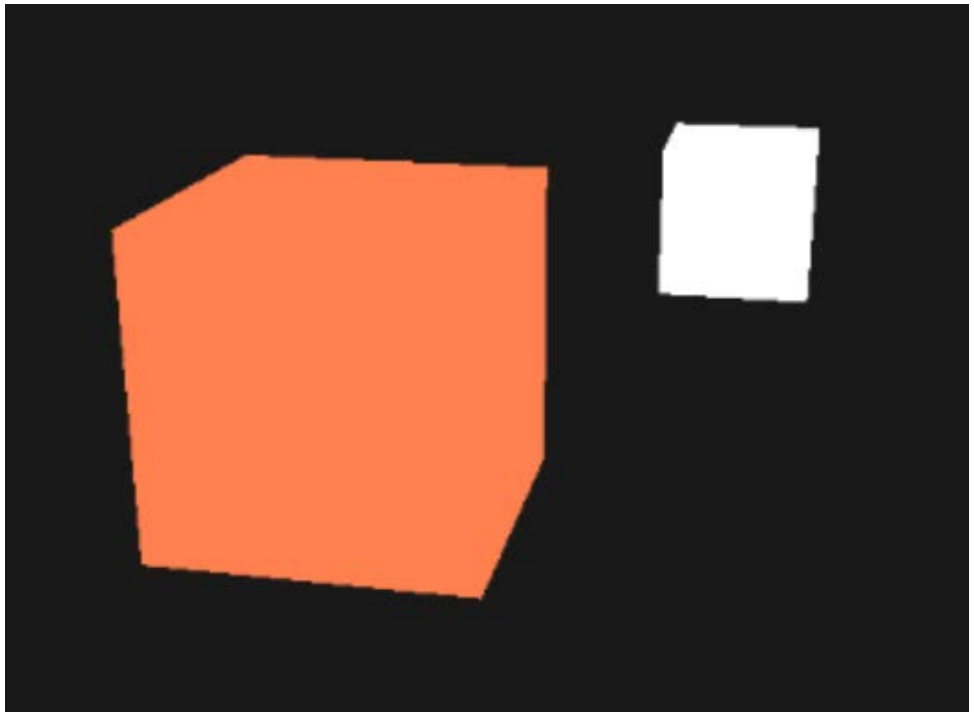
Let's try one more example with a **dark olive-green** light:

```
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);
```

Prop. & Rend. – Colors - A lighting scene

The idea is to **display light sources** 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**.

