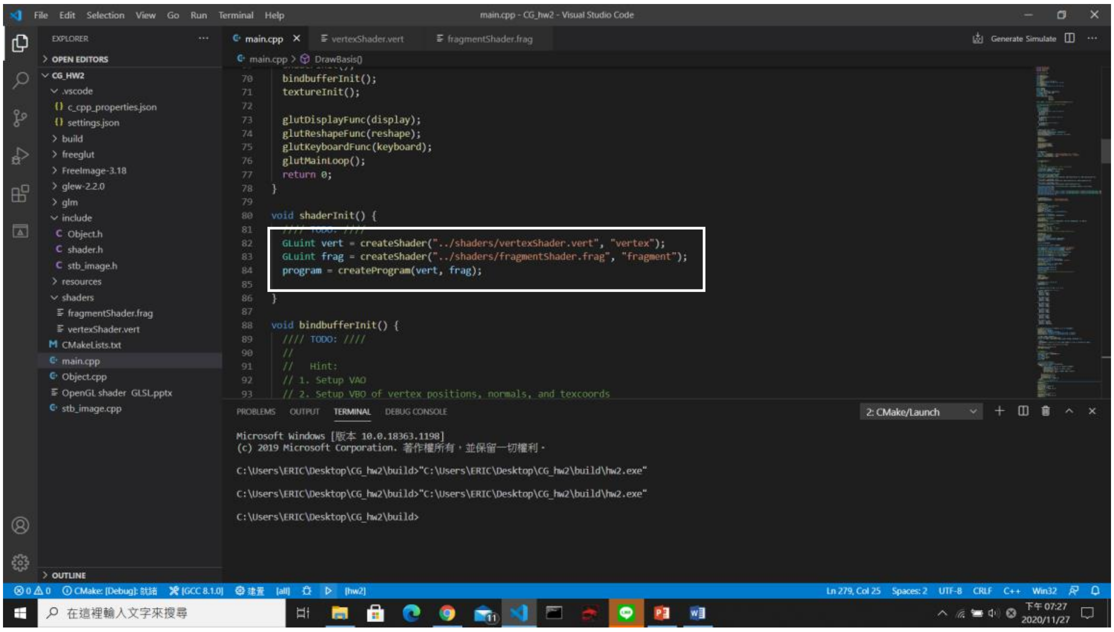
Computer Graphics HW2 Report

0716234 蕭彧

How to use GLSL

Snapshot1:

First thing of all you need to create shader and get a type call Gluint and then use the two shades vert and fragment to create a program.

