Computer Graphics
– Lighting I

J.-Prof. Dr. habil. Kai Lawonn

Introduction

- In the previous lectures, colors were briefly mentioned
- Now, we want to extensively discuss what colors are and start building the scene for the upcoming lighting tutorials

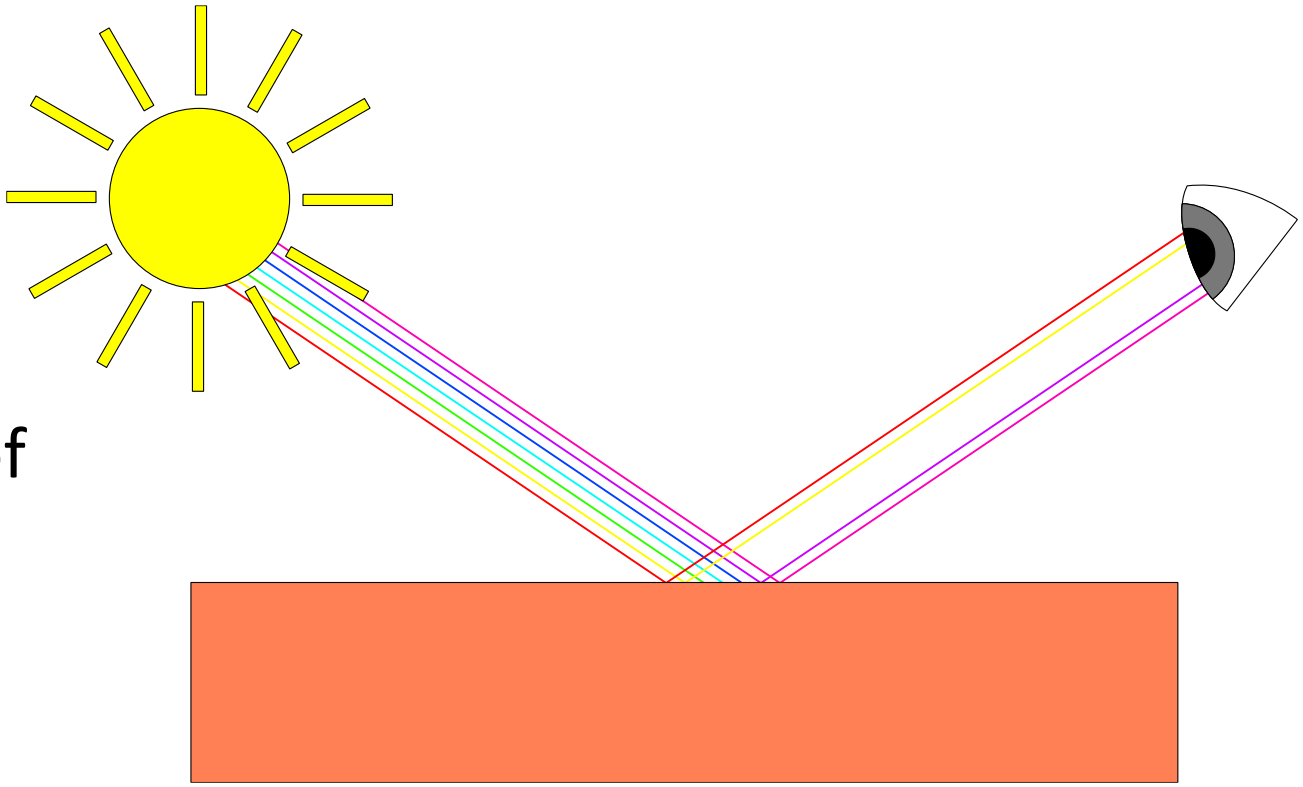
Introduction

- Real world: objects can have any known colors
- Digital world: need to map the (infinite) real colors to (limited) digital values → not all real-world colors can be represented digitally
- Can represent so many colors that we probably won't notice the difference
- Colors are digitally represented using a red, green and blue component (RGB) → combination represents almost any color:

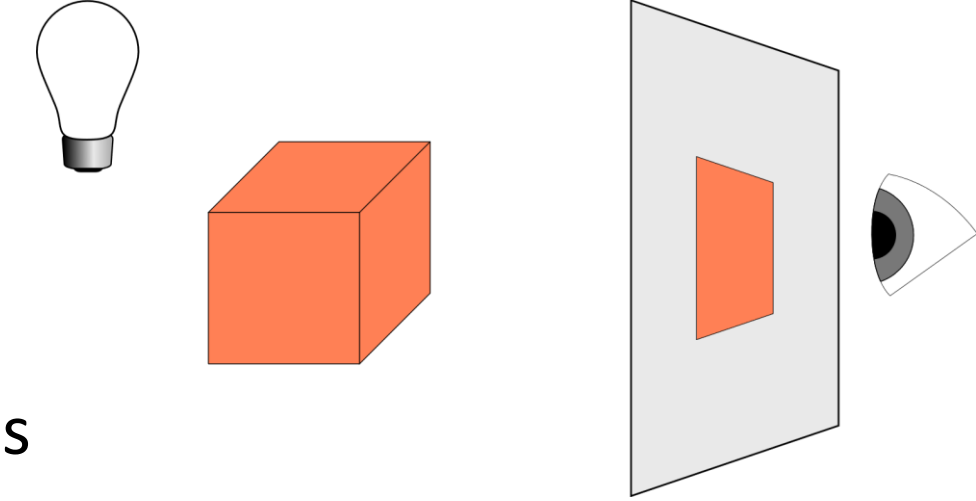
```
glm::vec3 coral(1.0f, 0.5f, 0.31f);
```

Introduction

- Colors in real life are not the colors the objects have, but the colors that are reflected
- The light of the sun is perceived as a white light (combined sum of many different colors)
- Shine the white light on a blue box absorbs all the white color's sub-colors except blue (right example of coral box)



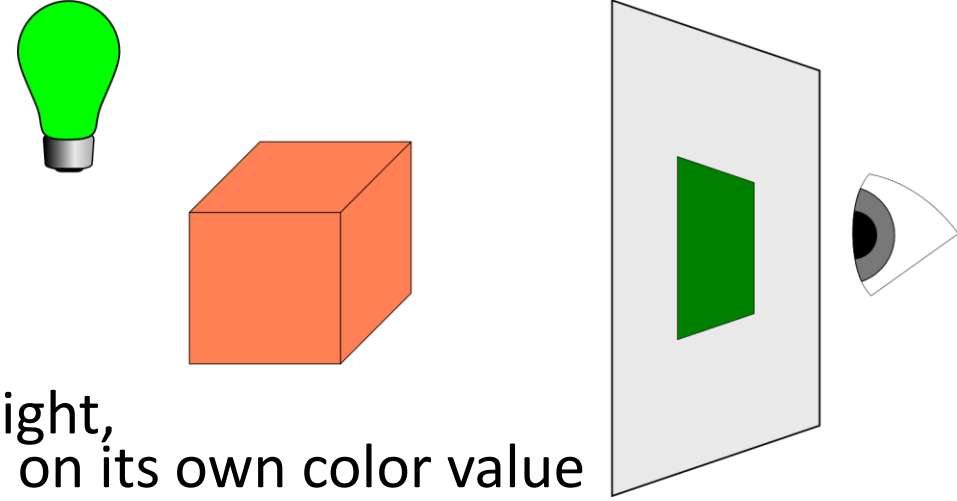
Introduction



- The reflection rule applies directly in graphics
- When defining a light source in OpenGL, it needs a color
- Previously, a white color was used, so give the light source a white color
- Multiplying the light source's color with an object's color value, results in a reflected color of the object (its perceived color)
- Apply this to the box:

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

Introduction

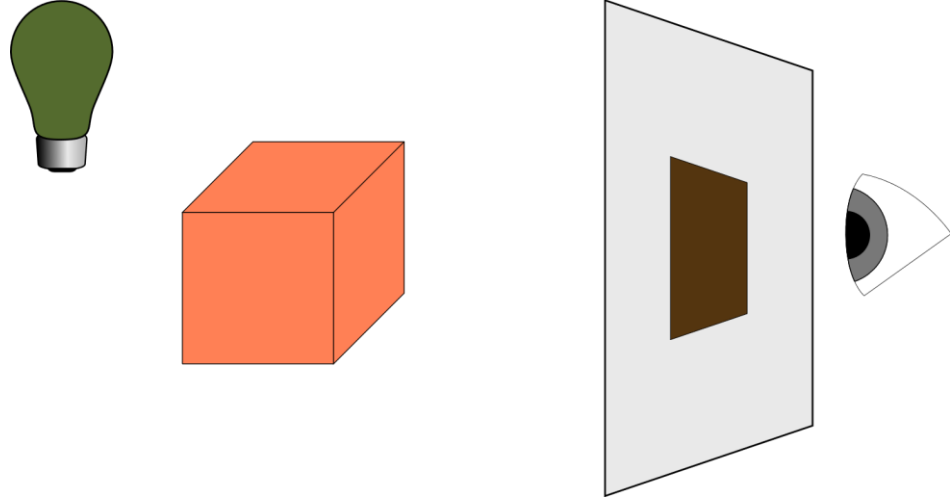


- The box's color absorbs a large portion of the white light, but reflects several red, green and blue values based on its own color value
- This is a representation of how colors would work in real life
- Now what would happen if we used a green light?

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

- The box had no red and blue light to absorb and/or reflect, it also absorbs half of the light's green value, but also still reflects half of the light's green value → perceived color is dark-greenish
- Using a green light, only the green color components can be reflected and thus perceived

Introduction



- Example with a dark olive-green light:

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

- Can get unexpected colors from objects by using different light colors

A Lighting Scene

- We want to use light sources and display them as visual objects in the scene and add at least one object to simulate the lighting on
- Need an object to cast the light on: cube from previous lectures
- Also need a light object to show where the light source is located (for simplicity use a cube, too)

A Lighting Scene

- Again, a vertex shader is needed to draw the box
- Vertex positions remain the same (do not need texture coordinates):

```
#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

- For creating a lamp cube, generate a new lamp VAO (could use the same, but for later purposes, we use a new one):

```
unsigned int lightCubeVAO;
glGenVertexArrays(1, &lightCubeVAO);
glBindVertexArray(lightCubeVAO);
// we only need to bind to the VBO, the container's VBO's data already
// contains the correct 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);
```

A Lighting Scene

- The box and the lamp cube is created, now define the fragment shader
- The light's color is multiplied with the object's (reflected) color:

```
#version 330 core
out vec4 FragColor;

uniform vec3 objectColor;
uniform vec3 lightColor;

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

A Lighting Scene

- The fragment shader accepts both an object color and a light color from a uniform variable:

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

A Lighting Scene

- When we start to change the vertex and fragment shaders, the lamp cube will unwantedly change as well
- Lamp object's color should be kept (constant bright color)
- → Need a second set of shaders that for the lamp, thus being safe from any changes to the lighting shaders
- Vertex shader is the same, fragment shader (bright):

```
#version 330 core
out vec4 FragColor;

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

A Lighting Scene

- Drawing objects (e.g., the box) → lighting shader; Drawing the lamp →lamp's shaders
- Gradually update the shaders to slowly achieve more realistic results
- The main purpose of the lamp cube is to show where the light comes from
- Usually have to define a light source's position somewhere in the scene, but this is simply a position that has no visual meaning (draw the lamp cube at the same location of the light source):

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

A Lighting Scene

- Translate the lamp's cube to the light source's position and scale it down (make it less dominant) before drawing it:

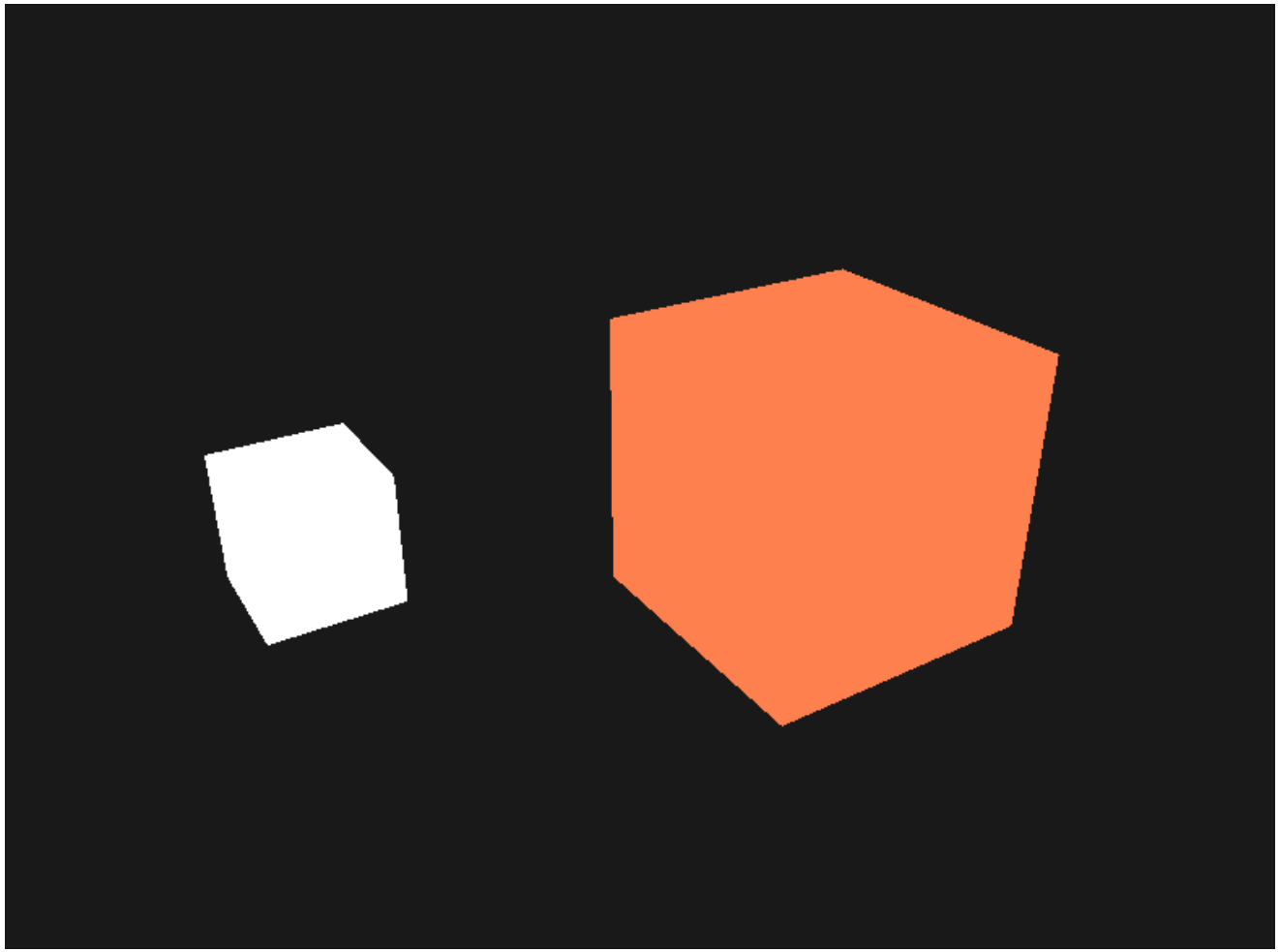
```
model = glm::mat4(1.0f);
model = glm::translate(model, lightPos);
model = glm::scale(model, glm::vec3(0.2f)); // a smaller cube
```

- The resulting drawing code for the lamp should then look something like this:

