# Computer Graphics – Lighting II

J.-Prof. Dr. habil. Kai Lawonn

# Introduction

- Last lecture, objects having a unique material of its own that reacts differently to light
- In the real world, objects consist of several materials, e.g., a car: exterior consists of a shiny fabric, but has also windows and tires
- The car also has diffuse and ambient colors that are not the same for the entire object
- All by all, such an object has different material properties for each of its different parts
- So need to extend the previous lecture by introducing diffuse and specular maps to influence the diffuse and the specular component

# Diffuse Maps

- Goal: set the diffuse color of an object for each individual fragment
- Instead of a color, we use a texture and apply a diffuse lighting (diffuse map)
- This time, store the texture as a sampler2D inside the Material struct (replace the defined vec3 diffuse color)

# Diffuse Maps

**Sampler2D is a so called opaque type, which means we can't instantiate these types, but only define them as uniforms.**

**If we would instantiate this struct 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.**

# Diffuse Maps

- Remove the ambient material color because the ambient color is in almost all cases equal to the diffuse:

```
struct Material {
    sampler2D diffuse;
    vec3 specular;
    float shininess;
};
…
uniform Material material;
```

# Diffuse Maps

**If you want to set the ambient colors to a different value (not the diffuse value), keep the ambient vec3, but then it is a global color for the entire object → better use another texture for ambient values**

# Diffuse Maps

- Again, texture coordinates are needed in the fragment shader (extra in variable)
- Then retrieve the fragment's ambient/diffuse color value:

```
in vec2 TexCoords;
…
vec3 ambient = light.ambient * texture(material.diffuse, TexCoords).rgb;
…
vec3 diffuse = light.diffuse * diff * texture(material.diffuse, TexCoords).rgb;
```

# Diffuse Maps

- Again 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 (Lec6)
- Update the vertex shader to accept texture coordinates as a vertex attribute and forward them to the fragment shader:

```
…
layout (location = 2) in vec2 aTexCoords;
…
out vec2 TexCoords;
…
void main()
{…
TexCoords = aTexCoords;
}
```

# Diffuse Maps

- Before drawing the cube, assign the texture 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);
```

# F5…

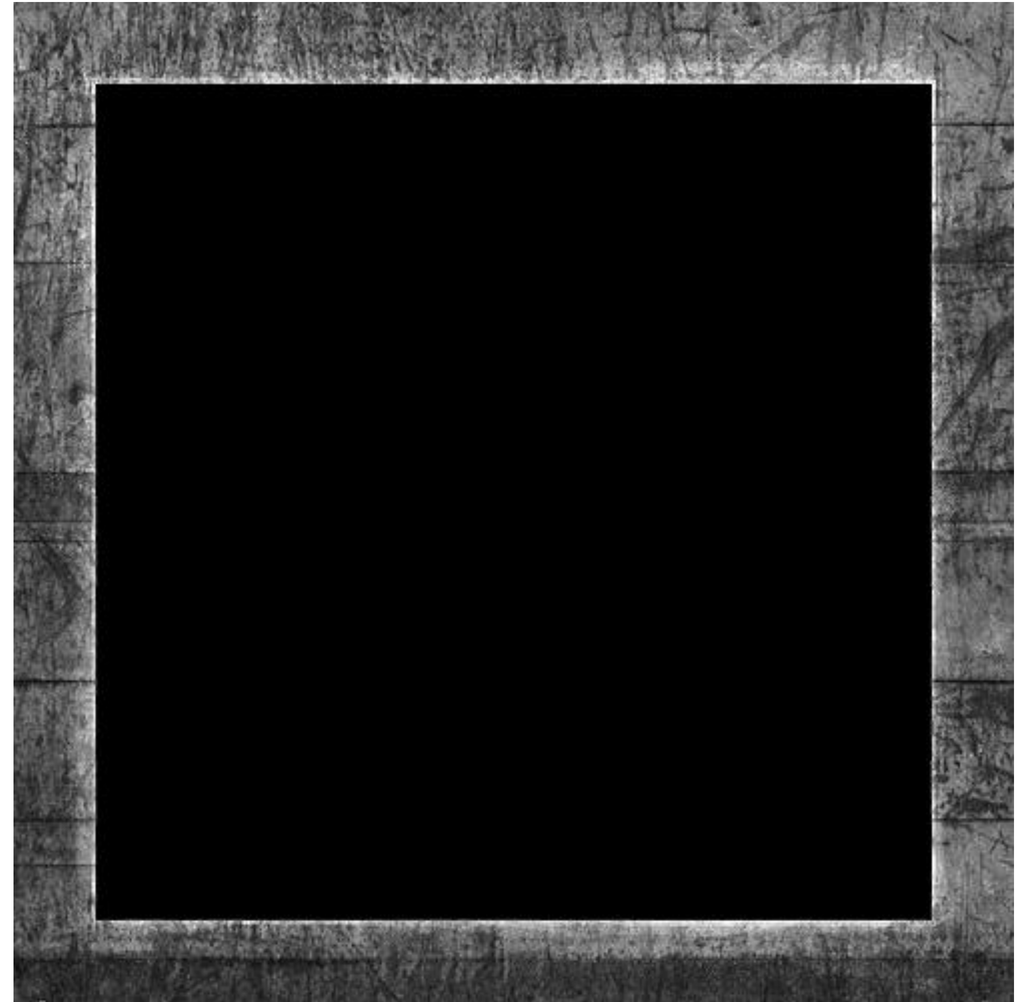
- …much better!

# Specular Maps

- Specular highlights looks strange because the wood material doesn’t give such specular highlights
- If we set the specular material of the object to vec3(0.0), that would help, but then the steel borders have also no specular highlights as well
- Thus, we need different materials

# Specular Maps

- Use a texture map just for specular highlights
- Need to generate a black and white (or colors) texture that defines the specular intensities of each part of the object:



# Specular Maps

- Specular highlight is retrieved by the brightness of each pixel in the image, e.g., black represents the color vector vec3(0.0) and gray vec3(0.5)
- The fragment shader samples color values and multiplies it with the light's specular intensity → the whiter the pixel the brighter the specular component
- Wood no specular highlights (entire wooden section of the diffuse texture was converted to black → no specular highlight)
- Steel border has varying specular intensities (steel has, cracks not)

# Specular Maps

**Wood has specular highlights with a much lower shininess value (more light scattering) and less impact, but as a simplification, pretend wood doesn't have any reaction to specular light.**

# Sampling Specular Maps

- A specular map is yet another texture (similar code as the diffuse map code)
- Using another texture sampler in the same fragment shader: use a different texture unit:

```
lightingShader.use();
lightingShader.setInt("material.diffuse", 0);
lightingShader.setInt("material.specular", 1);
…
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, diffuseMap);

glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, specularMap);
```

# Sampling Specular Maps

- 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;
};
```

# Sampling Specular Maps

- Finally, sample the specular map to retrieve the fragment's corresponding specular intensity:

```
vec3 ambient = light.ambient * texture(material.diffuse, TexCoords).rgb;
vec3 diffuse = light.diffuse * diff * texture(material.diffuse, TexCoords).rgb;
vec3 specular = light.specular * spec * texture(material.specular, TexCoords).rgb;

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

# Sampling Specular Maps

- Using a specular map allows to specify with enormous detail what parts of an object actually have shiny properties and can set their intensity
- Specular maps give an added layer of control on top of the diffuse map

# Sampling Specular Maps

**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 (to completely) 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).**

# F5…

- … that is better!

# Light Casters

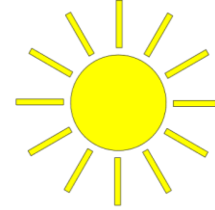
# Introduction

- Used lighting so far came from a single source, a single point in space
- In the real world, several types of light that act different
- A light source that casts light upon objects is called a light caster
- In this lecture, several different types of light casters:
  - Directional light
  - Point light
  - Spotlights

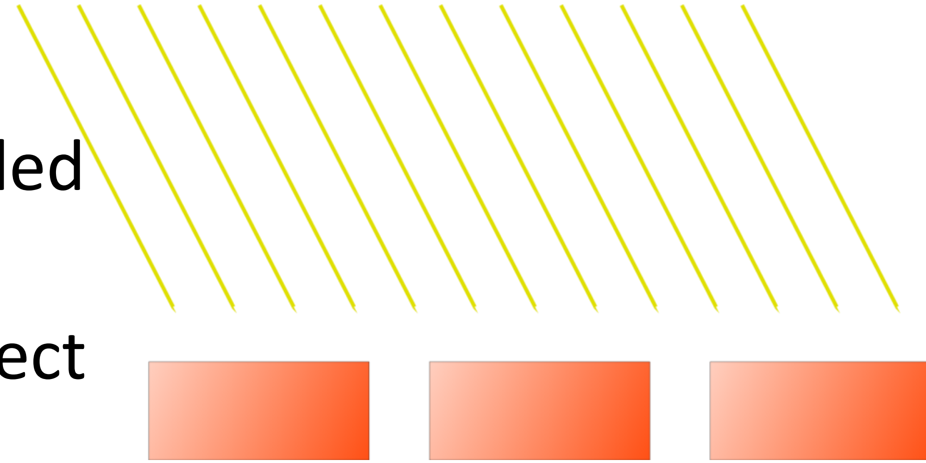
# Directional Light

- When a light source is far away the light rays coming from the light source are close to parallel to each other
- When a light source is modeled to be infinitely far away it is called a directional light (light rays have the same direction)
- Example: the sun (not infinitely far away, but it is so far away that we can perceive it as being infinitely far away)

# Directional Light



- All the light rays from the sun are then modelled as parallel light rays:
- All the light rays are parallel → relation of object and light source's does not matter (light's direction vector stays the same)
- Lighting calculations will be similar for each object in the scene



# Directional Light

- Model such a directional light by defining a light direction vector instead of a position vector

