hw2_report_109550017

Task1: OBJ file parser (10%)

加載模型

cube、mug、bottle的加載方式相同，在`loadModels`載入obj模型、材質與進行縮放或旋轉，完成模型定義後放入`ctx.models`

```
Model* m = Model::fromObjectFile("../assets/models/cube/cube.obj");
m->textures.push_back(
                createTexture("../assets/models/cube/texture.bmp"));
m->modelMatrix = glm::scale(m->modelMatrix,
                            glm::vec3(0.4f, 0.4f, 0.4f));
ctx.models.push_back(m);
```

建立Object

有了模型後，我們可以根據模型在場景中建立Object，利用剛剛在`ctx.models`中的index，與translate定義擺放位置後，將Object放入`ctx.objects`中，並用`(*ctx.objects.rbegin())->material`定義該Object的Material。

```
ctx.objects.push_back(new Object(0,
                        glm::translate(glm::identityglm::mat4(),
                        glm::vec3(1.5, 0.2, 2))));
(*ctx.objects.rbegin())->material = mFlatwhite;
```

Objects的Material定義在`loadMaterial`中，其中設定了Material的ambient、diffuse、specular與shininess值。

```
mFlatwhite.ambient = glm::vec3(0.0f, 0.0f, 0.0f);
mFlatwhite.diffuse = glm::vec3(1.0f, 1.0f, 1.0f);
mFlatwhite.specular = glm::vec3(0.0f, 0.0f, 0.0f);
mFlatwhite.shininess = 10;
```

定義OBJ File Parser

由於我們先前載入的模型是使用已存在的Obj檔案，我們需要透過OBJ File Parser取出OBJ 檔案中對模型的定義，其中包含頂點位置、Texture Coordinate及法線與面資訊(分別對應v、vt、vn、f)，是構建和渲染模型所需的數據。

首先，我們取得所有 Vertex、Texture Coordinate 和 Normal的資訊，最後利用 對 Face 的定義把上述資訊整合，建構出完整的模型。

`Model::fromObjectFile` **函式：**OBJ File Parser

參考 OBJ 檔案格式可以發現，在定義開頭會以 v、vt 、 vn 與 f 標註資料特性，因此以讀取開頭文字的方式進行資料分類。

File format [edit]

Anything following a hash character (#) is a comment.

```
# this is a comment
```

An OBJ file may contain vertex data, free-form curve/surface attributes, elements, free-form curve/surface body statements, connectivity between free-form surfaces, grouping and display/render attribute information. The most common elements are geometric vertices, texture coordinates, vertex normals and polygonal faces:

```
# List of geometric vertices, with (x, y, z, [w]) coordinates, w is optional and defaults to 1.0.
v 0.123 0.234 0.345 1.0
v ...
...
# List of texture coordinates, in (u, [v, w]) coordinates, these will vary between 0 and 1. v, w are optional and
default to 0.
vt 0.500 1 [0]
vt ...
...
# List of vertex normals in (x,y,z) form; normals might not be unit vectors.
vn 0.707 0.000 0.707
vn ...
...
# Parameter space vertices in (u, [v, w]) form; free form geometry statement (see below)
vp 0.310000 3.210000 2.100000
vp ...
...
# Polygonal face element (see below)
f 1 2 3
f 3/1 4/2 5/3
f 6/4/1 3/5/3 7/6/5
f 7//1 8//2 9//3
f ...
```

在 OBJ 檔案格式中，開頭字母 **v** 後接的是頂點資訊，包含 vx、vy、vz，並利用 v 記錄頂點資訊。

```
/* first character specifies data*/
if (prefix == "v") {
  float vx, vy, vz;
```

```
  ss >> vx >> vy >> vz;
  v.push_back(vx);
  v.push_back(vy);
  v.push_back(vz);
...
```

