111550139

郭芷安

ICG HW3 Report

Bling-Phong Shader

Fragment shader:

```
out vec4 FragColor;

in vec2 TexCoord;
in vec3 fragPos;
in vec3 fragNormal;

uniform sampler2D ourTexture;
uniform vec3 cameraPos;
uniform vec3 lightPos;
uniform vec3 lightAmb;
uniform vec3 lightDiff;
uniform vec3 lightSpec;
uniform vec3 matAmb;
uniform vec3 matDiff;
uniform vec3 matSpec;
uniform float matGloss;


void main() {
    vec3 normal = normalize(fragNormal);
    vec3 light = normalize(lightPos - fragPos);
    vec3 view = normalize(cameraPos - fragPos);
    vec3 halfVec = normalize((light + view));
    vec3 color = texture(ourTexture, TexCoord).rgb;

    vec3 ambient = lightAmb * matAmb * color;

    vec3 diffuse = lightDiff * matDiff * dot(light, normal) * color;

    vec3 specular = lightSpec * matSpec * pow(max(0.0, dot(normal, halfVec)), matGloss);

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

vertex shader:

```
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoord;

out vec2 TexCoord;
out vec3 fragPos;
out vec3 fragNormal;

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

void main() {
    TexCoord = aTexCoord;
    vec4 worldPos = model * vec4(aPos, 1.0);
    fragPos = worldPos.xyz;
    fragNormal = mat3(transpose(inverse(model))) * aNormal;
    gl_Position = projection * view * worldPos;
}
```

For bling-phone shading, the picture on the left-hand side is fragment shader, and the picture on the right-hand side is vertex shader.

Bling-phone shading is to calculate the halfway vector between viewer and light vector, and use dot(N, H) to replace dot(R, V) to reduce the complexity of calculation.

In vertex shader, I simply convert the model from model view to world view, and pass world position, fragment position and fragment normal vector to fragment shader.

In fragment shader, calculate the light vector and view vector using light position, fragment position and viewer position first, and calculate the halfway vector using light and view vector, then generate model color using model texture. Calculate ambient, diffuse and specular using the following formula:

$$ambient = L_{ambient} \times K_{embient} \times modelColor$$
$$diffuse = L_{diffuse} \times K_{diffuse} \times modelColor$$
$$specular = L_{specular} \times K_{specular} \times (V \cdot R)^{\alpha}$$

and add them together to produce the final fragment color.

Gouraud Shader

Vertex shader

```
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoord;

out vec2 TexCoord;
out vec3 ambient;
out vec3 diffuse;
out vec3 specular;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
uniform vec3 cameraPos;

uniform vec3 lightPos;
uniform vec3 lightAmb;
uniform vec3 lightDiff;
uniform vec3 lightSpec;
uniform vec3 matAmb;
uniform vec3 matDiff;
uniform vec3 matSpec;
uniform float matGloss;

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

    TexCoord = aTexCoord;

    vec3 normal = normalize(mat3(transpose(inverse(model))) * aNormal);
    vec4 worldPos = model * vec4(aPos, 1.0);

    ambient = lightAmb * matAmb;

    vec3 light = normalize(lightPos - vec3(worldPos));
    diffuse = lightDiff * matDiff * dot(light, normal);

    vec3 viewVec = normalize(cameraPos - vec3(worldPos));
    vec3 reflect = normalize(reflect(-light, normal));
    specular = lightSpec * matSpec * pow(max(0.0, dot(viewVec, reflect)), matGloss);
}
```

Fragment shader

```
out vec4 FragColor;

in vec2 TexCoord;
in vec3 ambient;
in vec3 diffuse;
in vec3 specular;

uniform sampler2D ourTexture;



void main() {
    vec3 color = texture(ourTexture, TexCoord).rgb;

    vec3 amb = ambient * color;

    vec3 diff = diffuse * color;

