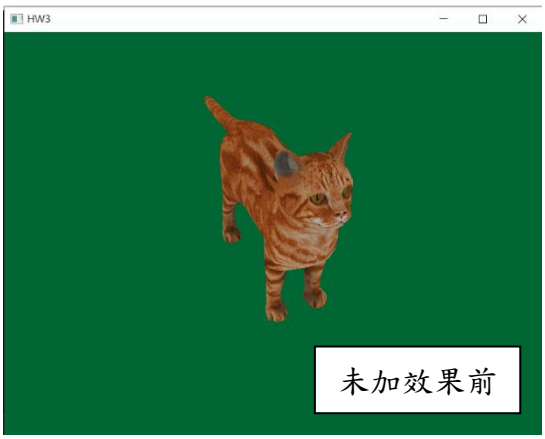
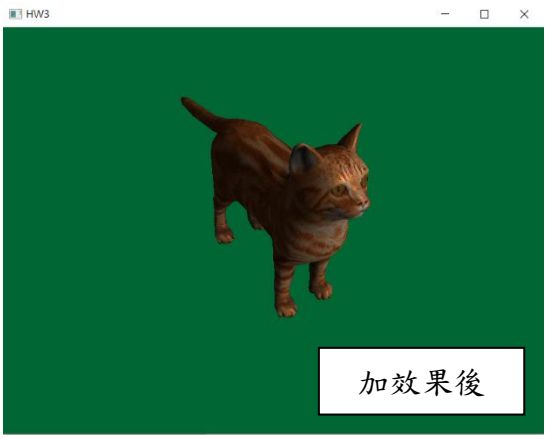
ICG Homework3

