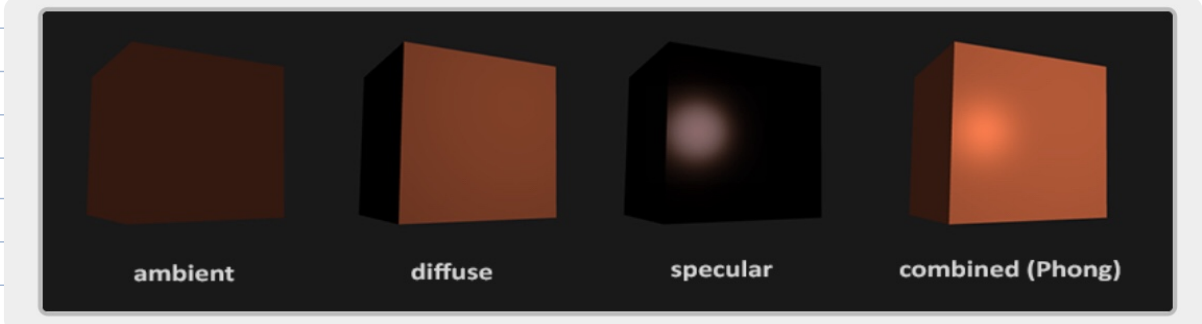
PHONG MODEL

HJALMAR BASILE
JAN 2019

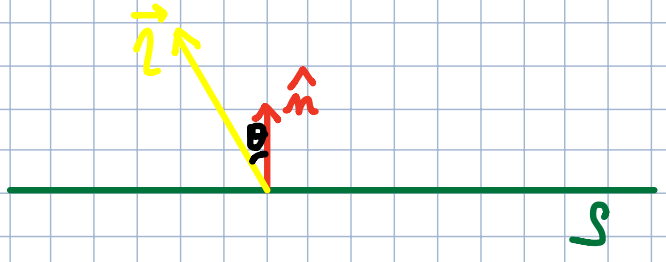
Phong is a basic lighting model which consists of 3 components:

AMBIENT , DIFFUSE , SPECULAR



AMBIENT: This is some constant illumination source coming from the environment. It is a cheap approximation of real physics models of light scattering.

DIFFUSE: The brightness of a surface depends from the angle between an incident ray of light and the normal to the surface.

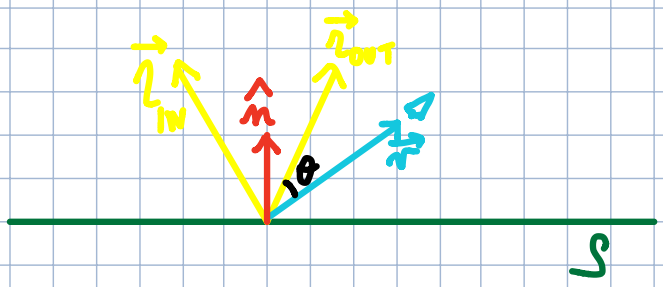


The diffuse intensity is given by $\cos\theta$, which is indeed maximized for $\theta = 0$, or 0 when $|\theta| > \frac{\pi}{2}$*

To compute it we simply compute $\max\left(0, \hat{n} \cdot \text{norm}(\vec{l})\right)$

*(the surface is lighted only when $0 \leq |\theta| \leq \frac{\pi}{2}$)

SPECULAR: An incident ray of light will reflect, and the brightness perceived by the viewer will depend from the angle between the reflected ray and the viewer view direction.



We will compute the specular intensity as

$$\max\left(0, \text{norm}(\vec{v}) \cdot \text{norm}(\vec{r}_{OUT})\right)^{\text{SHININESS}}$$

where shininess is a value depending on the material of the surface.

