2020 CG HW2 Report

一、 環境

完全遵照 HW0 所介紹的環境進行搭建。
OpenGL
Visual Studio 編寫 C++ 以及 GLSL
32 bit GLFW
32 bit freeglut

二、 作業內容

1. 使用 VBO 及 VAO 渲染有材質的模型
2. 使用 Shader 實作 Phong Shading
3. 使用 Shader 實作 Dissolving Effect
4. 使用 Shader 實作 Ramp Effect

三、 實作

1. 初始化視窗
2. 初始化材質
   將所需要用到的材質檔案讀取進來。共有三個檔案需要讀取。
3. 讀取 model
   將 model 檔案讀取進來，本次作業採用的是將節點座標、法向量、材質座標分別讀取到三個不同 vector 中。在這裡的讀取方式會決定之後存取 VBO 的方式。
4. 初始化 shader
   首先要將 shader 檔案讀取進來，有 vertex 以及 fragment 兩種 shader，前者主要負責頂點的運算，將頂點對應至畫面上的二維座標，後者則是計算出需要繪製的像素顏色值。

```
char* vertexShaderSource = ReadShader("Resources/shaders/Phong.vert");
char* fragmentShaderSource = ReadShader("Resources/shaders/Phong.frag");
CreateShader(vertexShader, GL_VERTEX_SHADER, vertexShaderSource);
CreateShader(fragmentShader, GL_FRAGMENT_SHADER, fragmentShaderSource);
CreateProgram(PhongProgram, 2, vertexShader, fragmentShader);
```

5. 初始化 Buffer
   Buffer 的用途為存取 Data，也就是先前讀取 model 的 vector。但此時 Opengl 仍不知道這些 Data 究竟有些什麼意義，因此需要利用 VAO，賦予這些 Data 實際意義。
   (1.) 首先產生一個 VAO 及一個 VBO
   (2.) Bind VAO 與 VBO
   (3.) 將 vector 的起點 pointer 傳給 VBO，正式將 Data 存入 VBO
   實際上我使用了 Subbuffer，目的是將數個不同 vector 或 array（此處為 vector）存進同一個 buffer 中。
   (4.) 設定節點屬性，正式將 VBO bind 在 VAO 上。各個 Data 實際上要做什麼運算則是在 Shader 檔中設定。
   (5.) 啟動結點屬性
   (6.) Unbind VAO 與 VBO（也可不 Unbind，但之後若有需要設定 VAO 與 VBO 需注意目前 bind 在誰身上）
6. 顯示視窗
7. 設定視窗
8. 繪製模型
   重點說明 Shader 實作部分，請見第四大點
9. 設定鍵盤按鍵與滑鼠
   此次的設計中有：
   (1.) WASD 控制畫面前左後右
   (2.) 4~9 控制旋轉
   (3.) B 切換 Phong、溶解與漸層

(4.)　（在溶解效果中）+ 號增加物體可見部分，- 號減少物體可見部分
(5.)　滑鼠拖曳可以轉動視角

四、 繪製模型

先前提到的 VAO 賦予 Data 意義，實際上都是在 Shader files 裡面才具體實現出來。

1. 用 mode 變數決定使用哪個效果
2. 計算 Model Matrix 並將名稱設定為 modelMatrix
3. 使用 getV() 取得 View Matrix，將名稱設定為 viewMatrix
4. 使用 getP() 取得 Projection Matrix，將名稱設定為 projectionMatrix
5. 將主要材質名稱設定為 mainTex
6. Phong Shading
   (1.) Uniform
      (a.) WorldLightPos、WorldCamPos 分別為光源位置、視線位置
      (b.) Ka、Kd、Ks 為 absorption coefficient
      (c.) La、Ld、Ls 為 intensity coefficient
      (d.) gloss（α）為光澤度
   (2.) Vertex Shader
      (a.) 傳入 Position、Normal、TexCoord，分別為物件本身的座標、物件本身的法向量以及材質座標

```
24 void main() {
25     texCoord = TexCoord;
26     worldPos = vec3(modelMatrix * vec4(Position, 1.0));
27     normal = vec3(transpose(inverse(viewMatrix * modelMatrix)) * vec4(Normal, 1.0));
28 
29     gl_Position = projMatrix * (viewMatrix * (modelMatrix * vec4(Position, 1.0)));
30 }
```

