109550175 許登豪 HW3 report

1. 實作方式 ：

a. Line 19

這是函式宣告, DrawModel 傳入 render 貓咪所需要的 model , view , perspective 的 matrix , 當下 render 這一幀畫面所要用的 shader program , 貓咪的 VAO , 以及貓咪的 texture。

```
void DrawModel(glm::mat4 M, glm::mat4 V, glm::mat4 P, unsigned int program , unsigned int VAO , unsigned int texture);
```

b. Line 31

這是整支程式需要的全域變數 . 用以 identify 按鍵被按下時所呈現的對應 shading 方式 , 1 , 2 , 3 , 4 分別為 Phong , Gouraud , Toon 及 Edge。

```
int Effect_num = 1;
```

c. Line 33 – Line 91

首先先 use 這一幀要用的對應的 program , 再傳入每一種 shading 方式都需要傳入的 4 個 uniform 變數 Model , View , Perspective , 以及 Texture。

接下來就是基於不同的 shading 方式 , 要傳入額外的 uniform 變數進 vertex 或 fragment shader 內。

如果是 Phong 或是 Gouraud , 則需要傳入物體的 Ka , Kd , Ks 以及 gloss , 還有光線的 La , Ld , Ls 以及 light pos , camera pos 以用來

算 phong shading model 所需要的向量 L , V 以及 R。

如果是 Toon shading , 則只需要傳入物體的 Kd , 以及 light pos 以用來算所需要的向量 L。

如果是 Edge effect , 則只需要傳入 camera 的位置即可。

最後再 active texture 以及 bind 要用的 texture 及 VAO , 最後畫完這一幀再傳入 glUseProgram(0)以暫停使用此 shader program 即可。

```
void DrawModel(glm::mat4 M, glm::mat4 V, glm::mat4 P, unsigned int program , unsigned int VAO , unsigned int texture) {
    glUseProgram(program);

    unsigned int Modelloc, Viewloc, Perspectiveloc;
    Modelloc = glGetUniformLocation(program, "Model");
    Viewloc = glGetUniformLocation(program, "View");
    Perspectiveloc = glGetUniformLocation(program, "Perspective");
    glUniformMatrix4fv(Modelloc, 1, GL_FALSE, glm::value_ptr(M));
    glUniformMatrix4fv(Viewloc, 1, GL_FALSE, glm::value_ptr(V));
    glUniformMatrix4fv(Perspectiveloc, 1, GL_FALSE, glm::value_ptr(P));
    glUniform1i(glGetUniformLocation(program, "Texture"), 0);
```

```
if (Effect_num == 1 | Effect_num == 2) {
    unsigned int mambientloc, mdiffuseloc, mspecularloc, mglossloc;
    mambientloc = glGetUniformLocation(program, "ambient_m");
    mdiffuseloc = glGetUniformLocation(program, "diffuse_m");
    mspecularloc = glGetUniformLocation(program, "specular_m");
    mglossloc = glGetUniformLocation(program, "gloss");
    glUniform3fv(mambientloc, 1 , glm::value_ptr(material.ambient));
    glUniform3fv(mdiffuseloc, 1 , glm::value_ptr(material.diffuse));
    glUniform3fv(mspecularloc, 1, glm::value_ptr(material.specular));
    glUniform1f(mglossloc , material.gloss);

    unsigned int lambientloc, ldiffuseloc, lspecularloc, lposloc;
    lambientloc = glGetUniformLocation(program, "ambient_l");
    ldiffuseloc = glGetUniformLocation(program, "diffuse_l");
    lspecularloc = glGetUniformLocation(program, "specular_l");
    lposloc = glGetUniformLocation(program, "light_pos");
    glUniform3fv(lambientloc, 1, glm::value_ptr(light.ambient));
    glUniform3fv(ldiffuseloc, 1, glm::value_ptr(light.diffuse));
    glUniform3fv(lspecularloc, 1, glm::value_ptr(light.specular));
    glUniform3fv(lposloc, 1, glm::value_ptr(light.position));

    unsigned int cameraloc;
    cameraloc = glGetUniformLocation(program, "camera_pos");
    glUniform3fv(cameraloc, 1, glm::value_ptr(cameraPos));
}
```

```
else if (Effect_num == 3) {
    unsigned int lposloc, mdiffuseloc;
    lposloc = glGetUniformLocation(program, "light_pos");
    glUniform3fv(lposloc, 1, glm::value_ptr(light.position));
    mdiffuseloc = glGetUniformLocation(program, "diffuse_m");
    glUniform3fv(mdiffuseloc, 1, glm::value_ptr(material.diffuse));
}
else {
    unsigned int cameraloc;
    cameraloc = glGetUniformLocation(program, "camera_pos");
    glUniform3fv(cameraloc, 1, glm::value_ptr(cameraPos));
}
```