```
lightingShader.use();
…
// render the cube
glBindVertexArray(cubeVAO);
glDrawArrays(GL_TRIANGLES, 0, 36);
```

F5…

- … two cubes



Basic Lighting

Introduction

- Lighting in the real world is extremely complicated (depends on too many factors → can't afford to calculate)
- Lighting in OpenGL is therefore based on approximations using simplified models → easier to process and look relatively similar
- Lighting models based on the physics of light one of those models is called the Phong lighting model

Introduction

- Phong model consist of 3 components (fltr): ambient, diffuse and specular lighting
- Rightmost image is the (combined) result

Introduction

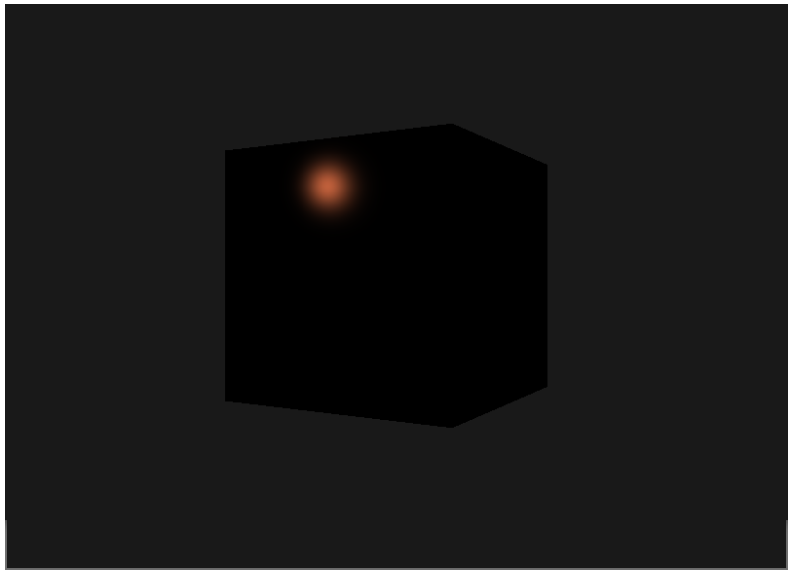
- Ambient lighting:
- Even when it is dark, usually still some light somewhere in the world (the moon, a distant light)
- Objects almost never completely dark
- To simulate this, an ambient lighting constant is used

Introduction

- Diffuse lighting:
- Simulates the directional light impact
- This is the most visually significant component of the lighting model
- The more a part of an object faces the light source, the brighter it becomes

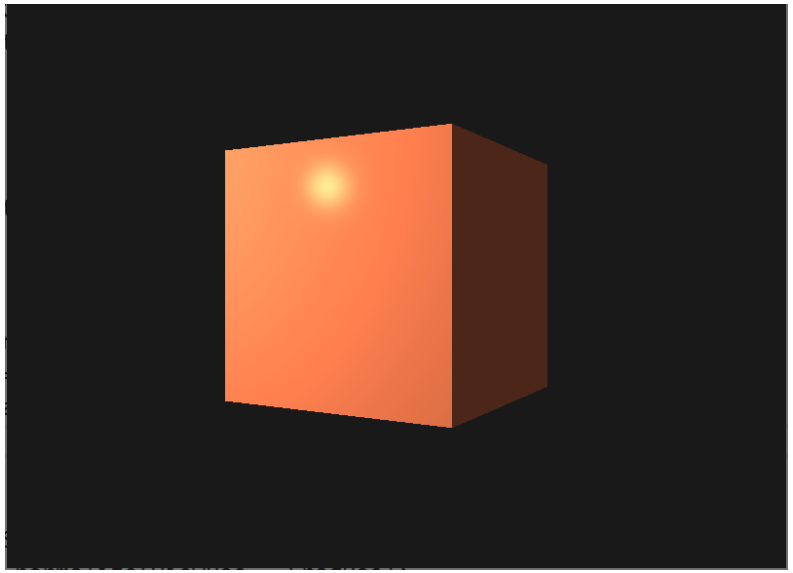
Introduction

- Specular lighting:
- Simulates bright spot of a light appearing on shiny objects
- Specular highlights are often more inclined to the color of the light than the color of the object



Introduction

- Phong model combines all components



Ambient Lighting

- Light usually comes from many light sources scattered all around
- Light can scatter and bounce (reflect from other surfaces) in many directions → reach spots that aren't in its direct vicinity
- Algorithms that consider this are called global illumination algorithms (too expensive and/or complicated)

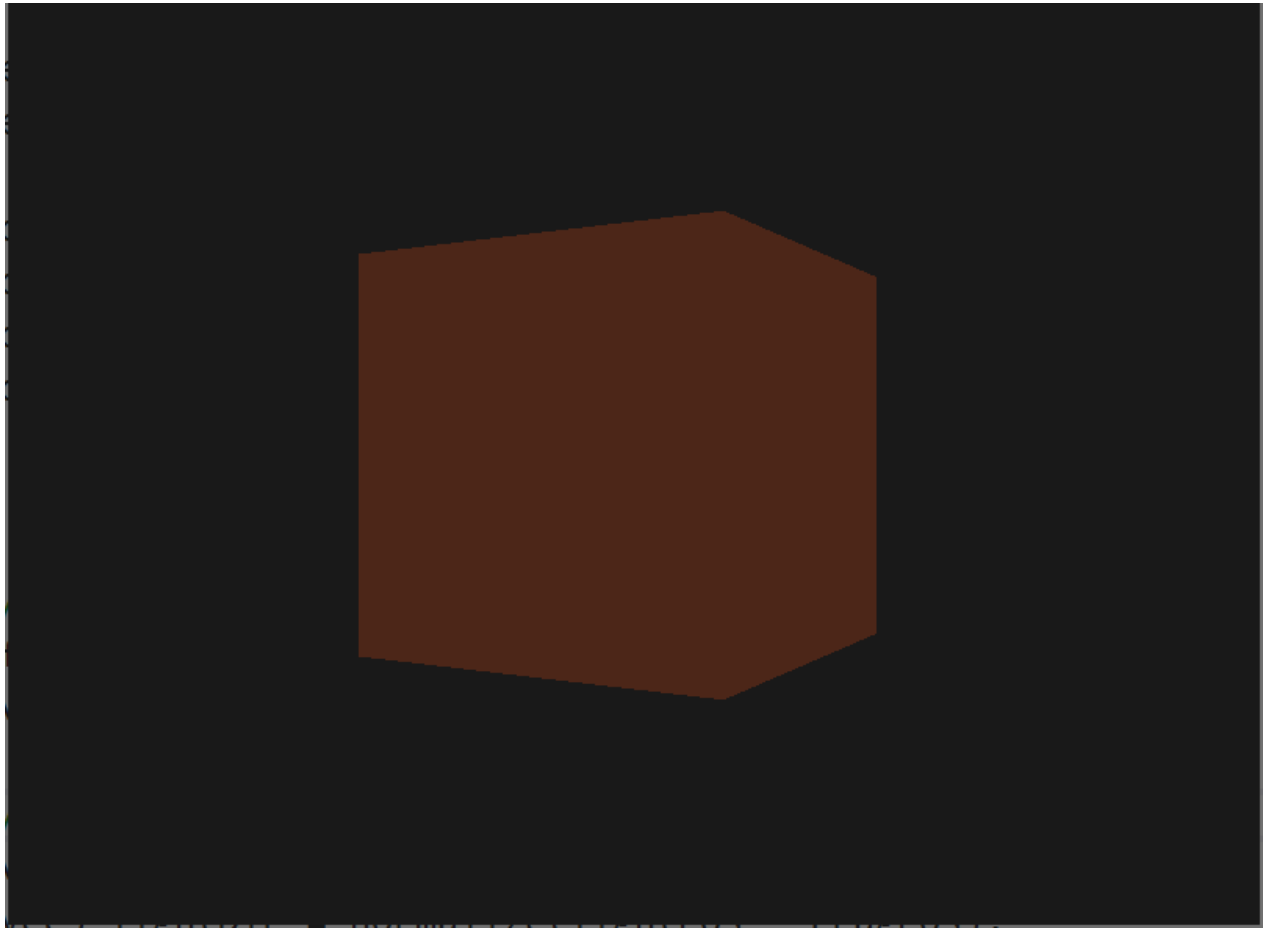
Ambient Lighting

- Start by using a very simplistic model: ambient lighting
- To pretend scattered light, a small constant (light) color is used (add to the final color)
- Take the light's color, multiply it with a small constant ambient factor, multiply this with the object's color and use it as the fragment's color:

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

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

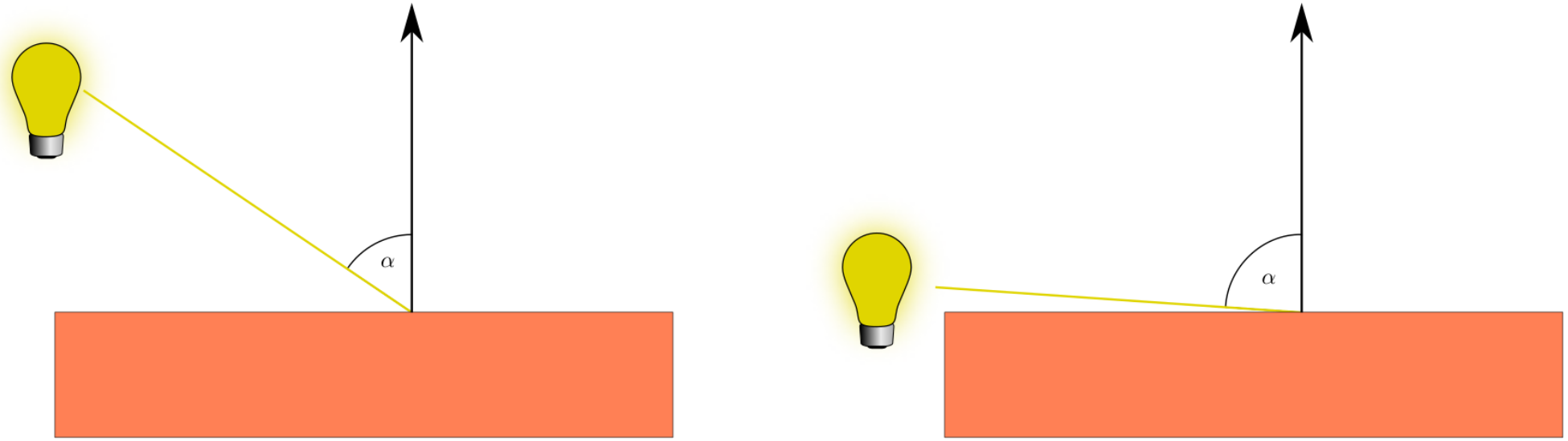
F5…

- … remember?



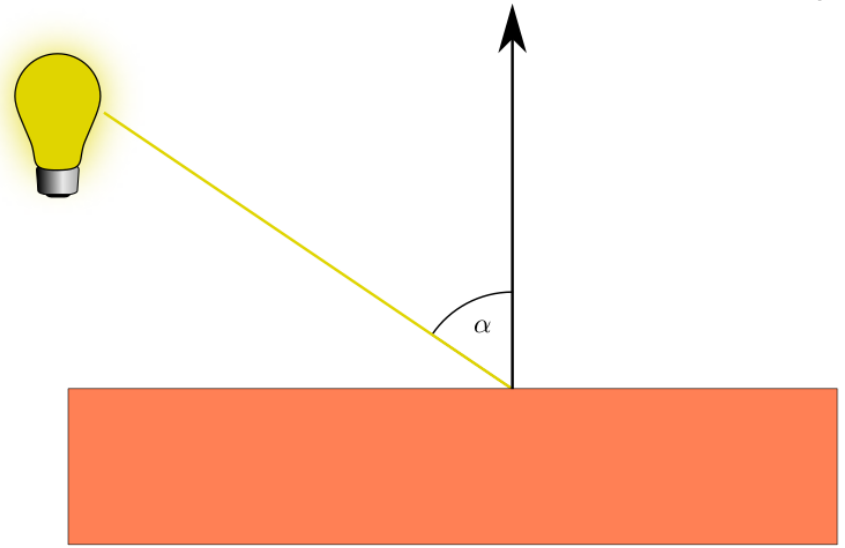
Diffuse Lighting

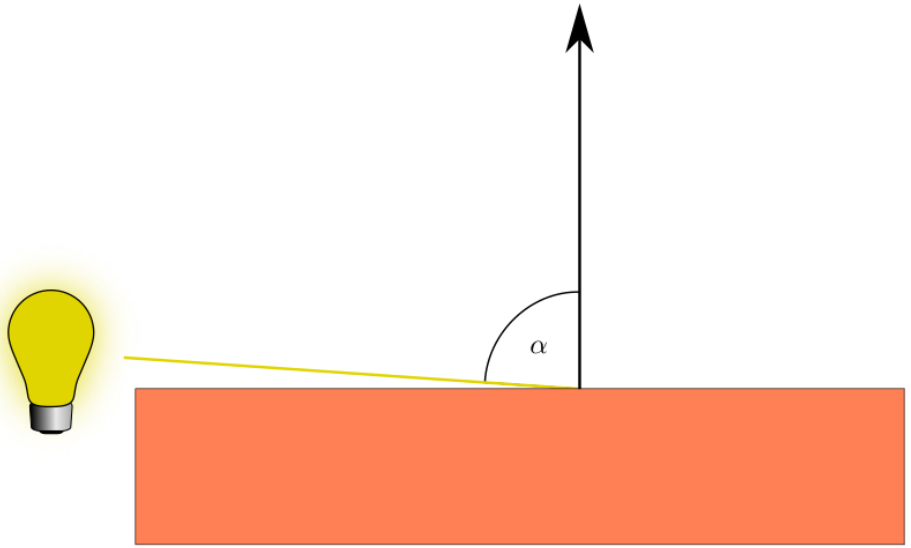
- Ambient lighting by itself is not very appealing
- Diffuse lighting improves the visual impact on the object
- It gives the object more brightness the closer its fragments are aligned to the light rays from a light source:



Diffuse Lighting

- A light source's ray targeted at a single fragment
- If the light ray is parallel to the object's normal → light has the greatest impact
- Intensity depends on the angle → angle between the two vectors can be calculated with the dot product





Diffuse Lighting

The cosine of the angle between two unit vectors is the dot product:

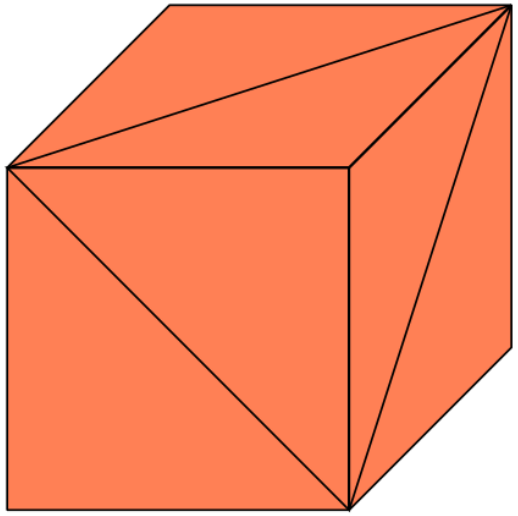
$$v \cdot w = |v| \cdot |w| \cdot \cos(\alpha) \overset{|v|=|w|=1}{=} \cos(\alpha)$$

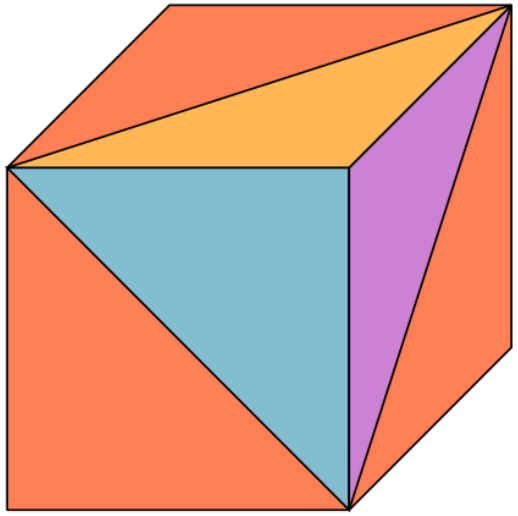
Diffuse Lighting

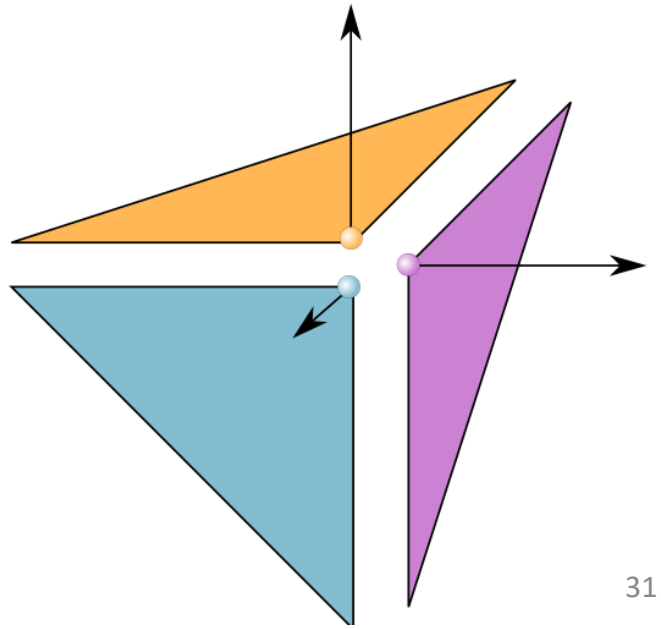
- To calculate diffuse lighting, we need:
  - Normal vector (perpendicular to the surface)
  - Directed light ray (light's position minus fragment's position)

Normal Vector

- A normal vector is a (unit) vector that is perpendicular to the surface
- Have to define normal vectors for each vertex
- For each vertex, we assign the normal of the incident triangle (can be calculated by the cross product)







Normal Vector

- First three coordinates are the vertex‘s position, then we have the normal vector