    FragColor = vec4((amb + diff + specular), 1.0);
}
```

For gouraud shading, the picture on the left-hand side is vertex shader, and the picture on the right-hand side is fragment shader.

Gouraud shading is a shading method which implements phong lighting model at each vertex, so we have to complete the calculations of ambient, diffuse and specular in vertex shader using the formula provided in the picture below, and pass these three parameters and texture coordinate to fragment shader.

$$\text{Ambient} = L_{ambient} \times K_{ambient}$$
$$\text{Diffuse} = L_{diffuse} \times K_{diffuse} \times (L \cdot N)$$
$$\text{Specular} = L_{specular} \times K_{specular} \times (V \cdot R)^{a}$$

In fragment shader, get model color using model texture, and multiply ambient and diffuse passed from vertex shader with model color. At the end, add up the final ambient, diffuse and specular to create the final fragment color.

Environment Cubemap

main

```
glm::mat4 cubemapView = glm::mat4(glm::mat3(view));
cubemapShader->use();
cubemapShader->set_uniform_value("projection", projection);
cubemapShader->set_uniform_value("view", cubemapView);
glDepthFunc(GL_LEQUAL); // Render skybox after all objects
glBindVertexArray(cubemapVAO);
glBindTexture(GL_TEXTURE_CUBE_MAP, cubemapTexture);
glDrawArrays(GL_TRIANGLES, 0, 36);
glBindVertexArray(0);
glDepthFunc(GL_LESS);
```

Vertex shader

```
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoord;

out vec3 TexCoord;

uniform mat4 view;
uniform mat4 projection;

void main() {
    TexCoord = aPos;
    vec4 pos = projection * view * vec4(aPos, 1.0);
    gl_Position = pos.xyww;
}
```

fragment shader

```
out vec4 FragColor;

in vec3 TexCoord;

uniform samplerCube ourTexture;

void main() {
    FragColor = texture(ourTexture, TexCoord);
}
```

The render environment cubemap, pass the necessary uniform values to the shader, bind the cubemap texture and render cubemap in main.

The picture on the left-hand side is vertex shader, and the picture on the right-hand side is fragment sharder.

In vertex shader, calculate pos to transform the vertex from model view to world view, and let gl_position equals pos.xyww. Thus, we ensure that the skybox appears infinitely far away, which x and y still undergo perspective scaling, and z is fixed to a constant value 1.0, which ensure the skybox depth is always farthest, and w controls the perspective division, ensure the cube's position respects the camera's perspective while always enclosing the viewer.

In fragment shader, simply generate the fragment color using the cube texture and texture coordinate passed by the vertex shader.

Metallic Shader

```
void main() {
    vec3 normal = normalize(fragNormal);
    vec3 light = normalize(lightPos - fragPos);
    vec3 view = normalize(cameraPos - fragPos);

    // calculate reflection
    vec3 reflection = (-view) - (2 * dot(-view, normal) * normal);
    vec3 envColor = texture(envTexture, reflection).rgb;
    vec3 modelColor = texture(ourTexture, TexCoord).rgb;

    // calculate B and fragColor
    float intensity = 1.0;
    float bias = 0.2;
    float alpha = 0.4;
    float B = max(dot(light, normal) * intensity, 0.0) + bias;
    FragColor = vec4((alpha * B * modelColor) + ((1 - alpha) * envColor), 1.0);
}
```

Because the vertex shader of metallic shading is identical to the vertex shader of Bling-Phong shading, so I am only going to explain fragment shader in this part.

The provided picture is the fragment shader, and to complete metallic shading, we have to calculate the reflection of environment and model color by the corresponding texture.

The reflection of environment color is computed using the formula: $R = I - 2 \times (I \cdot N) \times N, I = -viewVector$, and environment color = texture(envTexture, reflection).rgb.

After computing the value of B using the formula $B = \max\big((L \cdot N)I_l, 0\big) + bias$. complete the final fragment color by combining the model color and reflection of environment color together using the formula: $fragmentColor = alpha * B * modelColor + (1 - alpha) * environmentColor$.

Glass Shader – Schlick Approximation