```
struct Light {
    //vec3 position;
    vec3 direction;

    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};
```

# Directional Light

- The shader calculations remain mostly the same (now directly use the light’s direction instead of calculating the lightDir vector with the light’s position vector):

```
vec3 lightDir = normalize(-light.direction);
```

# Directional Light

- Negate the light.direction vector because lighting calculations expect the light direction to be a direction from the fragment towards the light source
- Some people generally prefer to specify a directional light as a global direction pointing from the light source, then negate the global light direction vector to switch its direction
- Now a direction vector pointing towards the light source
- Be sure to normalize the vector

# Directional Light

- To clearly demonstrate that a directional light has the same effect on all multiple objects, use multiple objects (Lec 6):

```
for (unsigned int i = 0; i < 10; i++)
{
    glm::mat4 model = glm::mat4(1.0f);
    model = glm::translate(model, cubePositions[i]);
    float angle = 20.0f * i;
    model = glm::rotate(model, glm::radians(angle), glm::vec3(1.0f, 0.3f, 0.5f));
    lightingShader.setMat4("model", model);

    glDrawArrays(GL_TRIANGLES, 0, 36);
}
```

# Directional Light

- Don’t forget to specify the direction of the light source (note that we define the direction as a direction from the light source):

```
lightingShader.setVec3("light.direction", -0.2f, -1.0f, -0.3f);
```

# Directional Light

**We used the light's position and direction vectors as vec3s**

**Some people tend to prefer to keep all the vectors defined as vec4.**

**For position vectors it is important to set the w component to 1.0 (so translation and projections are applied)**

**When defining a direction vector as a vec4 define the w component to be 0.0 (don't want translations)**

# Directional Light

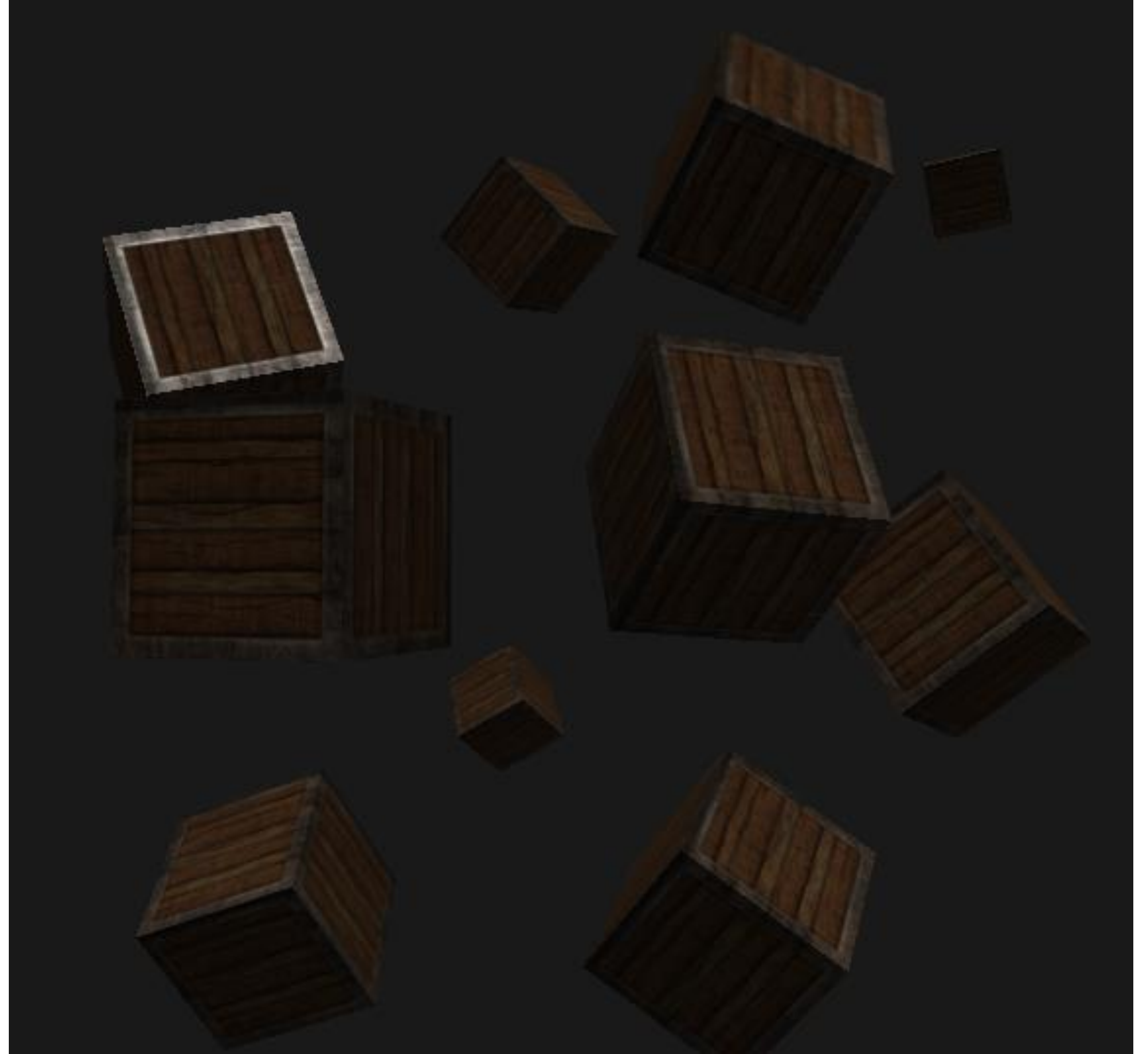
**Direction vectors are represented like: vec4(0.2f, 1.0f, 0.3f, 0.0f)**

**This can function as an easy check for light types: if the w component is equal to 1.0 we have a light's position vector and if w is equal to 0.0 we have a light's direction vector:**

