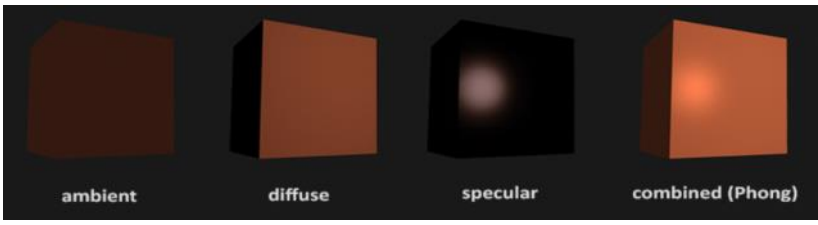
BASIC LIGHTING

Lighting models in computer graphics are a dumbed-down interpretation of real-world lighting, since real-world light is incredibly complex and would be overtly expensive to calculate exactly.
A common lighting model based on the physics of real-world lighting is the **Phong lighting model**. It consists of 3 components:

- Ambient lighting: Even when it is dark, there is usually still some light present (e.g. moonlight), so models are almost never completely dark.
  - We use an ambient lighting constant to give all objects some small amount of color.
- Diffuse lighting: Light comes from a light source, which means the light shone on an object comes from a direction relative to the object. The more a part of an object faces a light source, the brighter is becomes.
- Specular lighting: Some objects can shine, reflecting a bright spot of light (that is the light source). Specular highlights are more inclined to the color of the light than the color of the object.



Ambient Lighting

Light does not usually come from a single light source, but from multiple light sources scattered all around us, even if they are not immediately visible.

- Light can scatter and bounce in many directions, so objects that are not directly within line-of-sight of a light source can still be lit by another non-light-source object that is, since the light will bounce off of that object.
  - Algorithms that take this into consideration are called **global illumination** algorithms, but these are complicated and expensive to calculate.

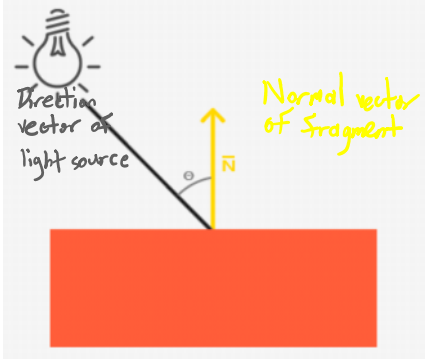
Ambient lighting is a simplified model of global illumination that just applies a constant light value to objects, imitating scattered light.
In the previous chapter, the light we made is essentially an ambient light.

Modify the intensity of our ambient light vector and apply it to the fragment color.

- Ambient Lighting

Diffuse Lighting

Diffuse lighting gives an object more brightness the closer its fragments are aligned to the light rays from a light source.



Using the image on the right:

- To the left is a light source with a light ray (the gray vector) targeted at a single fragment of our object.
  - We need to measure at what angle the light ray touches the fragment.
    - If the light ray is perpendicular to the object's surface, the light has the greatest impact. However, if the light ray is parallel to or behind the object's surface, the light has no impact.
    - To measure the angle between the light ray and the fragment, we use a normal vector, which is a vector perpendicular to the fragment's surface (the yellow vector).
    - The angle between the light ray vector and normal vector can be easily calculated with the dot product.
      - *Remember that the lower the angle between two unit vectors, the more the dot product is inclined towards a value of 1. So, when the angle (Θ) between both vectors is 90 degrees, the dot product becomes 0.*
      - **NOTE:** To get only the cosine of the angle between both vectors, we will work with unit vectors. Therefore, make sure all the vectors are normalized, otherwise the dot product returns more than just the cosine.
      - The resulting dot product thus returns a scalar that we can use to calculate the light's impact on the fragment's color, resulting in differently lit fragments based on their orientation towards the light.

What you need to calculate diffuse lighting:

1. Normal vector: A vector that is perpendicular to the vertex's surface.
2. The directed light ray vector: A direction vector that is the difference vector between the light's position and the fragment's position.

Normal Vectors

A normal vector is a unit vector that is perpendicular to the surface of a vertex.
Since a vertex by itself has no surface (it's just a single point in space), we retrieve a normal vector by using its surrounding vertices to figure out the surface of the vertex.

- We could calculate the normal vectors of the cube's vertices by using the cross product, but, since a 3D cube is not a complicated shape, we can simply manually add them to the vertex data.
  - Vertex Data w/ Normals

Adjust your code to take into account the newly added Normals vertex attribute.

- Adding Normals to Vertex Attributes

Calculating the Diffuse Color

Now that we have the normal vector for each vertex, we need to calculate the light source direction vector using the light's position vector and the fragment's position vector.

Calculate the light source direction vector.

