Student ID: 101062319
Name: 巫承威

Homework 3 - Report

Lighting

• How to operate my program?

As the Figure 1. shown, there are several new header files and source files in my project, please make sure they are all under the project folder.

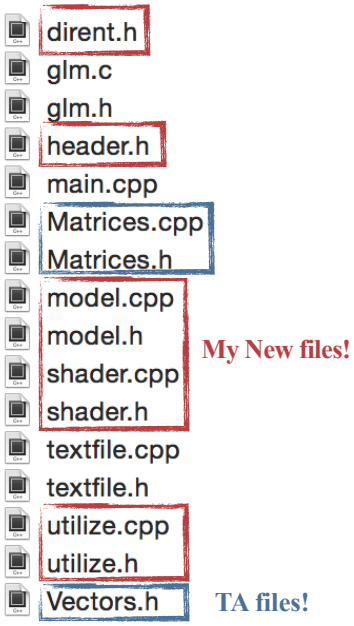

Figure 1.
Header Files and Source Files

Figure 2.
ScreenShot of Microsoft Visual Studio 2013

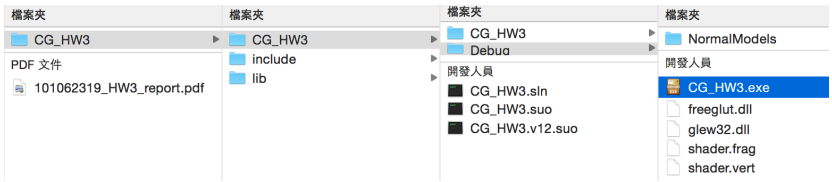

Figure 3.
Path of the "CG_HW3.exe"

All the user need to do to run my program is just open the .sln poject file via Microsoft Visual Studio 2013 (higher version may be fine). Then click the buttons "開始偵錯" or "啟動但不偵錯", and the execution window will appear. Or just execute the "CG_HW3.exe" as the Figure 3. shown. Like the example that TAs provided, user can type some button to control the display window and the object on it. These are the commands(note that I also allow the lowercase case letter to control the model as the uppercase letter does):

h / H - Show Help Menu
c / C - Clear Console
w / W - Switch between Solid / Wired
Left / Right Arrow - previous / next model
t / T - Start Translation Mode
s / S - Start Scaling Mode
r / R - Switch on / off Model Rotation
e / E - Start Eye Mode
p / P - Parallel / Perspective Projection

1	**- Switch on / off Directional Light**	**5**	**- Start Move Directional Light**
2	**- Switch on / off Point Light**	**6**	**- Start Move Point Light**
3	**- Switch on / off Spot Light**	**7**	**- Start Move Spot Light**
4	**- Switch Gouraud / Phong Shading**	**8 / 9**	**- Start Change Spot Light Cone / Exp**

• Implementation and problems I met

Based on my previous homework's framework, I add some attributes to my class "Model" and "ModelManager". All of the matrices' calculations are the same with the previous version. The main modifications are shown as follows:

```
27  extern GLint iLocNormal;
28  extern GLint iLocMVP;
29  extern GLint iLocVMN;
30  extern GLint iLocMN;
31  extern GLint iLocV;
32
33  const float ASPECT = 1.0;
34
35  extern Matrix4 M;
36  extern Matrix4 V;
37  extern Matrix4 P;
38
39  extern GLint iLocMDiffuse,  iLocMAmbient,  iLocMSpecular,  iLocMShininess;
40  extern GLint iLocLDAmbient, iLocLDDiffuse, iLocLDSpecular, iLocLDPosition, iLocLDSpotcutoff, iLocLDCAttenuation;
41  extern GLint iLocLPAmbient, iLocLPDiffuse, iLocLPSpecular, iLocLPPosition, iLocLPSpotcutoff, iLocLPCAttenuation, iLocLPLAttenuation, iLocLPQAttenuation;
42  extern GLint iLocLSAmbient, iLocLSDiffuse, iLocLSSpecular, iLocLSPosition, iLocLSSpotcutoff, iLocLSCAttenuation, iLocSSpotDirection, iLocLSLAttenuation, iLoc
43  extern GLint iLocLType;
44  extern GLint eyePos;
45  extern GLfloat lightType[4];
46
47  enum LightModeControl{Translation, Scaling, Rotation, Eye, Directional, Point, Spot, SpotEx };
```

Figure 4.

