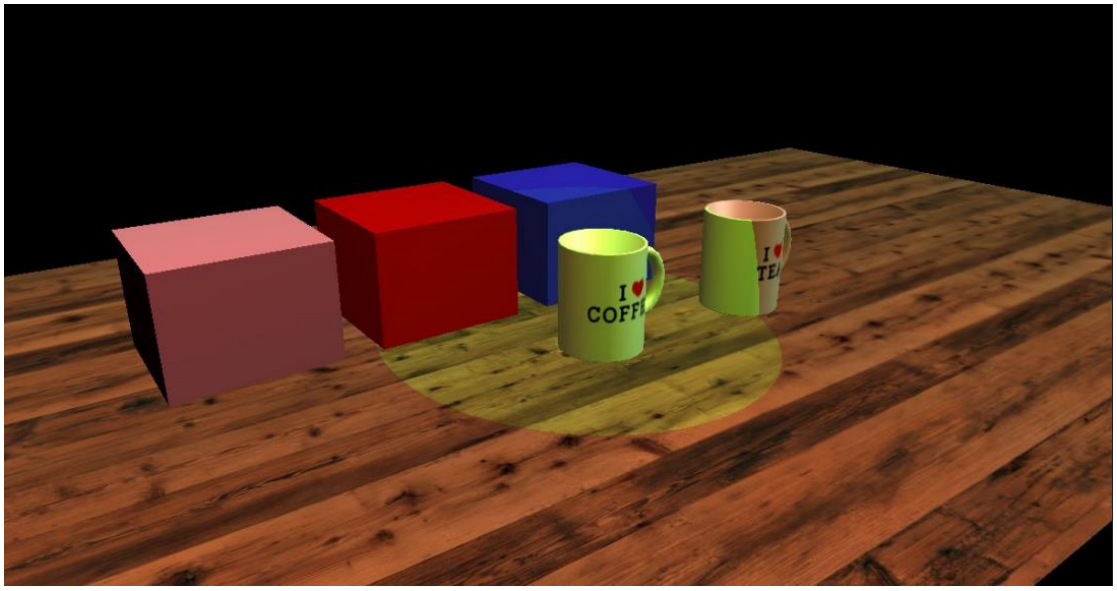
Homework #2 Report

[CSIC30108] Computer Graphics 2022

Student ID: 311605004 **Name:** 劉子齊 **Date:** 2022.11.11

1. Introduction:

In this homework, our goal is to implement a program capable in performing several shader effects, such as basic rendering, direction light, point light, and spotlight, based on the provided template with OpenGL and GLSL. According to the figure provided by the specification attached below, the final result of our implementation should contain 3 cubes, 2 mugs, and a wooden floor.



2. Implementation Details:

First of all, as shown in the following figures, I start with passing the textures, model matrices, and the objects to the context, a.k.a. “ctx”, which made us able to load the model data through the obj files in the further steps.

```
74  Model* m = Model::fromObjectFile("../assets/models/cube/cube.obj");
75  m->textures.push_back(createTexture("../assets/models/cube/texture.bmp"));
76  m->modelMatrix = glm::scale(m->modelMatrix, glm::vec3(0.4f, 0.4f, 0.4f));
77  ctx.models.push_back(m);
78
79  m = Model::fromObjectFile("../assets/models/Mugs/Models/Mug_obj3.obj");
80  m->textures.push_back(createTexture("../assets/models/Mugs/Textures/Mug_C.png"));
81  m->textures.push_back(createTexture("../assets/models/Mugs/Textures/Mug_T.png"));
82  m->modelMatrix = glm::scale(m->modelMatrix, glm::vec3(6.0f, 6.0f, 6.0f));
83  ctx.models.push_back(m);
```

```
121 ctx.objects.push_back(new Object(0, glm::translate(glm::identity<glm::mat4>(), glm::vec3(1.5, 0.2, 2))));
122 (*ctx.objects.rbegin())->material = mFlatwhite;
123 ctx.objects.push_back(new Object(0, glm::translate(glm::identity<glm::mat4>(), glm::vec3(2.5, 0.2, 2))));
124 (*ctx.objects.rbegin())->material = mShinyred;
125 ctx.objects.push_back(new Object(0, glm::translate(glm::identity<glm::mat4>(), glm::vec3(3.5, 0.2, 2))));
126 (*ctx.objects.rbegin())->material = mClearblue;
127 ctx.objects.push_back(new Object(1, glm::translate(glm::identity<glm::mat4>(), glm::vec3(3, 0.3, 3))));
128 ctx.objects.push_back(new Object(1, glm::translate(glm::identity<glm::mat4>(), glm::vec3(4, 0.3, 3))));
129 (*ctx.objects.rbegin())->textureIndex = 1;
```

After passing data to the “ctx” properly, I started parsing model data from OBL file. In this project, we only need to deal with “v”, “vt”, “vn”, and “f”, we can directly ignore the other fields. The following figures were how I implemented parsing of the OBJ files, and I will make further explanation in the next page.