- **NOTE:** When calculating lighting, we usually do not care about the magnitude or position of a vector; we only care about its direction. Because of this, almost all calculations are done with unit vectors since it simplifies most calculations (like the dot product).
- Light Source Direction Vector

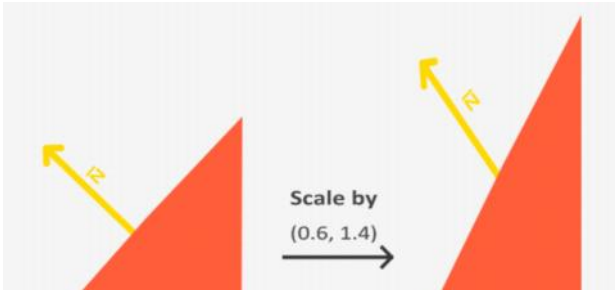
Calculate the diffuse lighting vector and apply it to the fragment color.

- Diffuse Light Vector

One Last Thing

Notice how we don't transform the normal vector of a vertex to world coordinates. That's because:

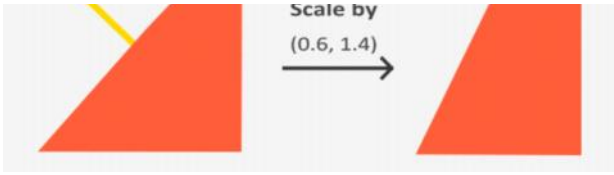
1. Normal vectors are just direction vectors and do not represent a specific position in space.
   - Normal vectors do not have a homogeneous coordinate (the w component of a vertex position) and, therefore, should not be affected by translations.
     - So, if we want to multiply the normal vectors with a model matrix, we want to remove the translation part of the matrix by taking the upper-left 3x3 matrix of the model matrix.
     - *Remember that translations occur in the 4th column of a 3D transformation matrix.*
2. If the model matrix would perform a non-uniform scale (e.g. scaling by `(0.6, 1.4, 1.6)`), the vertices would be changed in such a way that the normal vector is not perpendicular to the surface anymore (as seen in the image to the right), which would distort lighting.



This behavior can be fixed by using a different model matrix specifically tailored for normal vectors, called a normal matrix.

A normal matrix uses a few linear algebraic operations to remove the effect of wrongly scaling the normal vectors.

light), which would distort lighting.



This behavior can be fixed by using a different model matrix specifically tailored for normal vectors, called a normal matrix.

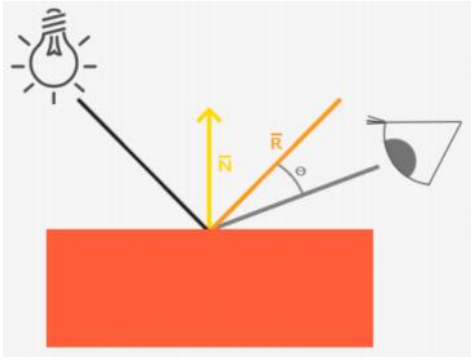
A normal matrix uses a few linear algebraic operations to remove the effect of wrongly scaling the normal vectors.

- **Normal Matrix:** The transpose of the inverse of the upper-left 3x3 part of the model matrix.
  - Don't worry too much about what inversing and transpositioning a matrix right now. It will be discussed in detail in a later chapter.
  - **NOTE:** Inversing matrices is expensive for shaders, so it is recommended to calculate the normal matrix on the CPU; however, for this example, we'll be inversing the normal matrix on the GPU.

Calculate the normal vector using a normal matrix.

- Normal Matrix

Specular Lighting



Similar to diffuse lighting, specular lighting is based on the light's direction vector and the object's normal vectors, but this time it is also based on the view direction (e.g. from what direction the player is looking at the fragment).
Specular lighting is based on the reflective properties of surfaces. So, if we think of the object's surface as a mirror, the specular lighting is the strongest wherever we would see the light reflected on the surface.

We calculate a reflection vector by reflecting the light direction vector around the normal vector. Then, we calculate the angular distance between this reflection vector and the view direction.

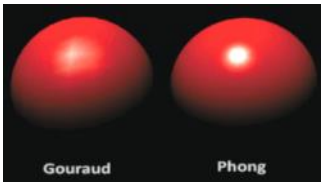
- The closer the angle between the reflection vector and the view direction vector, the greater the impact of the specular light.
- The resulting effect is a slight highlight when we're looking at the light's direction reflected via the surface.

NOTE: We're going to do lighting calculations in world space, but most people prefer doing lighting in view space. The advantage of calculating lighting in view space is that the viewer's position is always at `(0, 0, 0)`, so you've already got the position of the viewer for free. However, calculating lighting in world space is more intuitive. If you want to calculate lighting in view space, you should transform all the relevant vectors with the view matrix as well (don't forget to change the normal matrix too).

Calculate the specular lighting vector and apply it to the fragment color.