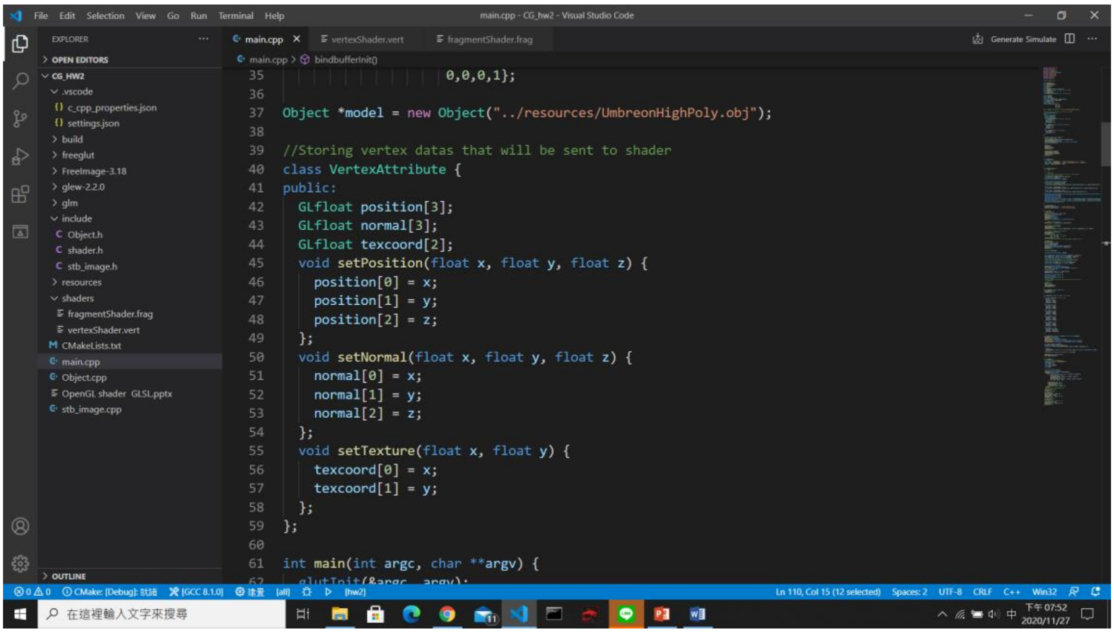


```
  bindbufferInit();
  textureInit();

  glutDisplayFunc(display);
  glutReshapeFunc(reshape);
  glutKeyboardFunc(keyboard);
  glutMainLoop();
  return 0;
}

void shaderInit() {
  //// TODO: ////
  GLuint vert = createShader("../shaders/vertexShader.vert", "vertex");
  GLuint frag = createShader("../shaders/fragmentShader.frag", "fragment");
  program = createProgram(vert, frag);

}

void bindbufferInit() {
  //// TODO: ////
  //
  //   Hint:
  // 1. Setup VAO
  // 2. Setup VBO of vertex positions, normals, and texcoords
```

```
Microsoft Windows [版本 10.0.18363.1198]
(c) 2019 Microsoft Corporation. 著作權所有，並保留一切權利。

C:\Users\ERIC\Desktop\CG_hw2\build>"C:\Users\ERIC\Desktop\CG_hw2\build\hw2.exe"

C:\Users\ERIC\Desktop\CG_hw2\build>"C:\Users\ERIC\Desktop\CG_hw2\build\hw2.exe"

C:\Users\ERIC\Desktop\CG_hw2\build>
```

Snapshot2:

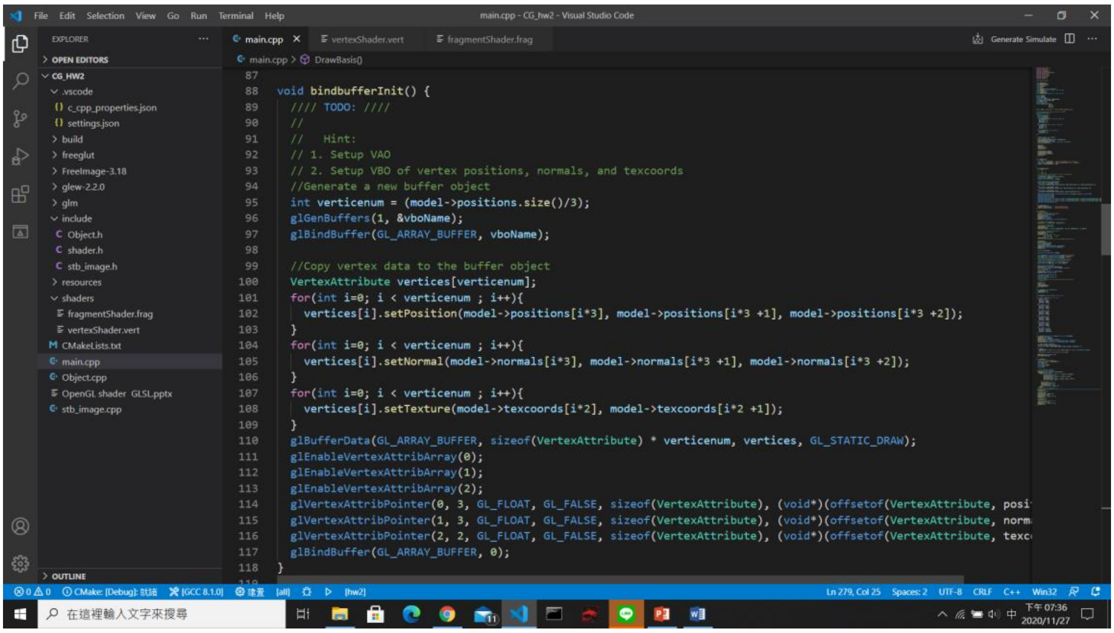
Next, I create a class call VertexAttribute to store the data of the vertex, and it has three variable position, normal and texcoord which can load the vertex attribute in them and three functions setposition, setnormal and settexcoord. As their name they can set the attribute of the vertex in the three variables in this class. I will use this class later to load the vertex attribute.



```
                    0,0,0,1};

Object *model = new Object("../resources/UmbreonHighPoly.obj");

//Storing vertex datas that will be sent to shader
class VertexAttribute {
public:
  GLfloat position[3];
  GLfloat normal[3];
  GLfloat texcoord[2];
  void setPosition(float x, float y, float z) {
    position[0] = x;
    position[1] = y;
    position[2] = z;
  };
  void setNormal(float x, float y, float z) {
    normal[0] = x;
    normal[1] = y;
    normal[2] = z;
  };
  void setTexture(float x, float y) {
    texcoord[0] = x;
    texcoord[1] = y;
  };
};

int main(int argc, char **argv) {
  glutInit(&argc, argv);
```