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture);
glBindVertexArray(VAO);
glDrawArrays(GL_TRIANGLES, 0, catModel->positions.size());
glBindVertexArray(0);

glUseProgram(0);
```

d. Line 122 – Line 146

創各個 shading 方式需要的 vertex shader , fragment shader 以及 shaderProgram。

```
// TODO:
// Create shaders
unsigned int vertexShader_Phong, fragmentShader_Phong;
unsigned int shaderProgram_Phong;
vertexShader_Phong = createShader("shaders/Phong.vert", "vert");
fragmentShader_Phong = createShader("shaders/Phong.frag", "frag");
shaderProgram_Phong = createProgram(vertexShader_Phong, fragmentShader_Phong);

unsigned int vertexShader_Gouraud, fragmentShader_Gouraud;
unsigned int shaderProgram_Gouraud;
vertexShader_Gouraud = createShader("shaders/Gouraud.vert", "vert");
fragmentShader_Gouraud = createShader("shaders/Gouraud.frag", "frag");
shaderProgram_Gouraud = createProgram(vertexShader_Gouraud, fragmentShader_Gouraud);

unsigned int vertexShader_Toon, fragmentShader_Toon;
unsigned int shaderProgram_Toon;
vertexShader_Toon = createShader("shaders/Toon.vert", "vert");
fragmentShader_Toon = createShader("shaders/Toon.frag", "frag");
shaderProgram_Toon = createProgram(vertexShader_Toon, fragmentShader_Toon);

unsigned int vertexShader_Edge, fragmentShader_Edge;
unsigned int shaderProgram_Edge;
vertexShader_Edge = createShader("shaders/Edge.vert", "vert");
fragmentShader_Edge = createShader("shaders/Edge.frag", "frag");
shaderProgram_Edge = createProgram(vertexShader_Edge, fragmentShader_Edge);
```

e. 創貓咪要用的 VAO , VBO , VBO[3]分別給予貓咪每個 vertex 的位置 , normal 以及 texture coordinate。

```
// VAO, VBO
unsigned int VAO, VBO[3];
glGenVertexArrays(1, &VAO);
glBindVertexArray(VAO);
glGenBuffers(3, VBO);

glBindBuffer(GL_ARRAY_BUFFER, VBO[0]);
glBufferData(GL_ARRAY_BUFFER, sizeof(GL_FLOAT) * (catModel->positions.size()), &(catModel->positions[0]), GL_STATIC_DRAW);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(GL_FLOAT) * 3, 0);
glEnableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, 0);

glBindBuffer(GL_ARRAY_BUFFER, VBO[1]);
glBufferData(GL_ARRAY_BUFFER, sizeof(GL_FLOAT) * (catModel->normals.size()), &(catModel->normals[0]), GL_STATIC_DRAW);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(GL_FLOAT) * 3, 0);
glEnableVertexAttribArray(1);
glBindBuffer(GL_ARRAY_BUFFER, 0);