Add Variables for Shader

First, add the variables which will be used in shader. This process is similar with the previous homework's step, and I also modify the way of mode controlling.

```
GLfloat mn[16]; Matrix4 MN; // MN = (T * S * R) * N
GLfloat vmn[16];Matrix4 VMN; // VM = V * M * N
GLfloat v[16];  Matrix4 V;
```

Figure 5.

Add Matrices for Transformation (in <model>)

Second, since I need to do several transformations for the vertices, normals and other calculations, I store the matrices for transforming above values to either world space or eye space.

```
321     glUniform3fv(eyePos, 1, eyeVec);
```

Figure 6.

Send Eye Position to Shader

Third, owing to that I need to calculate the relation between light source and vertex, so I also pass the eye position to shader.

```
// MN = (T * S * R) * N
this->model->MN = M * (this->model->T) * (this->model->S) * (this->model->R) * (this->model->N);
this->model->mn[0] = this->model->MN[0];  this->model->mn[4] = this->model->MN[1];   this->model->mn[8] = this->model->MN[2];    this->model->mn[12] = th
this->model->mn[1] = this->model->MN[4];  this->model->mn[5] = this->model->MN[5];   this->model->mn[9] = this->model->MN[6];    this->model->mn[13] = th
this->model->mn[2] = this->model->MN[8];  this->model->mn[6] = this->model->MN[9];   this->model->mn[10] = this->model->MN[10];   this->model->mn[14] = t
this->model->mn[3] = this->model->MN[12]; this->model->mn[7] = this->model->MN[13];  this->model->mn[11] = this->model->MN[14];   this->model->mn[15] = t
```

Figure 7.

Put the Values of the Matrix “MN” into the Array which will be Passed to Shader

```
// VMN = V * (T * S * R) * N
this->model->VMN = V *(this->model->VM) * M * (this->model->T) * (this->model->S) * (this->model->R) * (this->model->N);
this->model->vmn[0] = this->model->VMN[0];  this->model->vmn[4] = this->model->VMN[1];   this->model->vmn[8] = this->model->VMN[2];    this->model->vmn[
this->model->vmn[1] = this->model->VMN[4];  this->model->vmn[5] = this->model->VMN[5];   this->model->vmn[9] = this->model->VMN[6];    this->model->vmn[
this->model->vmn[2] = this->model->VMN[8];  this->model->vmn[6] = this->model->VMN[9];   this->model->vmn[10] = this->model->VMN[10];   this->model->vmn
this->model->vmn[3] = this->model->VMN[12]; this->model->vmn[7] = this->model->VMN[13];  this->model->vmn[11] = this->model->VMN[14];   this->model->vmn
```

Figure 8.

Put the Values of the Matrix “VMN” into the Array which will be Passed to Shader

```
// V = VM = VT * VR
this->model->v[0] = this->model->VM[0];  this->model->v[4] = this->model->VM[1];   this->model->v[8] = this->model->VM[2];    this->model->v[12] = this-
this->model->v[1] = this->model->VM[4];  this->model->v[5] = this->model->VM[5];   this->model->v[9] = this->model->VM[6];    this->model->v[13] = this-
this->model->v[2] = this->model->VM[8];  this->model->v[6] = this->model->VM[9];   this->model->v[10] = this->model->VM[10];   this->model->v[14] = this
this->model->v[3] = this->model->VM[12]; this->model->v[7] = this->model->VM[13];  this->model->v[11] = this->model->VM[14];   this->model->v[15] = this
```

Figure 9.

Put the Values of the Matrix “V” into the Array which will be Passed to Shader

And the above 3 figures show that I follow the same step of passing matrix “MVP” to shader.

```
void::ModelManager::updateModelGroup()
{
    GLfloat triangle_normals[9];
    GLfloat triangle_vertex[9];
    GLfloat* ambient;
    GLfloat* diffuse;
    GLfloat* specular;
    GLfloat shininess;

    GLMgroup* mGroup = this->displayModel->groups;
```

Figure 10.

