Homework 3 - Report

> **Name：朱驛庭**
> **Student ID：111550093**

Bling-phong Shader

- **Purpose**: Implements the Blinn-Phong lighting model.
- **Implementation**:
  - Calculates ambient, diffuse, and specular lighting.

$$
\begin{aligned}
I_{\text{ambient}} &= k_a \cdot I_a \\
I_{\text{diffuse}} &= k_b \cdot I \cdot (n \cdot l) \\
I_{\text{specular}} &= k_s \cdot I \cdot (n \cdot h)^{\alpha}
\end{aligned}
$$

    - n: normal vector, passed from vertex shader
    - l: light vector, gotten from the light position and fragment position (`lightPos - FragPos`)
    - v: view vector, gotten from the camera position and fragment position (`cameraPos - FragPos`)
    - h: halfway vector between view and light vector $(l + v/\|l + v\|)$
  - Uses the halfway vector for specular highlights.
  - Suitable for shiny surfaces with distinct highlights.

```glsl
#version 330 core

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

out vec3 FragPos;
out vec3 Normal;
out vec2 TexCoord;

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

void main()
{
    // Calculate fragment position in world space
    FragPos = vec3(model * vec4(aPos, 1.0));

    // Transform normal to world space (excluding translation)
    Normal = mat3(transpose(inverse(model))) * aNormal;

    // Calculate final position
    gl_Position = projection * view * model * vec4(aPos, 1.0);

    // Pass texture coordinates to fragment shader
    TexCoord = aTexCoord;
}
```

Vertex shader

Calculate the fragment position and normal vector in worldspace for fragement shader.

```glsl
#version 330 core

in vec3 FragPos;
in vec3 Normal;
in vec2 TexCoord;

out vec4 FragColor;

// Add material properties
uniform sampler2D ourTexture;
uniform vec3 lightPos;
uniform vec3 ambientColor;
uniform vec3 ambientLight;
uniform vec3 diffuseColor;
uniform vec3 diffuseLight;
uniform vec3 specularColor;
uniform vec3 specularLight;
uniform vec3 cameraPos;
uniform float gloss;
void main()
{
    // Retrieve the color from the texture at TexCoord
    vec4 color = texture(ourTexture, TexCoord);

    // Ambient
    vec3 ambient = ambientLight * ambientColor;

    // Diffuse
    vec3 normVec = normalize(Normal);
    vec3 lightVec = normalize(lightPos - FragPos);
    float diff = max(dot(normVec, lightVec), 0.0);
    vec3 diffuse = diffuseLight * diffuseColor * diff;

    // Specular
    vec3 viewVec = normalize(cameraPos - FragPos);
    vec3 halfwayVec = normalize(lightVec + viewVec);
    float spec = pow(max(dot(normVec, halfwayVec), 0.0), gloss);
    vec3 specular = specularLight * specularColor * spec;

    // Combine results
    vec3 result = ambient + diffuse + specular;
    FragColor = clamp(vec4(result, 1.0) * color, 0.0, 1.0);
}
```

Fragment shader

Use the formula mentioned above to calculate for ambient, diffuse and specular. Then sum them all to get the result.

Gouraud Shader

- **Purpose**: Implements Gouraud shading.
- **Implementation**:
  - Calculates lighting per vertex and interpolates across the surface. Nearly the same with Bling-phong shader except for specular caculation.