glBindBuffer(GL_ARRAY_BUFFER, VBO[2]);
glBufferData(GL_ARRAY_BUFFER, sizeof(GL_FLOAT) * (catModel->texcoords.size()), &(catModel->texcoords[0]), GL_STATIC_DRAW);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(GL_FLOAT) * 2, 0);
glEnableVertexAttribArray(2);
glBindBuffer(GL_ARRAY_BUFFER, 0);

glBindVertexArray(0);
```

f. Line 185 – Line 210

Render Loop , 每次要 render 這一幀時都先清空 color buffer , 並且拿到 model , view 跟 perspective , 最後根據全域變數 Effect_num 看現在採用的 shading 方式是什麼 , 丟入對應的 matrices 跟 shaderProgram 及 VAO , texture 進 DrawModel 這個 function 裡面。

```
while (!glfwWindowShouldClose(window))
{
    glClearColor(0.0f, 0.4f, 0.2f, 0.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // TODO:
    // Draw the cat with current active shader
    glm::mat4 M = glm::rotate(glm::mat4(1.0f), (float)glfwGetTime() * glm::radians(45.0f), glm::vec3(0, 1, 0));
    glm::mat4 V = getView();
    glm::mat4 P = getPerspective();
    if (Effect_num == 1) {
        DrawModel(M, V, P, shaderProgram_Phong , VAO, catTexture);
    }
    else if (Effect_num == 2) {
        DrawModel(M, V, P, shaderProgram_Gouraud, VAO, catTexture);
    }
    else if (Effect_num == 3) {
        DrawModel(M, V, P, shaderProgram_Toon, VAO, catTexture);
    }
    else {
        DrawModel(M, V, P, shaderProgram_Edge, VAO, catTexture);
    }

    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

g. Line 216 – Line 238

按下 1 , 2 , 3 , 4 更改全域變數 Effect_num 以採用不同的 shading 效果。

```
void keyCallback(GLFWwindow* window, int key, int scancode, int action, int mods)
{
    if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);
    if (key == GLFW_KEY_1 && action == GLFW_PRESS) {
        Effect_num = 1;
    }

    if (key == GLFW_KEY_2 && action == GLFW_PRESS) {
        Effect_num = 2;
    }

    if (key == GLFW_KEY_3 && action == GLFW_PRESS) {
        Effect_num = 3;
    }

    if (key == GLFW_KEY_4 && action == GLFW_PRESS) {
        Effect_num = 4;
    }

}
```

h. Phong vert. and Phong frag.

在 vertex shader 裡面 ， 宣告 3 個 layout , 分別代表藉由 VAO + VBO 傳入的 3 個 attribute position , normal 以及 texture coordinate。

宣告 3 個 uniform 變數分別為 Model , View 以及 Perspective matrix , 宣告 out 變數 texCoord 的原因在於要在 fragment shader 做 texture 的上色 ， 值直接往下送即可 ； 宣告 out 變數 Normal 的原因在於計算 Phong Model 時要使用到每個點的 normal , 但不可直接乘上 Model matrix 就往下送 ， 可能會導致在 world space 下(乘完 model matrix 後)normal 跟點並非垂直。所以要做以下操作(公式 from 講義投影片 ， 原理是找到一個 matrix 使得 normal , vertex 同時乘以 model matrix 後仍能保持垂直) ; 宣告 out 變數 Pos 的原因在於計算 Phong Model 時要使用到每個點的位置 ， 但不可直接往下送 ， 因為

```
#version 330 core

// TODO:
// Implement 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 Perspective ;

out vec2 texCoord;
out vec3 Normal;
out vec3 Pos;
void main()
{
    gl_Position = Perspective * View * Model * vec4(aPos, 1.0);
    texCoord = aTexCoord;
    Normal = mat3(transpose(inverse(Model))) * aNormal;
    Pos = mat3(Model) * aPos;
}
```