The Most Important Function of Traversing the Groups of the Model

This function “updateModelGroup” is the most important part in the C program since it will traverse all the groups of the model, pass vertices of the triangles of each group to shader in their corresponding order, pass normals of the vertices of the triangles of each group to shader in their corresponding order, store each group’s material’s features such like ambient and diffuse and finally call “glDrawArrays” to draw the vertices in the order of the previous sending order. While I call this “updateModelGroup” function in another loop function “onDisplay”, it will be called repeatedly.

```
484     while (mGroup){
485         ambient = this->displayModel->materials[mGroup->material].ambient;
486         diffuse = this->displayModel->materials[mGroup->material].diffuse;
487         specular = this->displayModel->materials[mGroup->material].specular;
488         shininess = this->displayModel->materials[mGroup->material].shininess;
489         for (int i = 0; i < (int)mGroup->numtriangles; i++){
490             int triangleID = mGroup->triangles[i];
491 
492             // the index of each vertex
493             int indv1 = this->displayModel->triangles[triangleID].vindices[0];
494             int indv2 = this->displayModel->triangles[triangleID].vindices[1];
495             int indv3 = this->displayModel->triangles[triangleID].vindices[2];
496 
497             // the index of each color
498             int indn1 = this->displayModel->triangles[triangleID].nindices[0];
499             int indn2 = this->displayModel->triangles[triangleID].nindices[1];
500             int indn3 = this->displayModel->triangles[triangleID].nindices[2];
501 
502             // vertices
503             triangle_vertex[0] = this->displayModel->vertices[indv1 * 3 + 0];
504             triangle_vertex[1] = this->displayModel->vertices[indv1 * 3 + 1];
505             triangle_vertex[2] = this->displayModel->vertices[indv1 * 3 + 2];
506 
507             triangle_vertex[3] = this->displayModel->vertices[indv2 * 3 + 0];
```

Figure 11.

View of Traversing Every Triangle's Vertices in Each Group of the Model

```
519             triangle_normals[3] = this->displayModel->normals[indn2 * 3 + 0];
520             triangle_normals[4] = this->displayModel->normals[indn2 * 3 + 1];
521             triangle_normals[5] = this->displayModel->normals[indn2 * 3 + 2];
522 
523             triangle_normals[6] = this->displayModel->normals[indn3 * 3 + 0];
524             triangle_normals[7] = this->displayModel->normals[indn3 * 3 + 1];
525             triangle_normals[8] = this->displayModel->normals[indn3 * 3 + 2];
```

Figure 12.

View of Traversing Every Normal of Triangle's Vertices in Each Group of the Model

Both of the above 2 figures all in the function "updateModelGroup". In conclusion, I will traverse all the groups of the model first, and then further traverse the material and the all the triangles of each group, and finally, traverse the vertices and normals. After traversing the above information, I bind them to the variables which will be used in shader. Another important thing is that I put this function inside the function "onDisplay" so that I will call it again and again.

All of the steps above are the main modifications of the C/C++ program. Except setting the variables in the function "setShaders" shown on the next page, the remaining codes are all in the vertex shade and fragment shader, which will be used to calculate all the relations between each vertex and light source.

```
61      iLocPosition = glGetAttribLocation (p, "av4position");
62      iLocNormal   = glGetAttribLocation (p, "av3normal");
63      iLocMVP      = glGetUniformLocation(p, "mvp");
64      iLocVMN      = glGetUniformLocation(p, "vmn");
65      iLocMN       = glGetUniformLocation(p, "mn");
66      iLocV        = glGetUniformLocation(p, "v");
```

Figure 13.