在 OBJ 檔案格式中，開頭字母 **vt** 後接的是 Testure Coordinate 資訊，包含 vtx、vty，並利用 vt 記錄 Testure Coordinate 資訊。

```
else if (prefix == "vt") {
  float vtx, vty;
  ss >> vtx >> vty;
  vt.push_back(vtx);
  vt.push_back(vty);
```

在 OBJ 檔案格式中，開頭字母 **vn** 後接的是 Normal 資訊，包含 vnx、vny、vnz，並利用 vn 記錄 Normal 資訊。

```
else if (prefix == "vn") {
  float vnx, vny, vnz;
  ss >> vnx >> vny >> vnz;
  vn.push_back(vnx);
  vn.push_back(vny);
  vn.push_back(vnz);
}
```

在 OBJ 檔案格式中，開頭字母 **f** 後接的是 Face 資訊，由於 Face 資訊參雜頂點、Texture Coordinate 與 Normal 資訊，因此透過函式 `face_parser` 處理，並利用 vi、vti、vni 分別記錄頂點、Texture Coordinate、Normal 資訊。

一個面可能具有三個或四個點，因此除了用 for loop 讀取三次 vi、vti、vni 的組合，還會再測試一次是否有尚未讀到的資料，或有則增加點的資訊。

Vertex normal indices [edit]

Optionally, normal indices can be used to specify normal vectors for vertices when defining a face. To add a normal index to a vertex index when defining a face, one must put a second slash after the texture coordinate index and then put the normal index. A valid normal index starts from 1 and matches the corresponding element in the previously defined list of normals. Each face can contain three or more elements.

```
f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3 ...
```

```
else if (prefix == "f") {
  for (int i = 0; i < 3; i++) {
    std::string p;
    ss >> p;
    int vi, vti, vni;
    face_parser(p, vi, vti, vni);
    ...
  }
  std::string p;
  if (ss >> p) {
    int vi, vti, vni;
    face_parser(p, vi, vti, vni);

    /*vertex*/
    m->positions.push_back(v[(vi - 1) * 3]);
    m->positions.push_back(v[(vi - 1) * 3 + 1]);
    m->positions.push_back(v[(vi - 1) * 3 + 2]);
    /*normal*/
    m->normals.push_back(vn[(vni - 1) * 3]);
    m->normals.push_back(vn[(vni - 1) * 3 + 1]);
    m->normals.push_back(vn[(vni - 1) * 3 + 2]);
    /*texture coordinate*/
    m->texcoords.push_back(vt[(vti - 1) * 2]);
    m->texcoords.push_back(vt[(vti - 1) * 2 + 1]);

    m->numVertex++;
  }
}
```

`face_parser` **函式:** 取得資訊的index

藉由本函式，會讀取當前 Face 的 Vertex、Texture Coordinate 和 Normal 資訊的 index，並在先前取得的 v、vt、vn 中尋找。由 OBJ 檔案的規範格式可看出， vi、vti、vni 之間以 **/** 分隔，因此當讀取到 **/** 時變換記錄對象，此外，由於讀取格式為 string，利用 `v * 10 + (int)(str[i] - '0')` 的方式把文字轉為數字。

```
void face_parser(std::string str, int& v, int& vt, int& vn) {
  v = 0;
  vt = 0;
  vn = 0;
  int i = 0;
  for (; i < str.size() && str[i] != '/'; i++) {
    v = v * 10 + (int)(str[i] - '0');
  }
  i++;
  for (; i < str.size() && str[i] != '/'; i++) {
    vt = vt * 10 + (int)(str[i] - '0');
  }
  i++;
```

```
  for (; i < str.size() && str[i] != '/'; i++) {
    vn = vn * 10 + (int)(str[i] - '0');
  }
}
```