/ ——— /

Below an example implementation of the model

13:17 Sat 19 Jan 28%

github.com

HjalmarBasile Implement Phong shading in view space 646a0ef 7 days ago

1 contributor

31 lines (23 sloc) | 819 Bytes Raw Blame History

```
#version 330 core

layout(location = 0) in vec3 position;
layout(location = 1) in vec3 normal;

out vec3 passFragViewSpacePos;
out vec3 passNormal;
out vec3 passLightViewSpacePos;

uniform mat4 u_View;
uniform mat4 u_ModelView;
uniform mat4 u_MVP;
uniform vec3 u_LightPosition;

void main()
{
        vec4 posToVec4 = vec4(position, 1.0f);
        gl_Position = u_MVP * posToVec4;

        /* Compute position of the vertex in view space */
        vec4 fragViewSpacePos = u_ModelView * posToVec4;
        /* N.B. fragViewSpacePos.w is 1 by construction */
        passFragViewSpacePos = fragViewSpacePos.xyz;

        /* Transform normal so that it is still ortogonal to the surface */
        /* N.B. Beware of ill-conditioned matrices */
        passNormal = mat3(transpose(inverse(u_ModelView))) * normal;

        passLightViewSpacePos = vec3(u_View * vec4(u_LightPosition, 1.0f));
}
```

PHONG VERTEX SHADER

HjalmarBasile Implement Phong shading in view space

1 contributor

37 lines (28 sloc) | 1.19 KB Raw Blar

```
1  #version 330 core
2
3  out vec4 color;
4
5  in vec3 passFragViewSpacePos;
6  in vec3 passNormal;
7  in vec3 passLightViewSpacePos;
8
9  uniform vec3 u_ObjectColor;
10 uniform vec3 u_AmbientColor;
11 uniform vec3 u_LightColor;
12
13 uniform float u_AmbientStrenght;
14 uniform float u_DiffuseStrenght;
15 uniform float u_SpecularStrenght;
16 uniform float u_SpecularShininess;
17
18 void main()
19 {
20         /* Compute ambient light component */
21         vec3 ambientColor = u_AmbientStrenght * (0.9f * u_AmbientColor + 0.1f * u_LightColor);
22
23         /* Compute diffuse light component */
24         vec3 nPassNormal = normalize(passNormal);
25         vec3 nLightDir = normalize(passLightViewSpacePos - passFragViewSpacePos);
26         float diffuseIntensity = max(0.0f, dot(nPassNormal, nLightDir));
27         vec3 diffuseColor = u_DiffuseStrenght * diffuseIntensity * u_LightColor;
28
29         /* Compute specular light component */
30         vec3 nViewDir = normalize(-passFragViewSpacePos); /* Viewer is at origin in View space */
31         vec3 nReflectDir = reflect(-nLightDir, nPassNormal);
32         float specularIntensity = pow(max(0.0f, dot(nViewDir, nReflectDir)), u_SpecularShininess);
33         vec3 specularColor = u_SpecularStrenght * specularIntensity * u_LightColor;
34
35         color = vec4(u_ObjectColor * (ambientColor + diffuseColor + specularColor), 1.0f);
36 }
```

PHONG FRAGMENT SHADER

FRAGMENT: We can recognize the 3 components described above. Notice how we assign a strenght to each component as a float value. We can actually generalize this and define a material property, where we have vec3 instead of float for the strenght.

Also notice that to compute diffuse and specular colors we will multiply by the light-color, and in the end the sum of all the components will be multiplied by the object color to get the final fragment color.

VERTEX: Notice that we are passing the normals applying a transformation, this is because the ModelView matrix may include rotations and scalings, so the normals in View space are going to be different from the ones in Local space.

For sake of simplicity let's call $A = View \cdot Model$ (JUST THE 3x3 UPPER PART).

Notice that A is a composition of scalings, rotations and translations, so it will be invertible.

We would like to show that, if the vertices of a triangle are transformed by A, the normal of the triangle will be transformed by $(A^{-1})^T$ (except for rescaling).