運管 11 0713216 龔祐萱

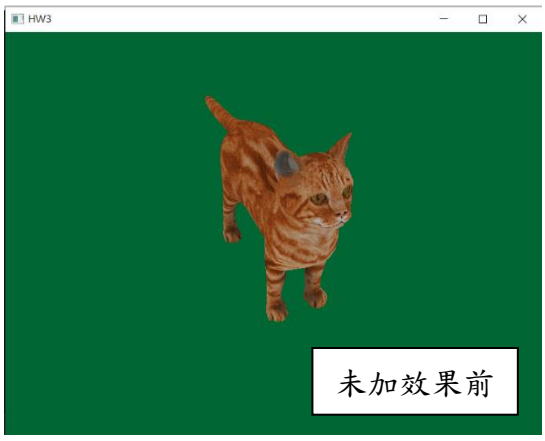
一、效果與觸發建

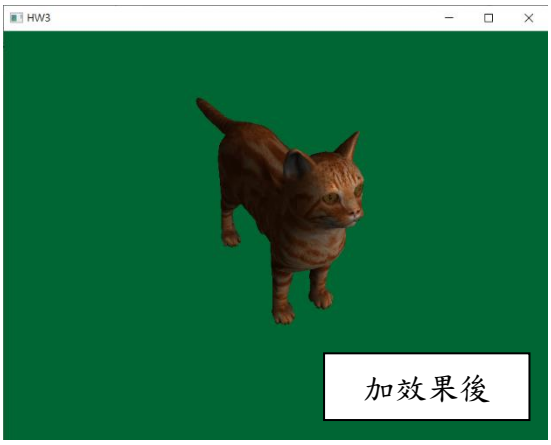
1. 按鍵 1 -> Phong shading



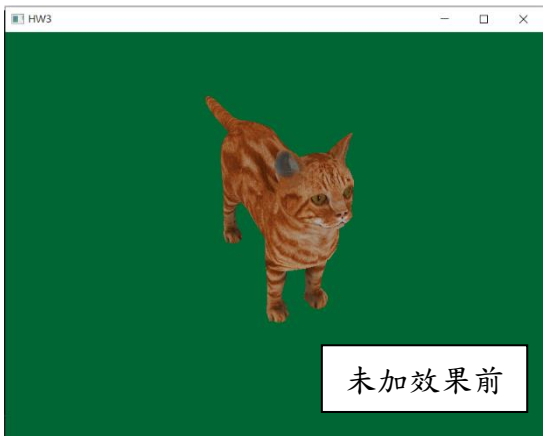


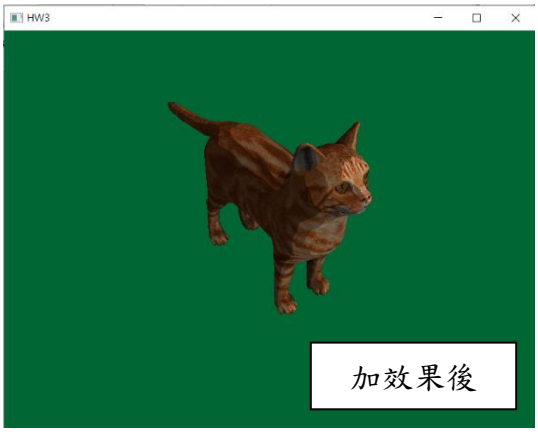
2. 按鍵 2 -> Gouraud shading



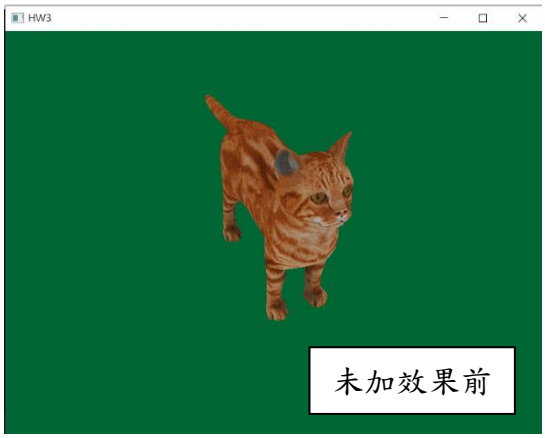


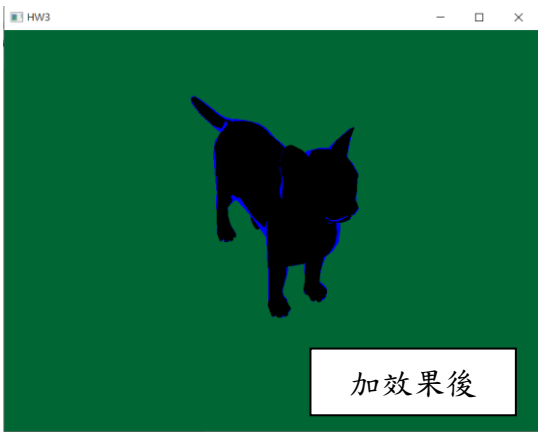
3. 按鍵 3 -> Toon shading





4. 按鍵 4 -> Edge effect





二、實作說明

1. Phong shading

Phong shading 所計算光的方法為將每個 vertex 的 normal 進行內插後輸至 Fragment shader，再由 fragment shader 計算 Ambient light、Diffuse light 以及 Specular light，最後與 texture 一起輸出顏色。

- Vertex shader

(1) 把物件的 vertex 位置、normal 以及 texCoord 輸入 vertex shader，並將物件的 model matrix、view matrix 以及 perspective matrix 以 uniform 的形式輸到 vertex shader。
(2) 將 vertex 乘上 model matrix、view matrix 以及 perspective matrix 使其可以在畫面中移動，並將 texCoord 輸出到 fragment shader。
(3) 將物件的 vertex 位置乘上 model matrix 計算物件在世界座標中的位置。
(4) 計算物件每個 vertex 在世界座標中的 normal，並送到 fragment shader。

```
void main()
{
    gl_Position = P * V * M * vec4(aPos, 1.0);
    texCoord = aTexCoord;
    WorldPos = M * vec4(aPos, 1.0);
    //Normal = aNormal;
    mat4 normal_transform = transpose(inverse(M));
    Normal = normalize((normal_transform * vec4(aNormal, 0.0)).xyz);;
}
```