找出 Face 中三個點的 Vertex、Texture Coordinate 和 Normal 資訊後，將其賦值至 model 中，完成 model 定義。

```
/*vertex*/
m->positions.push_back(v[(vi - 1) * 3]);
m->positions.push_back(v[(vi - 1) * 3 + 1]);
m->positions.push_back(v[(vi - 1) * 3 + 2]);
/*texture coordinate*/
m->texcoords.push_back(vt[(vti - 1) * 2]);
m->texcoords.push_back(vt[(vti - 1) * 2 + 1]);
/*normal*/
m->normals.push_back(vn[(vni - 1) * 3]);
m->normals.push_back(vn[(vni - 1) * 3 + 1]);
m->normals.push_back(vn[(vni - 1) * 3 + 2]);
```

返回：

- `Model::fromObjectFile` 返回 `Model` 的 pointer。

Task2: Basic shader program (20%)

Render objects with texture

利用 Basic Shader 可以將基本材質貼上 Objects。

basic.vert

在 Vertex Shader 中，需要定義好頂點位置與 Texture Coordinate的資訊，此處將頂點乘上 Model Matrix、View Matrix 與 Projection Matrix，自 world coordinates 轉為 Clip Coordinates。Texture Coordinate則保持不變。

```
gl_Position = Projection * ViewMatrix * ModelMatrix
                                      * vec4(position, 1.0);
TexCoord = texCoord;
```

basic.frag

在 Fragment Shader 中定義 pixel 的顏色，透過 `texture` 函式，藉由紋理坐標 TexCoord 從 ourTexture中讀取顏色，並將該顏色賦值給 color。

```
color = texture(ourTexture, TexCoord);
```

basic.cpp

1. `BasicProgram::load()`

   為了提高渲染效率，我們使用 VAO 及 VBO 進行運算，利用 VAO 對數據進行設置，而 VBO 是實際儲存資料的地方，我們需要先建立及綁定 VAO 後再建立及綁定 VBO，方便地組織和管理頂點數據，增加渲染效率。

   ## 創建 VAO

   ```
   bool BasicProgram::load() {
     //create program
     programId = quickCreateProgram(vertProgramFile, fragProgramFIle);
     int num_model = (int)ctx->models.size();

     //create VAO
     VAO = new GLuint[num_model];
     glGenVertexArrays(num_model, VAO);
   ...
   ```

   在名為 ” VAO “ 的容器中，存有各個模型的 VAO，接著使用一個 for loop 對每個模型進行配置及輸入資料。

   ## bind VAO

   透過綁定，我們告訴OpenGL使用特定的 VAO，讓後續的設置及數據操作只作用於這個 VAO。

   ```
   for (int i = 0; i < num_model; i++) {
       // bind VAO
       glBindVertexArray(VAO[i]);
   ```

取得資料

取得當前 Model 的資料，包含 Vertex Position、Texture Coorfinate、Normal，並輸入到一維陣列中，方便後續處理。

```
// get model
Model* model = ctx->models[i];

// combine positions, normals, textures to one vector
std::vector<float> combined;
for (int j = 0; j < model->numVertex; j++) {
  combined.push_back(model->positions[j * 3]);
  combined.push_back(model->positions[j * 3 + 1]);
  combined.push_back(model->positions[j * 3 + 2]);
  combined.push_back(model->normals[j * 3]);
  combined.push_back(model->normals[j * 3 + 1]);
  combined.push_back(model->normals[j * 3 + 2]);
  combined.push_back(model->texcoords[j * 2]);
  combined.push_back(model->texcoords[j * 2 + 1]);
}
```

創建及綁定 VBO

```
// create and bind VBO
GLuint VBO[1];
glGenBuffers(1, VBO);
glBindBuffer(GL_ARRAY_BUFFER, VBO[0]);
```

輸入資料至 VBO

將 Vertex Position、Texture Coordinate、Normal 資料藉由 combined 陣列輸入至 VBO 中。

```
// pass data to VBO
glBufferData(GL_ARRAY_BUFFER, sizeof(float) * combined.size(), combined.data(),
                                                                GL_STATIC_DRAW);
```