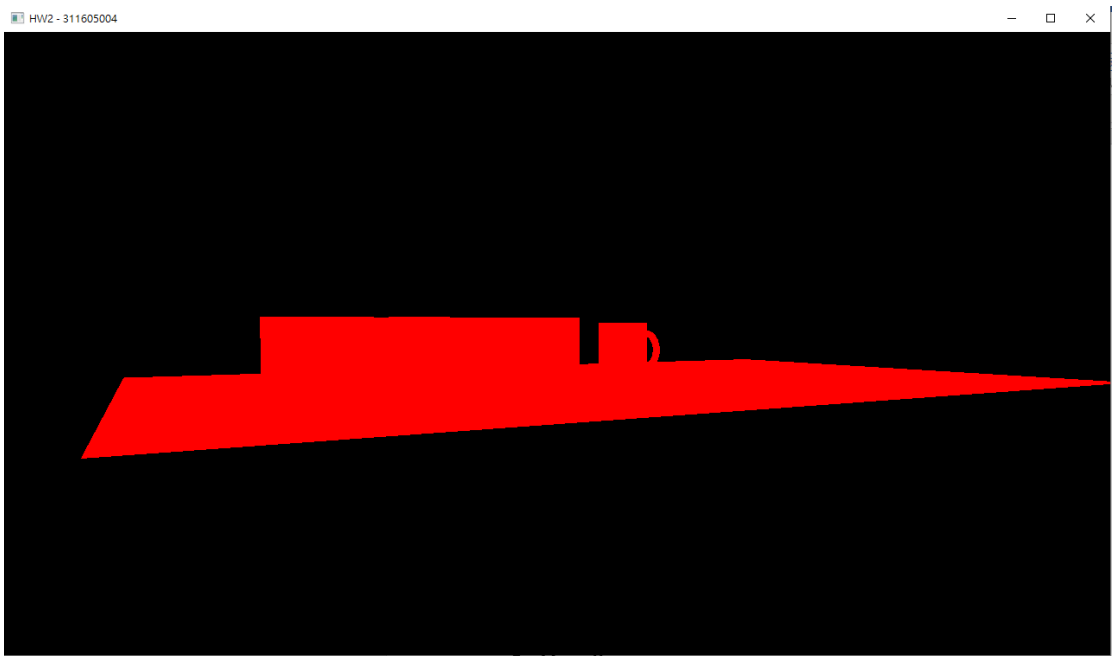
```
39  std::string line = "";
40  std::string prefix = "";
41  std::stringstream ss;
42
43  std::vector<glm::vec3> tmp_positions;
44  std::vector<glm::vec3> tmp_normals;
45  std::vector<glm::vec2> tmp_textcoords;
46
47  glm::vec2 temp_vec2;
48  glm::vec3 temp_vec3;
49
50  tmp_positions.push_back(temp_vec3);
51  tmp_normals.push_back(temp_vec3);
52  tmp_textcoords.push_back(temp_vec2);
53
54  int tmp_vertexnums = 0;
55  int total = 0;
56
57  GLint temp_glint = 0;
58
59  while (getline(ObjFile, line)) {
60      ss.clear();
61      ss.str(line);
62      ss >> prefix;
63
64      if (prefix == "v") {
65          ss >> temp_vec3.x >> temp_vec3.y >> temp_vec3.z;
66          tmp_positions.push_back(temp_vec3);
67      }
68      else if (prefix == "vn") {
69          ss >> temp_vec3.x >> temp_vec3.y >> temp_vec3.z;
70          tmp_normals.push_back(temp_vec3);
71      }
72      else if (prefix == "vt") {
73          ss >> temp_vec2.x >> temp_vec2.y;
74          tmp_textcoords.push_back(temp_vec2);
```

```
76      else if (prefix == "f") {
77          total++;
78          int counter = 0;
79          while (ss >> temp_glint) {
80              if (counter == 0){
81                  m->positions.push_back(tmp_positions[temp_glint].x);
82                  m->positions.push_back(tmp_positions[temp_glint].y);
83                  m->positions.push_back(tmp_positions[temp_glint].z);
84              }
85              else if (counter == 1){
86                  m->texcoords.push_back(tmp_textcoords[temp_glint].x);
87                  m->texcoords.push_back(tmp_textcoords[temp_glint].y);
88              }
89              else if (counter == 2){
90                  m->normals.push_back(tmp_normals[temp_glint].x);
91                  m->normals.push_back(tmp_normals[temp_glint].y);
92                  m->normals.push_back(tmp_normals[temp_glint].z);
93              }
94
95              if (ss.peek() == '/') {
96                  ++counter;
97                  ss.ignore(1, '/');
98              }
99              else if (ss.peek() == ' ') {
100                 ++counter;
101                 ss.ignore(1, ' ');
102             }
103
104             if (counter > 2) counter = 0;
105         }
106     }
107 }
108
109 m -> numVertex = total*3;
110
111 if (ObjFile.is_open()) ObjFile.close();
112
113 return m;
```