Snapshot3:

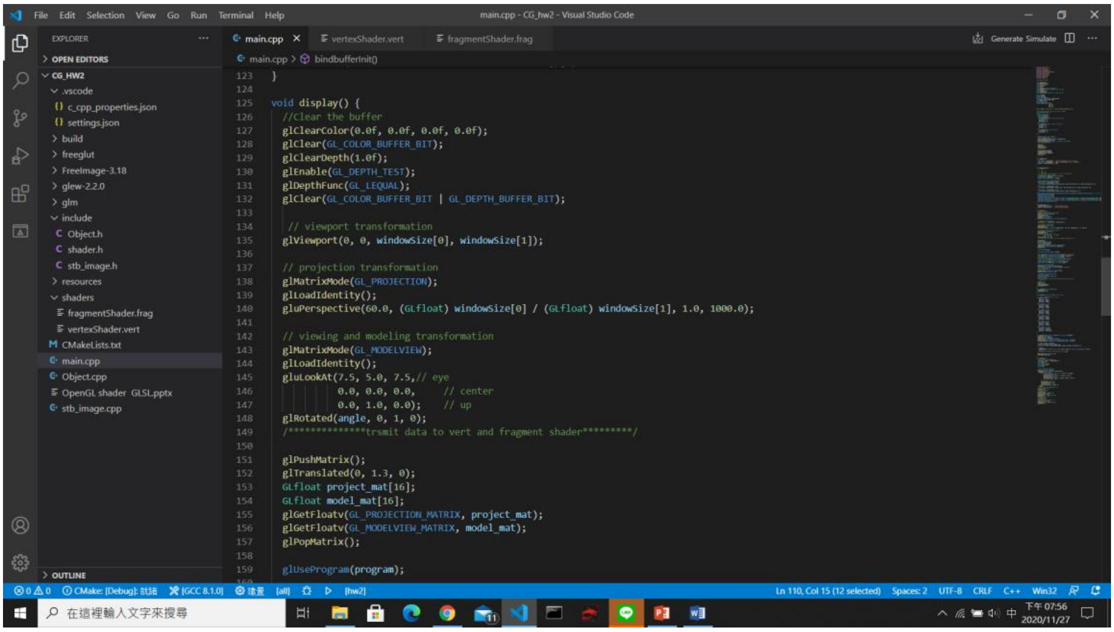
The next thing to do is to create a VAO and then bind the the vertices of model on it, so in the code we can see “vboName” is the VAO object I create and I bind it on GL_ARRAY_BUFFER. Moreover, I use three for loop to input three data, position, normal and texcoord into the class I write above with the functions setPosition and so on, next I use glBufferdata to create and initialize a buffer object's data store, in the end I enable three locations that I will transmit data to vert shader on, and then bind the three data position, normal and texcoord on the locations I enable so the vertex shader can get these three data from the locations I bind. So far, we had bind our data on GL_ARRAY_BUFFER and latter I will show you how to get the data by the vertex shader.



```cpp

void bindbufferInit() {
  //// TODO: ////
  //
  //   Hint:
  // 1. Setup VAO
  // 2. Setup VBO of vertex positions, normals, and texcoords
  //Generate a new buffer object
  int verticenum = (model->positions.size()/3);
  glGenBuffers(1, &vboName);
  glBindBuffer(GL_ARRAY_BUFFER, vboName);

  //Copy vertex data to the buffer object
  VertexAttribute vertices[verticenum];
  for(int i=0; i < verticenum ; i++){
    vertices[i].setPosition(model->positions[i*3], model->positions[i*3 +1], model->positions[i*3 +2]);
  }
  for(int i=0; i < verticenum ; i++){
    vertices[i].setNormal(model->normals[i*3], model->normals[i*3 +1], model->normals[i*3 +2]);
  }
  for(int i=0; i < verticenum ; i++){
    vertices[i].setTexture(model->texcoords[i*2], model->texcoords[i*2 +1]);
  }
  glBufferData(GL_ARRAY_BUFFER, sizeof(VertexAttribute) * verticenum, vertices, GL_STATIC_DRAW);
  glEnableVertexAttribArray(0);
  glEnableVertexAttribArray(1);
  glEnableVertexAttribArray(2);
  glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(VertexAttribute), (void*)(offsetof(VertexAttribute, posi
  glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(VertexAttribute), (void*)(offsetof(VertexAttribute, norm
  glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(VertexAttribute), (void*)(offsetof(VertexAttribute, texc
  glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```

Snapshot4:

This screenshot show that I have store the projection and modelview matrix to two matrices I create which call project_mat and model_mat, so latter I will pass these two matrices to vertex shader.


File Edit Selection View Go Run Terminal Help
main.cpp - CG_hw2 - Visual Studio Code
EXPLORER
OPEN EDITORS
CG_HW2
main.cpp
vertexShader.vert
fragmentShader.frag
Generate Simulate
main.cpp > bindbufferInit()
Ln 110, Col 15 (12 selected) Spaces: 2 UTF-8 CRLF C++ Win32
在這裡輸入文字來搜尋
下午 07:56
2020/11/27