```
void main() {
    vec3 normal = normalize(fragNormal);
    vec3 light = normalize(lightPos - fragPos);
    vec3 view = normalize(cameraPos - fragPos);

    // calculate reflect
    vec3 reflection = (-view) - (2 * dot(-view, normal) * normal);
    vec3 reflectColor = texture(envTexture, reflection).rgb;

    // calcuate refraction based on Snell's law
    float n1 = 1.0;
    float n2 = 1.52;
    float eta = n1 / n2;  // airRef / objRef
    float cosTheta = dot(-view, normal);
    if (cosTheta > 0.0) {
        eta = n2 / n1;
    }
    float k = 1.0 - eta * eta * (1.0 - cosTheta * cosTheta);
    vec3 refracted = vec3(0, 0, 0);
    if (k >= 0.0) {
        refracted = eta * (-view) - (eta * cosTheta + sqrt(k)) * normal;
    }
    vec3 refraColor = texture(envTexture, refracted).rgb;

    // calculate ratio
    float R0 = pow(((n1 - n2) / (n1 + n2)), 2);
    float R = R0 + (1 - R0) * pow((1 + cosTheta), 5);
    FragColor = vec4(R * reflectColor + (1 - R) * refraColor, 1.0);
}
```

Because the vertex shader of glass shading is identical to the vertex shader of Bling-Phong shading, so I am only going to explain fragment shader in this part as well.

To complete glass shading with schlick approximation, we have to complete the reflection of environment color and the refraction of environment color.

The method of calculating reflection color is explained in metallic shading, and the method to complete refraction color is to follow the formula: $T = \eta I - \left(\eta(I \cdot N) + \sqrt{k}\right)N, k = 1 - \eta^2(1 - (I \cdot N)^2), I = -viewVector$, and refractionColor = texture(envTexture, T).

After complete refraction color and reflect color, calculate R_θ using the formula

$$R_\theta = R_0 + (1 - R_0)(1 + I \cdot N)^5$$

$$R_0 = \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2, \text{ where } n \text{ is the refractive index}$$

And compute fragColor as R_θ * reflectColor + (1- R_θ) * refraColor.

Glass Shader – Empricial Approximation

```
void main() {
    vec3 normal = normalize(fragNormal);
    vec3 light = normalize(lightPos - fragPos);
    vec3 view = normalize(cameraPos - fragPos);

    // calculate reflect
    vec3 reflection = (-view) - (2 * dot(-view, normal) * normal);
    vec3 reflectColor = texture(envTexture, reflection).rgb;

    // calcuate refraction based on Snell's law
    float n1 = 1.0;
    float n2 = 1.52;
    float eta = n1 / n2;  // airRef / objRef
    float cosTheta = dot(-view, normal);
    if (cosTheta > 0.0) {
        eta = n2 / n1;
    }
    float k = 1.0 - eta * eta * (1.0 - cosTheta * cosTheta);
    vec3 refracted = vec3(0, 0, 0);
    if (k >= 0.0){
        refracted = eta * (-view) - (eta * cosTheta + sqrt(k)) * normal;
    }
    vec3 refraColor = texture(envTexture, refracted).rgb;

    // calculate ratio
    float scale = 0.7;
    float power = 2.0;
    float bias = 0.2;
    float R = max(0.0, min(1, bias + scale * pow((1 + cosTheta), power)));
    FragColor = vec4(R * reflectColor + (1 - R) * refraColor, 1.0);
}
```

Because the vertex shader of glass shading is identical to the vertex shader of Bling-Phong shading, so I am only going to explain fragment shader in this part.

To complete glass shading with empricial approximation, we have to complete the reflection of environment color and the refraction of environment color. and the method of complete reflection color and refraction color is already explained in the above scope.

After the reflection color and refraction color is completed, we have to calculate R_θ using the formula: $R_\theta = max(0, min(1, bias + scale \times (1 + I \cdot N)^{power}))$. And compute the fragColor fragColor as R_θ * reflectColor + (1- R_θ) * refraColor.