- Fragment shader

(1) 將 normal 進行標準化，變成單位向量。
(2) 計算 light vector，為光的位置減去物件所在的位置，並將其標準化，變成單位向量。
(3) 計算 view vector，為相機的位置減去物件所在的位置，並將其標準化，變成單位向量。
(4) 計算 reflect vector，為物件反射光的向量，使用 GLSL 中內建的 reflect() 實作。
(5) 將 Ambient light 分成 RGB 分量分別進行計算，計算公式為 Ambient light = Ka (Ambient reflectivity) * Ia (Ambient intensity)。
(6) 將 Diffuse light 分成 RGB 分量分別進行計算，計算公式為 Diffuse light = Kd (Diffuse reflectivity) * Id (Diffuse intensity)。
(7) 將 Specular light 分成 RGB 分量分別進行計算，計算公式為 Specular light = Ks (Specular reflectivity) * Is (Specular intensity)。
(8) 將各個 RGB 分量的 Ambient light、Diffuse light 以及 Specular light 相加，變成 Phong shading，並物件的 texture 一起輸出為 FragColor。

```
void main(){

    vec3 n = normalize(Normal);
    vec3 l = (normalize(vec4(lightPos, 1.0) - WorldPos)).xyz;
    vec3 v = normalize(cameraPos - WorldPos.xyz);
    vec3 r = reflect(-l, n);

    //Ambient light
    float Ambient_light_r =   Ambient_int.x * Ambient_ref.x ;
    float Ambient_light_b =   Ambient_int.y * Ambient_ref.y ;
    float Ambient_light_g =   Ambient_int.z * Ambient_ref.z ;

    //Diffuse light
    float Diffuse_light_r =   Diffuse_int.x * Diffuse_ref.x * max(dot(l, n), 0.0);
    float Diffuse_light_g =   Diffuse_int.y * Diffuse_ref.y * max(dot(l, n), 0.0);
    float Diffuse_light_b =   Diffuse_int.z * Diffuse_ref.z * max(dot(l, n), 0.0);

    //Specular light
    float Specular_light_r =  Specular_int.x * Specular_ref.x * pow(max(dot(v, r), 0.0), Gloss);
    float Specular_light_g =  Specular_int.y * Specular_ref.y * pow(max(dot(v, r), 0.0), Gloss);
    float Specular_light_b =  Specular_int.z * Specular_ref.z * pow(max(dot(v, r), 0.0), Gloss);


    vec3 result = vec3(Ambient_light_r + Diffuse_light_r + Specular_light_r,
                       Ambient_light_g + Diffuse_light_g + Specular_light_g,
                       Ambient_light_b + Diffuse_light_b + Specular_light_b);

    FragColor = vec4(result, 1.0) * texture(Texture, texCoord);

}
```

2. Gouraud shading

- Vertex shader

Gouraud shading 的 vertex shader 為 Phong shading 的 vertex shader 和 fragment shader 的綜合，因為 Gouraud shading 為先計算好顏色在進行內插，因此會在 vertex shader 中計算物件的 Ambient light、Diffuse light 以及 Specular light，在輸出到 fragment shader 中輸出顏色，而計算 Ambient light、Diffuse light 以及 Specular light 的公式與 Phong shading 相同。

```
void main()
{
    gl_Position = P * V * M * vec4(aPos, 1.0);
    texCoord = aTexCoord;
    vec4 WorldPos = M * vec4(aPos, 1.0);
    mat4 normal_transform = transpose(inverse(M));
    vec3 Normal = normalize((normal_transform * vec4(aNormal, 0.0)).xyz);

    vec3 n = normalize(Normal);
    vec3 l = (normalize(vec4(lightPos, 1.0) - WorldPos)).xyz;
    vec3 v = normalize(cameraPos - WorldPos.xyz);
    vec3 r = reflect(-l, n);

    //Ambient light
    float Ambient_light_r =   Ambient_int.x * Ambient_ref.x ;
    float Ambient_light_b =   Ambient_int.y * Ambient_ref.y ;
    float Ambient_light_g =   Ambient_int.z * Ambient_ref.z ;

    //Diffuse light
    float Diffuse_light_r =   Diffuse_int.x * Diffuse_ref.x * max(dot(l, n), 0.0);
    float Diffuse_light_g =   Diffuse_int.y * Diffuse_ref.y * max(dot(l, n), 0.0);
    float Diffuse_light_b =   Diffuse_int.z * Diffuse_ref.z * max(dot(l, n), 0.0);

    //Specular light
    float Specular_light_r =  Specular_int.x * Specular_ref.x * pow(max(dot(v, r), 0.0), Gloss);
    float Specular_light_g =  Specular_int.y * Specular_ref.y * pow(max(dot(v, r), 0.0), Gloss);
    float Specular_light_b =  Specular_int.z * Specular_ref.z * pow(max(dot(v, r), 0.0), Gloss);


    result = vec3(Ambient_light_r + Diffuse_light_r + Specular_light_r,
                  Ambient_light_g + Diffuse_light_g + Specular_light_g,
                  Ambient_light_b + Diffuse_light_b + Specular_light_b);

}
```