設置 VAO

在 VBO 取得 Model 資訊後，對 VAO 進行屬性設置。

透過 `glEnableVertexAttribArray` 可以指定 VAO 中的一個屬性並進行啟用，接著透過`glVertexAttribPointer`對資料格式進行設定。

`glVertexAttribPointer`的參數設定如下：

```
glVertexAttribPointer(
    GLuint index,           // 屬性 index
    GLint size,             // 屬性維度，例如 3 表示 3 個分量 (x, y, z)
    GLenum type,            // 屬性數據類型，例如 GL_FLOAT 表示浮點數
    GLboolean normalized,   // 是否進行 normalize
    GLsizei stride,          // 步長，即每個頂點的數據在數據陣列中所占的總字節數
    const GLvoid * pointer  // 數據偏移，即從頂點數據的起始位置到當前屬性數據的偏移
);
```

這些設定向OpenGL解釋如何存儲在VAO中的頂點數據，幫助 OpenGL 對數據進行渲染。Vertex Position、Texture Coordinate、Normal 的設置大致相同，不同處在於 Texture Coordinate 為 2 維資料，而三組資料的 pointer 起始位置也因順序有些許不同。

```
// set VAO attributes
// position
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)0);
// normal
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float),
                                                  (void*)(3 * sizeof(float)));
// texture
glEnableVertexAttribArray(2);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(float),
                                                    (void*)(6 * sizeof(float)));
```

2. `BasicProgram::doMainLoop`

   主要進行渲染的部分，在這裡我們將定義好的模型及材質資訊交給 Vertex Shader 與 Fragment Shader 做計算。

Bind model VAO

對於場景中的每個 Objects，從 VAO 中得到該 Objects 使用的 Model，並將該 Model 的 VAO 進行綁定，讓該 VAO 的資訊作用在 Object 上。

```
void BasicProgram::doMainLoop() {
  glUseProgram(programId);
  int obj_num = (int)ctx->objects.size();

  // for every objects
  for (int i = 0; i < obj_num; i++) {

    // bind model VAO
    int modelIndex = ctx->objects[i]->modelIndex;
    glBindVertexArray(VAO[modelIndex]);
...
```

頂點資料送入 Vertex Shader

接著我們將 Vertex Position 送入 Vertex Shader 中，由於頂點轉換以

`gl_Position = Projection * ViewMatrix * ModelMatrix * vec4(position, 1.0);`的方式進行，我們須一一將要用到的 Projection Matrix、View Matrix、Model Matrix 傳入 Vertex Shader 中。

首先我們取得轉換矩陣與 Vertex Shader 中名為 "Projection" 的 uniform 變數位置，最後使用`glUniformMatrix4fv`函數將投影矩陣數據傳遞給 Vertex Shader中的 "Projection" uniform變數。

View Matrix 與 Model Matrix 的作法也相同，其中，傳入 Vertex Shader 的 Model Matrix 為 transformMatrix 與原先 Model 預設 Model Matrix 相乘的結果。

```
// projection matrix
const float* p = ctx->camera->getProjectionMatrix();
GLint pmatLoc = glGetUniformLocation(programId,
                                     "Projection");
glUniformMatrix4fv(pmatLoc, 1, GL_FALSE, p);


// view matrix
const float* v = ctx->camera->getViewMatrix();
GLint vmatLoc = glGetUniformLocation(programId,
                                     "ViewMatrix");
glUniformMatrix4fv(vmatLoc, 1, GL_FALSE, v);
```

```
// model matrix
Model* model = ctx->models[modelIndex];
const float* m = glm::value_ptr(ctx->objects[i]
                    ->transformMatrix * model->modelMatrix);
GLint mmatLoc = glGetUniformLocation(programId,
                                          "ModelMatrix");
glUniformMatrix4fv(mmatLoc, 1, GL_FALSE, m);
```

貼上材質

綁定當前 Texture，使用 `glDrawArrays` 將 Texture 貼上 Model。

```
// texture
glBindTexture(GL_TEXTURE_2D, model->
                     textures[ctx->objects[i]->textureIndex]);
glDrawArrays(model->drawMode, 0, model->numVertex);
```