```
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};
```

Normal Vector

- Update the lighting's vertex shader:

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

- Update the vertex attribute pointers as well
- Note that the lamp object uses the same vertex array (not using the newly added normal vectors, but need to modify the vertex attribute pointers to reflect the new vertex array's size):

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

Normal Vector

It may look inefficient using vertex data that is not completely used by the lamp shader, but the vertex data is already stored in the GPU's memory from the container object

→ More efficient compared to allocating a new VBO specifically for the lamp.

Normal Vector

- Lighting calculations performed in the fragment shader
- Forward the normal vectors from the vertex to the fragment shader

```
out vec3 Normal;

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

- Declare the corresponding input variable in the fragment shader:

```
in vec3 Normal;
```

Calculating the Diffuse Color

- Still need the light's and the fragment's position vector
- Light's position is just a single static variable → uniform:

```
uniform vec3 lightPos;
```

- Update the uniform (outside the game loop since it doesn't change):

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

Calculating the Diffuse Color

- Now, the actual fragment's position is needed
- Lighting calculations are applied in world space → need a vertex position in world space:

```
out vec3 FragPos;
out vec3 Normal;

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

Calculating the Diffuse Color

- Add the input variable to the fragment shader:

```
in vec3 FragPos;
```

- 1. Calculate the direction vector between the light and the fragment's position
- Make sure all the relevant vectors end up as unit vectors

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

Calculating the Diffuse Color

When doing lighting calculations, make sure you always normalize the relevant vectors to ensure they're actual unit vectors.

Forgetting to normalize a vector is a popular mistake.

Calculating the Diffuse Color

- 2. Calculate the actual diffuse impact the light by taking the dot product of the norm and lightDir vector
- The resulting value is then multiplied with the light's color:

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

- If $\alpha > 90$ then the dot product will become negative → negative diffuse component
- Therefore, use the max function that returns the highest of both its parameters → never negative

Calculating the Diffuse Color

- Now add the ambient and diffuse color and then multiply the result with the color of the object:

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

F5…

- … remember?

One last thing

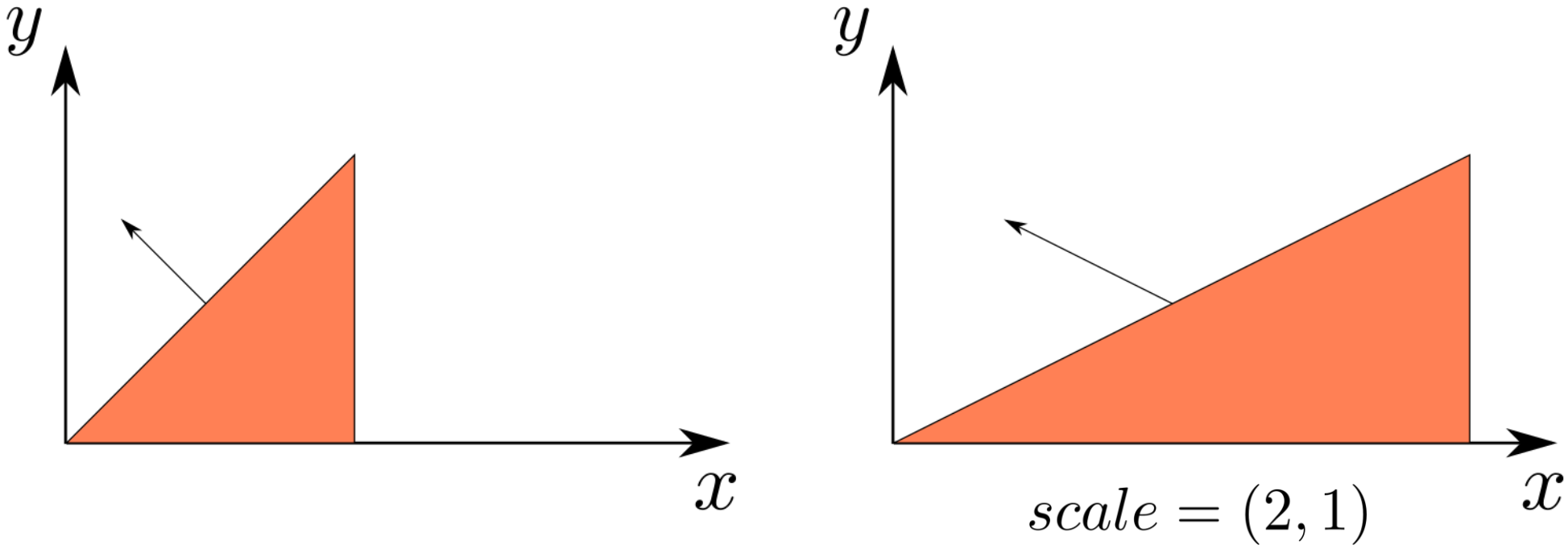
- Currently, the normal vectors are directly passed from the vertex shader to the fragment shader
- Calculations in the fragment shader are all done in world space
- The questions raises: Shouldn’t we transform the normal vectors to world space coordinates as well?
- Basically yes, but it’s not as simple as simply multiplying it with a model matrix

One last thing

- Normal vectors are only direction vectors without a homogeneous coordinate (w component) → Translations do and should not have any effect on the normal vectors
- When multiplying normal vectors with a model matrix, translation part should be removed (only take the upper-left 3x3 matrix or set the w component of a normal vector to 0)
- The only transformations to be applied to normal vectors are scale and rotation transformations

One last thing

- If the model matrix would perform a non-uniform scale → the normal vector may be not perpendicular to the surface anymore
- So applying such a model matrix will not work



One last thing

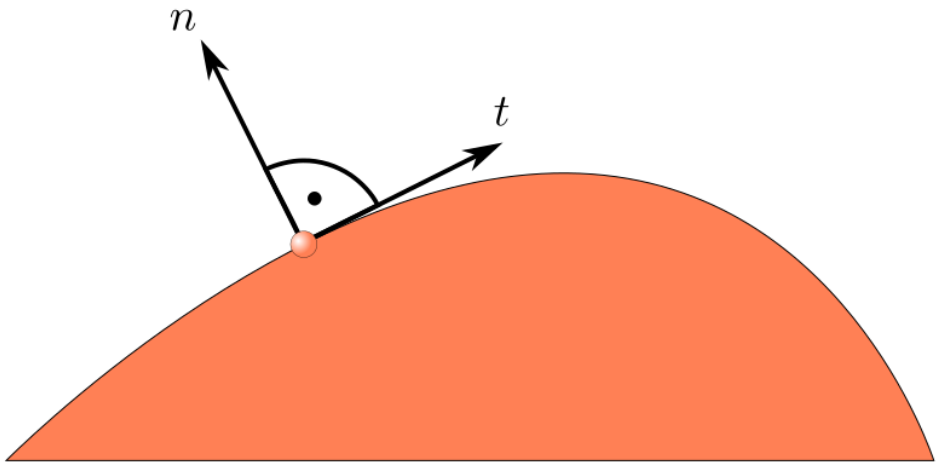
- Applying a non-uniform scale (uniform scales won't hurt the normal after normalizing them) the normal vectors are not perpendicular to the surface anymore → distorts the lighting
- Therefore, we use the normal matrix

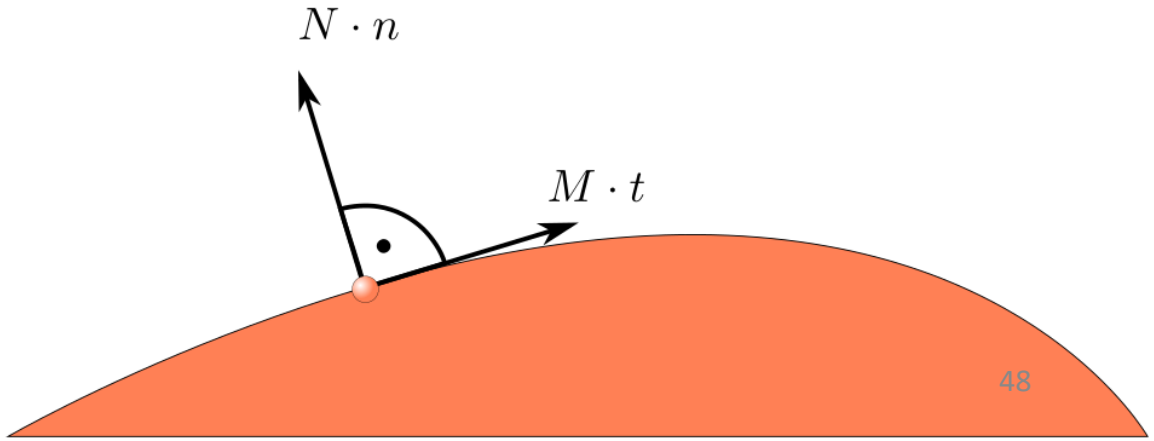
Normal Matrix*

Normal Matrix

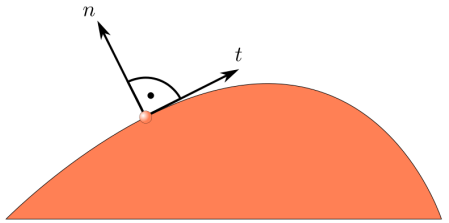
- Assume we have a surface and a normal vector n, with the normal vector we can also find a tangent vector t
- After applying the model matrix M to the tangent vector $M \cdot t$, we have to find a normal matrix N such that:

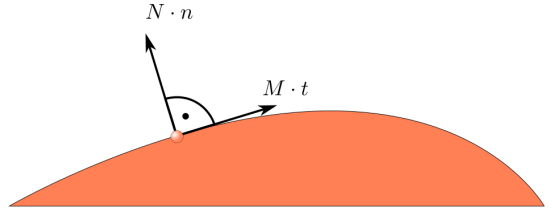
$$\langle t, n \rangle = \langle M \cdot t, N \cdot n \rangle = 0$$





Normal Matrix

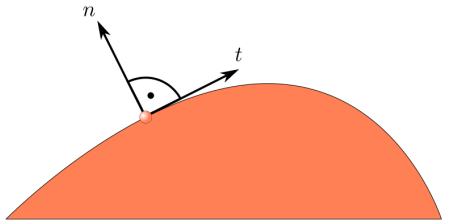


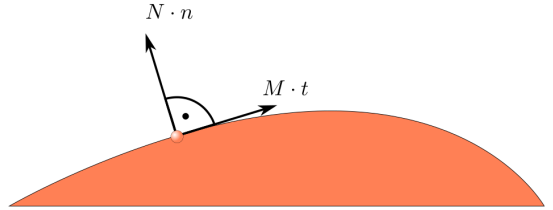


- Note, that we only consider the upper-left 3x3 matrix of the model matrix as M
- Then, we get:

$$
\begin{aligned}
0 &= \langle M \cdot t, N \cdot n \rangle \\
&= (M \cdot t)^T \cdot (N \cdot n) \\
&= t^T \cdot M^T \cdot N \cdot n \\
&= t^T (M^T N) n
\end{aligned}
$$

Normal Matrix





- If $Id = M^T N$, then we get $t^T n$, which is 0:

$$M^T N = Id$$

$$N = (M^T)^{-1}$$

- Note, M consists of rotations and scaling, thus, the inverse exists
- Property: $(A^T)^{-1} = (A^{-1})^T$

Normal Matrix

- Let‘s go back…

One last thing

- In the vertex shader, this normal matrix can be calculated:

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

- In the diffuse lighting section the lighting was correct (did not use any scaling)
- For a non-uniform scale, it is essential to multiply the normal vector with the normal matrix

One last thing

Inversing matrices is a costly operation even for shaders.

Try to avoid doing inverse operations in shaders (have to be done on each vertex of the scene).

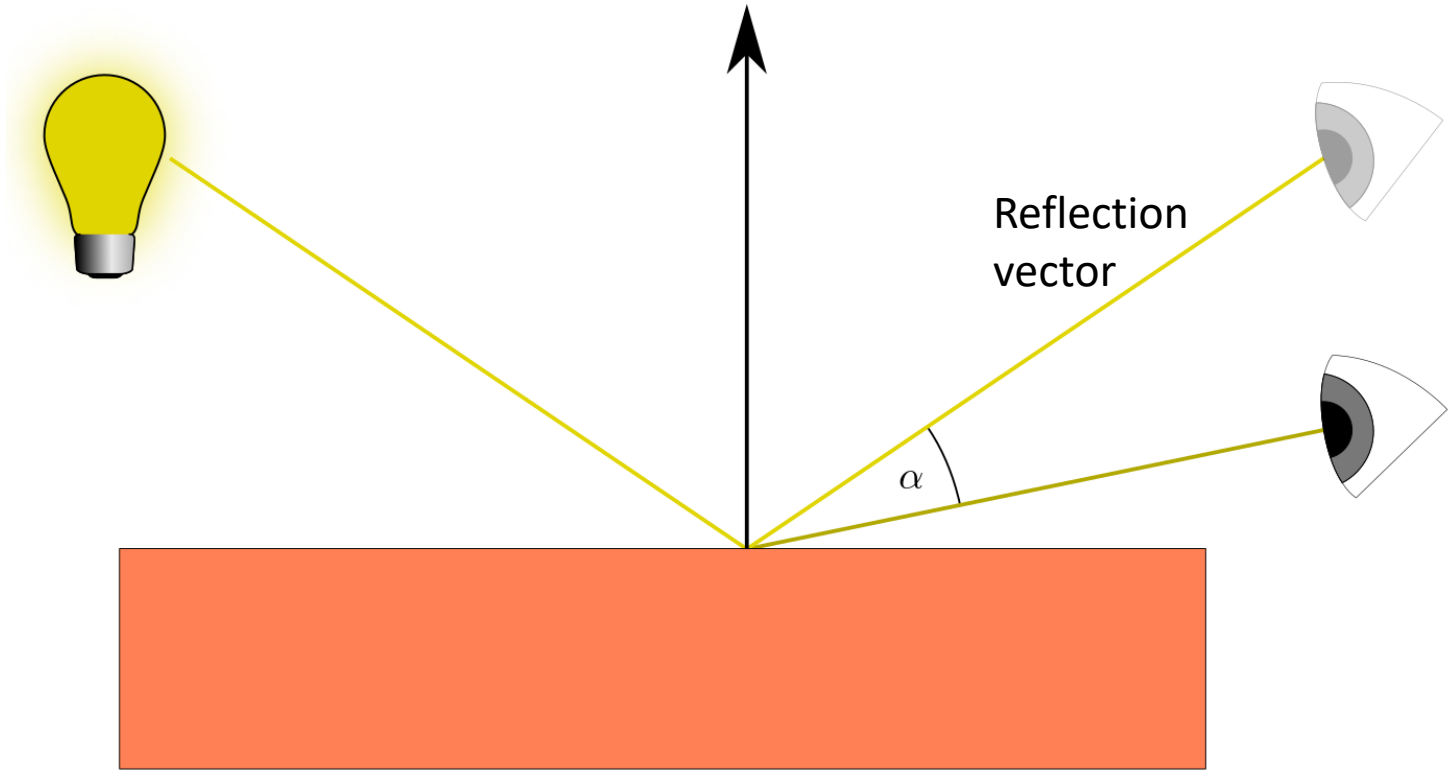
For an efficient application calculate the normal matrix on the CPU and send it to the shaders via a uniform before drawing (just like the model matrix).

Specular Lighting

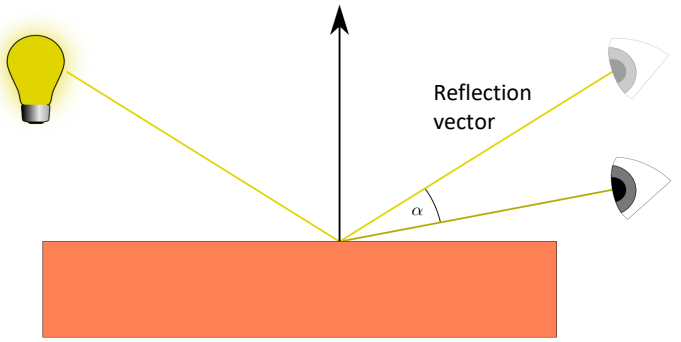
- Like diffuse lighting, specular lighting is based on the light's direction and the object's normal
- Now, it is also based on the view direction (from what direction the camera is looking at the fragment)

Specular Lighting

- Specular lighting is based on the reflective properties of light
- Think of the object's surface as a mirror, the specular lighting the light reflected on the surface:



Specular Lighting



- Reflection vector: reflecting the light direction around the normal vector
- Then, calculate the angular distance the reflection vector and the view direction
- The smaller the angle the greater the specular light (highlight)
- The view vector can be calculated using the viewer's world space position and the fragment's position
- Then, determine the specular's intensity, multiply this with the light color and add this to the resulting ambient and diffuse components

Specular Lighting

- **We do the lighting calculations in world space, but mostly it is done in view space.**
- **The advantage (view space) is that the viewer's position is always at (0,0,0) → got the position of the viewer for free.**
- **Lighting in world space is more intuitive for learning purposes.**
- **To calculate lighting in view space, transform all the relevant vectors with the view matrix as well (don't forget to change the normal matrix too).**

Specular Lighting

- For the world space coordinates of the viewer add another uniform to the fragment shader and pass the corresponding camera position vector to the fragment shader:

```
uniform vec3 viewPos;
```

- Send it to the shader:

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

Specular Lighting

- To calculate the specular lighting, a specular intensity value must be defined (specular highlight a medium-bright color so that it doesn't have too much of an impact):

```
float specularStrength = 0.5;
```

- Setting this to 1.0f would result in a too bright specular highlight
- Next, calculate the view direction and the reflection vector along the normal:

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

Specular Lighting

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

- The reflect function expects the first vector to point from the light source towards the fragment's position (currently reversed → negative)
- The second argument expects a normal vector

Specular Lighting

- Specular component:

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

- First calculate the dot product between the view direction and the reflect direction (and make sure it's not negative) and then raise it to the power of 32
- This 32 value is the shininess value of the highlight
- The higher the shininess value of an object, the more it properly reflects the light instead of scattering it all around and thus the smaller the highlight becomes

Shininess Values (2,4,16,32,64,128)

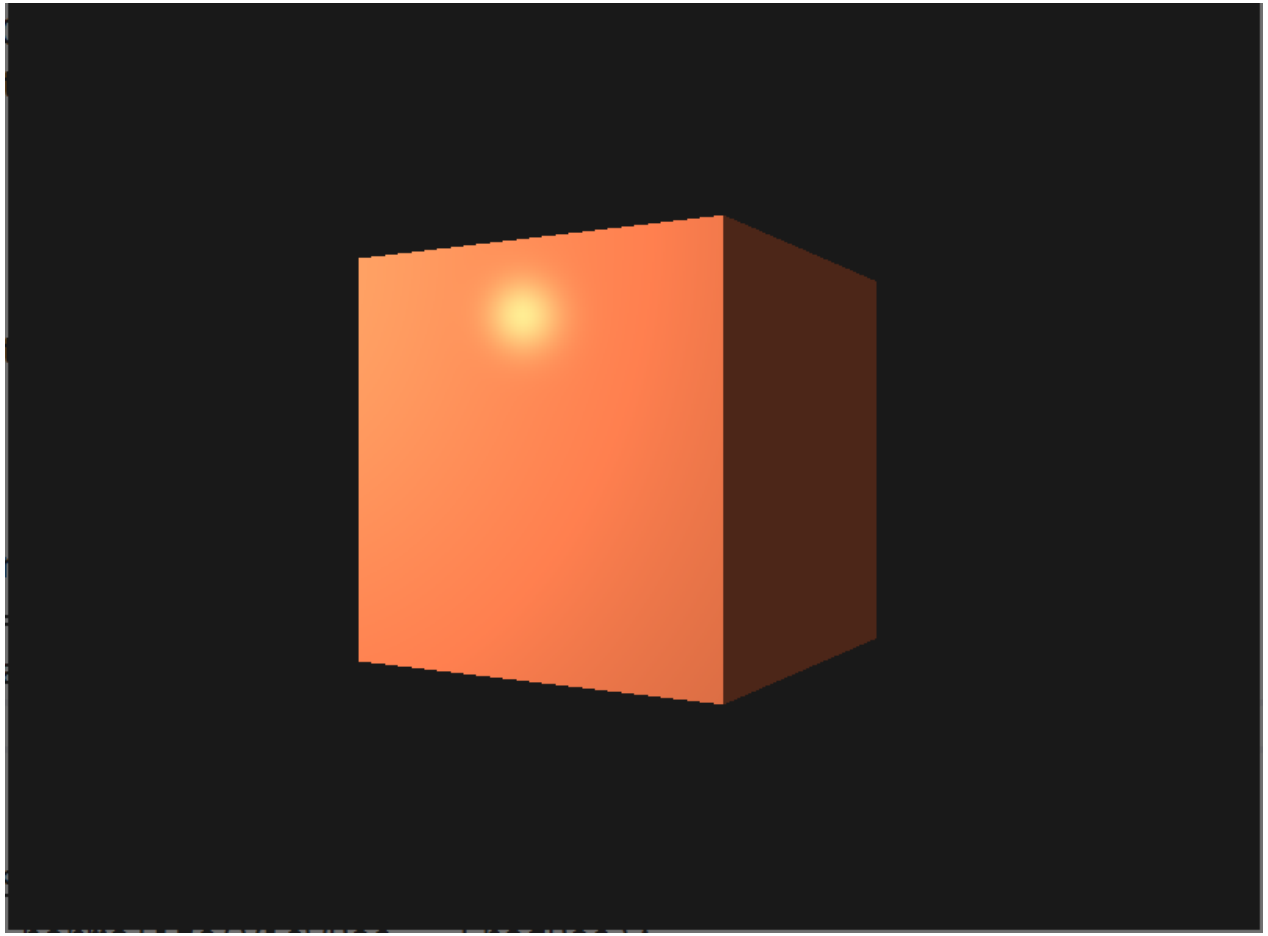
Specular Lighting

- The only thing left to do is to add it to the ambient and diffuse components and multiply the combined result with the object's color:

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

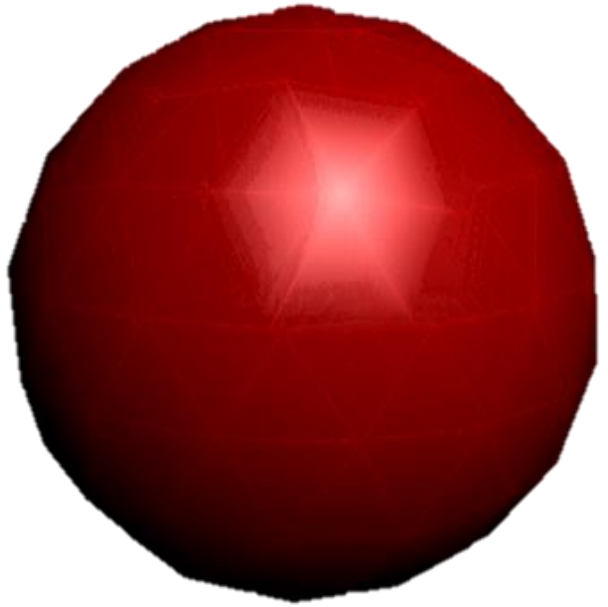
F5…

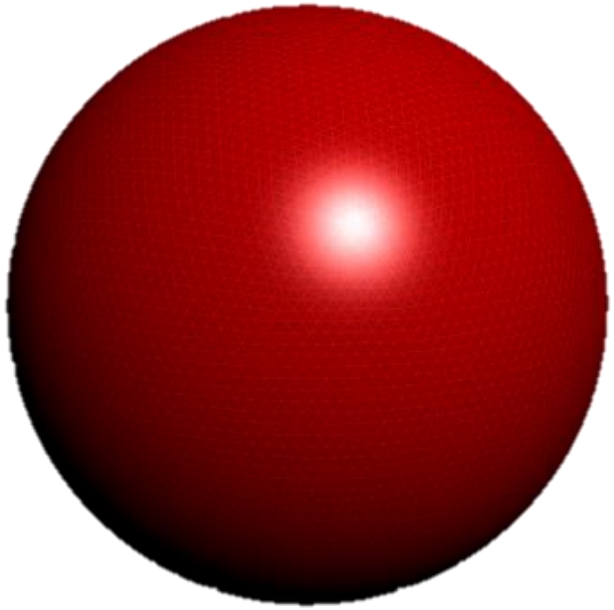
- … remember?



Gouraud Lighting

- In the past, developers implement the Phong lighting model in the vertex shader → more efficient (less vertices than fragments, so the (expensive) lighting calculations are done less frequently)
- The resulting color value in the vertex shader is the resulting lighting color of that vertex only and the color values of the surrounding fragments are then the result of interpolated lighting colors
- The result was not very realistic unless large amounts of vertices were used
- Phong lighting model is implemented in the vertex shader it is called Gouraud shading





https://en.wikipedia.org/wiki/Gouraud_shading

Materials

Introduction

- In the real world, each object reacts differently to light
- Steel objects are often shinier than a clay vase and a wooden object
- Each object also responds differently to specular highlights:
  - Some reflect the light without too much scattering (small highlights)
  - Others scatter a lot giving the highlight a larger radius
- To simulate several types of objects in OpenGL, material properties needs to be defined to specific each object

Introduction

- When describing objects, a material color should be defined for each of the 3 lighting components: ambient, diffuse and specular
- This gives fine-grained control over the color output of the object.
- Now add a shininess component to those 3 colors (fragment s.):