在 model->position 裡面的數值都是在 model space 上的(local)，所以要將 vertex 轉到跟 camera pos , light pos 一樣的 world space 上 ， 這樣計算向量 L , V 等才正確 ，所以要乘上 Model matrix 去做轉換。

```
#version 330 core

uniform sampler2D Texture;
in vec2 texCoord;
in vec3 Normal;
in vec3 Pos;

out vec4 fragColor ;

uniform vec3 ambient_m;
uniform vec3 diffuse_m;
uniform vec3 specular_m;
uniform float gloss;

uniform vec3 ambient_l;
uniform vec3 diffuse_l;
uniform vec3 specular_l;
uniform vec3 light_pos;

uniform vec3 camera_pos;

void main()
{
    vec3 L = normalize(light_pos - Pos);
    vec3 V = normalize(camera_pos - Pos);
    vec3 N = normalize(Normal);
    vec3 R = normalize(2*(dot(L , N))*N-L) ;

    vec3 Ka = ambient_m;
    vec3 Kd = diffuse_m;
    vec3 Ks = specular_m;

    vec3 La = ambient_l;
    vec3 Ld = diffuse_l;
    vec3 Ls = specular_l;

    vec4 object_color = texture(Texture, texCoord);
    vec3 ambient = La * Ka * vec3(object_color.x , object_color.y , object_color.z);
    vec3 diffuse = Ld * Kd * vec3(object_color.x , object_color.y , object_color.z) * max(dot(L,N) , 0);
    vec3 specular = Ls * Ks *  pow(max(dot(V , R) , 0) , gloss) ;
    fragColor = vec4(ambient + diffuse + specular , 1.0);
}
```

在 fragment shader 裡面宣告 3 個由 vertex shader 傳下來的 in 變數 ，1 個 texture sampler 以及要輸出的 1 個 out 變數 fragColor。

接下來就宣告對應於 Ka , Kd , Ks , La , Ld , Ls , alpha 以及計算向量需要的一些 uniform 變數。其中 L 在 phong model 指的是”物體上的某一點指向光源的單位向量”所以是拿 light_pos – Pos , 注意一下這個傳

下來的 Pos 跟 light_pos 都在 world space 上，所以直接相減沒有問題。其中 V 在 phong model 指的是”物體上的某一點指向眼睛(觀察處)的單位向量”所以是拿 camera_pos – Pos，注意一下這個傳下來的 Pos 跟 camera_pos 都在 world space 上，所以直接相減沒有問題。其中 N 在 phong model 指的是”物體上的某一點的單位法向量”所以 normalize 傳下來的變數即可。其中 R 在 phong model 指的是”L 的完美鏡反射方向的單位向量”。公式套用第 5 章講義的第 16 頁，此公式原理在於向量 L，向量 R 的相加可以畫出一個”菱形”，而菱形的對角線的一半長度剛好會是向量 L 跟物體上一點法向量 N 的內積值。接下來幾行就是純粹套用公式，要注意的是計算 diffuse 和 specular 的部分要分別考慮到 L,N 的夾角有沒有超過 90 度(又因為 L , N 是單位向量，內積值等於 cos 夾角值)以及 V , R 的夾角有沒有超過 90 度，而 cos 90 度 = 0，所以才會有以下 max 的寫法，當然如果上述某一夾角大於 90 度就可以割捨對應的那一項，因此整項乘上 0 等於 0 (0 的 gloss 次方會是 0)，意即忽略這一項對於呈色的影響。可以探討的一點是因為著色是三個顏色通道有各自的 Ka , Kd , Ks, La , Ld 等等，而 glsl 的 vector 相乘正好是 element wise 的，所以可以直接這樣寫。最後把 fragColor 打包成一個四維向量(最後一項透明度給 1 即可)就大功告成。

i. Gouraud vert. and Gouraud frag.

