实验一　三维网格模型操作

严铖　517021910823

1　实验目的与基本要求

1. 掌握 Obj 文件的读入；(读入提供的 dragon.obj 文件)
2. 利用给定的数据结构类，建立读入网格模型数据结构；
3. 利用 OpenGL 类库，对三维模型进行绘制；
4. 利用 OpenGL 类库，增加采用鼠标交互方式对三维模型进行旋转、放缩、平移等操作；
5. 利用 OpenGL 类库，添加光照，渲染效果；
6. 利用 OenGL 类库，进行材质设定，实现半透明效果。

2　实验步骤

本次实验基于 LearnOpenGL CN 教程完成,教程网址为:https://learnopengl-cn.github.io。在此向教程的作者表示由衷的感谢。

实验中使用的是 GLFW 环境以及开源库 GLAD，均使用的是最新版本。此外，OpenGL 的数学库 glm 也会在实验中被使用。

以下，根据实验的基本要求，我们分别讨论各项所需的实验步骤

2.1　obj 文件的读入

实验所提供的 obj 文件主要由以下两部分组成：各个顶点在世界坐标系下的位置以及每个三角形面所用到的顶点。考虑 OpenGL 的数据结构特性，我们将所读的数据存储在两个一维向量`vertices`, `faces` 中 (相同点的三个分量坐标按顺序存储，一个顶点全部存储完了再存储下一个顶点)。但

由于在之后的数组索引中顶点下标应从 0 开始，而 obj 文件内顶点索引从 1 开始，所以在读入时对索引进行减一的预处理。

因此我们定义函数:`void read_data(string filename);` 其中`filename` 为文件所在的路径。该函数将文件读入并把数据存储在向量里面。

函数的实现： 函数基于 c++ 的`fstream` 库读入文件，对文件进行逐行读取。文件内每行的第一个字符可作为数据的标识符（点为 v，面为 f），逐行处理，直到文件读完为止。函数具体的实现可参见源代码。

2.2 模型的绘制

有了存储三维模型信息的两个向量`vertices`，`faces`，我们便可以使用 OpenGL 对模型进行绘制了。

2.2.1 窗口的建立

首先当然应该构建显示模型的窗口，经过相关资料的查询，代码截图如下所示

```
52  int main()
53  {
54      glfwInit();     // Initialization
55      glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);  // GL version constraint
56      glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);  // use core profile
57      glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
58
59      // create a window
60      GLFWwindow* window = glfwCreateWindow(WIDTH, HEIGHT, "Dragon", NULL, NULL);
61      if (window == NULL)
62      {
63          cout << "Failed to create GLFW window" << endl;
64          glfwTerminate();
65          return -1;
66      }
67      glfwMakeContextCurrent(window);
68      glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
```

图 1: 窗口创建代码

该段代码对 GLFW 进行了初始化，并建立了一个窗口变量`windows`

类似的，我们也对 glad 进行初始化，具体过程不再赘述。

2.2.2 着色器

在实验中，我们使用一个自己写的着色器类。创建一个该类的对象需要有两个参数：顶点着色器代码文件路径以及片段着色器代码文件路径。着色

器对象创建时会编译这两个代码文件，具体的实现可见源代码
`shader.h, vs.txt, fs.txt`

2.2.3 VBO,VAO,EBO 的建立

VBO 为顶点缓冲对象，用于管理储存顶点数据的内存。我们会调用相关的函数，将已经读入顶点坐标数据的向量 `vertices` 缓存到 VBO 对象中

VAO 为顶点数组对象，随后的顶点属性调用均会存储在 VAO 对象之中。

EBO 为索引缓冲对象,用于管理绘制模型所需要的各项索引（即为`faces`内的顶点标识）。与 VBO 类似，我们也将向量`faces` 与 EBO 对象进行绑定。

相关的代码如下图所示：

```
95    unsigned int VBO, VAO;
96    glGenVertexArrays(1, &VAO);
97    glGenBuffers(1, &VBO);
98
99    glBindVertexArray(VAO);
100
101   glBindBuffer(GL_ARRAY_BUFFER, VBO);
102   glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(float), &vertices[0], GL_STATIC_DR
103
104   unsigned int EBO;
105   glGenBuffers(1, &EBO);
106   glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
107   glBufferData(GL_ELEMENT_ARRAY_BUFFER, faces.size() * sizeof(unsigned int), &faces[0], GL_
108
109
110   // position attribute
111   glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
112   glEnableVertexAttribArray(0);
113   // normal attribute
114   glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(flo
115   glEnableVertexAttribArray(1);
```

