111550149 林悅揚

ICGHW3

一、程式實作:

基本函式的實作:

TODO1 Set uniform:

```
// TODO 1
// Set uniform value for each shader program

shaderPrograms[shaderProgramIndex]->set_uniform_value("light.position", light.position);
shaderPrograms[shaderProgramIndex]->set_uniform_value("light.ambient", light.ambient);
shaderPrograms[shaderProgramIndex]->set_uniform_value("light.diffuse", light.diffuse);
shaderPrograms[shaderProgramIndex]->set_uniform_value("light.specular", light.specular);

shaderPrograms[shaderProgramIndex]->set_uniform_value("material.ambient", material.ambient);
shaderPrograms[shaderProgramIndex]->set_uniform_value("material.diffuse", material.diffuse);
shaderPrograms[shaderProgramIndex]->set_uniform_value("material.specular", material.specular);
shaderPrograms[shaderProgramIndex]->set_uniform_value("material.gloss", material.gloss);

shaderPrograms[shaderProgramIndex]->set_uniform_value("skybox", 1);

shaderPrograms[shaderProgramIndex]->set_uniform_value("viewPos",glm::vec3(cameraModel[3]));

helicopter.object->render();
shaderPrograms[shaderProgramIndex]->set_uniform_value("model", helicopterBladeModel);
helicopterBlade.object->render();
shaderPrograms[shaderProgramIndex]->release();
```

為當前選擇的著色器程序設置所需的 uniform 值（光源、材質參數、相機位置、Cubemap 紋理單位等），並使用這些設置渲染直升機的機身和旋翼模型，最後釋放該著色器程序。

TODO2 Bling-Phong Shader:

main.cpp src | bling-phong.vert | main.cpp C:\...\Rar$Dla25984.21828.rartemp

src > shaders > bling-phong.vert

```
#version 330 core

// TODO 2
// Implement Bling-Phong shading
layout(location = 0) in vec3 aPos;
layout(location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoord;

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

out vec2 texCoord;
out vec3 FragPos;
out vec3 Normal;

void main()
{
    FragPos = vec3(model * vec4(aPos, 1.0));
    texCoord = aTexCoord;
    Normal = mat3(transpose(inverse(model))) * aNormal;

    gl_Position = projection * view * vec4(FragPos, 1.0);

}
```

main.cpp src | bling-phong.frag | main.cpp C:\...\Rar$Dla25984.21828.rartemp | cubemap.vert | glass_empricial.frag

src > shaders > bling-phong.frag

```
#version 330 core

// TODO 2
// Implement Bling-Phong shading

struct Light {
    vec3 position;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

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

uniform Light light;
uniform Material material;
uniform vec3 viewPos;
uniform sampler2D texture1;

in vec2 texCoord;
in vec3 FragPos;
in vec3 Normal;

out vec4 FragColor;

void main()
{
    vec4 objColor = texture(texture1, texCoord);

    vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(light.position - FragPos);
    vec3 viewDir = normalize(viewPos - FragPos);
    vec3 halfwayDir = normalize(lightDir + viewDir);

    vec3 ambient = light.ambient * material.ambient * objColor.rgb;

    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = light.diffuse * (diff * material.diffuse) * objColor.rgb;

    float spec = pow(max(dot(norm, halfwayDir), 0.0), material.gloss);
    vec3 specular = light.specular * material.specular * spec ;

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

用頂點著色器將頂點的世界座標、法向量和紋理座標計算出來並傳遞給片段著色器，負責準備渲染所需的基本數據。在片段著色器中，實現了 Blinn-Phong 光影模型，主要包含三個部分：環境光、散射光和高光。根據公式，環境光是基於光源和材質的環境屬性混合物體顏色的結果；散射光則使用法向量與光線方向的點積計算表面受到的光線強度，並結合材質的散射屬性；高光部分使用

Blinn-Phong 模型計算觀察者視角與光源方向的中間向量的點積來模擬光滑表面的反射效果，並結合材質的光滑度參數進行調節。最終將這三部分混合，並結合貼圖顏色，生成物體的最終光影顏色輸出到屏幕上。

TODO3 Gouraud Shader:

src > shaders > gouraud.vert