```
#version 330 core

struct Material {
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float shininess;
};

uniform Material material;
```

Introduction

```
struct Material {
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float shininess;
};
```

- In the fragment shader, create a struct to store the material properties of the object (uniform values are also possible, but as a struct it is more organized)
- First, define the layout of the struct, then simply declare a uniform variable with the created struct as its type

Introduction

```
struct Material {
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float shininess;
};
```

- Here, define a color vector for each of the Phong lighting's components
- The ambient vector usually the same color as the object's color
- The diffuse vector defines the color of the object under diffuse lighting
- The diffuse color is (just like ambient lighting) set to the desired object's color
- The specular vector sets the color impact a specular light
- Lastly, the shininess impacts the scattering/radius of the specular highlight

Material Table

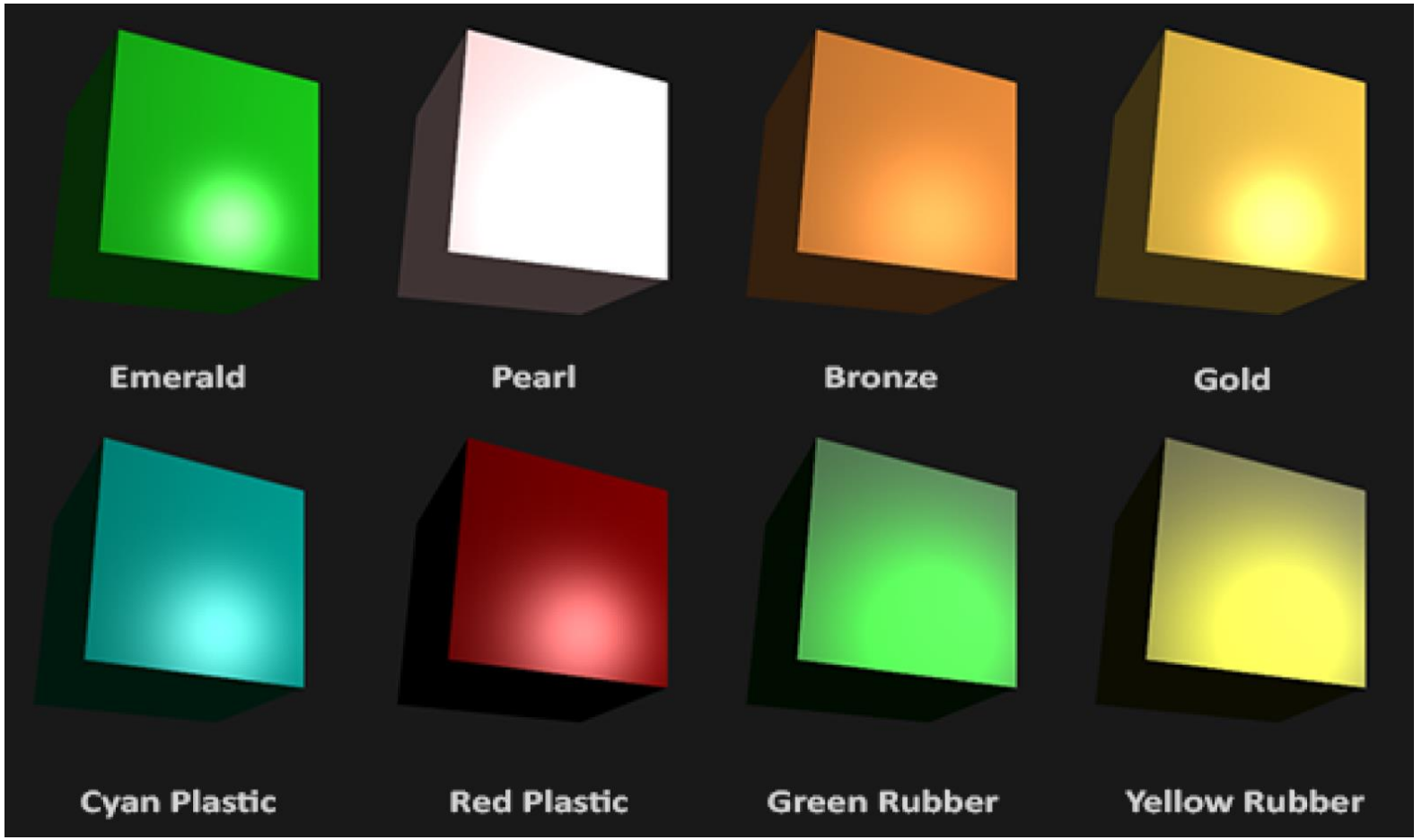
- Different values dependent on the material

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
copper	0.19125	0.0735	0.0225	0.7038	0.27048	0.0828	0.256777	0.137622	0.086014	0.1
gold	0.24725	0.1995	0.0745	0.75164	0.60648	0.22648	0.628281	0.555802	0.366065	0.4
silver	0.19225	0.19225	0.19225	0.50754	0.50754	0.50754	0.508273	0.508273	0.508273	0.4
black plastic	0.0	0.0	0.0	0.01	0.01	0.01	0.50	0.50	0.50	.25
cyan plastic	0.0	0.1	0.06	0.0	0.50980392	0.50980392	0.50196078	0.50196078	0.50196078	.25
green plastic	0.0	0.0	0.0	0.1	0.35	0.1	0.45	0.55	0.45	.25
red plastic	0.0	0.0	0.0	0.5	0.0	0.0	0.7	0.6	0.6	.25
white plastic	0.0	0.0	0.0	0.55	0.55	0.55	0.70	0.70	0.70	.25
yellow plastic	0.0	0.0	0.0	0.5	0.5	0.0	0.60	0.60	0.50	.25
black rubber	0.02	0.02	0.02	0.01	0.01	0.01	0.4	0.4	0.4	.078125
cyan rubber	0.0	0.05	0.05	0.4	0.5	0.5	0.04	0.7	0.7	.078125
green rubber	0.0	0.05	0.0	0.4	0.5	0.4	0.04	0.7	0.04	.078125
red rubber	0.05	0.0	0.0	0.5	0.4	0.4	0.7	0.04	0.04	.078125
white rubber	0.05	0.05	0.05	0.5	0.5	0.5	0.7	0.7	0.7	.078125
yellow rubber	0.05	0.05	0.0	0.5	0.5	0.4	0.7	0.7	0.04	.078125

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

Material Table

- Results of several of these real world materials



Setting Materials

- Change the lighting calculations to comply with the new material properties:

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

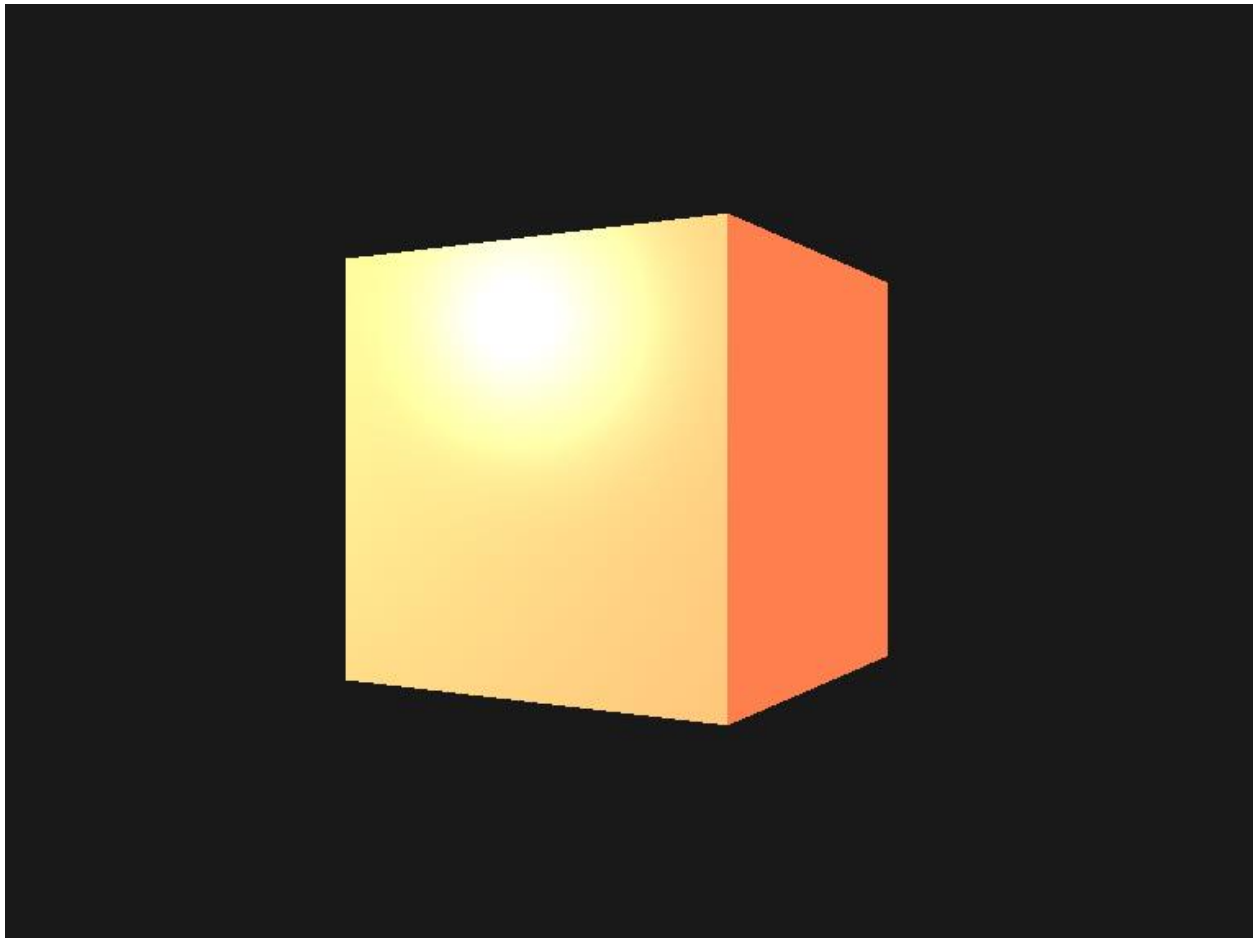
Setting Materials

- The material of the object can be set with the appropriate uniforms
- A struct in GLSL however is not special in any regard when setting uniforms
- A struct only acts as an encapsulation of uniform variables, to fill the struct individual uniforms have to be set, but this time 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);
```