- Specular Lighting

Gouraud Lighting Model



Back in the day, lighting used to be calculated per vertex instead of per fragment, since objects typically have far fewer vertices than fragments.

- This is much more efficient than Phong shading, but results in interpolated lighting colors for surrounding fragments, which doesn't look very realistic unless the object contains a large amount of vertices.

When the Phong lighting model is implemented in the vertex shader, it is called **Gouraud shading** instead.

EXERCISES

1. Right now, the light source is a boring, static light source that doesn't move. Try to move the light source around the scene over time using either sin of cos. Watching the lighting change over time gives you a good understanding of Phong's lighting model.
2. Play around with different ambient, diffuse, and specular strengths and see how they impact the result. Also, experiment with the shininess factor. Try to comprehend why certain values have a certain visual output.
3. Do Phong shading in view space instead of world space.
4. Implement Gourand shading instead of Phong shading. If you did things right, the lighting should look a bit off (especially with specular highlights) with the cube object. Try to reason why it looks so weird.

Ambient Lighting

Tuesday, March 29, 2022 5:08 PM

```
void main() {
   float ambientStrength = 0.5; // Value that sets the ambient light to half strength
   vec3 ambient = ambientStrength * lightColor; // Calculates the ambient lighting

   FragColor = vec4(ambient * objectColor, 1.0); // Applies ambient lighting to each fragment
}
```

Remember that the main function in GLSL is just like a main function in C. We can perform operations, declare and instantiate variable, etc.

Vertex Data w/ Normals

Tuesday, March 29, 2022 5:43 PM

```
// Vertex data of a cube
float vertices[] = {
   // Positions          Normals
   // Front face
   -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,

   // Back face
   -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,

   // Left Face
   -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,

   // Right face
    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,

    // Bottom face
    -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,

    // Top face
    -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
};
```

Adding Normals to Vertex Attributes

Tuesday, March 29, 2022 5:49 PM

default.vert

```
...

layout (location = 1) in vec3 aNormal;

...

out vec3 Normal;

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

default.frag

```
in vec3 Normal;
```

Update your vertex attribute pointers for both the object and light source.

Object VAO

```
// Positions
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// Normals
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float)));
glEnableVertexAttribArray(1);
```

Light Source VAO

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

Notice how we only changed the stride of the vertex attribute pointer for light cube vertex positions. Since the normals of the light cube aren't necessary, we can just omit them and only adjust the stride.

- **NOTE:** This is actually more efficient that creating a separate VBO for the light source object because the vertex data is already s tored in the GPU's memory from the object. Since we're using the same VBO, we don't have to store any new data into the GPU's memory.

Light Source Direction Vector

Tuesday, March 29, 2022 7:04 PM

default.vert

```
...
out vec3 FragPos;

void main() {
   ...
   FragPos = vec3(model * vec4(aPos, 1.0));
}
```

default.frag

```
...
in vec3 FragPos;
uniform vec3 lightPos;

void main() {
   ...
   vec3 norm = normalize(Normal);
   vec3 lightDir = normalize(lightPos - FragPos);
   ...
}
```

Diffuse Light Vector

Tuesday, March 29, 2022 7:14 PM

default.frag

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

vec3 finalColor = (diffuse + ambient) * objectColor;

FragColor = vec4(finalColor, 1.0);
```

If the angle between the normal vector and light direction vector is greater than 90 degrees, the dot product is actually negative, so we want to make sure the diffuse value doesn't go below 0.0.

default.frag code

```
#version 330 core

in vec3 Normal;
in vec3 FragPos;

out vec4 FragColor;

uniform vec3 objectColor;
uniform vec3 lightColor;
uniform vec3 lightPos;

void main() {
   float ambientStrength = 0.5;
   vec3 ambient = ambientStrength * lightColor;

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

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

   vec3 finalColor = (diffuse + ambient) * objectColor;

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

Normal Matrix

Tuesday, March 29, 2022 8:55 PM

default.vert

```
void main() {
...
Normal = mat3(transpose(inverse(model))) * aNormal;
```

By casting the model matrix to a mat3, we ensure that translations will not affect the normal vector and that it actually can multiply with the 3D normal vector.

Specular Lighting

Tuesday, March 29, 2022 9:46 PM

Render Loop

```
glUniform3fv(glGetUniformLocation(shaderProgram.id, "viewPos"), 1, glm::value_ptr(camera.position));
```

default.frag

```
...
uniform vec3 viewPos;

void main() {
   ...

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

   vec3 finalColor = (specular + diffuse + ambient) * objectColor;

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

`lightDir` points from the fragment to the light source, so we need to negate it (point the vector in the opposite direction) when we use the `reflect` function to get the correct reflection vector.
The pow function used to calculate `spec` affects the spread of the reflection of the light. A higher power focuses the light in a smaller, but brighter area, and a lower power disperses the light among a wider area.