```
#version 330 core

// TODO 3:
// Implement Gouraud shading

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

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

struct Light {
    vec3 position;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
uniform Material material;
uniform Light light;
uniform vec3 viewPos;

out vec4 ambientColor;
out vec4 diffuseColor;
out vec4 specularColor;
out vec2 texCoord;
```

```
void main()
{
    vec3 FragPos = vec3(model * vec4(aPos, 1.0));
    vec3 Normal = normalize(mat3(transpose(inverse(model))) * aNormal);

    vec3 lightDir = normalize(light.position - FragPos);
    vec3 viewDir = normalize(viewPos - FragPos);
    vec3 reflect = normalize(reflect(-lightDir, Normal));

    vec3 ambient = light.ambient * material.ambient;
    float diff = max(dot(Normal, lightDir), 0.0);
    vec3 diffuse = light.diffuse * (diff * material.diffuse);
    float spec = pow(max(dot(reflect, viewDir), 0.0), material.gloss);
    vec3 specular = light.specular * material.specular * spec;

    ambientColor = vec4(ambient, 1.0);
    diffuseColor = vec4(diffuse, 1.0);
    specularColor = vec4(specular, 1.0);

    texCoord = aTexCoord;

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

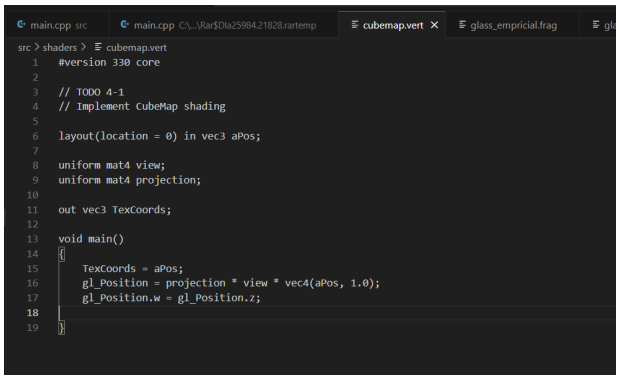
C:\...\Kar3Dla2998\2182\artemp | cubemap.vert | glass_empricial.frag | glass_schlick.frag | glass_schlick.vert

src > shaders > gouraud.frag

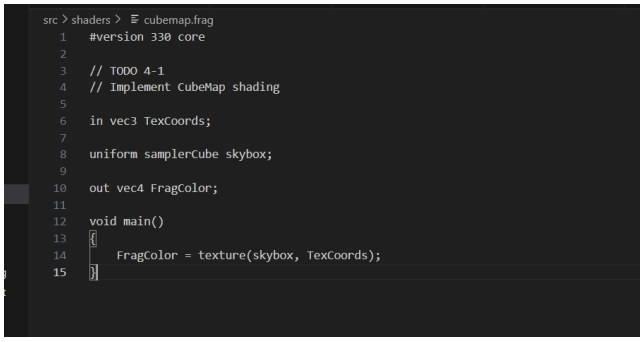
```
#version 330 core

// TODO 3:
// Implement Gouraud shading

uniform sampler2D texture1;

in vec4 ambientColor;
in vec4 diffuseColor;
in vec4 specularColor;

in vec2 texCoord;

out vec4 FragColor;

void main()
{
    vec4 texColor = texture(texture1, texCoord);

    FragColor = ambientColor * texColor + diffuseColor * texColor + specularColor;
}
```

通過將光影計算移至頂點著色器完成。在頂點著色器中，根據 spec 上的公式，計算環境光、散射光和高光三部分的顏色，並將這些結果作為輸出傳遞到片段著色器。在片段著色器中，通過紋理貼圖的取樣顏色，將頂點插值後的光影顏色進行混合，生成最終的像素顏色輸出。