      (b.) 材質座標直接往 Fragment Shader 傳
      (c.) 物件轉換至 worldPos 為 modelMatrix * 物件座標
      (d.) 相機座標下的法向量為法向量矩陣 * 物件法向量。而法向量矩陣為 ModelView Matrix 的反矩陣的 Transpose。
      (e.) 最重要的物件在顯示畫面上的位置，即為 MVP Matrix * 物件座標。MVP Matrix 來自 projMatrix * viewMatrix * modelMatrix。
   (3.) Fragment Shader

$$I = K_a L_a + K_d L_d (L \cdot N) + K_s L_s (V \cdot R)^{\alpha}$$

      (a.) Phong Shading 的公式即為上圖，各項代表的意義分別為 Ambient、Diffuse、Specular。
      (b.) N 代表 Vertex Shader 傳過來的法向量
      (c.) L 代表物體往光源的向量，即為 WorldLightPos – worldPos
      (d.) V 代表物體往視線的向量，即為 WorldCamPos – worldPos
      (e.) R 代表反射向量，公式如下

$$r = 2(l \cdot n)n - l$$

(f.) 材質顏色僅與前兩者相關，因此 Ambient 與 Diffuse 需要乘上該座標點位置的材質顏色值。

```
46 void main() {
47
48     N = normalize(normal);
49     L = normalize(WorldLightPos - worldPos);
50     V = normalize(WorldCamPos   - worldPos);
51     R = 2 * dot(L, N) * N - L;
52     H = (L + V) / abs(L + V);
53
54     albedo = texture2D(mainTex, texCoord);
55
56     ambient = La * Ka * vec3(albedo);
57     diffuse = Ld * Kd * vec3(albedo) * dot(L, N); // must > 0
58
59     specularPhong = Ls * Ks * pow(dot(V, R), gloss / 4.0);
60     specularBlinn = Ls * Ks * pow(dot(N, H), gloss);
61     specular = mix(specularPhong, specularBlinn, 0);
62     // change to 1 to see the difference between phong and blinn
63
64     color = vec4(ambient + diffuse + specular, 1.0);
65     // out color must be vec4
66 }
```

(g.) Blinn Shading 則是將 Specular 項的 $(V \bullet R)^{\alpha}$ 改為 $(N \bullet H)^{b}$

(h.) H 為 L 與 V 的標準化中央向量，公式如下

$$h = (l + v) / |l + v|$$

7. Dissolving Effect

實作中我將 Phong Shader 也加入了此效果中，讓兔子看起來更加真實

(1.) Uniform

(a.) 先前有在 Phong Shading 出現過的 Uniform 在這裡也同樣全部傳入

(b.) Dissolving noise texture 命名為 noiseTex

(c.) _Threshold 作為 Disssolve 程度的標竿

(d.) _EdgeLength 為溶解時的邊界寬度

(e.) _EdgeColor 為溶解時的邊界顏色

(2.) Vertex Shader

與 Phong Shading 的 Vertex Shader 一模一樣，差異會顯現在 Fragment Shader 中

(3.) Fragment Shader

(a.) 先計算出 Phong Shading 後的結果

(b.) 計算所有 noise 小於_Threshold 處，全部捨棄

(c.) 利用階梯函數的特性，設立一個 flag。若_Threshold + _EdgeLength - noise 的值小於_EdgeLength/2 則將 flag 設為 0，其餘則設為 1

(d.) 使用 flag 做為選擇 Phong Shading 後的結果或是_EdgeColor 的標準，若 flag 為 0 則顯示 Phong Shading，反之則為邊界顏色。