- Fragment shader

將 vertex shader 所輸出的顏色和物件的 texture 一起輸出到畫面上。

```
void main(){

    FragColor = vec4(result, 1.0) * texture(Texture, texCoord);

}
```

3. Toon shading

Toon shading 為依照 light vector dot normal vector 的大小來決定要給與物件甚麼顏色或甚麼強度的光源，以產生一層一層陰影的效果。

- Vertex shader

Toon shading 的 vertex shader 與 Phong shading 的 vertex shader 相同，分別會計算物件在畫面中移動的位置、輸出 texCoord、計算物件的世界座標位置、計算 normal。

```
void main()
{
    gl_Position = P * V * M * vec4(aPos, 1.0);
    texCoord = aTexCoord;
    WorldPos = M * vec4(aPos, 1.0);
    mat4 normal_transform = transpose(inverse(M));
    Normal = normalize((normal_transform * vec4(aNormal, 0.0)).xyz);;
}
```

- Fragment shader

(1) 計算 normal vector 並將其標準化變成單位向量。

(2) 使用 light vector 和 normal vector 做內積，計算光線的強度，並依照光線的強度給予不同程度的強度數值，使一個區間內的光線強度最後輸出的強度相同，產生一層一層陰影的效果。

```
void main(){

    vec3 n = normalize(Normal);
    vec3 l = (normalize(vec4(lightPos, 1.0) - WorldPos)).xyz;

    float intensity = dot(vec3(lightPos), n);
    if(intensity  > 15.0)
        intensity = 1.1;
    else if(intensity  > 7.0)
        intensity = 0.7;
    else if(intensity  > 3.0)
        intensity = 0.5;
    else
        intensity = 0.3;

    FragColor =  intensity *  texture(Texture, texCoord);

}
```

4. Edge effect

Edge effect 採用 view vector 和 normal vector 的內積來決定邊界的顏色，從內積的公式可知道，如果 view vector 和 normal vector 之間的夾角越接近 90 度，則內積出來的數值會越接近 0，也會越接近可看見的邊界，因此可以設定一個數

值，如果小於該數值則輸出藍色，其他數值輸出黑色，就可以達到 edge effect 的效果。

- Vertex shader

Edge shading 的 vertex shader 與 Phong shading 的 vertex shader 相同，分別會計算物件在畫面中移動的位置、輸出 texCoord、計算物件的世界座標位置、計算 normal。

```
void main()
{
    gl_Position = P * V * M * vec4(aPos, 1.0);
    texCoord = aTexCoord;
    WorldPos = M * vec4(aPos, 1.0);
    mat4 normal_transform = transpose(inverse(M));
    Normal = normalize((normal_transform * vec4(aNormal, 0.0)).xyz);;
}
```

- Fragment shader

使用標準化的 view vector 和 normal vector 做內插，如果內插數值小於 0.2，則輸出藍色；否則輸出黑色，以呈現邊線的效果。

```
void main(){

    vec3 n = normalize(Normal);
    vec3 l = (normalize(vec4(lightPos, 1.0) - WorldPos)).xyz;
    vec3 v = normalize(cameraPos - WorldPos.xyz);
    vec3 r = reflect(-l, n);

    float dot_product = dot(v, n);
    if(dot_product < 0.2)
        FragColor = vec4(0.0, 0.0, 1.0, 1.0);
    else
        FragColor = vec4(0.0, 0.0, 0.0, 1.0);

}
```

三、遇見的問題與解決方法

這次作業基本上沒遇到什麼問題，卡比較久的問題是將各個光的 reflectivity 和 intensity 作為 uniform 傳送到 vertex shader 和 fragment shader 時，發現無法順利地傳送過去，造成在執行程式時無法跑出正確的畫面。

起初是使用 glUniformMatrix3fv(glGetUniformLocation(Program,

"Ambitne_ref"), 1, GL_FALSE, glm::value_ptr(Ambient_ref));的形式作為 uniform 傳到 shader，後來上網查詢相關的 function 解釋後發現，上述的 functiong 是傳送 mat3 的，而要傳送的數值為 vec3 的形式，因此就將上述的程式改為 glUniform3fv(glGetUniformLocation(Program, "Ambient_ref"), 1, glm::value_ptr(Ambient_ref));，就可以正確地執行程式了。