這裡跟 phong 的區別僅在於把各個 shading 考慮到的要素如 ambient , diffuse , specular 都先在 vertex shader 算好往下送 , (所以要宣告這些 out 變數) , fragment shader 的各點拿到內插後的這些要素值並上色 , 所以 vertex shader 這邊便不贅述。

```
#version 330 core

// TODO:
// Implement Gouraud 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 Perspective ;

uniform vec3 ambient_m;
uniform vec3 diffuse_m;
uniform vec3 specular_m;
uniform float gloss;

uniform vec3 ambient_l;
uniform vec3 diffuse_l;
uniform vec3 specular_l;
uniform vec3 light_pos;

uniform vec3 camera_pos;

out vec3 ambient ;
out vec3 diffuse ;
out vec3 specular ;
out vec2 texCoord;

void main()
{
    gl_Position = Perspective * View * Model * vec4(aPos, 1.0);
    texCoord = aTexCoord;
    vec3 Normal = mat3(transpose(inverse(Model))) * aNormal;
    vec3 Pos = mat3(Model) * aPos;

    vec3 L = normalize(light_pos - Pos);
    vec3 V = normalize(camera_pos - Pos);
    vec3 N = normalize(Normal);
    vec3 R = normalize(2*(dot(L , N))*N-L) ;

    vec3 Ka = ambient_m;
    vec3 Kd = diffuse_m;
    vec3 Ks = specular_m;

    vec3 La = ambient_l;
    vec3 Ld = diffuse_l;
    vec3 Ls = specular_l;

    ambient = La * Ka ;
    diffuse = Ld * Kd  * max(dot(L,N) , 0);
    specular = Ls * Ks *  pow(max(dot(V , R) , 0) , gloss) ;

}
```

在 fragment shader 的部分也是宣告 in 變數去接這些內插完的值並且這些值補乘上貼上 texture 後的各點顏色以得到最終的 ambient , diffuse 以及 specular , 接著在三者相加並打包成 4 維即可。

```
#version 330 core

uniform sampler2D Texture ;

in vec2 texCoord;
in vec3 ambient ;
in vec3 diffuse ;
in vec3 specular ;
out vec4 fragColor ;

void main()
{
    vec4 object_color = texture(Texture , texCoord);
    vec3 a_color = ambient * object_color.xyz;
    vec3 d_color = diffuse * object_color.xyz;
    vec3 s_color = specular ;
    fragColor = vec4(a_color + d_color + s_color , 1.0);
}
```

j. Toon vert. and Toon frag.

Toon 的 vertex shader 就是一般的 vertex shader 處理 ， 但同樣考慮到如 phong vert.那邊所提到的是否在 world space , normal 變形的問題等等。

```
#version 330 core

// TODO:
// Implement Toon 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 Perspective ;

out vec2 texCoord;
out vec3 Normal;
out vec3 Pos;

void main()
{
    gl_Position = Perspective * View * Model * vec4(aPos, 1.0);
    texCoord = aTexCoord;
    Normal =  mat3(transpose(inverse(Model))) * aNormal;
    Pos = mat3(Model) * aPos;
}
```