```
if(lightVector.w == 0.0) // note: be careful for floating point errors
// do directional light calculations
else if(lightVector.w == 1.0)
// do light calculations using the light's position (like last tutorial)
```

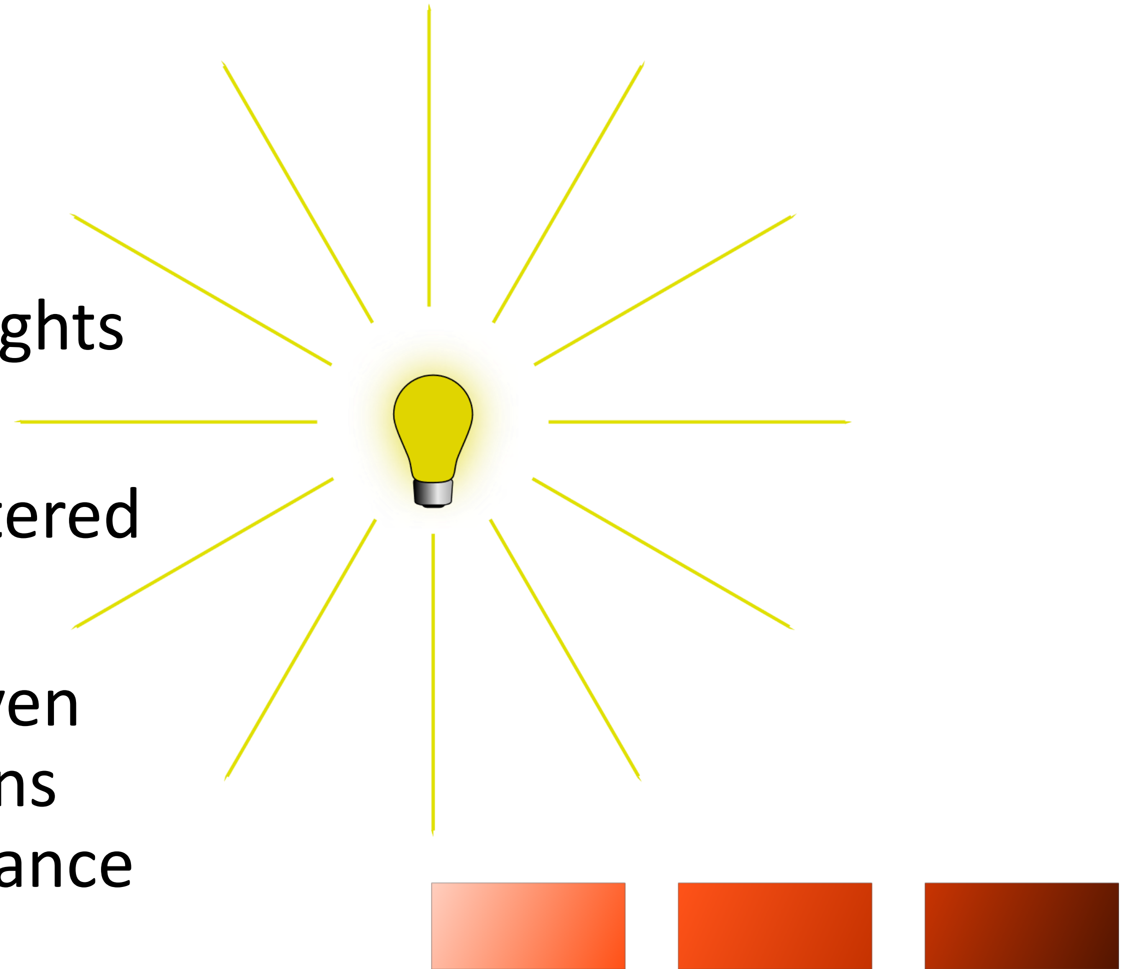
# F5…

- … like light source casting



# Point Light

- Directional lights are great for global lights that illuminate the entire scene
- But also want several point lights scattered throughout the scene
- A point light is a light source with a given position that illuminates in all directions where the light rays fade out over distance (like light bulbs and torches)

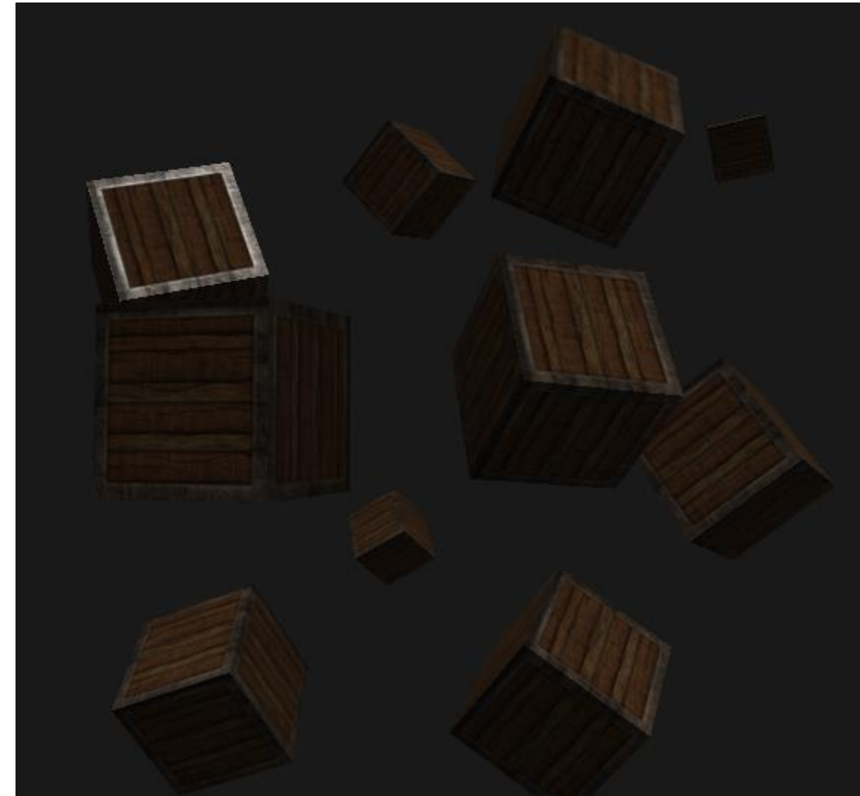


# Directional Light

- So far, had a light source at a given position that scatters light in all directions
- Light rays never fade out thus making it look like the light source is extremely strong
- In most 3D simulations simulate a light source that illuminates a certain area close to the light source and not the entire scene

# Directional Light

- All boxes are lit with the same intensity (independent if they are in the back or in the front)
- We want the container in the back to only be slightly lit in comparison to the containers close to the light source



# Attenuation

- The reduction of the intensity of light over distance is attenuation
- One way to reduce the light intensity over distance is to simply use a linear equation (linearly reduce the light intensity over the distance thus making sure that objects at a distance are less bright)
- Such a linear function tends to look a bit fake
- In the real world, lights are generally quite bright standing close by, but the brightness of a light source diminishes quickly at the start and the remaining light intensity more slowly diminishes over distance

# Attenuation

- We are thus in need of a different formula for reducing the light's intensity
- The following formula calculates an attenuation value based on a fragment's distance to the light source, which we later multiply with the light's intensity vector:

$$F_{att} = \frac{1}{K_c + K_l \cdot d + K_q \cdot d^2}$$

$$F_{att} = \frac{1}{K_c + K_l \cdot d + K_q \cdot d^2}$$

# Attenuation

- $d$ represents the distance from the fragment to the light source
- To calculate the attenuation value we define 3 (configurable) terms:
  - a constant term $K_c$, a linear term $K_l$ and a quadratic term $K_q$
- $K_c$ is usually kept at 1.0 (ensure the resulting denominator never gets smaller than 1)
- $K_l$ is multiplied with the distance value that reduces the intensity in a linear fashion
- $K_q$ is multiplied with the quadrant of the distance and sets a quadratic decrease of intensity for the light source

$$F_{att} = \frac{1}{K_c + K_l \cdot d + K_q \cdot d^2}$$

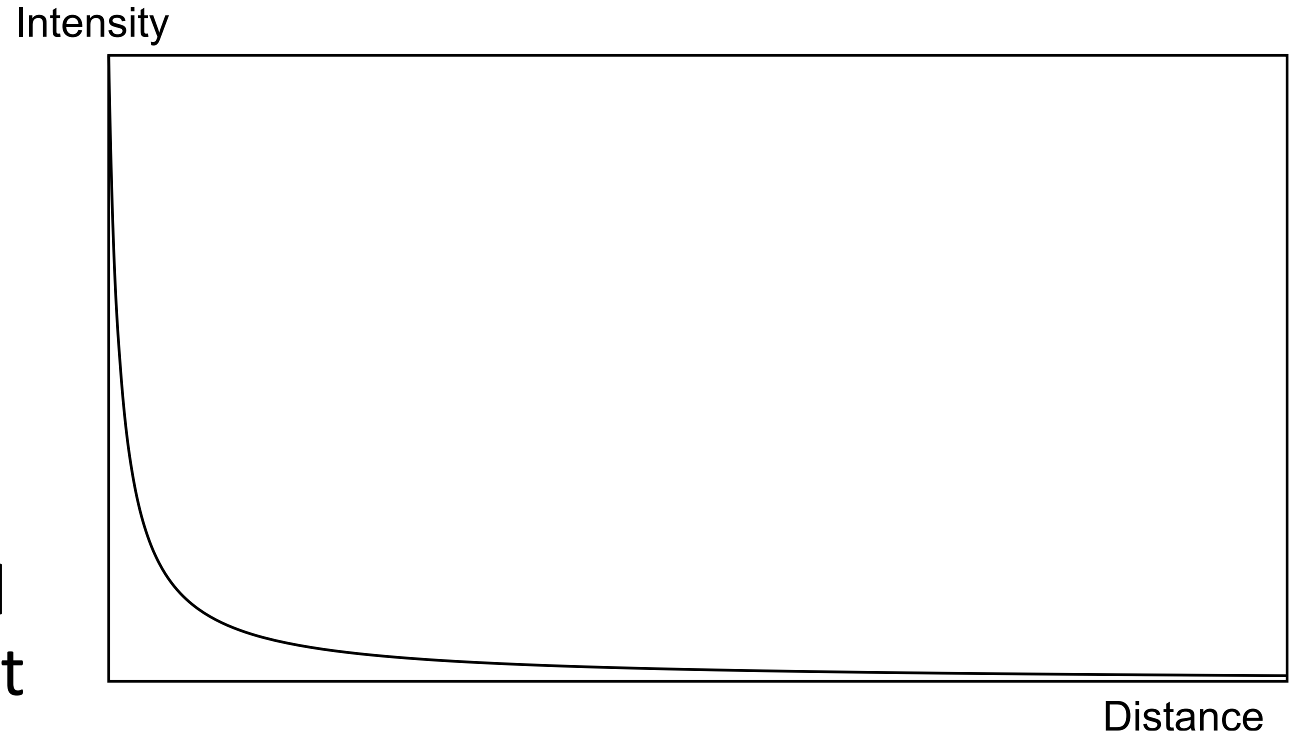
# Attenuation

- Due to $K_q$ the light will diminish mostly at a linear fashion until the distance becomes large enough for the quadratic term to surpass $K_l$ and then the light intensity will decrease a lot faster
- The resulting effect is that the light is quite intense when at a close range, but quickly loses it brightness over distance and eventually loses its brightness at a more slower pace

$$F_{att} = \frac{1}{K_c + K_l \cdot d + K_q \cdot d^2}$$

# Attenuation

- The following graph shows the effect such an attenuation has over a distance of 100:
- Light has the highest intensity when the distance is small
- As the distance grows its intensity is significantly reduced and slowly reaches 0 intensity at around a distance of 100



$$F_{att} = \frac{1}{K_c + K_l \cdot d + K_q \cdot d^2}$$

# Choosing the right Values

- What are good or right values?
- Depends on many factors: the environment, the distance you want a light to cover, the type of light etc.
- Mostly it is a question of experience and a moderate amount of tweaking
- These values are good starting points for most lights (Ogre3d.org)

| Distance | Constant | Linear | Quadratic |
|---|---|---|---|
| 7 | 1.0 | 0.7 | 1.8 |
| 13 | 1.0 | 0.35 | 0.44 |
| 20 | 1.0 | 0.22 | 0.20 |
| 32 | 1.0 | 0.14 | 0.07 |
| 50 | 1.0 | 0.09 | 0.032 |
| 65 | 1.0 | 0.07 | 0.017 |
| 100 | 1.0 | 0.045 | 0.0075 |
| 160 | 1.0 | 0.027 | 0.0028 |
| 200 | 1.0 | 0.022 | 0.0019 |
| 325 | 1.0 | 0.014 | 0.0007 |
| 600 | 1.0 | 0.007 | 0.0002 |
| 3250 | 1.0 | 0.0014 | 0.000007 |

http://www.ogre3d.org/tikiwiki/tiki-index.php?page=-Point+Light+Attenuation

# Choosing the right Values

$$F_{att} = \frac{1}{K_c + K_l \cdot d + K_q \cdot d^2}$$

- As you can see, $K_c$ is kept at 1.0
- $K_q$ is usually quite small to cover larger distances and $K_q$ is even smaller

| Distance | Constant | Linear | Quadratic |
|---|---|---|---|
| 7 | 1.0 | 0.7 | 1.8 |
| 13 | 1.0 | 0.35 | 0.44 |
| 20 | 1.0 | 0.22 | 0.20 |
| 32 | 1.0 | 0.14 | 0.07 |
| 50 | 1.0 | 0.09 | 0.032 |
| 65 | 1.0 | 0.07 | 0.017 |
| 100 | 1.0 | 0.045 | 0.0075 |
| 160 | 1.0 | 0.027 | 0.0028 |
| 200 | 1.0 | 0.022 | 0.0019 |
| 325 | 1.0 | 0.014 | 0.0007 |
| 600 | 1.0 | 0.007 | 0.0002 |
| 3250 | 1.0 | 0.0014 | 0.000007 |

http://www.ogre3d.org/tikiwiki/tiki-index.php?page=-Point+Light+Attenuation

# Attenuation

- To implement attenuation 3 extra values are needed in the fragment shader: the constant, linear and quadratic terms
- These are best stored in the Light struct
- Note, calculate lightDir as in the previous lecture (not a directional light)