(e.) 換言之，若某一點的 noise 值大於_Threshold + (_EdgeLength / 2)，則顯示兔子，反之則顯示邊界（超過邊界的部分已經在先前被 discard 掉了）

```
void main() {

    N = normalize(normal);
    L = normalize(WorldLightPos - worldPos);
    V = normalize(WorldCamPos   - worldPos);
    R = 2 * dot(L, N) * N - L;

    albedo = texture2D(mainTex, texCoord);

    ambient = La * Ka * vec3(albedo);
    diffuse = Ld * Kd * vec3(albedo) * dot(L, N); // must > 0

    specularPhong = Ls * Ks * pow(dot(V, R), gloss / 4.0);
    specular = mix(specularPhong, specularBlinn, 0);

    albedo = vec4(ambient + diffuse + specular, 1.0);

    noise = texture(noiseTex, texCoord).x;

    if(noise - _Threshold < 0.0) {
        discard;
    }

    // use EdgeLength / 2 as threshold to prevent
    // exactly _Threshold + _EdgeLength - noise is 0
    flag = step(_EdgeLength / 2, _Threshold + _EdgeLength - noise);

    color = mix(albedo, _EdgeColor, flag);
    // out color must be vec4
}
```

8. Ramp Effect
   (1.) Uniform
      (a.) Kd（Diffuse 的 absorbtion coefficient）
      (b.) Ramp texture 命名為 rampTex
   (2.) Vertex Shader
      與 Phong Shading 一模一樣
   (3.) Fragment Shader
      Ramp Effect 只計算 Diffusion。與 Phong Shading 的 Diffusion 不同的是，N 與 L 內積後的結果拿來做為座標點，尋找 rampTex 上所對應的值，並乘進 diffuse 結果。

五、 難點（疑問）與解法

1. GLSL 語法與 C++語法儘管相當類似，但要真正使用起來，還是有許多語法相關的問題需要查詢
2. 在做 glBufferData 時，不知道如何才能將三個分開的 vector 設定在一個 VBO 當中，最後找到了 glBufferSubData，才成功將三者串連起來。
3. 不知道其實 Shader file 也是實作中的一部分。在 Shader 都 Initial 好了之後，一直印出一隻綠色兔子，想了非常久都不知道為什麼沒有貼上石頭材質。直到拿 Triangle 來測試之後才發現，原來 Vertex Shader 和 Fragment Shader 檔案都是空的。
4. 在不了解 VAO 的情況下 glVertexAttributePointer 的各項參數不曉得如何設定。尤其是第一項，完全不能理解該編號是哪裡來的。後來在開始寫 Shader file 時才終於找到來源。
5. 在很多地方，例如：glBufferSubData、glUniformMatrix 等等地方，不知道可以直接將 Array 或 Vector 的首項的 Pointer 傳入。
   (1.) 在做 glBufferSubData 時，由於範例是使用 array 直接傳入，因此不曉得 vector 該如何處理。
   (2.) glUniformMatrix 時，也就是要 Uniform View Matrix 和 Projection Matrix 的地方，不知道可以直接將 mat4 資料型態傳入。原以為需要轉換 glMatrixMode，然後使用 glLoadMatrix 將新的 matrix load 上去，再使用 glGetFloat 將該 Matrix 值取出來，才能傳入。會有此種誤解是由於課程提供的 OpenGL_Shader.pdf 中，有一頁正是使用了 glGetFloat 才得到 Matrix 本身，並傳入 Uniform，導致我也以為需要做這些動作。事實上只需要直接傳入 mat4 即可。

六、 結語

與第一次作業相比，這次的思路已經相當清晰，但由於又突然從純粹的 C++加上 package，轉換成 GLSL 系統，又重新習慣了一下，因此仍花了不少時間在理解。相信日後的作業可以更加得心應手。在 Ramp Effect 的部分，做出來的結果與 Demo 影片中的差異相當大，但我個人認為這是因為所提供的 ramp texture 與範例的不同，才導致了不同結果。我希望這部分助教或老師可以特別講解一下，也希望我自己在 demo 時記得問這個問題。