Task3: Display texture plane (10%)

Draw obj (5%) (positions, normals, texcoords…)

預設平面模型為大小 2 X 2 且中心位於 (0, 0) 的四邊形，並設定點資訊（Vertex Position、Normal、Texture Coordinate）每個頂點都會包含這些資訊。

Vertex Position 為三維，又有四個點，因此有十二個元素；Normal 為三維，又有四個點，因此有十二個元素；Texture Coordinate 為二維，又有四個點，因此有八個元素。

```
// vertex position (4 vertices, 3 elements each)
std::vector<float> p = {-1.0, 0.0, -1.0, -1.0, 0.0, 1.0, 1.0, 0.0, 1.0, 1.0, 0.0, -1.0};
// normal (4 vertices, 3 elements each)
std::vector<float> n = {0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0};
// texture coordinate (4 vertices, 2 elements each)
std::vector<float> t = {0.0, 0.0, 0.0, 2.0, 2.0, 2.0, 2.0, 0.0};
```

設定完成後，宣告模型並將剛剛設定完的點資訊送入模型中，並將點數量設為4，將 drawMode 設為 GL_QUADS，且對 modelMatrix 縮放為需要的大小。

```
// create model
m = new Model();
// set model attributes: pos, normal, texcoord
m->positions.insert(m->positions.end(), p.begin(), p.end());
```

```
m->normals.insert(m->normals.end(), n.begin(), n.end());
m->texcoords.insert(m->texcoords.end(), t.begin(), t.end());

// set model attributes: numVertex, drawMode, modelMatrix
m->numVertex = 4;
m->drawMode = GL_QUADS;
// size: 8.096 X 5.12
m->modelMatrix = glm::scale(m->modelMatrix, glm::vec3(4.096f, 1.0f, 2.56f));
```

Texture (5%) (assets/models/Wood_maps/AT_Wood.jpg)

將平面模型的 Texture 存入模型，最後，將完成設定的模型存入 Models。

```
m->textures.push_back(createTexture("../assets/models/Wood_maps/AT_Wood.jpg"));
ctx.models.push_back(m);
```

將平面放入場景

將平面 Object 中心放在場景中 (4.096, 0.0, 2.56) 的位置。

```
ctx.objects.push_back(new Object(3, glm::translate(glm::identity<glm::mat4>(),
                                                   glm::vec3(4.096, 0.0, 2.56))));
```

Task4: Light shader program (three light source mixed)

light.vert

在 Vertex Shader 中，需要定義好頂點位置與 Texture Coordinate的資訊，此處將頂點乘上 Model Matrix、View Matrix 與 Projection Matrix，自 world coordinates 轉為 Clip Coordinates。Texture Coordinate則保持不變。

用 FragPos、Normal 記錄物體表面及法向量在世界坐標下的值，以便後續 Fragment Shader 進行 Phong Shading 使用。

```
gl_Position = Projection * ViewMatrix * ModelMatrix * vec4(position, 1.0);
TexCoord = texCoord;
```

```
FragPos = vec3(ModelMatrix * vec4(position, 1.0f));
Normal = vec3(ModelNormalMatrix * vec4(normal, 1.0));
```

light.frag

Using phong shading

在 Phong Shading 的定義中，物體的表面反射光由三種光線所組成，分別為 Ambient(環境光)、 Diffuse(漫射光) 與 Specular(鏡面反射光)，其中各項光線的算式如下所示：

1. **環境光（Ambient Light）：**
   - **數學式：** $Ia = Ka \cdot Iambient$
   - **符號解釋：**
     - Ia：環境光的強度。
     - Ka：環境光系數，表示材質的環境光屬性。
     - $Iambient$：環境光的強度。