```
struct Light {
    vec3 position;

    vec3 ambient;
    vec3 diffuse;
    vec3 specular;

    float constant;
    float linear;
    float quadratic;
};
```

# Attenuation

- Then set the values in OpenGL (want the light to cover a distance of 50):

```
lightingShader.setFloat("light.constant", 1.0f);
lightingShader.setFloat("light.linear", 0.09f);
lightingShader.setFloat("light.quadratic", 0.032f);
```

# Attenuation

- Implementing attenuation in the fragment shader: calculate an attenuation value based on the formula and multiply this with the ambient, diffuse and specular components
- The distance to the light source can be retrieved by the difference vector between the fragment and the light source and take the resulting vector’s length:

```
float distance    = length(light.position - FragPos);
float attenuation = 1.0 / (light.constant + light.linear * distance +
light.quadratic * (distance * distance));
```

# Attenuation

- Attenuation value in is multiplied with the ambient, diffuse and specular colors

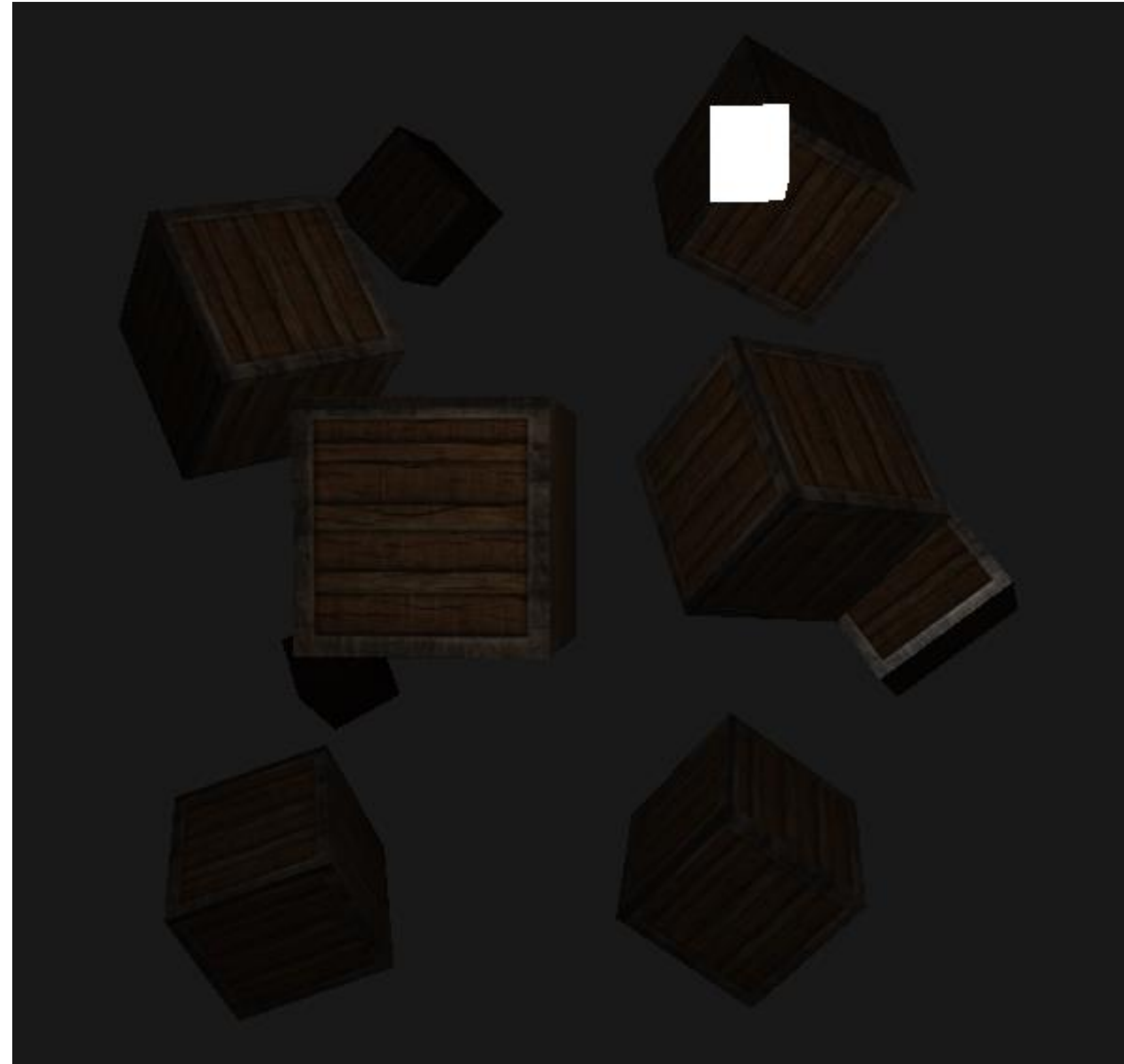
```
ambient  *= attenuation;
diffuse   *= attenuation;
specular *= attenuation;
```

# Attenuation

**Could leave the ambient component such that it is not decreased over distance, but if we were to use more than 1 light source all the ambient components will start to stack up so in that case we want to attenuate ambient lighting as well.**

# F5…

- front boxes are lit, boxes in the back are dark

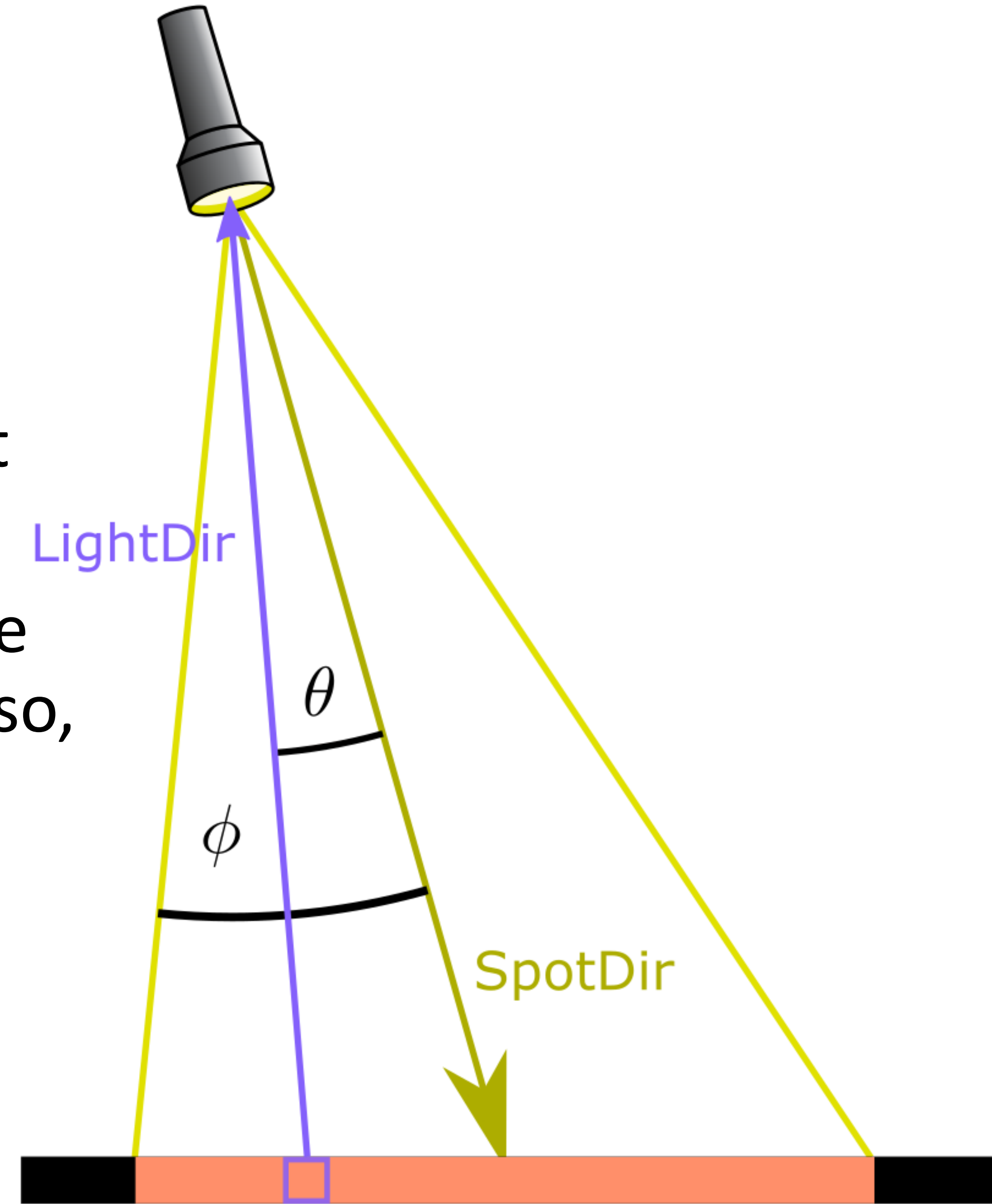


# Spotlight

- A spotlight is a light source that, instead of shooting light rays in all directions, only shoots them in a specific direction
- The result is that only the objects within a certain radius of the spotlight's direction are lit and everything else stays dark
- A good example of a spotlight would be a street lamp or a flashlight

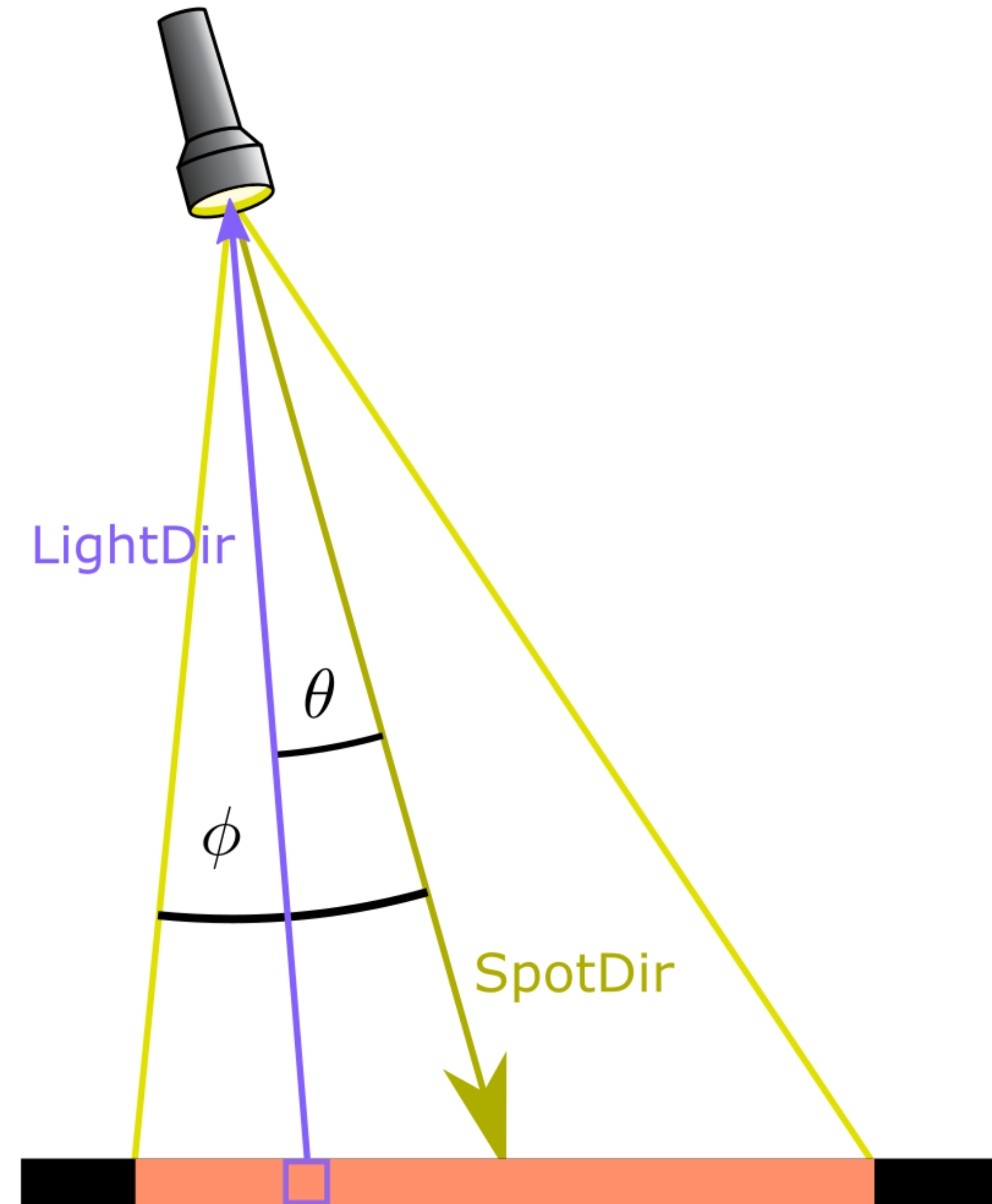
# Spotlight

- Spotlight is represented by a world-space position, a direction and a cutoff angle that specifies the radius of the spotlight
- For each fragment check if it is between the spotlight's cutoff directions (in its cone), if so, lit the fragment



# Spotlight

- LightDir: the vector pointing from the fragment to the light source
- SpotDir: the direction the spotlight is aiming at
- Phi $\phi$: the cutoff angle that specifies the spotlight's radius (outside this angle is not lit)
- Theta $\theta$: the angle between the LightDir vector and the SpotDir vector. $\theta$ should be smaller than $\phi$ to be inside the spotlight



# Spotlight

- Need to do calculate the dot product of two unit vectors (returns the cosine of the angle)
- Unit vectors are the LightDir vector and the SpotDir vector
- Compare this angle with the cutoff angle $\phi$

# Spotlight

- A flashlight is a spotlight located at the viewer's position and usually aimed straight ahead from the player's perspective
- So, the fragment shader needs the spotlight's position vector (to calculate the light's direction vector), the spotlight's direction vector and the cutoff angle:

```
struct Light {
    vec3 position;
    vec3 direction;
    float cutOff;
    …
```

# Spotlight

- Pass the values to the shaders:

```
lightingShader.setVec3("light.position", camera.Position);
lightingShader.setVec3("light.direction", camera.Front);
lightingShader.setFloat("light.cutOff", glm::cos(glm::radians(12.5f)));
```

# Spotlight

- Not setting an angle for the cutoff value, but calculate the cosine of an angle
- Instead of calculating the angle between the LightDir and the SpotDir vector, the inverse of the cosine needs to be determined (an expensive operation)
- To save some performance, compare the cosine values

# Spotlight

- Calculate the $\theta$ value and compare this with the cutoff $\phi$ value to determine if the fragment is in or outside the spotlight:

```
float theta = dot(lightDir, normalize(-light.direction));

if(theta > light.cutOff) // '>’ because of the cos
{
…
}
else
{
   FragColor = vec4(light.ambient * texture(material.diffuse, TexCoords).rgb, 1.0);
}
```

# Spotlight

- Calculate the dot product between the lightDir vector and the negated direction vector
- Be sure to normalize all the relevant vectors

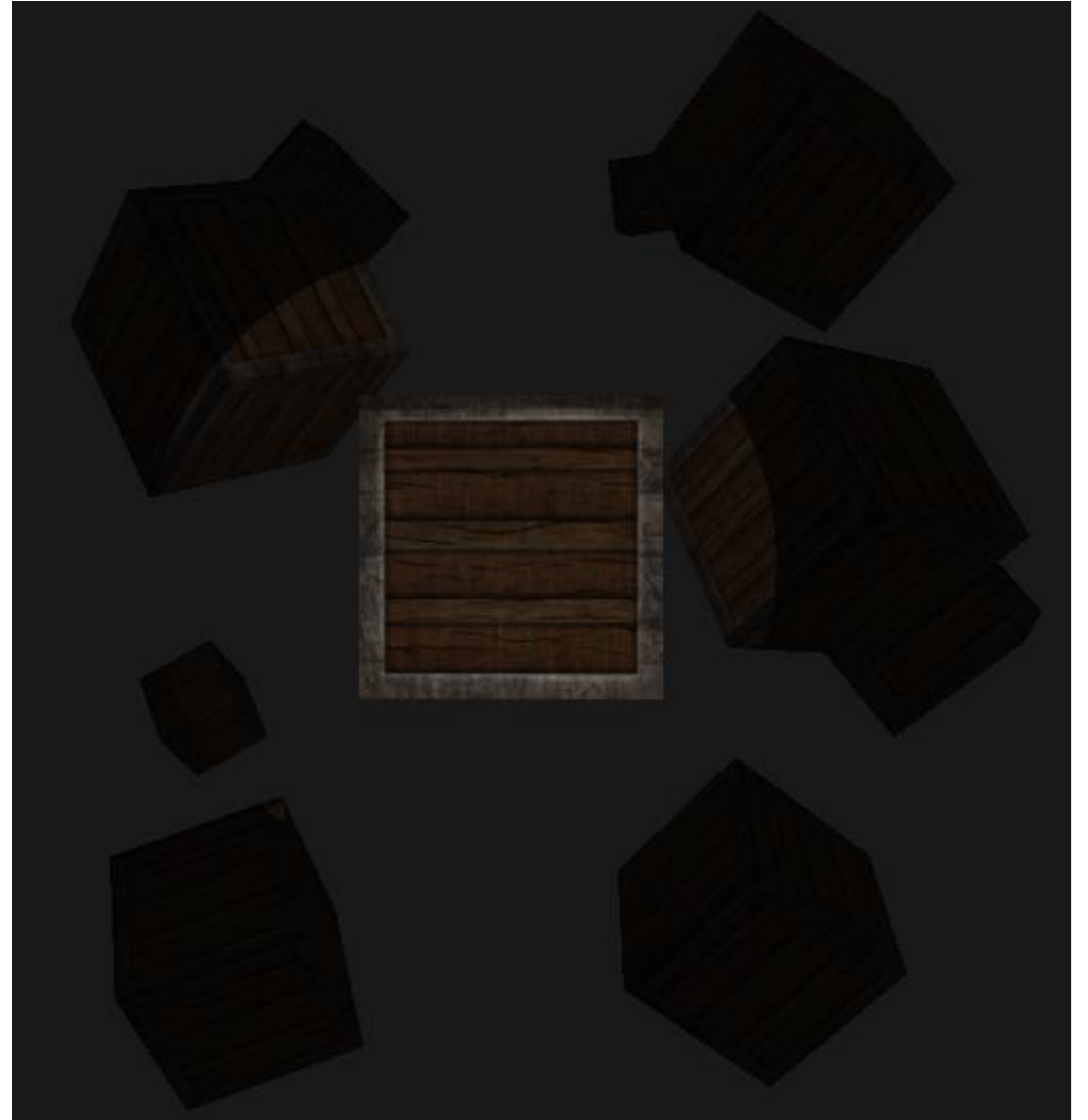
# Spotlight

**if(theta > light.cutOff)?**

**The cosine is decreasing in the interval [0,90°], thus, the greater the angle, the lower the cosine.**

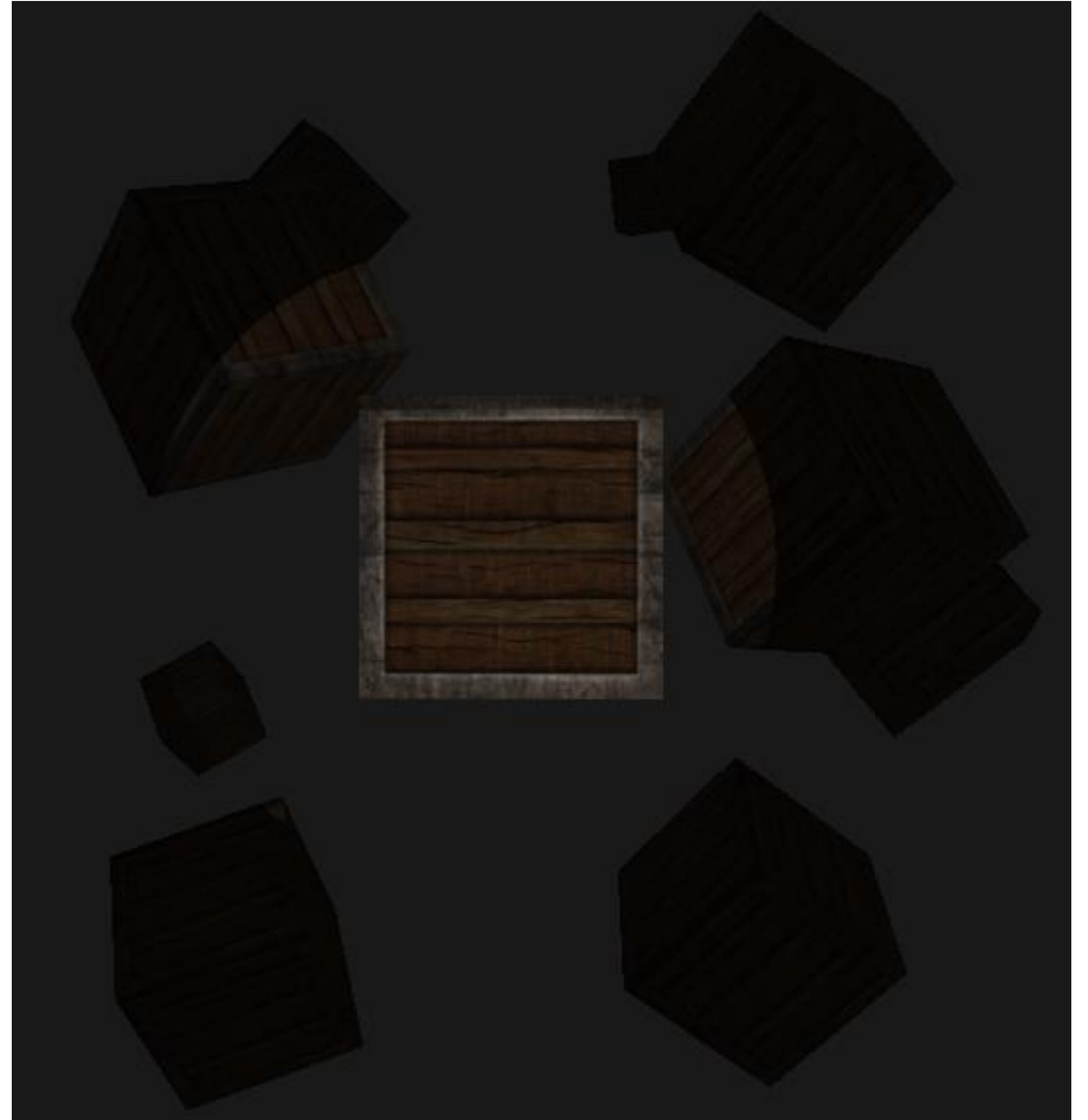
# F5…

- … a scary spotlight!



# F5…

- … looks strange, because the spotlight has hard edges
- Wherever a fragment reaches the edge of the spotlight’s cone it is completely dark instead of with a smooth fade



# Smooth/Soft Edges

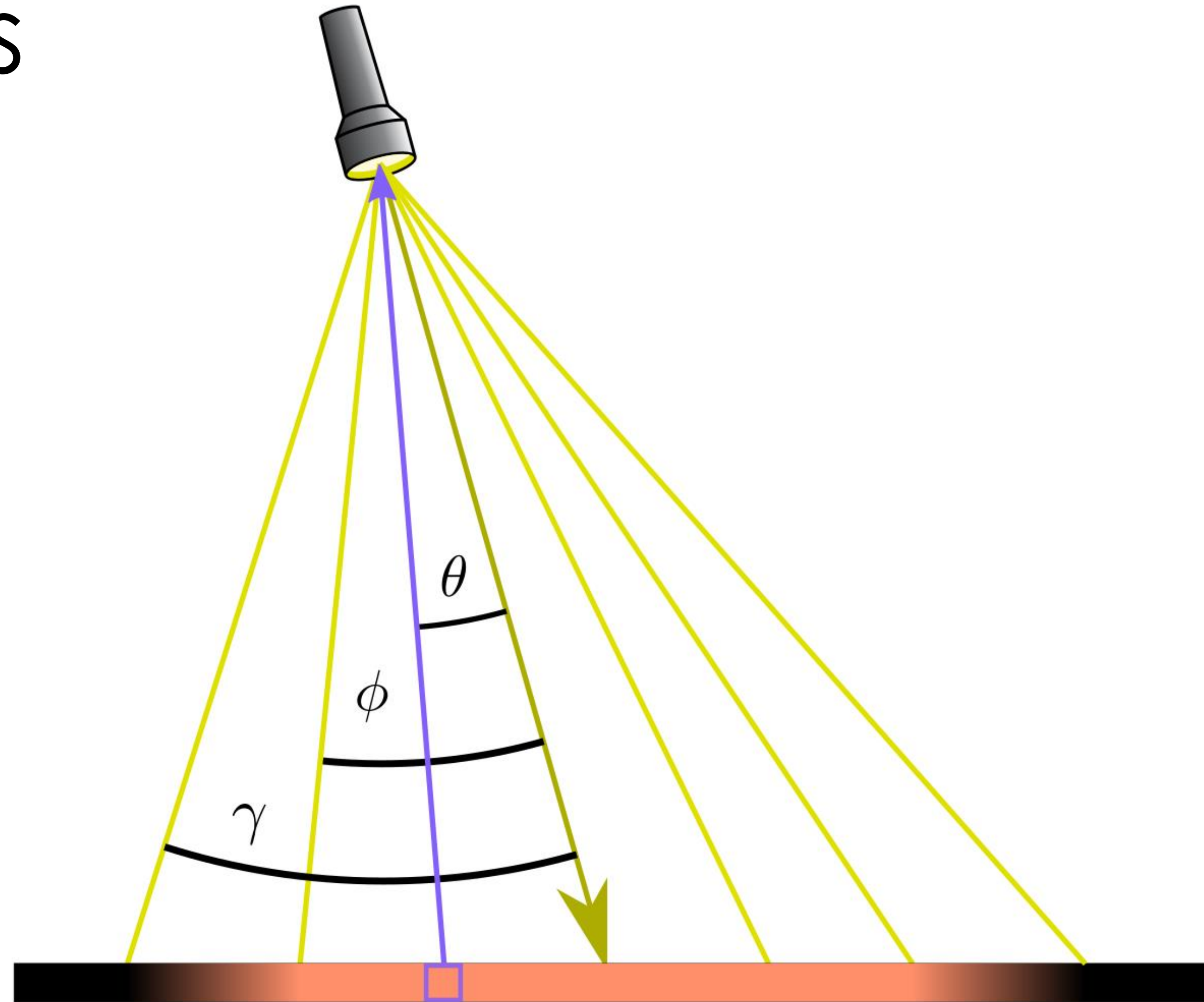