$$I_{\text{specular}} = k_s \cdot I \cdot (l \cdot v)^{\alpha}$$

  - Less computationally intensive than per-fragment shading.
  - Can result in less accurate specular highlights.

```glsl
#version 330 core

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

// Output the calculated color to be interpolated
out vec3 vertexColor;
out vec2 TexCoord;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
uniform vec3 lightPos;
uniform vec3 ambientColor;
uniform vec3 ambientLight;
uniform vec3 diffuseColor;
uniform vec3 diffuseLight;
uniform vec3 specularColor;
uniform vec3 specularLight;
uniform float gloss;
uniform vec3 cameraPos;

void main()
{
    // Transform vertex position to world space
    vec3 FragPos = vec3(model * vec4(aPos, 1.0));

    // Transform normal to world space
    vec3 normVec = mat3(transpose(inverse(model))) * aNormal;
    normVec = normalize(normVec);

    // Ambient
    vec3 ambient = ambientLight * ambientColor;

    // Diffuse
    vec3 lightVec = normalize(lightPos - FragPos);
    float diff = max(dot(normVec, lightVec), 0.0);
    vec3 diffuse = diffuseLight * diffuseColor * diff;

    // Specular
    vec3 viewVec = normalize(cameraPos - FragPos);
    vec3 reflectionVec = reflect(-lightVec, normVec);
    float spec = pow(max(dot(viewVec, reflectionVec), 0.0), gloss);
    vec3 specular = specularLight * specularColor * spec;

    // Calculate final color at vertex
    vertexColor = ambient + diffuse + specular;

    // Set final position
    gl_Position = projection * view * model * vec4(aPos, 1.0);

    // Pass texture coordinates to fragment shader
    TexCoord = aTexCoord;
}
```

Verrtex shader

Different from Bling-phong shader, the reflection color is calculated in vertex shader then pass to fragment shader.

```glsl
#version 330 core

// Receive interpolated color from vertex shader
in vec3 vertexColor;
in vec2 TexCoord;

out vec4 FragColor;

uniform sampler2D ourTexture;

void main()
{
    // Use the interpolated color directly
    FragColor = vec4(vertexColor, 1.0) * texture(ourTexture, TexCoord);
}
```

Fragment shader

Only need to multiply the relflection color from vertex shader and the texture color.

Cubemap

- **Purpose**: Renders a skybox to simulate an environment surrounding the scene.

- **Implementation**:

```
// Render skybox last
cubemapShader->use();

// Remove translation from view matrix
glm::mat4 viewWithoutTranslation = glm::mat4(glm::mat3(view));

// Set uniforms for cubemap shader
cubemapShader->set_uniform_value("view", viewWithoutTranslation);
cubemapShader->set_uniform_value("projection", projection);

// Draw skybox
glDepthFunc(GL_LEQUAL);  // Change depth function so depth test passes when values are equal to depth buffer's content
glBindVertexArray(cubemapVAO);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_CUBE_MAP, cubemapTexture);
glDrawArrays(GL_TRIANGLES, 0, 36);
glBindVertexArray(0);
glDepthFunc(GL_LESS); // Set depth function back to default

cubemapShader->release();
```

  - **Shader Usage**: Activates the cubemapShader to handle the rendering of the skybox.
  - **Uniforms**: Sets the view and projection matrices as uniforms for the shader.
  - **Depth Function**: Changes the depth function to GL_LEQUAL to ensure the skybox is rendered at the maximum depth.
  - **Drawing**: Binds the cubemap texture and vertex array, then draws the skybox using glDrawArrays.
  - **Reset Depth Function**: Restores the depth function to its default state after rendering.

This setup ensures the skybox is rendered correctly and appears as a distant environment, enhancing the realism of the scene.

```
#version 330 core

layout (location = 0) in vec3 aPos;

out vec3 TexCoords;

uniform mat4 projection;
uniform mat4 view;

void main()
{
    // Cubemap texture coordinates are the same as the vertex positions
    TexCoords = aPos;

    // Remove translation from view matrix by converting to mat3 and back to mat4
    mat4 viewNoTranslation = mat4(mat3(view));

    // Note: The position is multiplied by the projection and view matrix
    // We set w to 1.0 to ensure the skybox stays in the background
    vec4 pos = projection * viewNoTranslation * vec4(aPos, 1.0);
    gl_Position = pos.xyww; // ensure skybox is rendered at maximum depth
}
```

Verrtex shader

Remove translation from view matrix to prevent the skybox rendering affected by the camera pos.

Use pox.xyww to ensure the skybox is rendered at the maximum depth (`z=1`).

```
#version 330 core

in vec3 TexCoords;

out vec4 FragColor;

uniform samplerCube skybox;

void main()
{
    // Sample the cubemap using the texture coordinates
    FragColor = texture(skybox, TexCoords);
}
```

Fragment shader

Use `samplerCube` to get skybox texture, and use TexCoord to get the color.