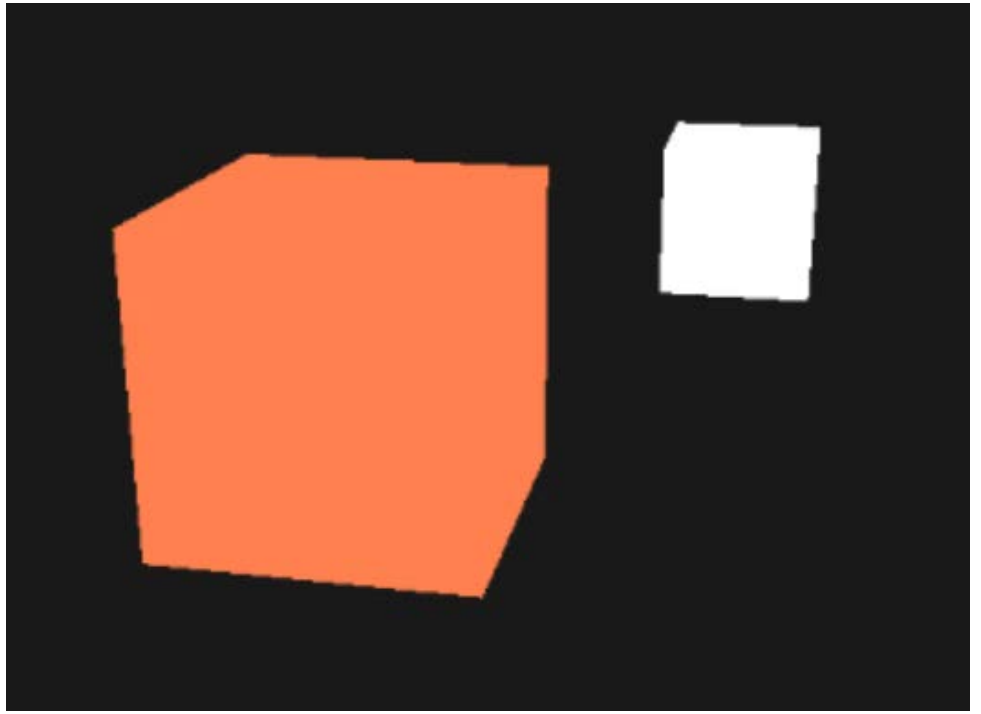


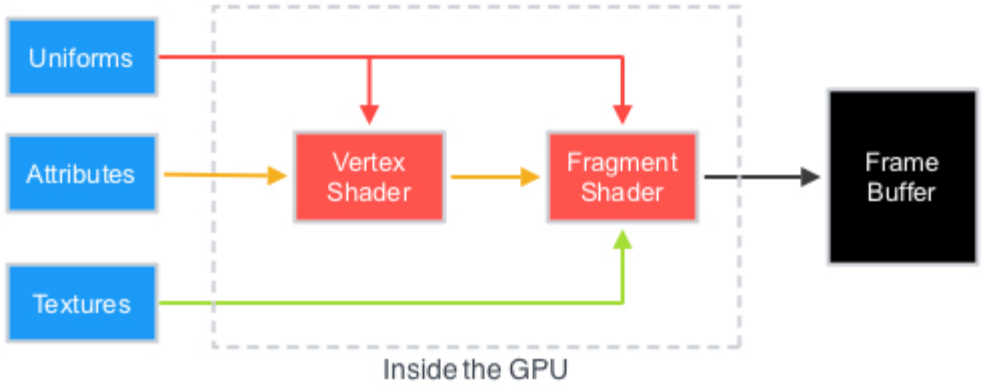
Prop. & Rend. – Colors - A lighting scene

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.

```
#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);
}
```

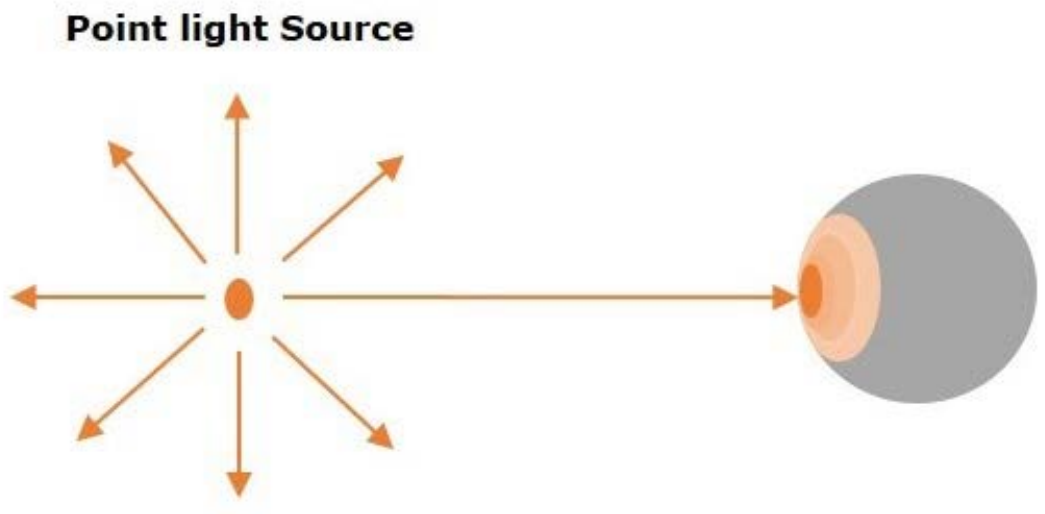




Make sure to **update the vertex data and attribute pointers** to match the new vertex shader

Prop. & Rend. – Colors - A lighting scene

Because we're also going to render a light source cube, we want to generate a **new VAO specifically for the light source**.