- To create the smooth transitions, simulate a spotlight having an inner and an outer cone
- Inner cone: defined in the previous section
- Outer cone: gradually dims the light from the inner to the edges of the outer cone

# Smooth/Soft Edges

- Therefore, define another cosine value that represents the angle between the spotlight's direction vector and the outer cone's vector (equal to its radius)
- Then, if a fragment is between the inner and the outer cone it should calculate an intensity value between 0.0 and 1.0
- If the fragment is inside the inner cone its intensity is equal to 1.0 and 0.0 if the fragment is outside the outer cone

# Smooth/Soft Edges

- Calculate $I$ (intensity)
- Goal:
  - If $\theta \geq \gamma$, then $I = 0$
  - If $\theta \leq \phi$, then $I = 1$
  - If $\theta \in [\phi, \gamma]$, then $I = 1 - \frac{\theta - \phi}{\gamma - \phi}$

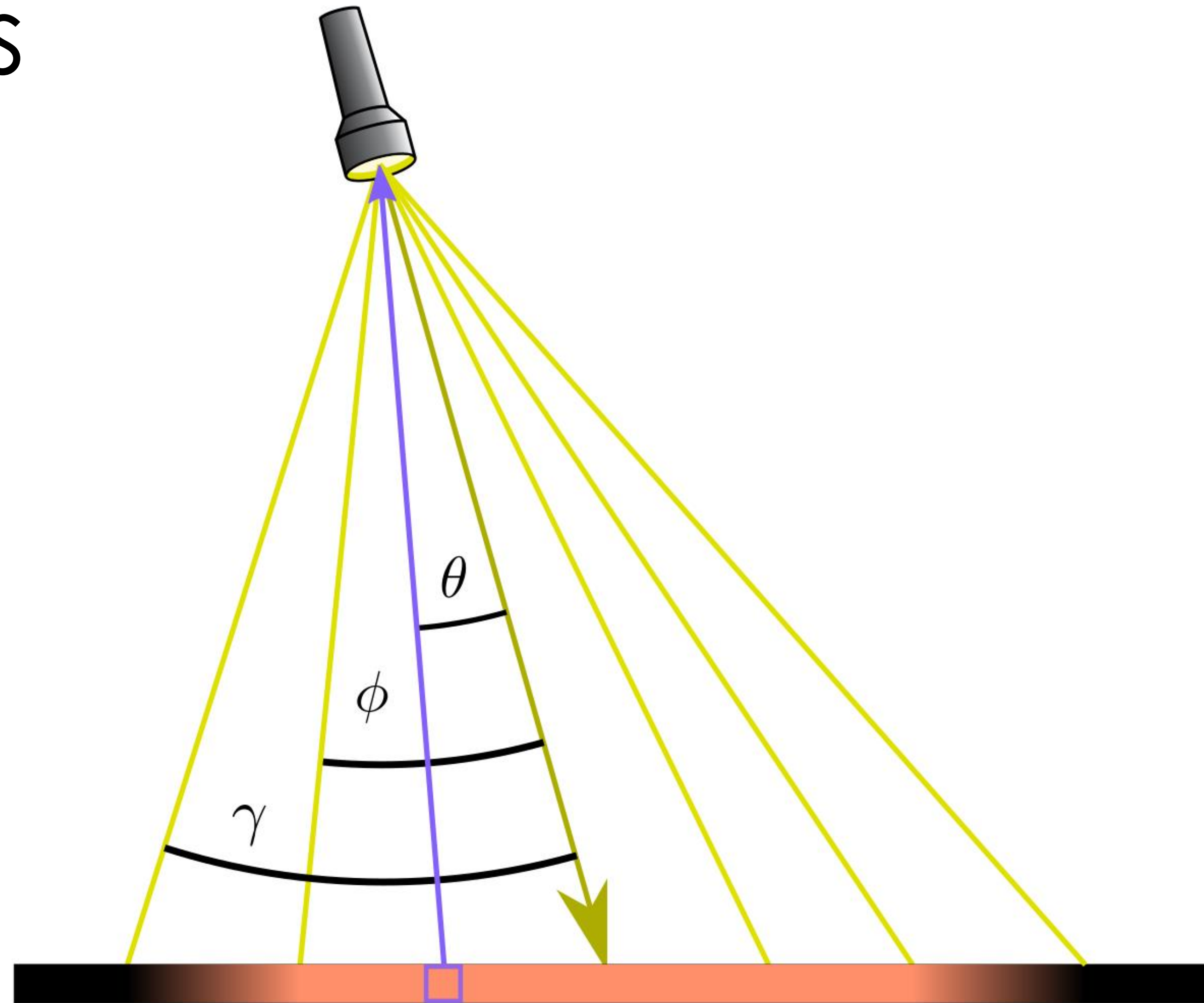


# Smooth/Soft Edges

- Calculate $I$ (intensity)
- Goal:
  - If $\theta \geq \gamma$, then $I = 0$
  - If $\theta \leq \phi$, then $I = 1$
  - If $\theta \in [\phi, \gamma]$, then

$$I = 1 - \frac{\theta - \phi}{\gamma - \phi} = \frac{\gamma - \phi - (\theta - \phi)}{\gamma - \phi}$$

$$= \frac{\gamma - \theta}{\gamma - \phi} = \frac{\theta - \gamma}{\phi - \gamma}$$



# Smooth/Soft Edges

$$I = \frac{\theta - \gamma}{\phi - \gamma}$$

- To bring the three conditions together, the clamp function is used:

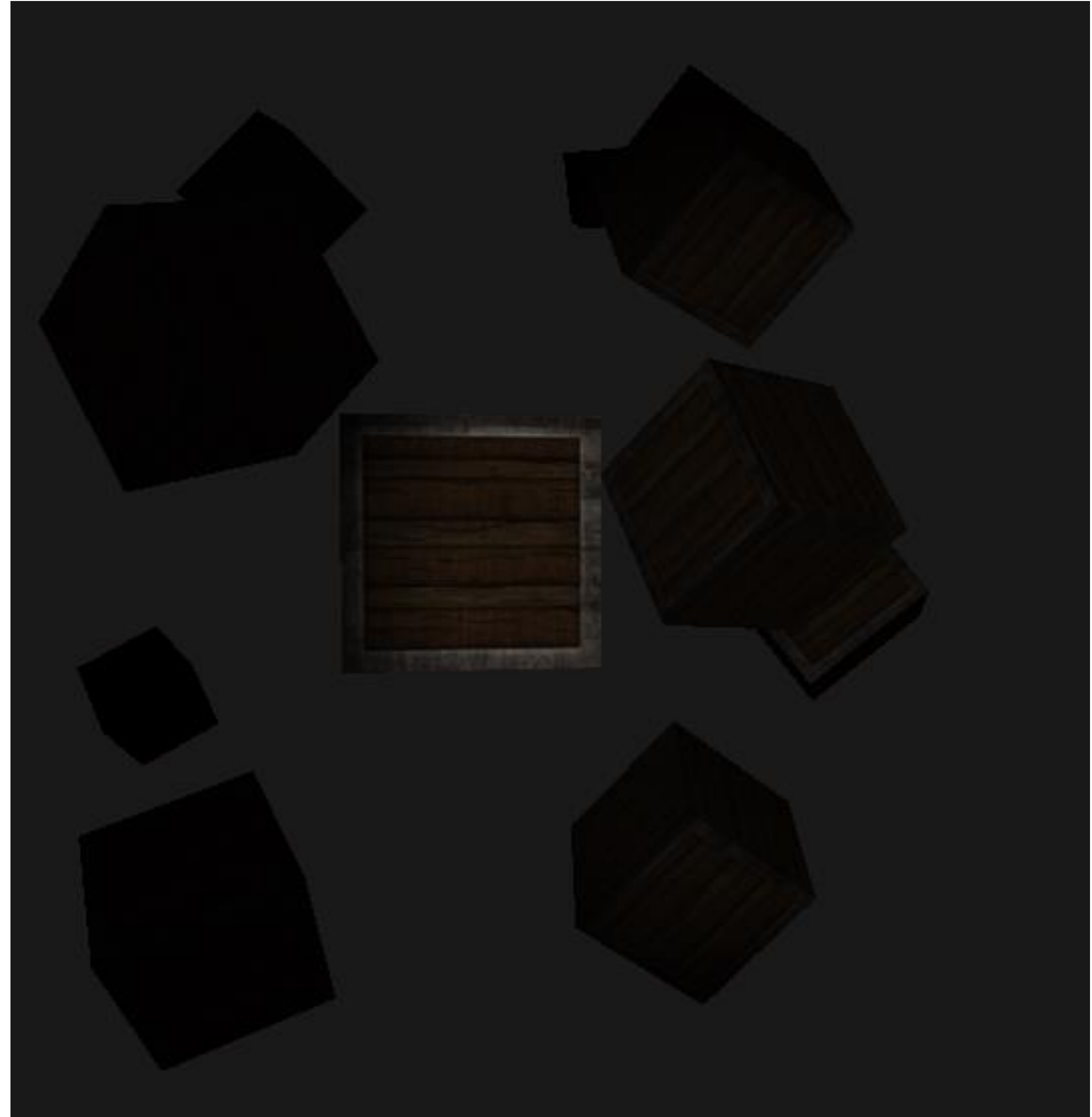
```
float theta = dot(lightDir, normalize(-light.direction));
float intensity = clamp((theta - light.outerCutOff) /
                              (light.cutOff - light.outerCutOff), 0.0, 1.0);