In the figures above, we cam see that if the first character in the line of the OBJ file is “v”, “vt”, or “vn”, I will correspondingly store them in tmp_positions, tmp_normals, and tmp_textcoords temporary.

Afterwards, when I found the first character of the new line is “f”, I would find the corresponding position, normal, or textcoords from its temporary storage through the indices provided in the line, and push them into the homologous label in the model.

After finishing the implementations above, we will obtain the result in the following figure, which we can see the shape of 3 cubes, 2 mugs, and a plane as the floor.



After getting the shape of the objects, I moved onto rendering the basic texture of the objects, which we need to implement the vertex and fragment shaders.

For the vertex shader, the output was gl_Position, also we had to pass the texture coordinate to the fragment shader. Since we only needed to deal with the textures first, the calculation of the position and the texture coordinate were quite simple here, which were shown in the following figure.

```
27    void main() {
28        gl_Position = Projection * ViewMatrix * ModelMatrix * vec4(position.x, position.y, position.z, 1.0);
29        TexCoord = texCoord;
30    }
```

For the fragment shader, I simply get the final color of the object through texture coordinate, the texture passed by the uniform and the texture2D function, which is shown in the following figure.

```
 9    void main() {
10        color = texture2D(ourTexture, TexCoord);
11    }
```

Afterwards, I start passing the vertex data to the vertex buffer. As shown in the following figure, I first generated the vertex array object, a.k.a. VAO, and bind them for each model. Then I generated 3 vertex buffer objects for each model and bind them. Through the VAO and VBO I just generated, I pass the vertex data I obtained from ctx→ models[i]→{positions, normal, textcoords} to the vertex buffer.

```
29  glGenVertexArrays(num_model, VAO);
30
31  for (int i = 0; i < num_model; i++) {
32      glBindVertexArray(VAO[i]);
33      Model* model = ctx->models[i];
34
35      GLuint VBO[3];
36      glGenBuffers(3, VBO);
37
38      glBindBuffer(GL_ARRAY_BUFFER, VBO[0]);
39      glBufferData(GL_ARRAY_BUFFER, sizeof(float) * model->positions.size(), model->positions.data(), GL_STATIC_DRAW);
40      glEnableVertexAttribArray(0);
41      glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0 * sizeof(float), (void*)0);
42
43      glBindBuffer(GL_ARRAY_BUFFER, VBO[1]);
44      glBufferData(GL_ARRAY_BUFFER, sizeof(float) * model->normals.size(), model->normals.data(), GL_STATIC_DRAW);
45      glEnableVertexAttribArray(1);
46      glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0 * sizeof(float), (void*)0);
47
48      glBindBuffer(GL_ARRAY_BUFFER, VBO[2]);
49      glBufferData(GL_ARRAY_BUFFER, sizeof(float) * model->texcoords.size(), model->texcoords.data(), GL_STATIC_DRAW);
50      glEnableVertexAttribArray(2);
51      glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0 * sizeof(float), (void*)0);
52  }
```

After passing the vertex data to the vertex buffer, I started rendering the objects with the shader implemented. As shown in the following figure, I first make use and bind the program through the programID obtained from the BasicProgram class.