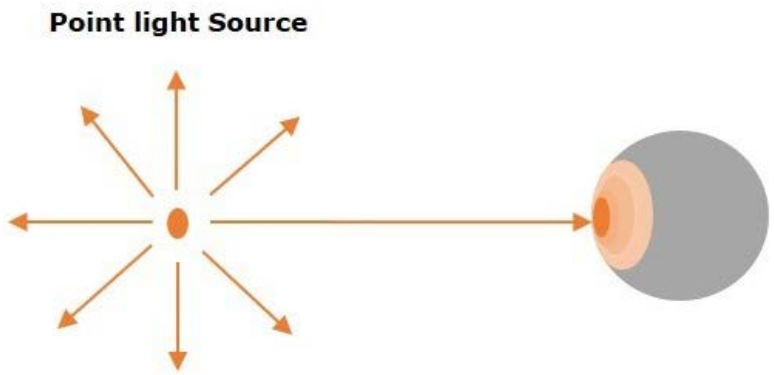


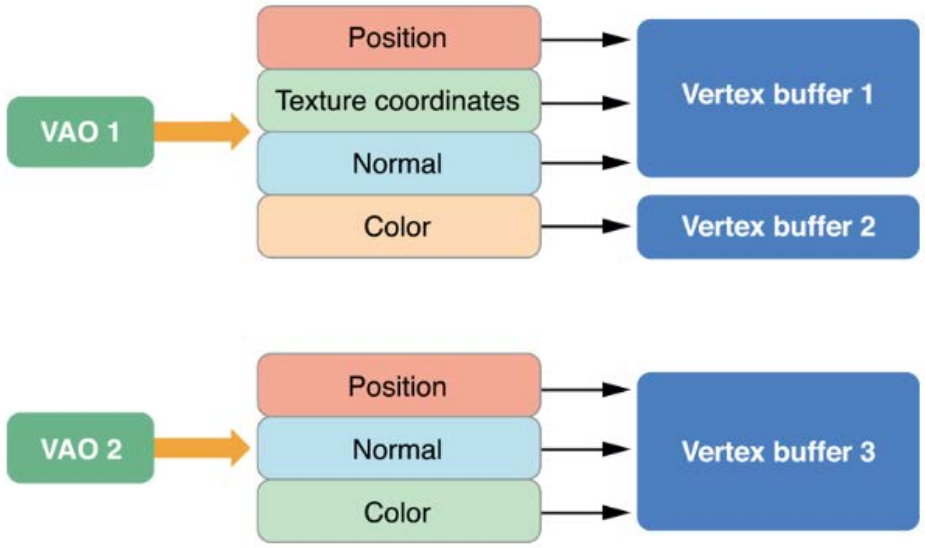

```
unsigned int lightVAO;
glGenVertexArrays(1, &lightVAO);
glBindVertexArray(lightVAO);
// we only need to bind to the VBO, the container's VBO's data already
contains the data.
glBindBuffer(GL_ARRAY_BUFFER, VBO);
// set the vertex attributes (only position data for our lamp)
glVertexAttribPointer(0,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

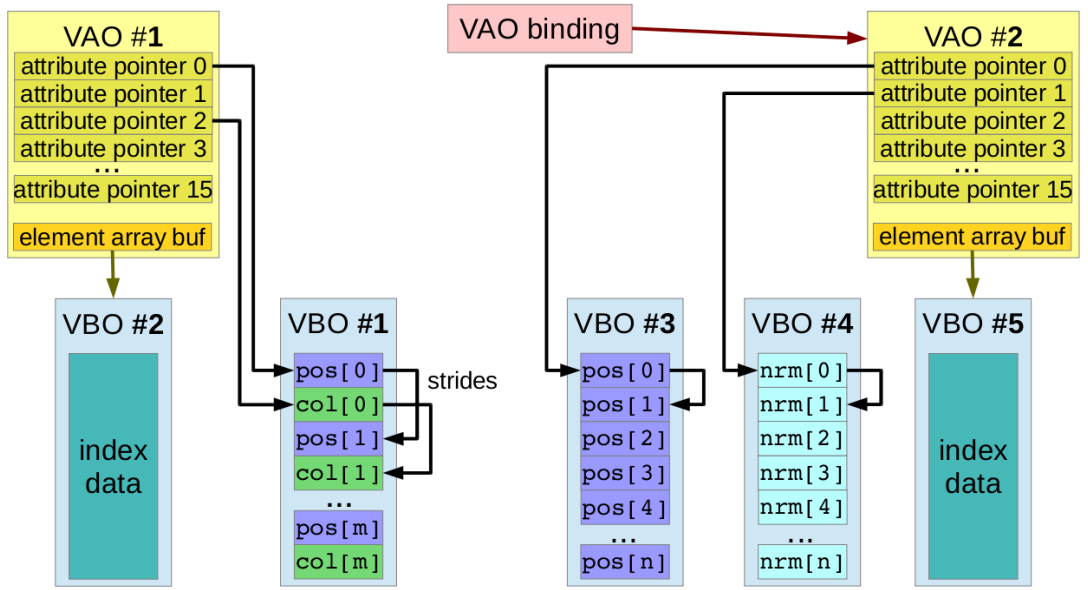
Prop. & Rend. – Colors - A lighting scene

Because we're also going to render a light source cube, we want to generate a **new VAO specifically for the light source**.