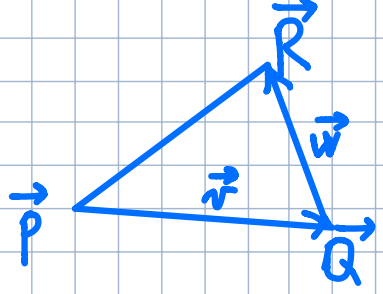
OSS.1 If $A \in \mathbb{R}^{m \times m}$ is invertible, then $(A^T)^{-1} = (A^{-1})^T$.

Indeed by definition we get $A \cdot A^{-1} = A^{-1} \cdot A = I \Rightarrow$

by transposing we get $(A^{-1})^T \cdot A^T = A^T \cdot (A^{-1})^T = I^T = I \Rightarrow$

by definition this means that A^T is invertible, with inverse $(A^{-1})^T$ ■

Let's consider the following triangle:



Let's define $\vec{v}=\vec{Q}-\vec{P}$, $\vec{w}=\vec{R}-\vec{Q}$, $\vec{n}=\text{norm}(\vec{v}\times\vec{w})$

and denote the transformed vertices with $\vec{P}', \vec{Q}', \vec{R}'$:

$\vec{P}'=A\cdot\vec{P}$, $\vec{Q}'=A\cdot\vec{Q}$, $\vec{R}'=A\cdot\vec{R}$ $\Rightarrow$

$\vec{v}'=\vec{Q}'-\vec{P}'=A\cdot\vec{Q}-A\cdot\vec{P}=A\cdot(\vec{Q}-\vec{P})=A\cdot\vec{v}$ and in the same way $\vec{w}'=A\vec{w}$.

We know that $\vec{v}\perp\vec{n}$, let's prove that $\vec{v}'\perp\left(A^{-1}\right)^T\vec{n}$ [SAME THING WILL HOLD FOR $\vec{w}$]

$\vec{v}\perp\vec{n}\Rightarrow v^T\cdot n=0\Rightarrow v^T\cdot\left(A^T\cdot(A^T)^{-1}\right)\cdot n=0\Rightarrow\left(v^T\cdot A^T\right)\cdot\left(\left(A^T\right)^{-1}\cdot n\right)=0$

$\Rightarrow\left(A\cdot v\right)^T\cdot\left(\left(A^{-1}\right)^T\cdot n\right)=0\Rightarrow A\vec{v}\perp\left(A^{-1}\right)^T\vec{n}$ ■

($A\vec{v}=\vec{v}'$)

We now know that the transformed normal will be orthogonal to the transformed triangle, but we are yet not satisfied:

will this transformed normal still point to the expected direction?

(i.e. the direction of $\vec{v}'\times\vec{w}'$)

To answer to this question we need the following lemma.

OSS.2 Let $A\in\mathbb{R}^{3\times3}$ invertible, $v,w\in\mathbb{R}^3$, then

$$\left(Av\right)\times\left(Aw\right)=\det A\cdot\left(A^{-1}\right)^T\cdot\left(v\times w\right).$$

[PROOF IS JUST A LONG CALCULATION]

Applying the above relation to our problem, we get

$$\vec{v}' \times \vec{w}' = A\vec{v} \times A\vec{w} = \det A \cdot (A^{-1})^T \cdot (\vec{v} \times \vec{w}), \text{ so,}$$

as long as $\det A > 0$, direction will be preserved.

Notice that rotations and translations will have det equal to 1, also scaling matrices will have positive det if all the scale factors are positive.

SUMMING UP We can use $(ModelView^{-1})^T$ to transform the normals, we just have to normalize the resulting vector.

FINAL NOTE Explicit matrix inverse calculation is always bad in numerical analysis. In our case we could take advantage of the known shape of the single factors which compose the Model and View matrix, and building the $(A^{-1})^T$ as a safe product of known matrices. I will get into more details if I should implement this approach in the future.