By iterating through all the objects, I load the VAO of the current model for the object. Then I passed projection, the view

matrix, and the model matrix to the shader for further calculation. Lastly, I draw the object with the glDrawArrays function.

```
82  glUseProgram(programId);
83  int obj_num = (int)ctx->objects.size();
84
85  for (int i = 0; i < obj_num; i++) {
86      int modelIndex = ctx->objects[i]->modelIndex;
87      glBindVertexArray(VAO[modelIndex]);
88
89      Model* model = ctx->models[modelIndex];
90
91      const float* p = ctx->camera->getProjectionMatrix();
92      GLint pmatLoc = glGetUniformLocation(programId, "Projection");
93      glUniformMatrix4fv(pmatLoc, 1, GL_FALSE, p);
94
95      const float* v = ctx->camera->getViewMatrix();
96      GLint vmatLoc = glGetUniformLocation(programId, "ViewMatrix");
97      glUniformMatrix4fv(vmatLoc, 1, GL_FALSE, v);
98
99      const float* m = glm::value_ptr(ctx->objects[i]->transformMatrix * model->modelMatrix);
100     GLint mmatLoc = glGetUniformLocation(programId, "ModelMatrix");
101     glUniformMatrix4fv(mmatLoc, 1, GL_FALSE, m);
102
103     glActiveTexture(GL_TEXTURE0 + ctx->objects[i]->textureIndex);
104     glBindTexture(GL_TEXTURE_2D, model->textures[ctx->objects[i]->textureIndex]);
105     glUniform1i(glGetUniformLocation(programId, "ourTexture"), ctx->objects[i]->textureIndex);
106
107     glDrawArrays(model->drawMode, 0, model->numVertex);
108 }
109 glUseProgram(0);
```

Afterwards, I started working on the plane model. I created a model and manually set the plane position, normal, and textcoords according to the texture provided. Since the provided texture is 4.096*2.56, but the size of our plane shape is 8.192*5.12, we have to set the plane position, normal, and textcoords properly to extend the texture, which is shown below.

```
95  m = new Model();
96  float pos[] = {-4.096, 0, -2.56, -4.096, 0, 2.56, 4.096, 0, 2.56, 4.096, 0, -2.56};
97  float nor[] = {0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0};
98  float texco[] = {0.0, 0.0, 0.0, 2.0, 2.0, 2.0, 2.0, 0.0};
99
100 for (int i = 0; i < 12; i++) {
101     m->positions.push_back(pos[i]);
102     m->normals.push_back(nor[i]);
103 }
104
105 for (int i = 0; i < 8; i++) {
106     m->texcoords.push_back(texco[i]);
107 }
108
109 m->textures.push_back(createTexture("../assets/models/Wood_maps/AT_Wood.jpg"));
110
111 m->numVertex = 12;
112 m->drawMode = GL_QUADS;
113
114 ctx.models.push_back(m);
```

```
132    // TODO#3-2: Put the plane into scene
133    ctx.objects.push_back(new Object(2, glm::translate(glm::identity<glm::mat4>(), glm::vec3(4.096, 0 ,2.56))));
```

After adding the plane model properly, I push the plane into the scene. With the implementations above, we will obtain the result in the following figure, which we can see the shape of 3 cubes, 2 mugs, and a plane with their corresponding texture.

After finishing the basic part, I start with implementing the light shader program, which contains directional light, point light, and spotlight using phong shading. Let us start with the light vertex shader. In comparison to the basic vertex shader explained above, we also need to calculate the FragPos and the Normal in the light vertex shader, which calculation is shown in the following figure.

```
39    void main() {
40      TexCoord = texCoord;
41      FragPos = vec3(ModelMatrix * vec4(position, 1.0));
42      Normal = vec3(ModelNormalMatrix * vec4(normal, 1.0));
43      gl_Position = Projection * ViewMatrix * ModelMatrix * vec4(position, 1.0);
44    }
```

For the fragment shader, I separated the directional light, point light, and spotlight into different functions. When the enable of each light was activated, I will call the corresponding function and add the return vector, which is the color of the corresponding light, to the

current output color. The implementation is shown in the following figure.

```
124  void main() {
125      color = vec4(0.0, 0.0, 0.0, 1.0);
126
127      if(dl.enable == 1){
128          color += CalcDirLight();
129      }
130
131      if(pl.enable == 1){
132          color += CalcPointLight();
133      }
134
135      if(sl.enable == 1){
136          color += CalcSpotLight();
137      }
138
139      if(dl.enable == 0 && pl.enable == 0 && sl.enable == 0){
140          color = vec4(material.ambient, 1.0) * texture(ourTexture, TexCoord);
141      }
142  }
```