图 2: VBO,VAO,EBO 创建代码

2.2.4 渲染循环

有了这些对象，我们就可以开始进行渲染了。由于我们希望实时看到画面，我们就需要写一个 while 循环对模型进行实时的渲染。在渲染循环中，首先进行窗口颜色的设置，其次激活我们刚刚定义的着色器，最后调用相关的画图函数就可以看到一只龙了。

但是由于还没有加入光照、材质、深度等相关信息，所以暂时只能看到一个三维模型的二维投影。

```
120     // render loop
121     while (!glfwWindowShouldClose(window))
122     {
123         float currentFrame = glfwGetTime();
124         deltaTime = currentFrame - lastFrame;
125         lastFrame = currentFrame;
126 
127         processInput(window);       // DETECT ESC
128 
129         glClearColor(0.2f, 0.3f, 0.3f, 0.0f);
130         glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
131         // glClear(GL_COLOR_BUFFER_BIT);
132         // be sure to activate shader when setting uniforms/drawing objects
133         ourShader.use();
134 
```

图 3: 渲染循环相关代码

2.3 键鼠交互

2.3.1 摄像机类

为了实现对于平移、旋转等交互方式的实现，我们需要自己定义一个摄像机类。摄像机类的对象定义了我们观察物体所在的视点，变换矩阵`model`，`view`，`projection` 可通过摄像机内计算得出的数据进行实时的改变以达到视角的变换。该类的具体实现详见源代码`camera.h`

2.3.2 平移

对于平移操作，我们使用键盘 WASD 四个键的输入进行控制。在每个渲染循环之中，我们会对四个键的状态进行分析，如果某个键被触发，就将我们的摄像机向某个方向以一个恒定的速率进行移动。处理平移的函数代码如下图所示：

其中`deltaTime` 为循环一次所相间隔的时间并以此为依据进行速度的调控。

2.3.3 缩放

缩放与平移完全类似。我们使用鼠标的滚轮进行画面的缩放，读取鼠标滚轮需要对缩放函数进行注册从而得到滚轮偏移的大小。之后将偏移值读入摄像机处理缩放的函数方法之中进行处理。

```
198  void processInput(GLFWwindow* window)
199  {
200      if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
201          glfwSetWindowShouldClose(window, true);
202
203      if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
204          camera.ProcessKeyboard(FORWARD, deltaTime);
205      if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
206          camera.ProcessKeyboard(BACKWARD, deltaTime);
207      if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
208          camera.ProcessKeyboard(LEFT, deltaTime);
209      if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
210          camera.ProcessKeyboard(RIGHT, deltaTime);
211  }
```

图 4: 平移处理

2.3.4 旋转

对物体的旋转与前两个交互略有不同。前两个的处理是对摄像机视角变换的处理，而旋转是对物体本身的变动。因此在旋转中应当直接用鼠标移动的偏移量去改变`model` 矩阵中的值而摄像机方位不变。类似的，我们也需要将旋转处理函数进行注册以得到鼠标移动的偏移量。旋转的实现函数如图所示：

```
173  void mouse_callback(GLFWwindow* window, double xpos, double ypos)
174  {
175      if (firstMouse)
176      {
177          lastX = xpos;
178          lastY = ypos;
179          firstMouse = false;
180      }
181
182      float xoffset = xpos - lastX;
183      float yoffset = lastY - ypos;
184      lastX = xpos;
185      lastY = ypos;
186
187      model = glm::rotate(model, glm::radians(0.2f * yoffset), glm::vec3(1, 0, 0));
188      model = glm::rotate(model, glm::radians(0.2f * xoffset), glm::vec3(0, 1, 0));
189
190      // camera.ProcessMouseMovement(xoffset, yoffset, true);
191  }
```

图 5: 旋转处理

2.4 光照

在加入光照效果之前，我们应当首先进行各个顶点法向量的计算。顶点的法向量定义为所有经过该顶点的面的法向量之和。因此我们要对我们的数据进行一些新的处理，在向量`vertices`之中，在每个顶点后加入其相应的标准化法向量的三维坐标（这样每一个顶点就有六个参数了）。与此同时，对一些相关的函数进行参数的调整。