Metallic Shader

- **Purpose**: Simulates metallic surfaces.
- **Features**:
  - Uses reflection to sample environment colors.
  - Mixes base color with reflection based on a metallic factor.
  - Creates realistic metal-like appearance.

```glsl
#version 330 core

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

out vec3 FragPos;
out vec3 Normal;
out vec2 TexCoord;
out vec3 Reflection;

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

void main()
{
    // Transform vertex position and normal to world space
    FragPos = vec3(model * vec4(aPos, 1.0));
    Normal = mat3(transpose(inverse(model))) * aNormal;

    // Calculate reflection vector
    vec3 viewVec = normalize(FragPos - cameraPos);
    Reflection = reflect(viewVec, normalize(Normal));

    // Pass texture coordinates
    TexCoord = aTexCoord;

    // Set final position
    gl_Position = projection * view * model * vec4(aPos, 1.0);
}
```

Verrtex shader

Use GLSL built in function `reflect` to get the reflection vector of view vector.

```glsl
#version 330 core

in vec3 FragPos;
in vec3 Normal;
in vec2 TexCoord;
in vec3 Reflection;

out vec4 FragColor;

uniform vec3 lightPos;
uniform sampler2D ourTexture;
uniform samplerCube skybox;

void main()
{
    // Get base color from texture
    vec4 modelColor = texture(ourTexture, TexCoord);

    // Calculate Lambertian reflection (diffuse)
    vec3 norm = normalize(Normal);
    vec3 lightVec = normalize(lightPos - FragPos);
    float diff = max(dot(norm, lightVec), 0.0);

    // Add bias to diffuse (as shown in the formula B = Bd + bias)
    float bias = 0.2;
    float B = diff + bias;

    // Sample environment reflection color from skybox
    vec3 reflectColor = vec3(texture(skybox, Reflection));

    // Mix the colors using the metallic ratio (alpha)
    float alpha = 0.4; // Adjust this value to control metallicness
    vec3 finalColor = alpha * B * modelColor.rgb + (1.0 - alpha) * reflectColor;

    // Output final color
    FragColor = vec4(finalColor, 1.0);
}
```

Fragment shader

Based on the formula in spec, calculate diffuse and multiply to model color. Then use reflection vector to get the color of environment from skybox. Finally mix the model color and reflection color by a ratio `alpha = 0.4`.

Glass Shader

- **Purpose**: Simulates glass surfaces using Schlick's approximation and empirical approximation.

$$C_{\text{final}} = R_\theta * C_{\text{reflect}} + (1 - R_\theta) * C_{\text{refract}}$$

- **Implementation:**
  - Calculates reflection and refraction.
  - Uses Fresnel effect to mix reflection and refraction.
  - Provides realistic glass-like appearance with dynamic reflections.
  - The difference between these two approximation methods is the R_θ calculation.

    For Schlick's approximation:

$$R_\theta = R_0 + (1 - R_0)(1 + l \cdot n)^5$$
$$R_0 = \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2$$

    For Empirical approximation:

$$R_\theta = \max(0, \min(1, \text{bias} + \text{scale} \times (1 + l \cdot n)^{\text{power}}))$$

Schlick's approximation

```glsl
#version 330 core

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

out vec3 FragPos;
out vec3 Normal;
out vec2 TexCoord;
out vec3 Reflection;
out vec3 Refraction;

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

const float AIR_COEFF = 1.0;
const float GLASS_COEFF = 1.52;

vec3 calculateReflection(vec3 I, vec3 N) {
    return I - 2.0 * dot(I, N) * N;
}

vec3 calculateRefraction(vec3 I, vec3 N, float eta) {
    float cosI = dot(-I, N);
    float sinT2 = eta * eta * (1.0 - cosI * cosI);
    if (sinT2 > 1.0) return vec3(0.0); // Total internal reflection
    float cosT = sqrt(1.0 - sinT2);
    return eta * I + (eta * cosI - cosT) * N;
}

void main()
{
    // Transform position and normal
    FragPos = vec3(model * vec4(aPos, 1.0));
    Normal = normalize(mat3(transpose(inverse(model))) * aNormal);

    // Calculate view direction
    vec3 viewDir = normalize(FragPos - cameraPos);

    // Calculate reflection and refraction vectors
    Reflection = calculateReflection(viewDir, Normal);
    Refraction = calculateRefraction(viewDir, Normal, AIR_COEFF / GLASS_COEFF);

    // Pass texture coordinates
    TexCoord = aTexCoord;

    gl_Position = projection * view * model * vec4(aPos, 1.0);
}
```

Verrtex shader

Here we implement two function to calculate reflection and refraction by ourselves instead of `reflect()` bulit in function. The formula is based on the spec.