```
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 attributes (only position data for our lamp)
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
```



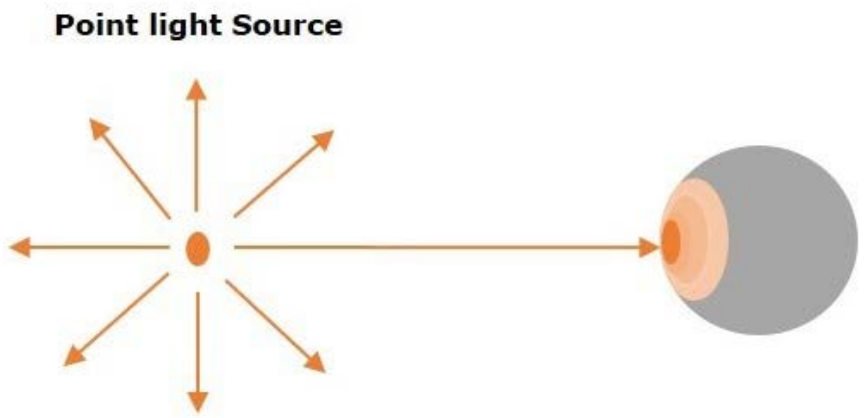




Prop. & Rend. – Colors - A lighting scene

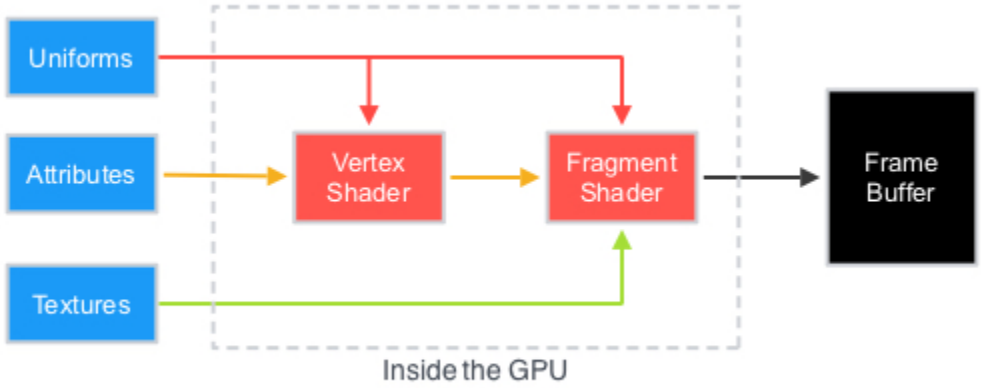
The code should be relatively straightforward. Now that we created both **the container and the light source cube** there is one thing left to define and that is the **fragment shader for both the container and the light source**:

```
#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**.

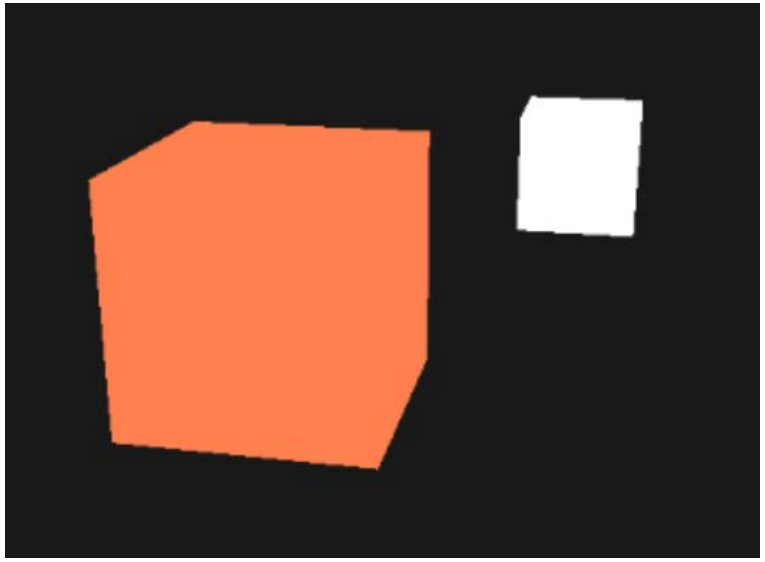
- Here we multiply the light's color with the object's (reflected) color.