我们使用上课所介绍的冯氏光照模型对我们的三维对象进行渲染。在主程序中，我们需要提供的是：光源的所在位置，视点所在的位置，光的颜色以及物体的颜色。将这四个参数传入着色器之中，着色器就可以根据我们既定的需要对物体进行渲染了。

改变后的片段着色器以实现冯氏光照模型的代码如下所示：

```
12  void main()
13  {
14      // ambient
15      float ambientStrength = 0.1;
16      vec3 ambient = ambientStrength * lightColor;
17
18      // diffuse
19      vec3 norm = normalize(Normal);
20      vec3 lightDir = normalize(lightPos - FragPos);
21      float diff = max(dot(norm, lightDir), 0.0);
22      vec3 diffuse = diff * lightColor;
23
24      // specular
25      float specularStrength = 0.5;
26      vec3 viewDir = normalize(viewPos - FragPos);
27      vec3 reflectDir = reflect(-lightDir, norm);
28      float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
29      vec3 specular = specularStrength * spec * lightColor;
30
31      vec3 result = (ambient + diffuse + specular) * objectColor;
32      FragColor = vec4(result, 0.7);
33  }
```

图 6: 实现了冯氏光照模型的片段着色器主函数

2.5 半透明材质

半透明材质的设定十分简单，在片段着色器之中，我们输出的颜色是一个有四个分量的向量。其中前三个分量为代表 RGB 的颜色三元组，而最后一个为物体的透明度 α，其值通常默认为 1。因此我们调整 α 至一个小于 1 的参数，并且将 OpenGL 的混合模式开启，就可以看到半透明材质的龙了。

最后所绘制的龙如图所示：

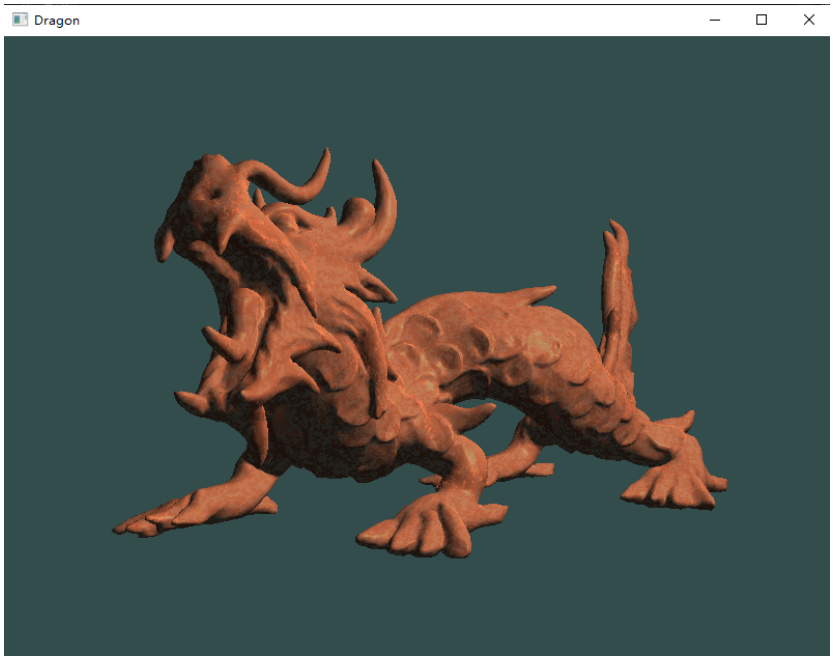

图 7: 实现了各项要求的最终模型

3 实验小结

本次实验我对 OpenGL 类库以及模型从构建到渲染等步骤都有了一个很深刻的认识。在实验之中由于对相关函数的不熟悉，走了许许多多的弯路。从创建窗口到看到画面之中的第一个二维模型就花了我两天的时间（一个参数忘记设置直接导致画面无法显示，让我对照相关材料好一会时间才找到自己的错误）。

实验让我对 c++ 编程以及面向对象的思想有了更好的掌握，有了相关的类，在未来调用一些方法以及对类内进行不断地扩充也会更加的方便。