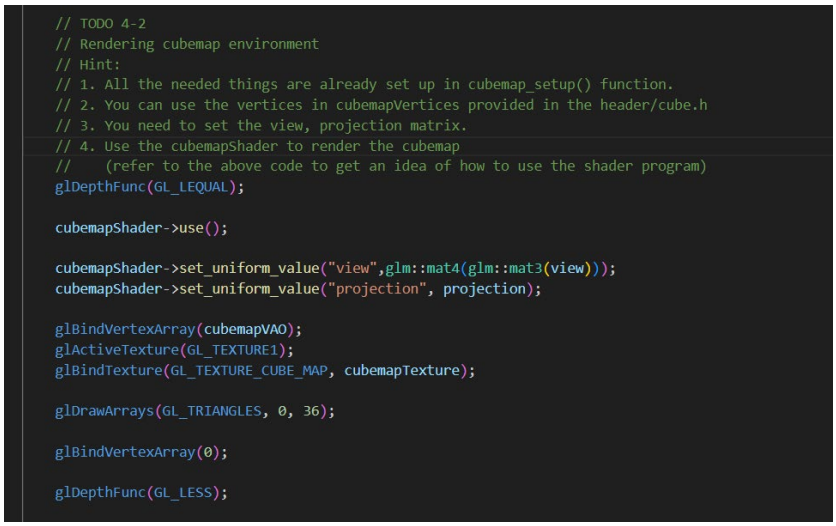
TODO4 Environment Cubemap:



```
src > shaders > cubemap.vert
#version 330 core

// TODO 4-1
// Implement CubeMap shading

layout(location = 0) in vec3 aPos;

uniform mat4 view;
uniform mat4 projection;

out vec3 TexCoords;

void main()
{
    TexCoords = aPos;
    gl_Position = projection * view * vec4(aPos, 1.0);
    gl_Position.w = gl_Position.z;

}
```



```
src > shaders > cubemap.frag
#version 330 core

// TODO 4-1
// Implement CubeMap shading

in vec3 TexCoords;

uniform samplerCube skybox;

out vec4 FragColor;

void main()
{
    FragColor = texture(skybox, TexCoords);
}
```



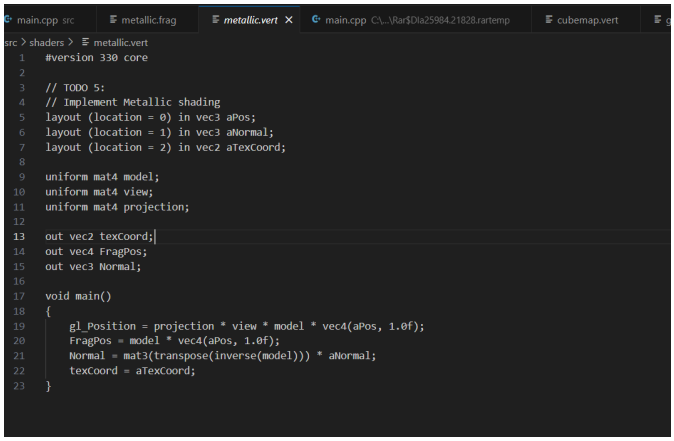
```
// TODO 4-2
// Rendering cubemap environment
// Hint:
// 1. All the needed things are already set up in cubemap_setup() function.
// 2. You can use the vertices in cubemapVertices provided in the header/cube.h
// 3. You need to set the view, projection matrix.
// 4. Use the cubemapShader to render the cubemap
//    (refer to the above code to get an idea of how to use the shader program)
glDepthFunc(GL_LEQUAL);

cubemapShader->use();

cubemapShader->set_uniform_value("view",glm::mat4(glm::mat3(view)));
cubemapShader->set_uniform_value("projection", projection);

glBindVertexArray(cubemapVAO);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_CUBE_MAP, cubemapTexture);

glDrawArrays(GL_TRIANGLES, 0, 36);

glBindVertexArray(0);

glDepthFunc(GL_LESS);
```

透過頂點著色器，將立方體的頂點位置進行投影矩陣和視圖矩陣的變換，計算出裁剪空間中的頂點位置，同時將每個頂點的方向向量傳遞給片段著色器。在片段著色器中，使用這些方向向量從 samplerCube 的立方體貼圖中取樣，獲取對應方向的顏色值，並輸出作為像素顏色。