View of Binding the Variables for Shader (Matrices for Transformation)

```
73      iLocLDAmbient      = glGetUniformLocation(p, "LightSource[0].ambient");
74      iLocLDDiffuse      = glGetUniformLocation(p, "LightSource[0].diffuse");
75      iLocLDSpecular     = glGetUniformLocation(p, "LightSource[0].specular");
76      iLocLDPosition     = glGetUniformLocation(p, "LightSource[0].position");
77      iLocLDSpotcutoff   = glGetUniformLocation(p, "LightSource[0].spotCutoff");
78      iLocLDCAttenuation = glGetUniformLocation(p, "LightSource[0].constantAttenuation");
```

Figure 14.

View of Binding the Variables for Shader (Parameters of the Directional Light)

```
80      iLocLPAmbient      = glGetUniformLocation(p, "LightSource[1].ambient");
81      iLocLPDiffuse      = glGetUniformLocation(p, "LightSource[1].diffuse");
82      iLocLPSpecular     = glGetUniformLocation(p, "LightSource[1].specular");
83      iLocLPPosition     = glGetUniformLocation(p, "LightSource[1].position");
84      iLocLPSpotcutoff   = glGetUniformLocation(p, "LightSource[1].spotCutoff");
85      iLocLPCAttenuation = glGetUniformLocation(p, "LightSource[1].constantAttenuation");
86      iLocLPLAttenuation = glGetUniformLocation(p, "LightSource[1].linearAttenuation");
87      iLocLPQAttenuation = glGetUniformLocation(p, "LightSource[1].quadraticAttenuation");
```

Figure 15.

View of Binding the Variables for Shader (Parameters of the Point Light)

```
89      iLocLSAmbient      = glGetUniformLocation(p, "LightSource[2].ambient");
90      iLocLSDiffuse      = glGetUniformLocation(p, "LightSource[2].diffuse");
91      iLocLSSpecular     = glGetUniformLocation(p, "LightSource[2].specular");
92      iLocLSPosition     = glGetUniformLocation(p, "LightSource[2].position");
93      iLocLSSpotcutoff   = glGetUniformLocation(p, "LightSource[2].spotCutoff");
94      iLocLSCAttenuation = glGetUniformLocation(p, "LightSource[2].constantAttenuation");
95      iLocSSpotDirection = glGetUniformLocation(p, "LightSource[2].spotDirection");
96      iLocLSLAttenuation = glGetUniformLocation(p, "LightSource[2].linearAttenuation");
97      iLocLSQAttenuation = glGetUniformLocation(p, "LightSource[2].quadraticAttenuation");
98      iLocSSpotExponent  = glGetUniformLocation(p, "LightSource[2].spotExponent");
```

Figure 16.

View of Binding the Variables for Shader (Parameters of the Spot Light)

```
100     iLocLType = glGetUniformLocation(p, "lightType");
101     eyePos    = glGetUniformLocation(p, "eyeP");
```

Figure 17.

View of Binding the Variables for Shader (Parameters of Eye Position and Light Type)

The most difficult problem is that to find the location of the traversing groups and how to do it. Since I compress all the values into 1-dimension array in their corresponding order and pass them to shader, I try to reproduce this way to this time at first; however, I discover that this method is difficult since I need to traverse all the groups of the model and then traverse its triangles, vertices and normals in proper order. If I insist on this way, I will need to allocate enough array to store them and the most important of all, I need to record the range of the material for vertices and normals…. So I finally give up this annoying method, and try to just traverse them again and again in the "onDisplay" function. That's much easier than the previous one since it follows the order of groups, and I only need to store this group's material, triangles, vertices and normals in each round.

• Other efforts I have done

Owing to the unfriendly codes which combine all the segment of the program into only one file, and also not so readable for the programmer, I decide to make it looked like the object-oriented program. Nevertheless, this idea cost me almost one afternoon to adjust them…. To finish this task, I need to figure out the operation between the model and the whole program and also how to manipulate each class. I think this is the most difficult part for transforming the original code to object-oriented type.

Another part I have done is to use the structure "DIR" and "dirent" which are already mentioned in above paragraph. It's more powerful and flexible than brute-force method because it can traverse all the files in the given folder's all subfolders. Also, I define new macro called "abs" to help get the length of the model. To avoid the re-define error of so many new header files, I add the "#pragma once" to make compiler correctly include the header file only once when they need.