```cpp
}

void display() {
  //Clear the buffer
  glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
  glClear(GL_COLOR_BUFFER_BIT);
  glClearDepth(1.0f);
  glEnable(GL_DEPTH_TEST);
  glDepthFunc(GL_LEQUAL);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   // viewport transformation
  glViewport(0, 0, windowSize[0], windowSize[1]);

  // projection transformation
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(60.0, (GLfloat) windowSize[0] / (GLfloat) windowSize[1], 1.0, 1000.0);

  // viewing and modeling transformation
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(7.5, 5.0, 7.5,// eye
            0.0, 0.0, 0.0,     // center
            0.0, 1.0, 0.0);    // up
  glRotated(angle, 0, 1, 0);
  /**************trsmit data to vert and fragment shader*********/

  glPushMatrix();
  glTranslated(0, 1.3, 0);
  GLfloat project_mat[16];
  GLfloat model_mat[16];
  glGetFloatv(GL_PROJECTION_MATRIX, project_mat);
  glGetFloatv(GL_MODELVIEW_MATRIX, model_mat);
  glPopMatrix();

  glUseProgram(program);
```

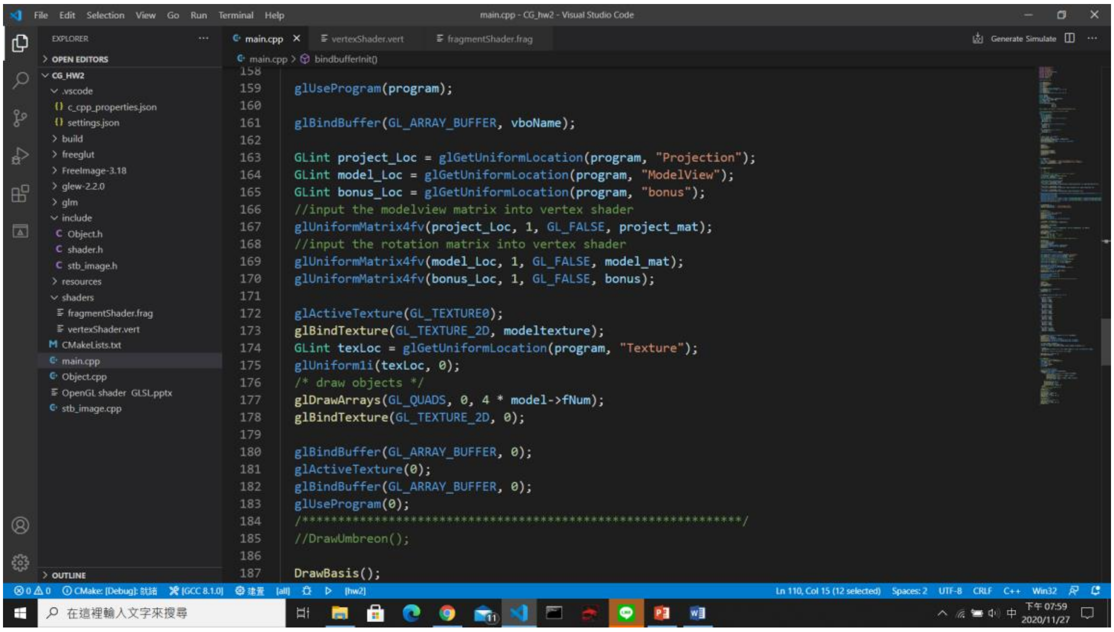
Snapshot5:

Here I start to transmit some data to the vertex shader. The first thing is to use the program I have created in screenshot 1, so the things I do below will have effects in the corresponding vertex and fragment shader. First I use glbindbuffer to transmit the VAO I have bond in screenshot3. Because we have already bund the data on location, the vertex shader can get the data from the corresponding location number.

Second part I use glGetuniformlocation to specify the matrices number so that that vertex shader can use the corresponding names to get the matrices. I pass three matrices here two are projection and modelview matrices and the third one is a bonus matrix that I will mention in bonus section.

Third part is to transmit the texture with binding the texture on GL_TEXTURE_2D by glBindTexture. The same as section two I use uniform to transmit, and then use glUniformi to specify the texture number as 0.

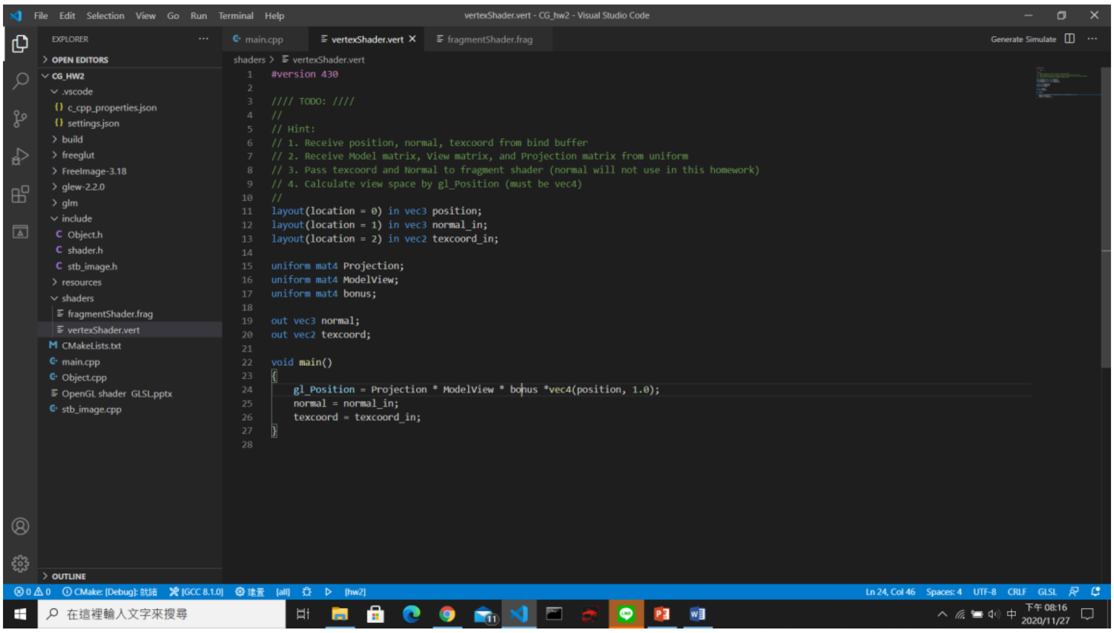
The last section is to draw the model with glDrawArrays, the function can get the output of the two shader and draw the result onto the monitor. Last thing for all is remember to unbind all binding objects by using buffer as 0.