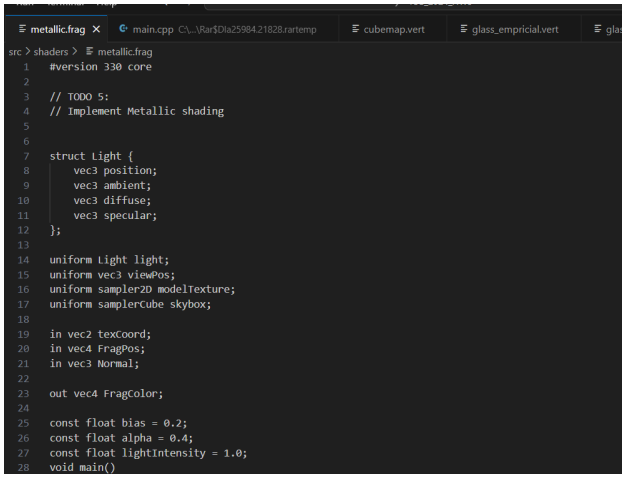
在主程式中，渲染過程先將深度測試模式設置為 GL_LEQUAL，確保天空盒僅在沒有其他物體的地方可見。接著啟用 CubemapShader，傳遞修正過的視圖矩陣和投影矩陣，確保天空盒保持靜止而不隨相機移動。然後綁定立方體的 VAO 和 CubemapTexture 貼圖資源，最後使用 glDrawArrays 渲染立方體的 36 個三角形面。渲染完成後，恢復深度測試模式為 GL_LESS，確保其他場景物體正常渲

染。整體流程實現了立方體天空盒的渲染效果，並且不會影響場景中其他物體的顯示。

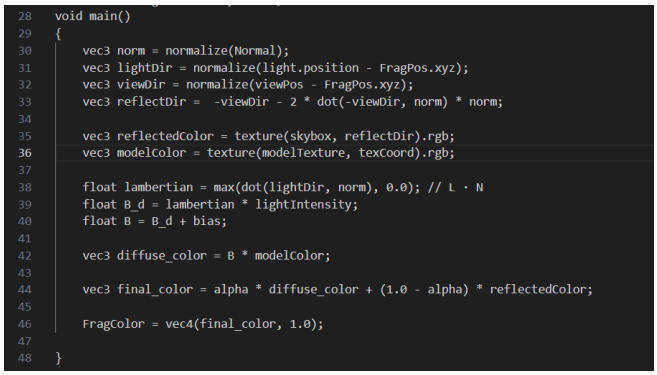
TODO5 Metallic Shader:



```
#version 330 core

// TODO 5:
// Implement Metallic shading
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoord;

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

out vec2 texCoord;
out vec4 FragPos;
out vec3 Normal;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0f);
    FragPos = model * vec4(aPos, 1.0f);
    Normal = mat3(transpose(inverse(model))) * aNormal;
    texCoord = aTexCoord;
}
```



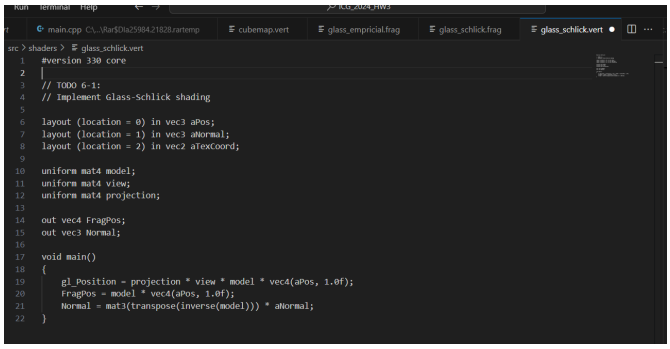
```
#version 330 core

// TODO 5:
// Implement Metallic shading


struct Light {
    vec3 position;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

uniform Light light;
uniform vec3 viewPos;
uniform sampler2D modelTexture;
uniform samplerCube skybox;

in vec2 texCoord;
in vec4 FragPos;
in vec3 Normal;

out vec4 FragColor;

const float bias = 0.2;
const float alpha = 0.4;
const float lightIntensity = 1.0;
void main()
```