2. **漫射光（Diffuse Reflection）：**
   - **數學式：** $Id = Kd \cdot I \cdot (L \cdot N)$
   - **符號解釋：**
     - Id：漫反射光的強度。
     - Kd：漫反射系數，表示材質的漫反射屬性。
     - I：光源的強度。
     - L：光源方向向量（單位向量）。
     - N：表面法向量（單位向量）。

3. **鏡面反射光（Specular Reflection）：**
   - **數學式：** $Is = Ks \cdot I \cdot (R \cdot V)^n$
   - **符號解釋：**
     - Is：鏡面反射光的強度。

- Ks：鏡面反射系數，表示材質的鏡面反射屬性。
- I：光源的強度。
- R：反射方向向量（單位向量）。
- V：視線方向向量（單位向量）。
- n：反光度指數，控制鏡面高光的散射程度。

在 Fragment Shader 中，以 Material 的結構建構基本的 Phong Shading 架構，其中包含 Ambient(環境光)、 Diffuse(漫射光) 與 Specular(鏡面反射光) 及 Shininess(反射係數)。

```
struct Material {
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float shininess;
};
```

1. Directional Light (15%)

首先將 Directional Light 組織為 Struct，方便之後對資料進行運算，其中包含確認 Directional Light 是否啟用、方向及光照顏色。

```
struct DirectionLight
{
    int enable;
    vec3 direction;
    vec3 lightColor;
};
```

當 Directional Light 確認開啟時，我們可以依照 Phong Shading 的算式計算 Ambient、 Diffuse 與 Specular。

```
if (dl.enable == 1) {

  // ambient: Ia = I_ambient * Ka
  dAmbient = vec4(dl.lightColor * material.ambient, 1.0);
```

```
  // diffuse: Id = I · Kd · (L · N)
  float ddCoef = dot(normalize((-1) * dl.direction), normalize(Normal)); // (L · N)
  dDiffuse = vec4(dl.lightColor * material.diffuse, 1.0) * max(ddCoef, 0.0);

  // specular: Is = I · Ks · (R · V) ^ n
  vec3 reflect_dir = reflect(dl.direction, normalize(Normal)); // R
  vec3 view_dir = viewPos - FragPos; // V
  float dsCoef = dot(normalize(reflect_dir), normalize(view_dir)); // (R · V)
  dSpecular = vec4(dl.lightColor * material.specular, 1.0) * pow(max(dsCoef, 0.0),
                                                                 material.shininess);
}
```

2. Point Light (15%)

Point Light 不同於 Directional Light 之處在於光照存在衰減現象，基於 Phong Shading 的計算需要再乘以衰減值，衰減值公式如下所示：

$$L_{distance} = \frac{L_1}{Attenuation_{constant} + Attenuation_{linear} * distance + Attenuation_{exp} * distance^2}$$

和 Directional Light 相同，首先將 Point Light 組織為 Struct，方便之後對資料進行運算，其中包含確認 Point Light 是否啟用、方向及光照顏色。由於 Point Light 存在衰減現象，我們另外以三個浮點數變數做為衰減常數。

```
struct PointLight {
  int enable;
  vec3 position;
  vec3 lightColor;

  // Paramters for attenuation formula
  float constant;
  float linear;
  float quadratic;
};
```

當 Point Light 確認開啟時，首先依照公式計算衰減常數，接著以 Phong Shading 的算式計算 Ambient、 Diffuse 與 Specular 並分別與衰減常數相乘。

```
if (pl.enable == 1) {
  // attenuation: 1/(constant + linear * dist + exp * dist * dist)
  float dist = length(FragPos - pl.position);
  float attenuation = 1.0 / (pl.constant + pl.linear * dist + pl.quadratic
                                                              * dist * dist);

  // ambient: Ia = I_ambient * Ka * attenuation
  pAmbient = vec4(pl.lightColor * material.ambient, 1.0) * attenuation;

  // diffuse: Id = I · Kd · (L · N) * attenuation
  vec3 light_dir = FragPos - pl.position;
  float pdCoef = dot(normalize((-1) * light_dir), normalize(Normal));
  pDiffuse = vec4(pl.lightColor * material.diffuse, 1.0) *
                                              max(pdCoef, 0.0) * attenuation;

  // specular: Is = I · Ks · (R · V) ^ n * attenuation
  vec3 reflect_dir = reflect(light_dir, normalize(Normal));
  vec3 view_dir = viewPos - FragPos;
  float psCoef = dot(normalize(reflect_dir), normalize(view_dir));
  pSpecular = vec4(pl.lightColor * material.specular, 1.0) *
                              pow(max(psCoef, 0.0), material.shininess) * attenuation;
}
```