在 fragment shader 裡面計算要用到的 L 跟 N(算法跟 phong model 裡面寫的一樣)，然後藉由 L, N 內積值(= 兩者的 cos 夾角值)去判定這塊區域對入射光線朝向的幅度 ， 又因為 cos theta 在 theta 介於 0 – 90 度的時候 ， 角度越小值越大 ， 代表很正對著光線 ， 因此 intensity 就給他越大 ， 就可以達到 Toon shading 的亮暗區隔效果。

```glsl
#version 330 core

uniform sampler2D Texture;
in vec2 texCoord;
in vec3 Normal;
in vec3 Pos;
out vec4 fragColor ;

uniform vec3 light_pos;
uniform vec3 diffuse_m;

void main()
{
    vec3 L = normalize(light_pos - Pos);
    vec3 N = normalize(Normal);
    vec3 Kd = diffuse_m;
    vec4 object_color = texture(Texture, texCoord);
    vec3 intensity = vec3(0 , 0 , 0);
    if(dot(L , N) >= 0.7){
        intensity = vec3(1 , 1 , 1) ;
    }
    else if(dot(L , N) <= 0.7 && dot(L , N) >= 0.4){
        intensity = vec3(0.7 , 0.7 , 0.7) ;
    }
    else if(dot(L , N) <= 0.1 && dot(L , N) >= 0.4){
        intensity = vec3(0.4 , 0.4 , 0.4) ;
    }
    else{
        intensity = vec3(0.2 , 0.2 , 0.2) ;
    }
    fragColor = vec4(Kd * vec3(object_color.x , object_color.y , object_color.z) * intensity , 1.0) ;
}
```

k. Edge vert. and Edge frag.

Edge 的 vertex shader 就是一般的 vertex shader 處理 ， 但同樣考慮到如 phong vert.那邊所提到的是否在 world space , normal 變形的問題等等。

```glsl
#version 330 core

// TODO:
// Implement Edge effect

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 Perspective ;

out vec3 Normal;
out vec3 Pos;
void main()
{
    gl_Position = Perspective * View * Model * vec4(aPos, 1.0);
    Normal = mat3(transpose(inverse(Model))) * aNormal;
    Pos = mat3(Model) * aPos;
}
```

Edge 的 fragment shader 會計算 V 跟 N 的夾角，計算方式同在 Phong , Goraud 的方式。具體的實作方式是讓 fragColor 只輸出不同程度的藍色，然而為了達到有點如 demo 影片的漸層效果，會根據 V 跟 N 的夾角越大(內積值越小),給予更藍的效果已達到漸層的 edge 效果。那為什麼 edge 效果會跟 V , N 掛勾呢?試想一下有個物體擺在我們面前，對我們而言物體的"最邊緣"上那些點的 normal 剛好會與我們視角成 90 度，再大我們就看不到(變成物體的背面)；另一個角度去思考，一個與我們正對的面(那顯然不是物體邊緣)，上面的點的 normal 會和我們視角方向夾角很小，因此這樣的邏輯就造就為何要算 V , N 內積以及為了營造漸層效果要如何給值的實作。

```
#version 330 core
uniform sampler2D Texture;
in vec2 texCoord;
in vec3 Normal;
in vec3 Pos;

uniform vec3 camera_pos;
out vec4 fragColor ;
void main()
{
    vec3 V = normalize(camera_pos - Pos);
    vec3 N = normalize(Normal);
    if(abs(dot(V , N)) < 0.15){
        fragColor = vec4(0 , 0 , 1 , 1);
    }
    else if(abs(dot(V , N)) < 0.3){
        fragColor = vec4(0 , 0 , abs(pow(dot(V , N) , 2)) , 1);
    }
    else if(abs(dot(V , N)) < 0.5){
        fragColor = vec4(0 , 0 , abs(pow(dot(V , N) , 4)) , 1);
    }
    else{
        fragColor = vec4(0 , 0 , 0 , 1);
    }
    //fragColor = vec4(0 , 0 , abs(pow(dot(V , N) , 2)) , 1);
}
```

2.遇到的問題 ：

一開始 normalize 這些 phong model 要用的向量時 ，不小心轉成 4 維的向量並且連第 4 維都一起 normalize 了 ，可以想見這樣的狀況下 render 出來的影像很暗 ，因為第四維必定小於 1。

還有不知道如何把一個四維的 matrix 直接變成三維的 ，因為三維的 matrix 才可以乘在三維的 Pos 以及 Normal。後來問朋友才知道 glsl 可以直接用 mat3()強制轉型。

當然還有沒考慮到 model->position , model->normal 等等的向量都是在 model space 上表示的 ，要轉到 world space 上才可以正確運行。