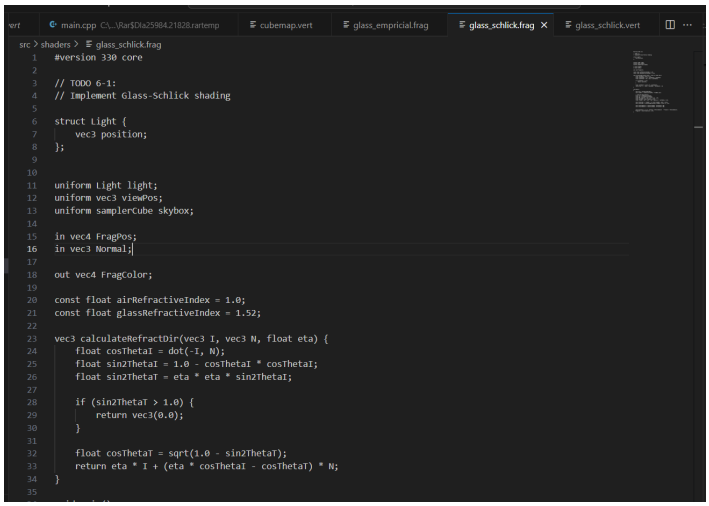
```
void main()
{
    vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(light.position - FragPos.xyz);
    vec3 viewDir = normalize(viewPos - FragPos.xyz);
    vec3 reflectDir =  -viewDir - 2 * dot(-viewDir, norm) * norm;

    vec3 reflectedColor = texture(skybox, reflectDir).rgb;
    vec3 modelColor = texture(modelTexture, texCoord).rgb;

    float lambertian = max(dot(lightDir, norm), 0.0); // L · N
    float B_d = lambertian * lightIntensity;
    float B = B_d + bias;

    vec3 diffuse_color = B * modelColor;

    vec3 final_color = alpha * diffuse_color + (1.0 - alpha) * reflectedColor;

    FragColor = vec4(final_color, 1.0);

}
```

在頂點著色器中，計算頂點的世界座標、法向量和紋理座標，傳遞到片段著色器。片段著色器中，首先使用法向量與觀察方向計算反射方向，從立方體貼圖（Cubemap）中取樣環境反射顏色。然後根據 spec 的公式去計算光線方向與法向量的點積計算 Lambertian 散射，用來模擬表面的基本漫反射效果，並引入偏置（bias）和光強度調整。最後，將漫反射顏色與環境反射顏色按照混合比例（alpha）進行混合，生成最終的材質顏色。

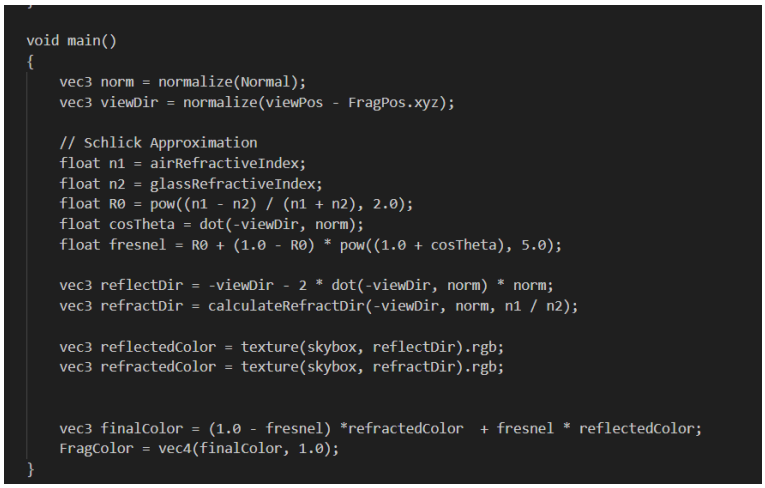
TODO6 Glass Shader- Schlick Approximation:



```glsl
#version 330 core

// TODO 6-1:
// Implement Glass-Schlick shading

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

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

out vec4 FragPos;
out vec3 Normal;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0f);
    FragPos = model * vec4(aPos, 1.0f);
    Normal = mat3(transpose(inverse(model))) * aNormal;
}
```



```glsl
#version 330 core

// TODO 6-1:
// Implement Glass-Schlick shading

struct Light {
    vec3 position;
};


uniform Light light;
uniform vec3 viewPos;
uniform samplerCube skybox;

in vec4 FragPos;
in vec3 Normal;

out vec4 FragColor;

const float airRefractiveIndex = 1.0;
const float glassRefractiveIndex = 1.52;

vec3 calculateRefractDir(vec3 I, vec3 N, float eta) {
    float cosThetaI = dot(-I, N);
    float sin2ThetaI = 1.0 - cosThetaI * cosThetaI;
    float sin2ThetaT = eta * eta * sin2ThetaI;

    if (sin2ThetaT > 1.0) {
        return vec3(0.0);
    }

    float cosThetaT = sqrt(1.0 - sin2ThetaT);
    return eta * I + (eta * cosThetaI - cosThetaT) * N;
}
```