Prop. & Rend. – Colors - A lighting scene

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



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).

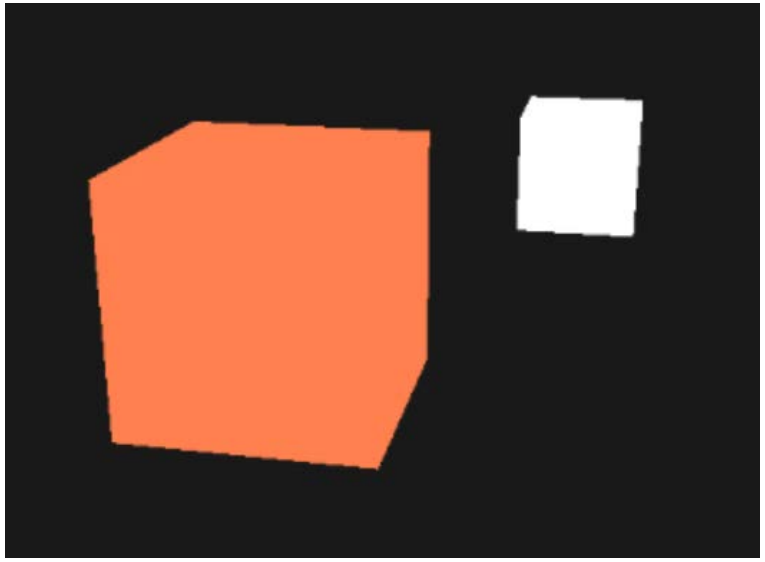
To accomplish this we need to create a **second set of shaders** that we'll use to **draw the light source cube**, thus being safe from any changes to the lighting shaders. The vertex shader is the same as the lighting vertex shader so you can simply copy the source code over. The fragment shader of 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
}
```

Prop. & Rend. – Colors - A lighting scene

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**.

- 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.

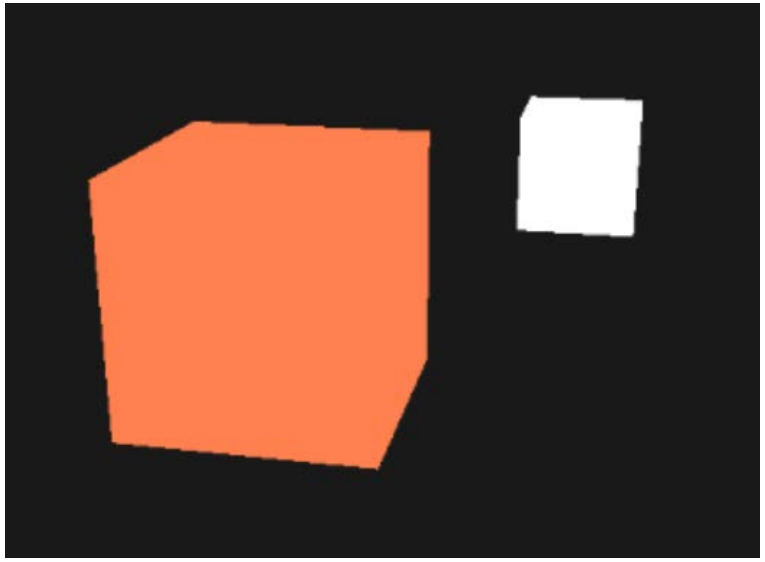
So let's declare a global vec3 variable that represents the light source's location in world-space coordinates:

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

Prop. & Rend. – Colors - A lighting scene

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.

Prop. & Rend. – Colors - A lighting scene

Exercise 13 Task 1: Create a lighting scene composed by a cube object and a light source.