diffuse  *= intensity;
specular *= intensity;
```

- Add the outerCutOff value to the Light struct

```
lightingShader.setFloat("light.outerCutOff", glm::cos(glm::radians(17.5f)));
```

# F5…

- … much better!



# Multiple Lights

# Introduction

- Now, combine everything we learned so far (Phong shading, materials, lighting maps and different types of light casters)
- Create a fully lit scene with 6 active light sources:
  - 1 sun-like light as a directional light source
  - 4 point lights scattered throughout the scene
  - 1 flashlight
- To use more than one light source in the scene, we use GLSL functions, otherwise the code quickly becomes difficult to understand
- Create a different function for each of the light types: directional lights, point lights and spotlights.

# Introduction

- When using multiple lights in a scene the approach is usually as follows: a single color vector represents the fragment's output color
- For each light, the light's contribution color of the fragment is added to the fragment's output color vector
- Each light in the scene will calculate its individual impact on the aforementioned fragment and contribute to the final output color

# Introduction

- A general structure would look something like this:

```
out vec4 FragColor;

void main()
{
// define an output color value
vec3 output;
// add the directional light's contribution to the output
output += someFunctionToCalculateDirectionalLight();
// do the same for all point lights
for(int i = 0; i < nr_of_point_lights; i++)
        output += someFunctionToCalculatePointLight();
// and add others lights as well (like spotlights)
output += someFunctionToCalculateSpotLight();

FragColor = vec4(output, 1.0);
}
```

# Directional Light

- Define a function in the fragment shader for the directional light: takes a few parameters and returns the calculated directional lighting color
- Set the required variables for a directional light source (as a struct):

```
struct DirLight {
    vec3 direction;

    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

uniform DirLight dirLight;
```

# Directional Light

- The dirLight uniform can be passed to a function with the following prototype:

```
vec3 CalcDirLight(DirLight light, vec3 normal, vec3 viewDir);
```

# Directional Light

**Like C and C++ if we want to call a function (in this case inside the main function) the function should be defined somewhere before the caller's line number.**

**In this case we'd prefer to define the functions below the main function so this requirement doesn't hold.**

**Therefore we declare the function's prototypes somewhere above the main function, just like we would in C.**

# Directional Light

- The function requires a DirLight struct and two other vectors required for its computation:

```
vec3 CalcDirLight(DirLight light, vec3 normal, vec3 viewDir)
{
    vec3 lightDir = normalize(-light.direction);
    // diffuse shading
    float diff = max(dot(normal, lightDir), 0.0);
    // specular shading
    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
    // combine results
    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));
    return (ambient + diffuse + specular);
}
```

# Point Light

- Define a struct for contribution of a point light including its attenuation:

```
struct PointLight {
    vec3 position;

    float constant;
    float linear;
    float quadratic;

    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};
```

# Point Light

- Use a pre-processor directive in GLSL to define the number of point lights
- Then use this NR_POINT_LIGHTS constant to create an array of PointLight structs:

```
#define NR_POINT_LIGHTS 4
uniform PointLight pointLights[NR_POINT_LIGHTS];
```

# Point Light

**Could also define one large struct (instead of different structs per light type) that contains all the necessary variables for all the different light types and use that struct for each function, and simply ignore the unneeded variables.**

**The current approach is more intuitive and aside from a few extra lines of code it could save up some memory since not all light types need all variables.**

# Point Light

- The prototype of the point light's function is as follows:

```
vec3 CalcPointLight(PointLight light, vec3 normal, vec3 fragPos, vec3 viewDir);
```

# Point Light

- The function takes all the data it needs as its arguments and returns a vec3 that represents the color:

```
vec3 CalcPointLight(PointLight light, vec3 normal, vec3 fragPos, vec3 viewDir)
{
    vec3 lightDir = normalize(light.position - fragPos);
    // diffuse shading
    float diff = max(dot(normal, lightDir), 0.0);
    // specular shading
    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
    // attenuation
    float distance = length(light.position - fragPos);
    float attenuation = 1.0 / (light.constant + light.linear * distance + light.quadratic * (distance * distance));
    // combine results
    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));
    ambient *= attenuation;
    diffuse *= attenuation;
    specular *= attenuation;
    return (ambient + diffuse + specular);
}
```

# Spot Light

- Define a struct for the spot light:

```
struct SpotLight {
    vec3 position;
    vec3 direction;
    float cutOff;
    float outerCutOff;

    float constant;
    float linear;
    float quadratic;

    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};
…
uniform SpotLight spotLight;
```

# Spot Light

- The corresponding function:

```
vec3 CalcSpotLight(SpotLight light, vec3 normal, vec3 fragPos, vec3 viewDir)
{
    vec3 lightDir = normalize(light.position - fragPos);
    // diffuse shading
    float diff = max(dot(normal, lightDir), 0.0);
    // specular shading
    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
    // attenuation
    float distance = length(light.position - fragPos);
    float attenuation = 1.0 / (light.constant + light.linear * distance + light.quadratic * (distance * distance));
    // spotlight intensity
    float theta = dot(lightDir, normalize(-light.direction));
    float epsilon = light.cutOff - light.outerCutOff;
    float intensity = clamp((theta - light.outerCutOff) / epsilon, 0.0, 1.0);
    // combine results
    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));
    ambient *= attenuation * intensity;
    diffuse *= attenuation * intensity;
    specular *= attenuation * intensity;
    return (ambient + diffuse + specular);
}
```

# All together

- Put it all together in the main function:

```
void main()
{
    // properties
    vec3 norm = normalize(Normal);
    vec3 viewDir = normalize(viewPos - FragPos);

    // phase 1: directional lighting
    vec3 result = CalcDirLight(dirLight, norm, viewDir);
    // phase 2: point lights
    for(int i = 0; i < NR_POINT_LIGHTS; i++)
        result += CalcPointLight(pointLights[i], norm, FragPos, viewDir);
    // phase 3: spot light
    result += CalcSpotLight(spotLight, norm, FragPos, viewDir);

    FragColor = vec4(result, 1.0);
}
```

# All together

- Need to define a position vector for each of the point lights
- Define another glm::vec3 array that contains the pointlights' positions:

```
glm::vec3 pointLightPositions[] = {
        glm::vec3( 0.7f,  0.2f,  2.0f),
        glm::vec3( 2.3f, -3.3f, -4.0f),
        glm::vec3(-4.0f,  2.0f, -12.0f),
        glm::vec3( 0.0f,  0.0f, -3.0f)
    };
```

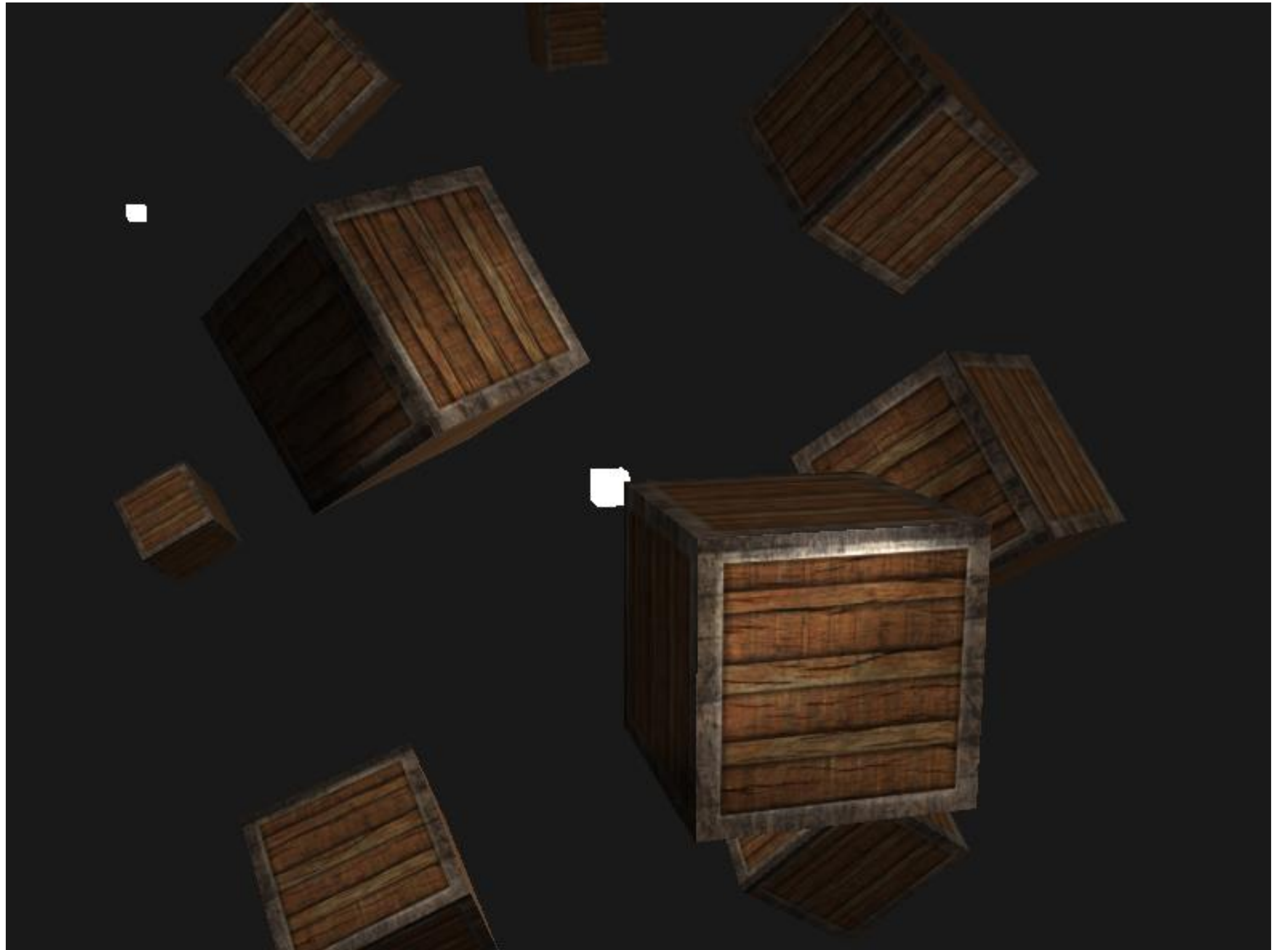
# All together

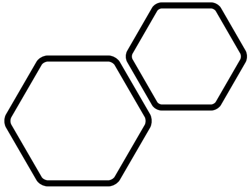
- Setting the uniforms for the directional light struct and the point lights:

```
lightingShader.setVec3("dirLight.direction", -0.2f, -1.0f, -0.3f);
lightingShader.setVec3("dirLight.ambient", 0.05f, 0.05f, 0.05f);
lightingShader.setVec3("dirLight.diffuse", 0.4f, 0.4f, 0.4f);
lightingShader.setVec3("dirLight.specular", 0.5f, 0.5f, 0.5f);
// point light 1
lightingShader.setVec3("pointLights[0].position", pointLightPositions[0]);
lightingShader.setVec3("pointLights[0].ambient", 0.05f, 0.05f, 0.05f);
lightingShader.setVec3("pointLights[0].diffuse", 0.8f, 0.8f, 0.8f);
lightingShader.setVec3("pointLights[0].specular", 1.0f, 1.0f, 1.0f);
lightingShader.setFloat("pointLights[0].constant", 1.0f);
lightingShader.setFloat("pointLights[0].linear", 0.09);
lightingShader.setFloat("pointLights[0].quadratic", 0.032);
…
```

# F5…

- …very neat!





# Questions???