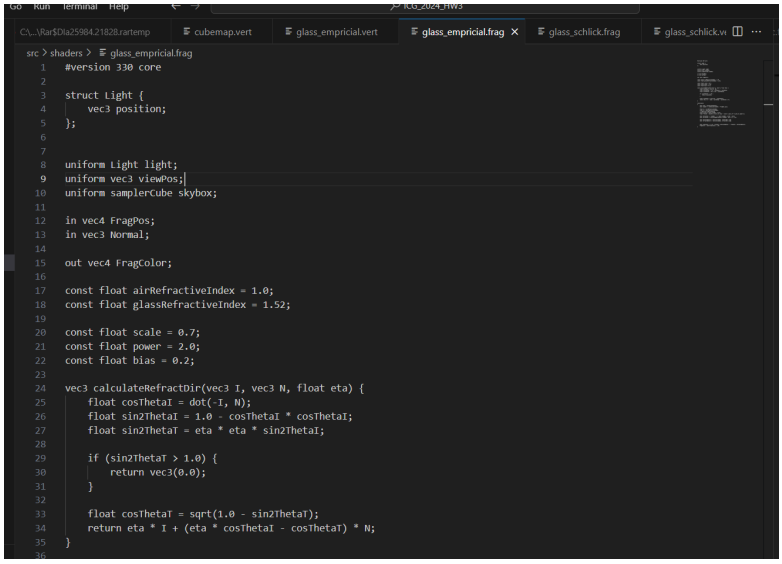


```glsl
void main()
{
    vec3 norm = normalize(Normal);
    vec3 viewDir = normalize(viewPos - FragPos.xyz);

    // Schlick Approximation
    float n1 = airRefractiveIndex;
    float n2 = glassRefractiveIndex;
    float R0 = pow((n1 - n2) / (n1 + n2), 2.0);
    float cosTheta = dot(-viewDir, norm);
    float fresnel = R0 + (1.0 - R0) * pow((1.0 + cosTheta), 5.0);

    vec3 reflectDir = -viewDir - 2 * dot(-viewDir, norm) * norm;
    vec3 refractDir = calculateRefractDir(-viewDir, norm, n1 / n2);

    vec3 reflectedColor = texture(skybox, reflectDir).rgb;
    vec3 refractedColor = texture(skybox, refractDir).rgb;


    vec3 finalColor = (1.0 - fresnel) *refractedColor  + fresnel * reflectedColor;
    FragColor = vec4(finalColor, 1.0);
}
```