3. Spot Light (15%)

Spot Light 與 Point Light 在光照計算上相同，不同之處在於 Spot Light 需要對範圍進行切除，因此在 Struct 中多加了浮點數變數，以 cos 角的方式記錄光照範圍。

```
struct Spotlight {
  int enable;
  vec3 position;
  vec3 direction;
  vec3 lightColor;
  float cutOff;

  // Paramters for attenuation formula
  float constant;
  float linear;
  float exp;
};
```

當光源為 Spot Light 時，會將 Objects 分類為「有被 Spot Light 照射到」與「沒有被 Spot Light 照射到」兩種類別，當 Objects 屬於「沒有被 Spot Light 照射到」的類別時，僅會有基本的 Ambient 造成影響，Diffuse 與 Specular 是不需要考慮的。因此，

我們需要以 cutOff 變數來判斷當前 fragment 與 Spot Light 的連線與 Spot Light 的照射角度夾角使否小於某個值，若是，則代表 Object 處於 Spot Light 的照射範圍內，需考慮 Diffuse 及 Specular。Ambient、 Diffuse 及 Specular 的計算方式與 Point Light 相同。

```
if (sl.enable == 1) {
  // attenuation: 1/(constant + linear * dist + exp * dist * dist)
  float dist = length(FragPos - sl.position);
  float attenuation = 1.0 / (sl.constant + sl.linear * dist + sl.quadratic
                                                              * dist * dist);

  // ambient: Ia = I_ambient * Ka * attenuation
  sAmbient = vec4(sl.lightColor * material.ambient, 1.0) * attenuation;


  // check if position is in the spotlight
  vec3 light_dir = FragPos - sl.position;
  if (dot(normalize(light_dir), normalize(sl.direction)) > sl.cutOff) {

      // diffuse: Id = I · Kd · (L · N) * attenuation
      float sdCoef = dot(normalize((-1) * light_dir), normalize(Normal));
      sDiffuse = vec4(sl.lightColor * material.diffuse, 1.0) * max(sdCoef, 0.0)
                                                                * attenuation;

      // specular: Is = I · Ks · (R · V) ^ n * attenuation
      vec3 reflect_dir = reflect(light_dir, normalize(Normal));
      vec3 view_dir = viewPos - FragPos;
      float ssCoef = dot(normalize(reflect_dir), normalize(view_dir));
      sSpecular = vec4(sl.lightColor * material.specular, 1.0) * pow(max(ssCoef, 0.0),
                                                    material.shininess) * attenuation;
  }
```