Empirical approximation

```glsl
#version 330 core

in vec3 FragPos;
in vec3 Normal;
in vec2 TexCoord;
in vec3 Reflection;
in vec3 Refraction;

out vec4 FragColor;

uniform vec3 cameraPos;
uniform samplerCube skybox;

const float AIR_COEFF = 1.0;
const float GLASS_COEFF = 1.52;

void main()
{
    vec3 viewDir = normalize(FragPos - cameraPos);

    // Calculate Schlick's approximation
    float ratio = AIR_COEFF / GLASS_COEFF;
    float cosTheta = max(dot(-viewDir, Normal), 0.0);
    float r0 = pow((AIR_COEFF - GLASS_COEFF) / (AIR_COEFF + GLASS_COEFF), 2.0);
    float schlickFactor = r0 + (1.0 - r0) * pow(1.0 - cosTheta, 5.0);

    // Sample both reflection and refraction colors
    vec3 reflectColor = texture(skybox, Reflection).rgb;
    vec3 refractColor = texture(skybox, Refraction).rgb;

    // Mix colors using Schlick approximation
    vec3 finalColor = mix(refractColor, reflectColor, schlickFactor);

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

Fragment shader

Use the formula mentioned above to calculate R_0 and R_θ (`schlickFactor`), finally treated R_θ as a mixture ratio to mix refraction color and reflection color.

```glsl
#version 330 core

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

out vec3 FragPos;
out vec3 Normal;
out vec2 TexCoord;
out vec3 Reflection;
out vec3 Refraction;

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

const float AIR_COEFF = 1.0;
const float GLASS_COEFF = 1.52;

vec3 calculateReflection(vec3 I, vec3 N) {
    return I - 2.0 * dot(I, N) * N;
}

vec3 calculateRefraction(vec3 I, vec3 N, float eta) {
    float cosI = dot(-I, N);
    float sinT2 = eta * eta * (1.0 - cosI * cosI);
    if (sinT2 > 1.0) return vec3(0.0); // Total internal reflection
    float cosT = sqrt(1.0 - sinT2);
    return eta * I + (eta * cosI - cosT) * N;
}

void main()
{
    // Transform position and normal
    FragPos = vec3(model * vec4(aPos, 1.0));
    Normal = normalize(mat3(transpose(inverse(model))) * aNormal);

    // Calculate view direction
    vec3 viewDir = normalize(FragPos - cameraPos);

    // Calculate reflection and refraction vectors
    Reflection = calculateReflection(viewDir, Normal);
    Refraction = calculateRefraction(viewDir, Normal, AIR_COEFF/GLASS_COEFF);

    // Pass texture coordinates
    TexCoord = aTexCoord;

    gl_Position = projection * view * model * vec4(aPos, 1.0);
}
```

Verrtex shader

The same with Schlick's approximation.

```glsl
#version 330 core

in vec3 FragPos;
in vec3 Normal;
in vec2 TexCoord;
in vec3 Reflection;
in vec3 Refraction;

out vec4 FragColor;

uniform vec3 cameraPos;
uniform samplerCube skybox;

// Empirical parameters from the image
const float SCALE = 0.7;
const float POWER = 2.0;
const float BIAS = 0.2;

void main()
{
    vec3 viewDir = normalize(FragPos - cameraPos);
    float cosTheta = max(dot(-viewDir, Normal), 0.0);

    // Calculate empirical factor
    float empiricalFactor = SCALE * pow(cosTheta, POWER) + BIAS;
    empiricalFactor = clamp(empiricalFactor, 0.0, 1.0);

    // Sample both reflection and refraction colors
    vec3 reflectColor = texture(skybox, Reflection).rgb;
    vec3 refractColor = texture(skybox, Refraction).rgb;

    // Mix colors using empirical approximation
    vec3 finalColor = mix(refractColor, reflectColor, empiricalFactor);

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

Fragment shader

Similar to Schilck's approximation, we use the formula mentioned above to calculate R_0 and R_θ (`empiricalFactor`), finally treated R_θ as a mixture ratio to mix refraction color and reflection color.