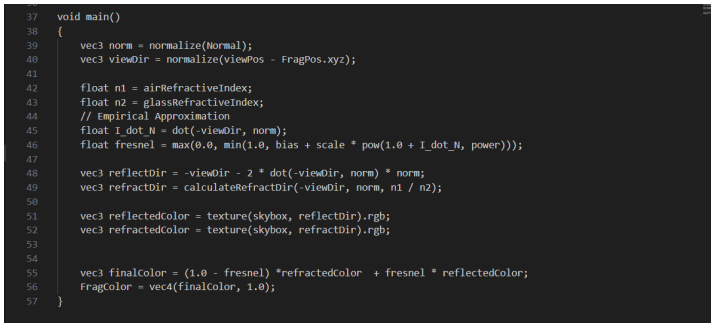
在頂點著色器中，計算頂點的世界座標和法向量，並將其傳遞到片段著色器。片段著色器中，根據觀察方向和法向量計算反射方向與折射方向，並從立方體貼圖（Cubemap）中取樣反射和折射的顏色。使用 Schlick 近似法，根據更新過後的公式去計算入射角和材質折射率，計算 Fresnel 系數來決定反射和折射的混合比例。最終，將兩者按比例混合生成玻璃效果的最終顏色，輸出作為片段顏色。

TODO6 Glass Shader- Empricial Approximation:

```glsl
#version 330 core

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

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

out vec4 FragPos;
out vec3 Normal;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0f);
    FragPos = model * vec4(aPos, 1.0f);
    Normal = mat3(transpose(inverse(model))) * aNormal;
}
```



```glsl
#version 330 core

struct Light {
    vec3 position;
};


uniform Light light;
uniform vec3 viewPos;
uniform samplerCube skybox;

in vec4 FragPos;
in vec3 Normal;

out vec4 FragColor;

const float airRefractiveIndex = 1.0;
const float glassRefractiveIndex = 1.52;

const float scale = 0.7;
const float power = 2.0;
const float bias = 0.2;

vec3 calculateRefractDir(vec3 I, vec3 N, float eta) {
    float cosThetaI = dot(-I, N);
    float sin2ThetaI = 1.0 - cosThetaI * cosThetaI;
    float sin2ThetaT = eta * eta * sin2ThetaI;

    if (sin2ThetaT > 1.0) {
        return vec3(0.0);
    }

    float cosThetaT = sqrt(1.0 - sin2ThetaT);
    return eta * I + (eta * cosThetaI - cosThetaT) * N;
}
```



```glsl
void main()
{
    vec3 norm = normalize(Normal);
    vec3 viewDir = normalize(viewPos - FragPos.xyz);

    float n1 = airRefractiveIndex;
    float n2 = glassRefractiveIndex;
    // Empirical Approximation
    float I_dot_N = dot(-viewDir, norm);
    float fresnel = max(0.0, min(1.0, bias + scale * pow(1.0 + I_dot_N, power)));

    vec3 reflectDir = -viewDir - 2 * dot(-viewDir, norm) * norm;
    vec3 refractDir = calculateRefractDir(-viewDir, norm, n1 / n2);

    vec3 reflectedColor = texture(skybox, reflectDir).rgb;
    vec3 refractedColor = texture(skybox, refractDir).rgb;


    vec3 finalColor = (1.0 - fresnel) *refractedColor  + fresnel * reflectedColor;
    FragColor = vec4(finalColor, 1.0);
}
```

在頂點著色器中，計算頂點的世界座標和法向量，並傳遞到片段著色器。在片段著色器中，根據觀察方向與法向量，計算光線的反射方向和折射方向。折射方向使用輔助函數 calculateRefractDir 計算，並根據折射率比值模擬光線的折射行為，同時處理全內反射的特殊情況。

程式採用了更新過後的公式，計算反射與折射的混合比例，通過引入偏置（bias）、縮放（scale）和指數（power）控制光線反射的強度和過渡效果。反射與折射方向分別從環境貼圖中取樣對應顏色，最後根據混合比例計算出玻璃材質的最終顏色。

二、問題及解決:

1.一開始在實作 glass 時，不管怎麼做都跟 demo 影片差一點，但是最後自己最後網路上的公式後，向助教確認，才發現是 spec 的公式有小差錯，最後也成功寫出來了頗有成就感。

2.在相機那邊跟助教的圖差了一點，後來跟室友確認才發現是自己的 view 設定錯誤了。