light.cpp

主要渲染工作於`LightProgram::doMainLoop`中進行。

`LightProgram::doMainLoop`

不同於 basic.cpp，light.cpp 因涉及到 Phong Shading 及不同的光照方式，需要傳入 shader 的變數增加，包含 Material、Directional Light、Point Light、Spot Light 相關資訊。

```
// pass variables to shader
Model* model = ctx->models[modelIndex];
Camera* camera = ctx->camera;
Object* object = ctx->objects[i];
```

```
glm::mat4 modelNormalMatrix = glm::transpose(glm::inverse(modelMatrix));
setMat4("ModelNormalMatrix", glm::value_ptr(modelNormalMatrix));
setVec3("viewPos", camera->getPosition());

// material
Material material = ctx->objects[i]->material;
setVec3("material.ambient", glm::value_ptr(material.ambient));
setVec3("material.diffuse", glm::value_ptr(material.diffuse));
setVec3("material.specular", glm::value_ptr(material.specular));
setFloat("material.shininess", material.shininess);

// directional light
setInt("dl.enable", ctx->directionLightEnable);
setVec3("dl.direction", glm::value_ptr(ctx->directionLightDirection));
setVec3("dl.lightColor", glm::value_ptr(ctx->directionLightColor));

// point light
setInt("pl.enable", ctx->pointLightEnable);
setVec3("pl.position", glm::value_ptr(ctx->pointLightPosition));
setVec3("pl.lightColor", glm::value_ptr(ctx->pointLightColor));
setFloat("pl.constant", ctx->pointLightConstant);
setFloat("pl.linear", ctx->pointLightLinear);
setFloat("pl.quadratic", ctx->pointLightQuardratic);

// spot light
setInt("sl.enable", ctx->spotLightEnable);
setVec3("sl.position", glm::value_ptr(ctx->spotLightPosition));
setVec3("sl.direction", glm::value_ptr(ctx->spotLightDirection));
setVec3("sl.lightColor", glm::value_ptr(ctx->spotLightColor));
setFloat("sl.cutOff", ctx->spotLightCutOff);
setFloat("sl.constant", ctx->spotLightConstant);
setFloat("sl.linear", ctx->spotLightLinear);
setFloat("sl.quadratic", ctx->spotLightQuardratic);
```

Control

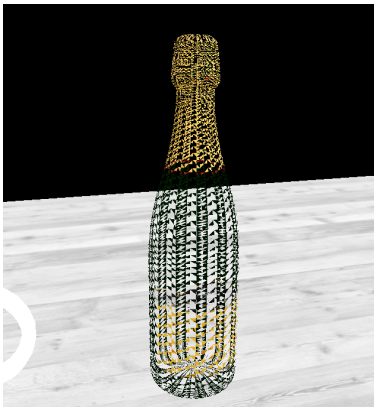
對shader與光源的控制如下所示：

- 1: 使用 example shader program
- 2: 使用 basic shader program
- 3: 使用 light shader program
- 4: 開啟/關閉 direction light
- 5: 開啟/關閉 point light
- 6: 開啟/關閉 spot light
- k/l: 改變 point light 位置
- h/j: 改變 point light 顏色

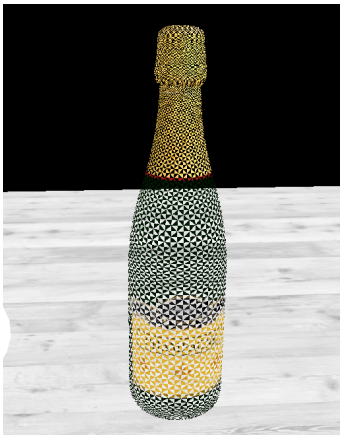
- i/o: 改變 spot light 位置
- y/u: 改變 spot light 顏色

問題

原先在畫面中的bottle會有繪製不完整的狀況出現，如圖一所示：



圖一：GL_QUADS



圖二：GL_TRIANGLES

圖三：GL_TRIANGLE_FAN

我之後認為是model的繪製模式錯誤，試了使用 GL_TRIANGLES(圖二) 或 GL_TRIANGLE_FAN(圖三) 可以改善狀況，但仍無法完美顯示材質。

之後思考或許是 vertex 處理有問題，因此回去詳細閱讀了 hint，發現有寫道「每個面可能有3或4個 vertex」，而我原先在遇到 face 時只處理3個 vertex 的情況，在加入判斷使否存在第4個 veretx 並增加 vertex 資訊後，便能成功使用 GL_QUADS繪製了。

Reference

https://ogldev.org/www/tutorial20/tutorial20.html

https://en.wikipedia.org/wiki/Wavefront_.obj_file

https://github.com/B10732009/computer-graphics/tree/main/hw2