F5…

- … again the cube, but it looks strange



Light Properties

- The object is way too bright → 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
- Thus, specify intensity vectors for each of the lighting components
- `lightColor` = vec3(1.0)

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

Light Properties

- Each material property of the object is returned with full intensity for each of the light's components
- Right now the ambient component of the object is fully influencing the color of the cube, but the ambient component shouldn't really have such a big impact on the final color:

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

Light Properties

- Influence the diffuse and specular intensity of the light source in the same way by using a struct:

```
struct Light {
    vec3 position;

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

Light Properties

- A light source has a different intensity for its ambient, diffuse and specular light
- Ambient light is usually set to a low intensity (not to be too dominant)
- Diffuse component of a light source is usually set to the exact color (often a bright white color)
- Specular component is usually kept at vec3(1.0) shining at full intensity

Light Properties

- Update the fragment shader:

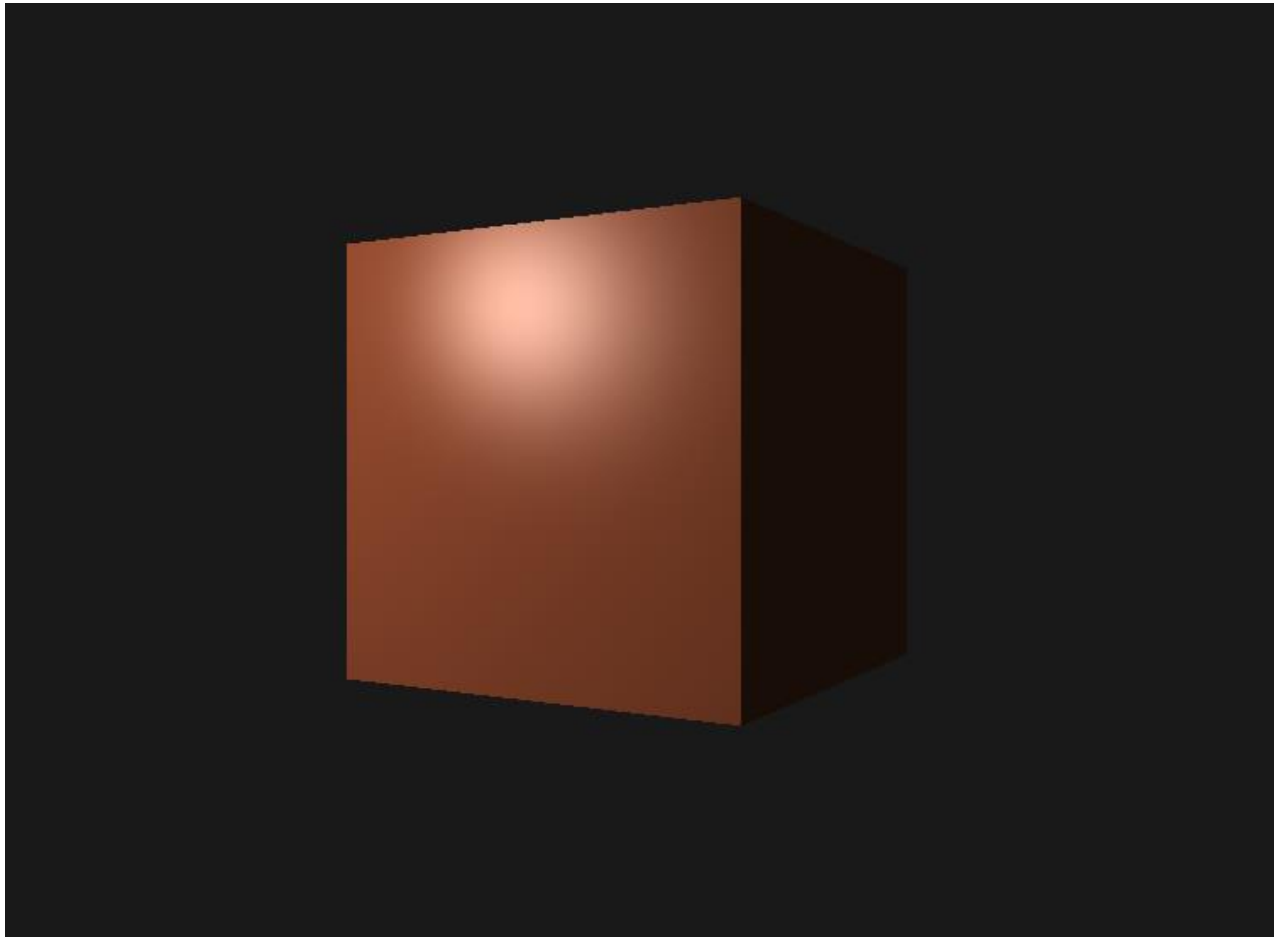
```
vec3 ambient = light.ambient * material.ambient;
vec3 diffuse = light.diffuse * (diff * material.diffuse);
vec3 specular = light.specular * (spec * material.specular);
```

- Set the light intensities in the application:

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

F5…

- … much better!



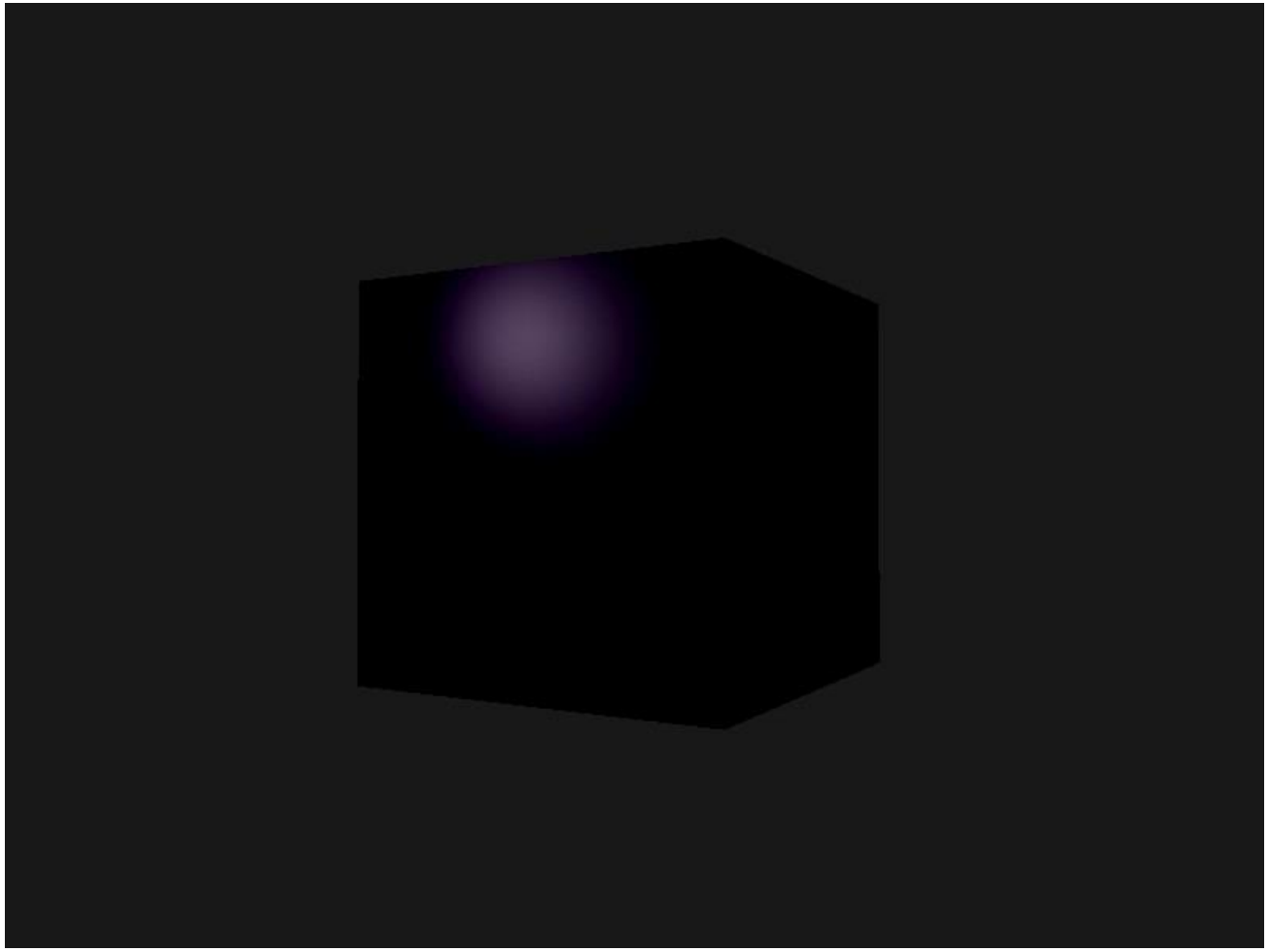
Different Light Colors

- So far, light colors range from white to gray to black
- Not affecting the actual colors of the object (only its intensity)
- Let's change their colors over time