What's more, I add the feature that my model can be scaled by adjusting the GLUT window. It means that if I change the GL window, my object model will also change its size to fit the size of window.

• Screenshots

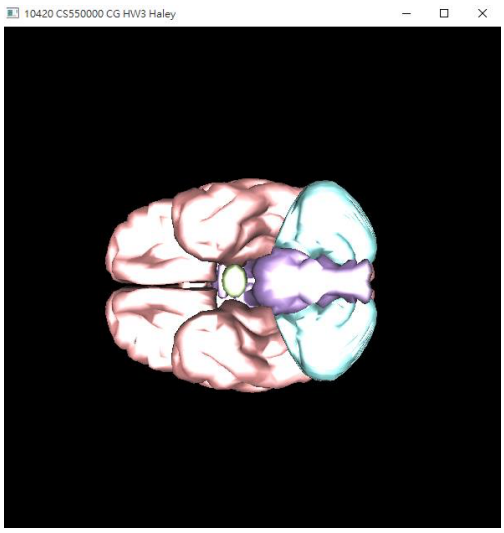

Figure 18.
./NormalModels/High/brain18KN.obj

• Bonus

The difference between Gouraud-Shading and Phong-Shading is that the Phong-Shading is "per pixel lighting". Thus, I copy the lighting function I have written in vertex shader to the fragment shader. The only difference is that I change the "vv4color" to be the normals sent to the vertex shader if I decide to use Phong-Shading; otherwise, I calculate the lighting values then set it to be the value of "vv4color".

• Screenshots

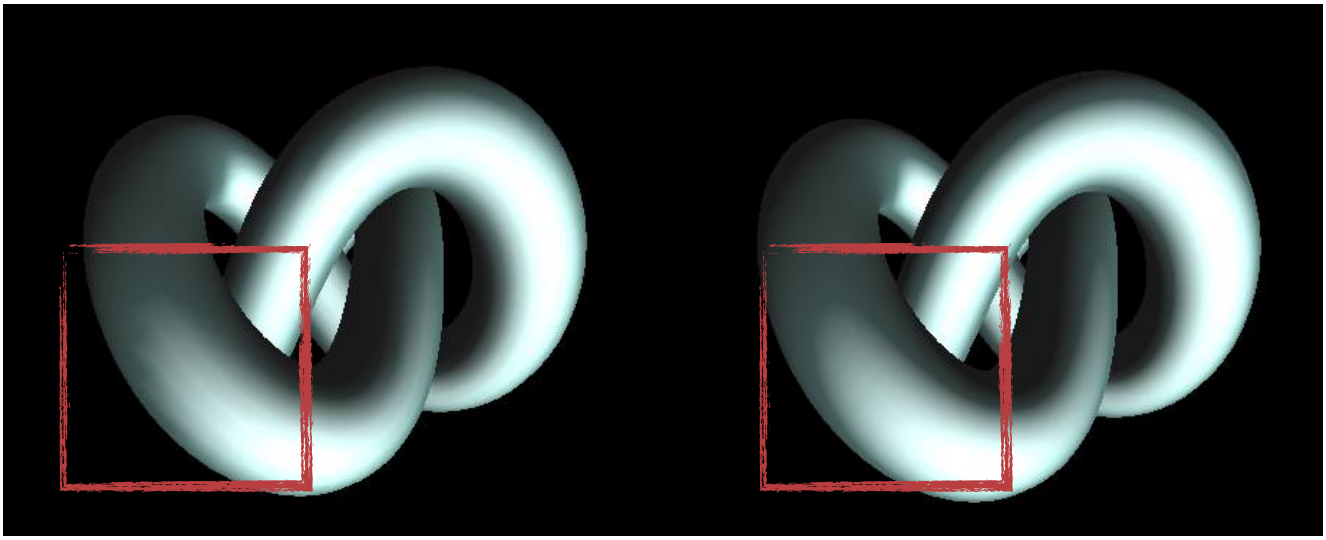

Figure 19.
Difference between Gouraud-Shading and Phong-Shading

(./NormalModels/High/texturedknot11KC.obj)

Left: Gouraud-Shading
Right: Phong-Shading