For the directional light, we didn't need to consider the attenuation problem. Since all the light rays are parallel it does not matter how each object relates to the light source's position since the light direction remains the same for each object in the scene. To get the output color, what we needed to do was simply multiply the light color of the directional light with the ambient, diffuse, and the specular of the material itself, which is shown in the following figure.

```
53   vec4 CalcDirLight(){
54
55       vec3 norm = normalize(Normal);
56       vec3 viewDir = normalize(viewPos - FragPos);
57       vec3 lightDir = normalize(-dl.direction);
58
59       vec3 reflectDir = reflect(-lightDir, norm);
60
61       float diff = max(dot(norm, lightDir), 0.0);
62       float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
63
64       vec4 ambient  = vec4(dl.lightColor * material.ambient, 1.0)  * texture(ourTexture, TexCoord);
65       vec4 diffuse  = vec4(dl.lightColor * material.diffuse * diff, 1.0) * texture(ourTexture, TexCoord);
66       vec4 specular = vec4(dl.lightColor * material.specular * spec, 1.0) * texture(ourTexture, TexCoord);
67
68       return (ambient + diffuse + specular);
69   }
```

For the point light, since point light has a light source with a given position, which made the light ray fade out over distance. We have to consider the effect of attenuation. Hence, as shown in the

figure below, we need to calculate the distance and the attenuation. Then multiply the diffuse and the specular with the attenuation and add them up with the ambient to get the final color of the point light.

```
70  vec4 CalcPointLight(){
71
72      vec3 norm = normalize(Normal);
73      vec3 lightDir = normalize(pl.position - FragPos);
74      vec3 viewDir = normalize(viewPos - FragPos);
75      vec3 reflectDir = reflect(-lightDir, norm);
76
77      float diff = max(dot(norm, lightDir), 0.0);
78      float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
79
80      float distance = length(pl.position - FragPos);
81      float attenuation = 1.0 / (pl.constant + pl.linear * distance + pl.quadratic * (distance * distance));
82
83      vec4 ambient  = vec4(pl.lightColor * material.ambient, 1.0)  * texture(ourTexture, TexCoord);
84      vec4 diffuse  = vec4(pl.lightColor * material.diffuse * diff, 1.0) * texture(ourTexture, TexCoord);
85      vec4 specular = vec4(pl.lightColor * material.specular * spec, 1.0) * texture(ourTexture, TexCoord);
86
87      diffuse  *= attenuation;
88      specular *= attenuation;
89
90      return (ambient + diffuse + specular);
91  }
```

The calculation of the color of the spotlight was basically the extension of the calculation of the point light. Difference from the point light, spotlight's light ray was shot in a specific direction, and will not fade out over distance. As shown in the following figure, besides attenuation, we also need to consider the theta and the cut off value. If the theta calculated is greater than the cut off value, we would display the calculated color of the spotlight. Otherwise, we would simply display the color of the ambient for the rest of the area.

```
93  vec4 CalcSpotLight(){
94
95      vec3 norm = normalize(Normal);
96      vec3 lightDir = normalize(sl.position - FragPos);
97      vec3 viewDir = normalize(viewPos - FragPos);
98      vec3 reflectDir = reflect(-lightDir, norm);
99
100     float diff = max(dot(norm, lightDir), 0.0);
101     float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
102
103     float distance = length(sl.position - FragPos);
104     float attenuation = 1.0 / (sl.constant + sl.linear * distance + sl.quadratic * (distance * distance));
105
106     float theta = dot(lightDir, normalize(-sl.direction));
107
108     vec4 ambient = vec4(sl.lightColor * material.ambient, 1.0) * texture(ourTexture, TexCoord);
109     vec4 diffuse = vec4(sl.lightColor * material.diffuse * diff, 1.0) * texture(ourTexture, TexCoord);
110     vec4 specular = vec4(sl.lightColor * material.specular * spec, 1.0) * texture(ourTexture, TexCoord);
111
112     diffuse *= attenuation;
113     specular *= attenuation;
```

```
114
115         if(theta > sl.cutOff){
116             return (ambient + diffuse + specular);
117         }
118         else{
119             return vec4(sl.lightColor * material.ambient, 1.0) * texture(ourTexture, TexCoord);
120         }
121     }
```