```
glUseProgram(program);

glBindBuffer(GL_ARRAY_BUFFER, vboName);

GLint project_Loc = glGetUniformLocation(program, "Projection");
GLint model_Loc = glGetUniformLocation(program, "ModelView");
GLint bonus_Loc = glGetUniformLocation(program, "bonus");
//input the modelview matrix into vertex shader
glUniformMatrix4fv(project_Loc, 1, GL_FALSE, project_mat);
//input the rotation matrix into vertex shader
glUniformMatrix4fv(model_Loc, 1, GL_FALSE, model_mat);
glUniformMatrix4fv(bonus_Loc, 1, GL_FALSE, bonus);

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, modeltexture);
GLint texLoc = glGetUniformLocation(program, "Texture");
glUniform1i(texLoc, 0);
/* draw objects */
glDrawArrays(GL_QUADS, 0, 4 * model->fNum);
glBindTexture(GL_TEXTURE_2D, 0);

glBindBuffer(GL_ARRAY_BUFFER, 0);
glActiveTexture(0);
glBindBuffer(GL_ARRAY_BUFFER, 0);
glUseProgram(0);
/*************************************************************/
//DrawUmbreon();

DrawBasis();
```

Snapshot6:

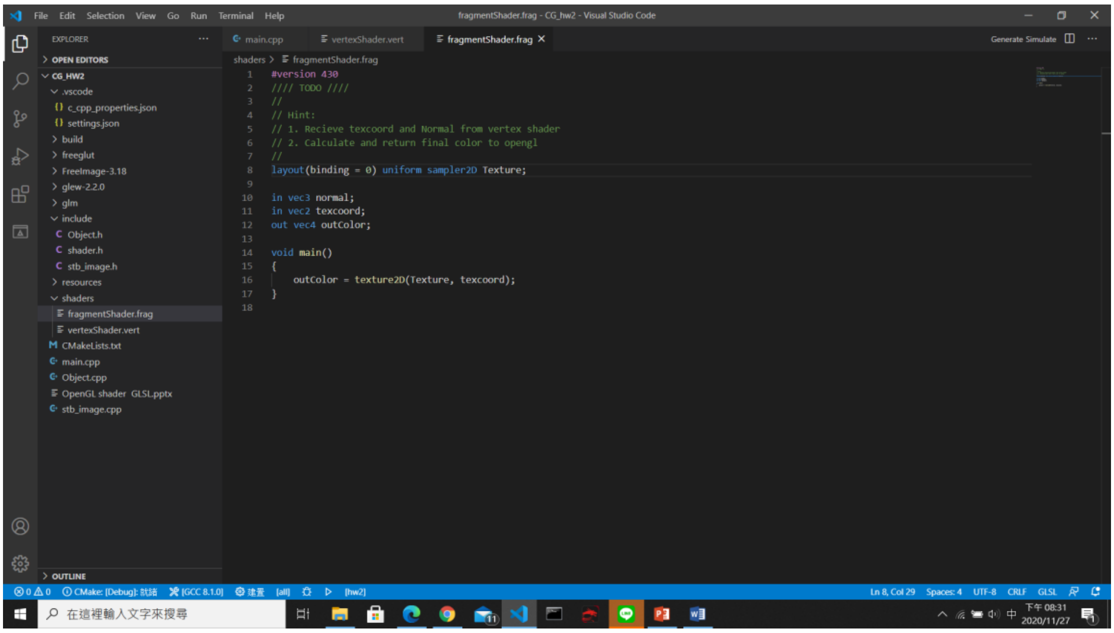
On the above part I transmit a lot of data to vertex shader, so this part is to show how to use vertex shader. The layout statement with location statement inside get the data I bind on the corresponding location which are position, normal and texcoord. Next, the uniform statement gets three matrices I transmit with the name I specified in main.cpp. In the main part of vertex shader I multiply all of the matrices with the data position and then give the final result to glposition which is the GLSL function to specify the coordinate in the space. The other two input just give them new name and then can transmit them to fragment shader by the out statement which specify the output data to fragment shader