```
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);
```

F5…

- … changing color!



Parametric Surfaces*

Introduction

- Assume, we want to generate a more interesting surface instead of a cube
- What about parametric surfaces?

Introduction

- To visualize parametric surfaces, we need:
- 1. The basics
- 2. We need a grid on the C++ site
- 3. We need index-based triangles
- 4. We need the coordinates of the vertices
- 5. We need the normal for a proper shading

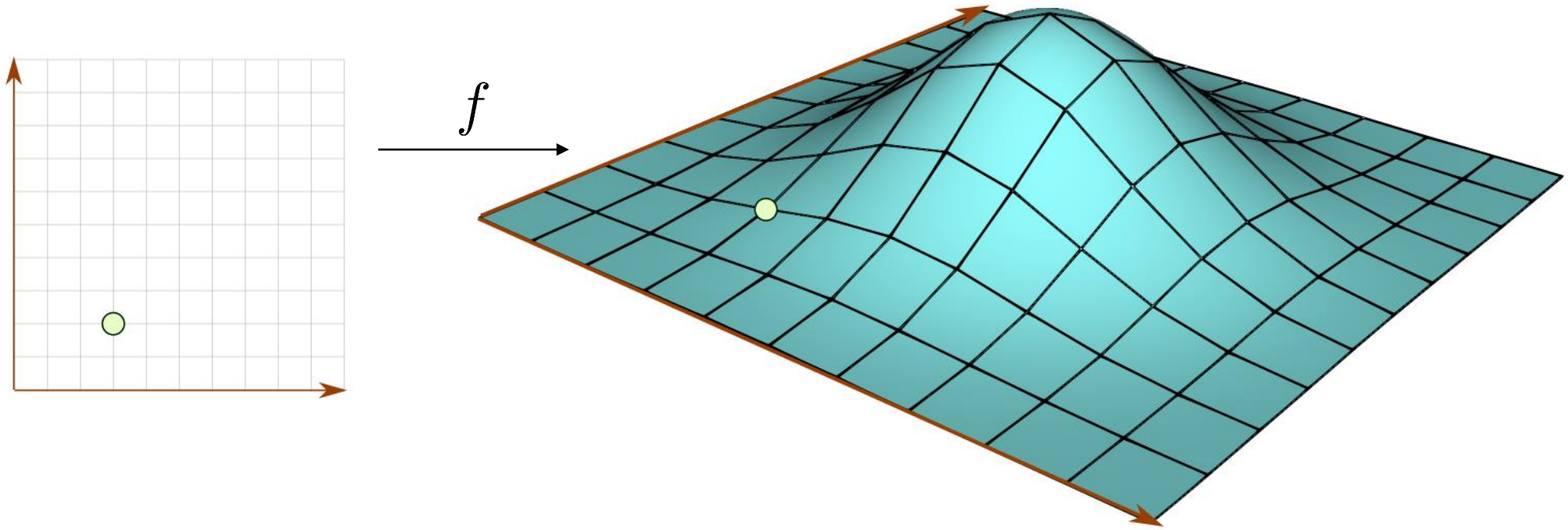
1. Basics

- Parametric representation
  - Range of a function
  - Surface patch

$$f : \mathbb{R}^2 \rightarrow \mathbb{R}^3$$

$$S_\Omega = f(x),\ x \in \Omega$$

Parametric Surface



Parametric Surface

- A surface $f : I \subset \mathbb{R}^2 \to \mathbb{R}^3$ is called a parametric surface if f is an immersion
- An immersion means that all partial derivatives $\frac{\mathrm{d}f}{\mathrm{d}x_i}$ are injective at each point (non-zero and linear independent)

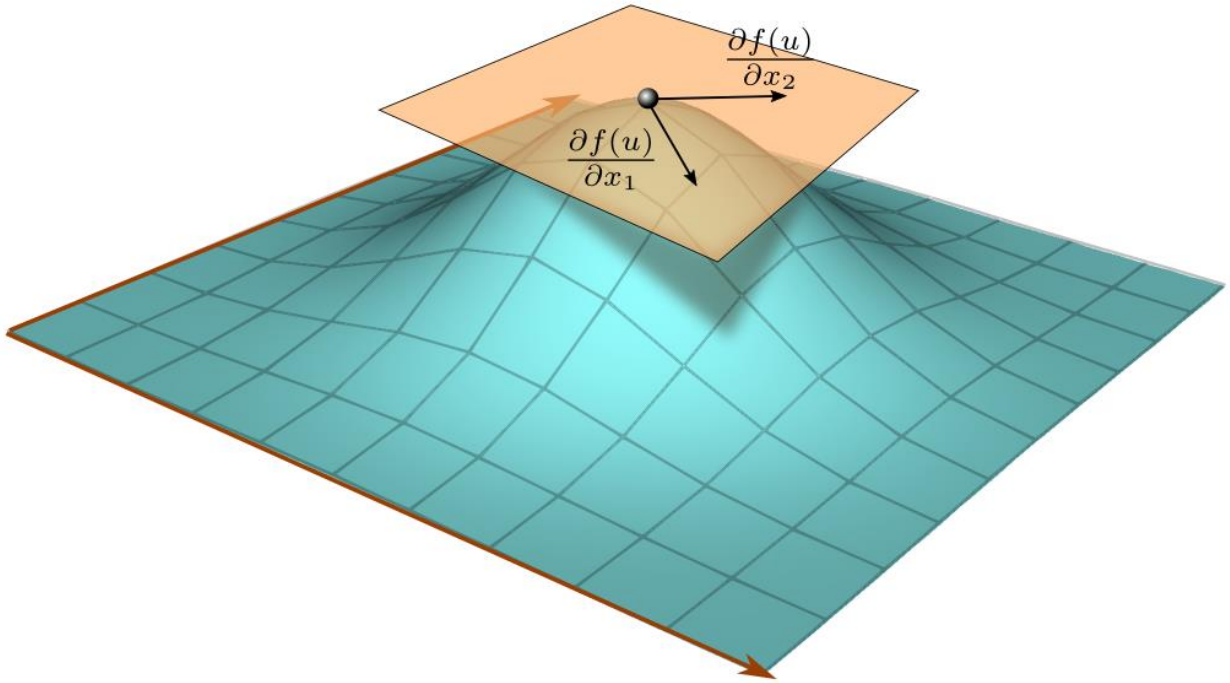
Parametric Surface

- The further calculations are mostly based on the tangent space of a surface
- The tangent space $T_p f$ of f is defined as the linear combination of the partial derivatives of f:

$$\mathcal{T}_u f = \text{span}\left\{ \frac{\partial f(u)}{\partial x_1}, \frac{\partial f(u)}{\partial x_2} \right\}$$

Parametric Surface

- At a certain point, the derivatives define the tangent vectors and the span defines the tangent space



Parametric Surface

- Given the tangent space, we can define the normal
- The normal is orthogonal to all elements in the tangent space:

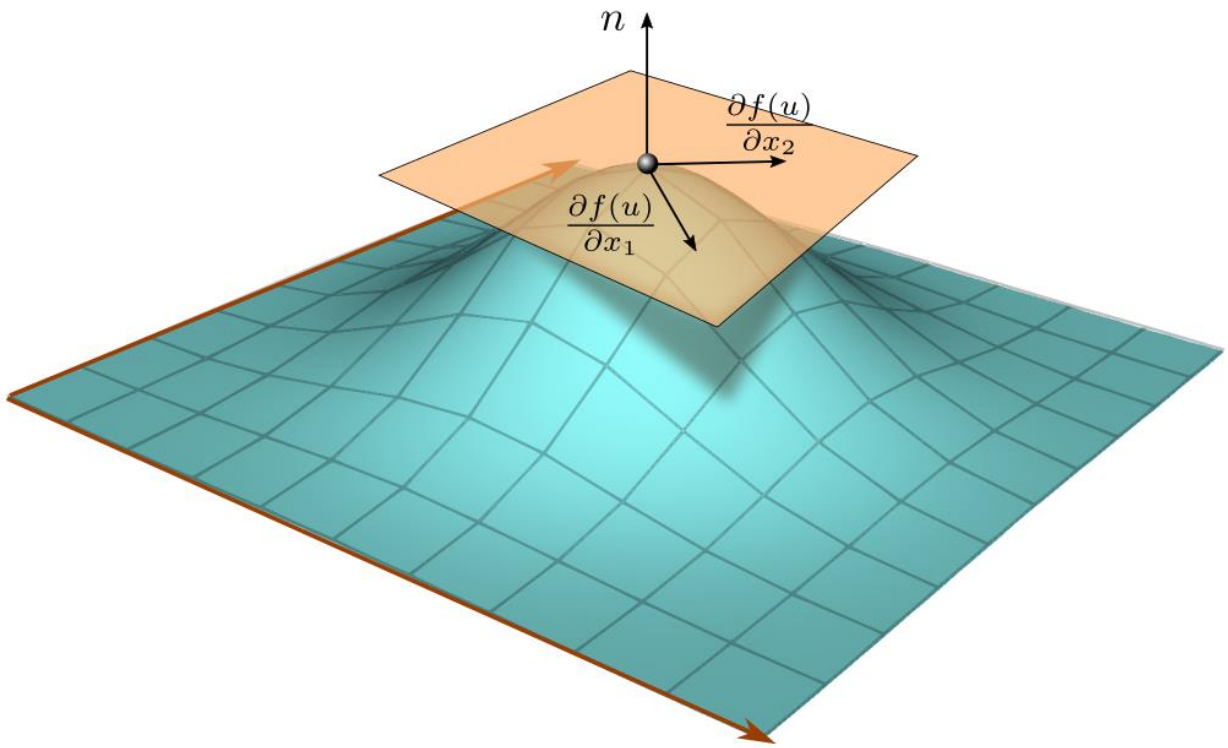
$$\langle \mathbf{n}(\mathbf{u}), \mathbf{v} \rangle = 0, \quad \mathbf{v} \in \mathcal{T}_{\mathbf{p}} f$$

- Then, orthonormal tangent is (up-to sign) uniquely defined as:

$$\mathbf{n}(u) = \pm \frac{\frac{\partial f(u)}{\partial x_1} \times \frac{\partial f(u)}{\partial x_2}}{\| \frac{\partial f(u)}{\partial x_1} \times \frac{\partial f(u)}{\partial x_2} \|}$$

Parametric Surface

- The cross product of the tangent vectors yield the normal vector



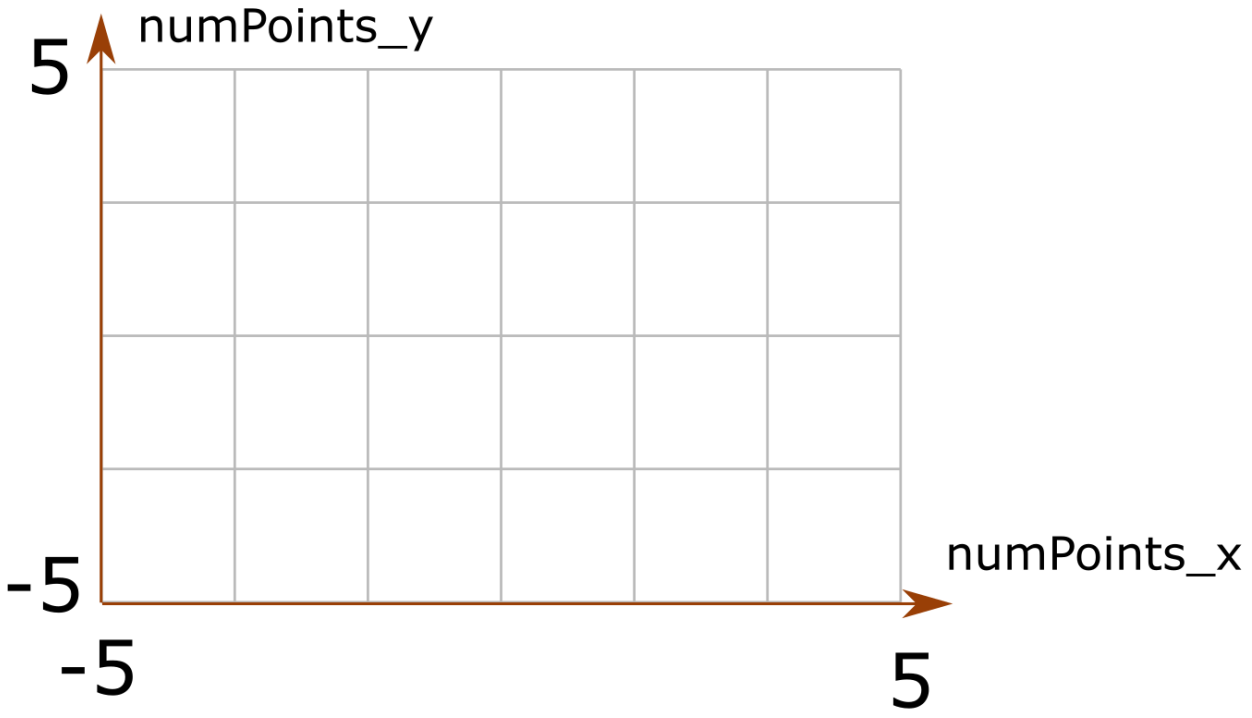
2. C++ Grid

- Let‘s define a grid and set the coordinates of the grid points:

```
const int numPoints_x = 100;
const int numPoints_y = 100;
float vertices[3 *  numPoints_x * numPoints_y] = {};

for (int i = 0; i < numPoints_x; i++)
    {
        for (int j = 0; j < numPoints_y; j++)
        {
            float x = static_cast<float>(i);
            float y = static_cast<float>(j);

            x = 10*(x / (numPoints_x - 1)-0.5); // x in [-5,5]
            y = 10*(y / (numPoints_y - 1)-0.5); // y in [-5,5]

            vertices[3 * (i * numPoints_y + j) + 0] = x;
            vertices[3 * (i * numPoints_y + j) + 1] = y;
            vertices[3 * (i * numPoints_y + j) + 2] = 0.0;
        }
    }
```

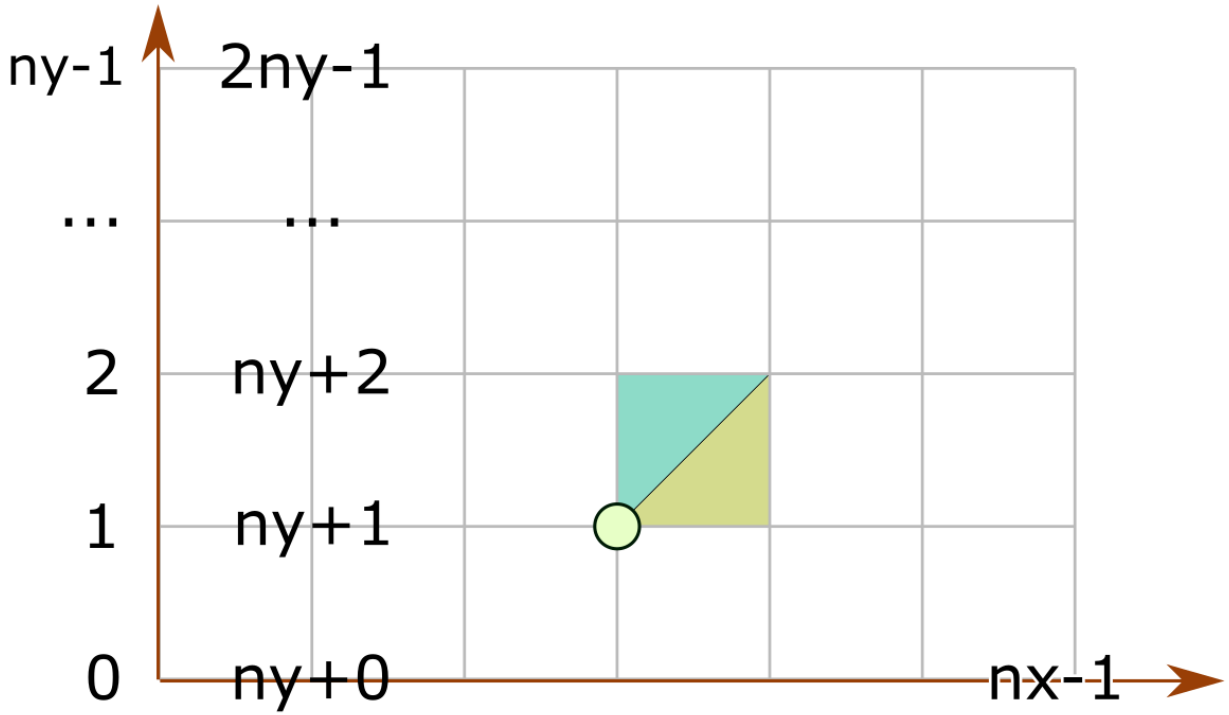
2. C++ Grid

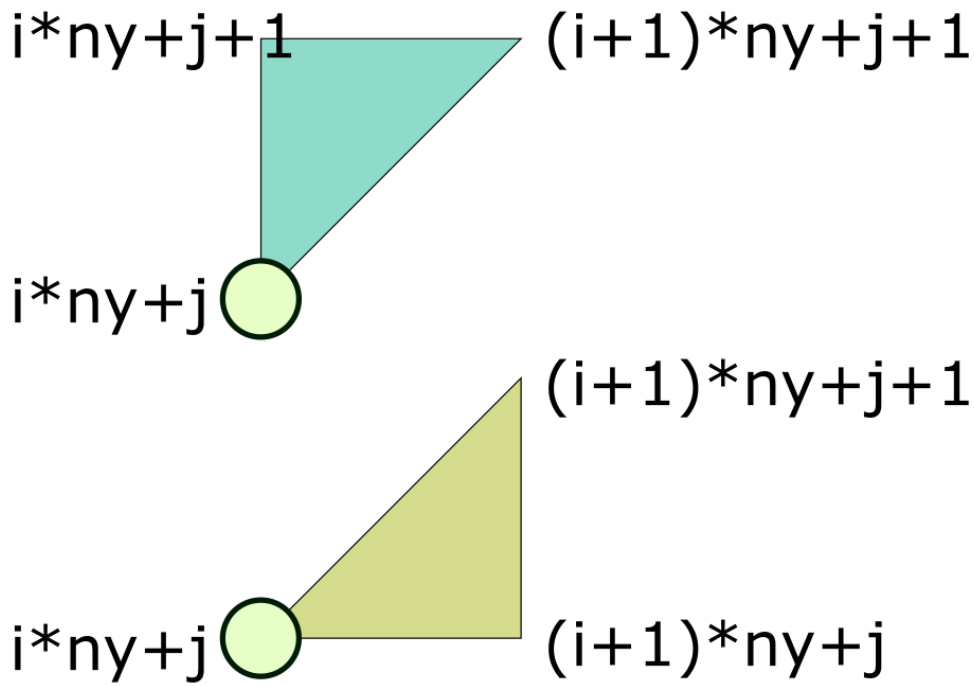
- Let‘s define a grid and set the coordinates of the grid points:



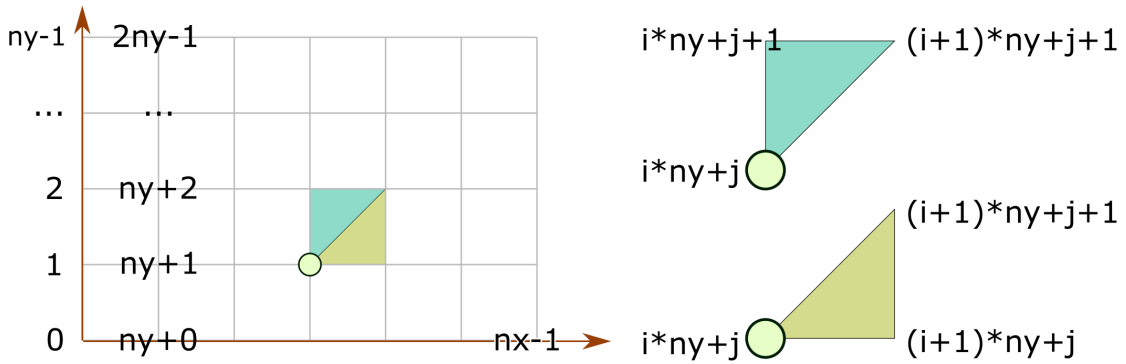
3. Index–based Triangles

- Based on the grid, we can observe the following triangle indices:





3. Index-based Tri.



- Set the indices for the triangles:

```
unsigned int indices[3 * (numPoints_x - 1) * (numPoints_y - 1) * 2] = {};
for (int i = 0; i < numPoints_x-1; i++)
    {
        for (int j = 0; j < numPoints_y-1; j++)
        {
            int curIndex = 6 * (i * (numPoints_y-1) + j);
            indices[curIndex + 0] = i * numPoints_y + j;
            indices[curIndex + 1] = i * numPoints_y + j + 1;
            indices[curIndex + 2] = (i+1) * numPoints_y + j + 1;
            indices[curIndex + 3] = i * numPoints_y + j;
            indices[curIndex + 4] = (i+1) * numPoints_y + j;
            indices[curIndex + 5] = (i+1) * numPoints_y + j + 1;
        }
    }
```

3. Index–based Triangles

- Do not forget to set up an EBO:

```
unsigned int EBO;
…
glGenBuffers(1, &EBO);
…
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices,
GL_STATIC_DRAW);
```

- Change the draw command:

```
glDrawElements(GL_TRIANGLES, sizeof(indices) / sizeof(indices[0]),
GL_UNSIGNED_INT, 0);
```

3. Index–based Triangles

- We also do not need the normal on the C++ site, we generate the normal vectors in the shader later

4. Vertex Positions

$$f(x,y) = \begin{pmatrix} x \\ y \\ \cos(2 \cdot \sqrt{x^2 + y^2}) \end{pmatrix}$$

- In the vertex shader, we alter the coordinates:

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

out vec3 FragPos;

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

void main()
{
    float x=aPos.x;
    float y=aPos.y;
    float z=cos(2.0 * sqrt(x * x + y * y));

    FragPos = vec3(model * vec4(x, y, z, 1.0));
    gl_Position = projection * view * vec4(FragPos, 1.0);
}
```

5. Normal Vectors

$$f(x,y) = \begin{pmatrix} x \\ y \\ \cos(2 \cdot \sqrt{x^2+y^2}) \end{pmatrix}$$

- We learned that we can determine the normal vector by calculating the derivatives:

$$\frac{\partial f_z(x,y)}{\partial x} = \frac{-2x \cdot \sin(2 \cdot \sqrt{x^2+y^2})}{\sqrt{x^2+y^2}}$$

$$\frac{\partial f_z(x,y)}{\partial y} = \frac{-2y \cdot \sin(2 \cdot \sqrt{x^2+y^2})}{\sqrt{x^2+y^2}}$$

5. Normal Vectors

$$\frac{\partial f_z(x,y)}{\partial x} = \frac{-2x \cdot \sin(2 \cdot \sqrt{x^2+y^2})}{\sqrt{x^2+y^2}}$$

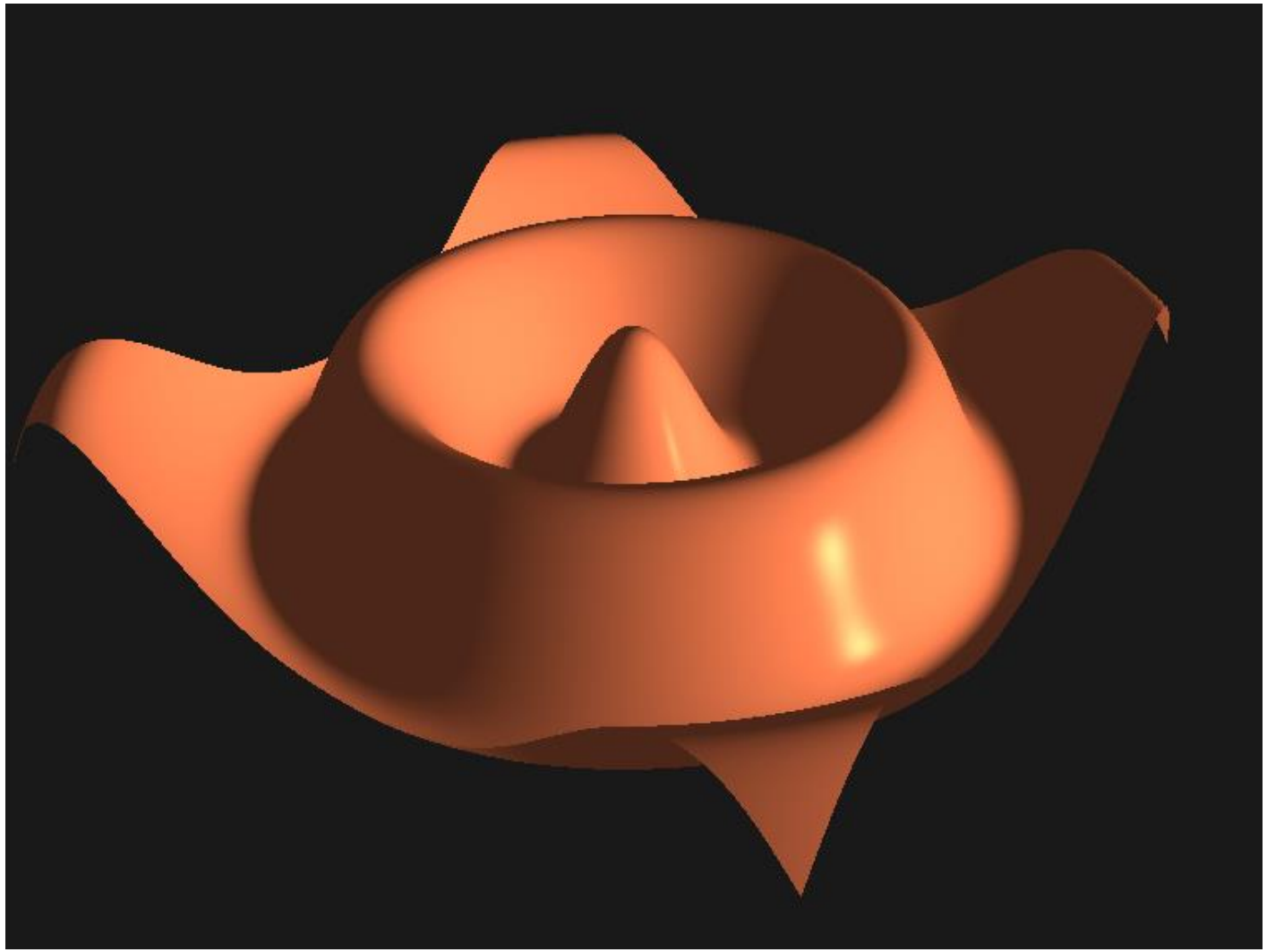
$$\frac{\partial f_z(x,y)}{\partial y} = \frac{-2y \cdot \sin(2 \cdot \sqrt{x^2+y^2})}{\sqrt{x^2+y^2}}$$

- Based on the tangent vectors, we can now generate the normal vector in the vertex shader:

```
float derive_x = -2.0 * x / sqrt(x * x + y * y) * sin(2.0 * sqrt(x * x + y * y));
float derive_y = -2.0 * y / sqrt(x * x + y * y) * sin(2.0 * sqrt(x * x + y * y));

vec3 t1 = vec3(1.0, 0.0, derive_x);
vec3 t2 = vec3(0.0, 1.0, derive_y);
vec3 normal = normalize(cross(t1, t2));

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

F5…

- … we get this beauty!
- (We kept the fragment shader from the section previous)
- (We also changed the lightPos and rotated the surface)



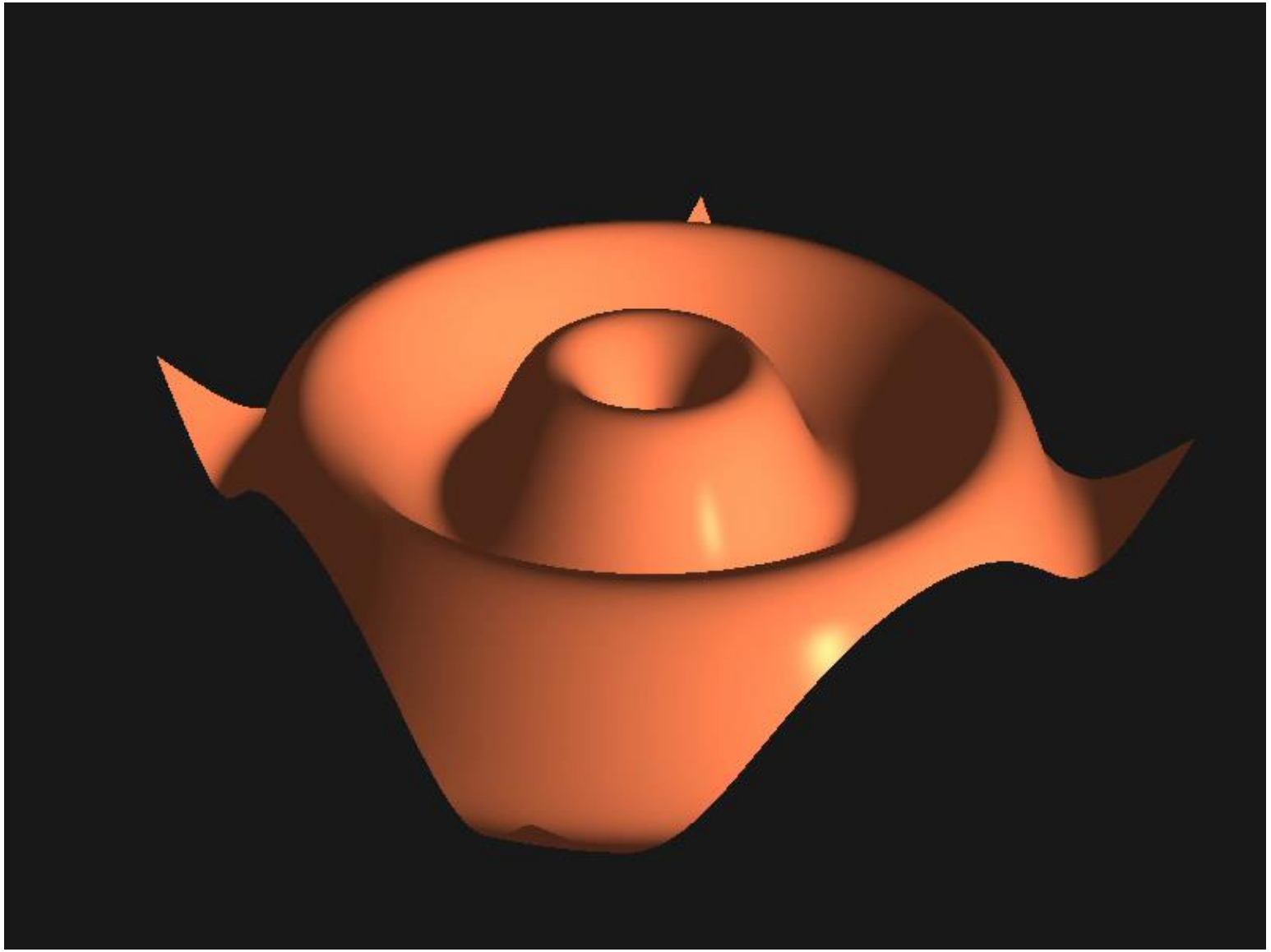
Animated Waves

- Now, we add a timer as n uniform and using `glfwGetTime()`

```
uniform float timer;
…
float z=cos(2.0*sqrt(x * x + y * y) + 2.0 * timer);
…
float derive_x = -2.0 * x / sqrt(x * x + y * y) * sin(2.0 * sqrt(x * x + y *
y) + 2.0 * timer);
float derive_y = -2.0 * y / sqrt(x * x + y * y) * sin(2.0 * sqrt(x * x + y *
y) + 2.0 * timer);
```

F5…

- … nice!



Questions???