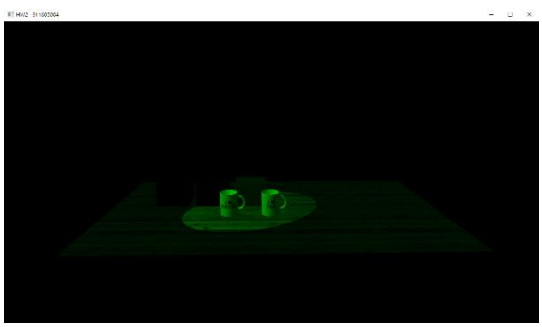
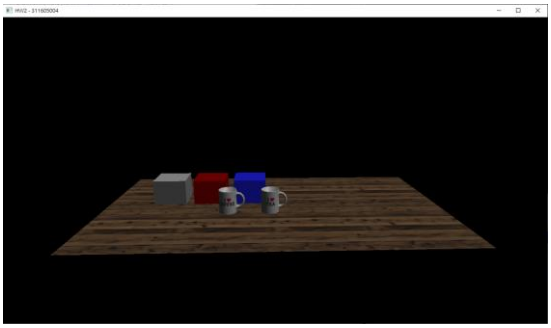
With the shaders implemented, I moved onto the procedure of passing the vertex data to the vertex buffer. This part was basically the same with the one of the basic shaders mentioned above. The following is how I implemented it.

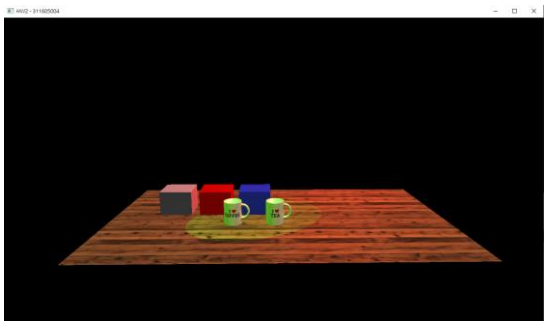
```
14      programId = quickCreateProgram(vertProgramFile, fragProgramFIle);
15
16      int num_model = (int)ctx->models.size();
17      VAO = new GLuint[num_model];
18      glGenVertexArrays(num_model, VAO);
19
20      for (int i = 0; i < num_model; i++) {
21          glBindVertexArray(VAO[i]);
22          Model* model = ctx->models[i];
23
24          GLuint VBO[3];
25          glGenBuffers(3, VBO);
26
27          glBindBuffer(GL_ARRAY_BUFFER, VBO[0]);
28          glBufferData(GL_ARRAY_BUFFER, sizeof(float) * model->positions.size(), model->positions.data(), GL_STATIC_DRAW);
29          glEnableVertexAttribArray(0);
30          glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0 * sizeof(float), (void*)0);
31
32          glBindBuffer(GL_ARRAY_BUFFER, VBO[1]);
33          glBufferData(GL_ARRAY_BUFFER, sizeof(float) * model->normals.size(), model->normals.data(), GL_STATIC_DRAW);
34          glEnableVertexAttribArray(1);
35          glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0 * sizeof(float), (void*)0);
36
37          glBindBuffer(GL_ARRAY_BUFFER, VBO[2]);
38          glBufferData(GL_ARRAY_BUFFER, sizeof(float) * model->texcoords.size(), model->texcoords.data(), GL_STATIC_DRAW);
39          glEnableVertexAttribArray(2);
40          glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0 * sizeof(float), (void*)0);
41      }
42
43      return programId != 0;
```

Lastly, we move onto rendering the objects with the shader. The beginning of this part was basically the same with the one explained above. However, we also have to make some effort passing the view matrix, the parameters of the material, the directional light, the point light, and the spotlight to the shader we just implemented in the rendering process here, which is shown in the figure below.

```
95          // Material Parameters
96          glUniform3f(glGetUniformLocation(programId, "material.ambient"),
97                      ctx->objects[i]->material.ambient.x,
98                      ctx->objects[i]->material.ambient.y,
99                      ctx->objects[i]->material.ambient.z);
```

When all the parts explained above were implemented correctly, we can get the results shown in the following figures, which are directional light, point light, and spotlight correspondingly.



3. Problems Encountered

- **Parsing the OBJ file**

  At the beginning of the project, I got some problem parsing the OBJ file. But then I found that I shouldn't push the data directly into the model. What I should do is store those data first, and push the corresponding data into the model when we detect the label "f". After discovering this, the problem was well solved.

- **Passing the data to the light shader by glUniform**

  Since I wasn't familiar with how uniform works in GLSL, and what exactly should I pass to the shader, I was stuck in the beginning of TODO 4 for about 4 days. However, after taken a deeper look of the specification of GLSL, my problem was solved and I successfully got the three light effects.