```glsl
#version 430

//// TODO: ////
//
// Hint:
// 1. Receive position, normal, texcoord from bind buffer
// 2. Receive Model matrix, View matrix, and Projection matrix from uniform
// 3. Pass texcoord and Normal to fragment shader (normal will not use in this homework)
// 4. Calculate view space by gl_Position (must be vec4)
//
layout(location = 0) in vec3 position;
layout(location = 1) in vec3 normal_in;
layout(location = 2) in vec2 texcoord_in;

uniform mat4 Projection;
uniform mat4 ModelView;
uniform mat4 bonus;

out vec3 normal;
out vec2 texcoord;

void main()
{
    gl_Position = Projection * ModelView * bonus *vec4(position, 1.0);
    normal = normal_in;
    texcoord = texcoord_in;
}
```

Snapshot7:

The final screenshot shows the content of fragment shader. First line I receive the picture of texture by the layout statement, and then receive the normal and texcoord data from vertex shader. In main function, I use the function texture2D which use texcoord to put corresponding texture coordinate to the model, so the final output is the result of this function. After these procedure I can finally use the gldrawarrays function to draw the result of the shaders.



```
#version 430
//// TODO ////
//
// Hint:
// 1. Recieve texcoord and Normal from vertex shader
// 2. Calculate and return final color to opengl
//
layout(binding = 0) uniform sampler2D Texture;

in vec3 normal;
in vec2 texcoord;
out vec4 outColor;

void main()
{
    outColor = texture2D(Texture, texcoord);
}
```

The problem I met and the solution

The problem I met is how to texture my basis with opengl. Because I draw my basis with glcylinder in hw1, I can just open the attribute of texturing the basis and it will texture the basis in a default way, texturing the picture around the basis, instead of customizing how to texture. In this situation, the first method jumping onto my brain is to modify the original basis.png to a picture which has the twenty copies of 1/20 width of basis.jpg. I then call my friend to help me modify the picture and test if it is feasible. Unfortunately, it failed. The color of my basis is gray instead of beautiful wood color. After the failure, I decide to draw the basis with glVertex3f and glTexCoord2f one by one, and finally success. But my roommate didn’t believe me, he modified the picture by himself and then to my surprise he success. I then google the reason of my failure. I found that the graphics software that my friend use may have the different color composition of the one I set. My setting of input texture is RGB, and the picture my friend made may have the composition of RAGB or what, so it couldn’t match my setting. That’s may be the reason why my picture is gray. This is a tricky finding that modify the picture can solve the problem in an easy way instead of mapping texture coordinate one by one, but in my code I still mapping one by one haha.

Bonus section

The bonus is passing an extra matrix by uniform statement into the vertex shader. The initial state of the matrix is an identity matrix, and when keying keyboard the matrix passed to the vertex shader will be modify. You can press the keyboard button “1” to “5” to modify the size of Eevee. “1” means the origin size of Eevee and the bigger the number is the bigger Eevee is. By the way, you can press the space button to let Eevee turn upside down. The above is the bonus I did.