计算机图形学实验报告
Assignment0

课程名称：　　计算机图形学

实验名称：　　Assignment0

学生学院：　　泰山学堂

学生班级：　　2017 级计算机取向

学生学号：　　201705301350

学生姓名：　　宋建涛

提交日期：　　2019.10.26

目录

实验目标

本实验是图形学系列试验中的第一个实验，本实验的实验要求主要由两个：

1. 读一个三角形网格数据文件（包含网格纹理）
2. 使用 OpenGL 绘制该模型（要求能够实现颜色的变换和简单的移动）

实验概览

首先根据实验要求，我们可以将大实验简单的分为四个阶段：

1. 读取模型文件并保存相关数据
2. 将读取的模型文件通过 OpenGL 绘制出来
3. 实现模型颜色的改变
4. 实现模型简单的移动（旋转）

接下来我们就分这四部分来看一下我们需要做什么。

读取模型文件并保存相关数据

认识 obj 模型

OBJ 文件是 Wavefront 公司为它的一套基于工作站的 3D 建模和动画软件"Advanced Visualizer"开发的一种文件格式，这种格式同样也以通过 Maya 读写。

OBJ 文件是一种文本文件，可以直接用写字板打开进行查看和编辑修改。

Obj 文件中每一行的行首通过关键字来说明本行说明的内容类型，常见的关键字如下

1.v 顶点

2.vt 纹理坐标

3.vn 顶点法向量

4.f 面

一、顶点

格式：v x y z

意义：每个顶点的坐标

二、纹理坐标

格式：vt u v w

意义：绘制模型的三角面片时，每个顶点取像素点时对应的纹理图片上的坐标。纹理图片的坐标指的是，纹理图片如果被放在屏幕上显示时，以屏幕左下角为原点的坐标。

注意：w 一般用于形容三维纹理，大部分是用不到的，基本都为 0。

三、顶点法向量

格式：vn x y z

意义：绘制模型三角面片时，需要确定三角面片的朝向，整个面的朝向，是由构成每个面的顶点对应的顶点法向量的做矢量和决定的（xyz 的坐标分别相加再除以 3 得到的）。

四、面

格式 ：f v/vt/vn v/vt/vn v/vt/vn（f 顶点索引 / 纹理坐标索引 / 顶点法向量索引）

意义：绘制三角面片的依据，每个三角面片由三个 f 构成，由 f 可以确定顶点、顶点的对应的纹理坐标（提取纹理图片对应该坐标的像素点）、通过三个顶点对应的顶点法向量可以确定三角面的方向。

补充：有些模型可能会出现四边形的绘制方式，那样的模型关于面的数据描述是这样的 f v/vt/vn v/vt/vn v/vt/vn v/vt/vn ，比三角面绘制方式多一项数据。

存储内容的结构

为了存储从文件中读取到的信息，我们一共定义了三种结构如下，

```
template<typename T>
class Ternary {
public:
    T x, y, z;
    Ternary() {};
    Ternary(T a, T b, T c);
};

class Surface {
public:
    Ternary<int> vertex;
    Ternary<int> texture;
    Ternary<int> normal;
    Surface(Ternary<int>, Ternary<int>, Ternary<int>);
};

class ObjLoader {
    public:
        vector<Ternary<float>> obj_vertexe;
        vector<Ternary<float>> obj_texture;
        vector<Ternary<float>> obj_normal;
        vector<Surface> obj_surface;
        ObjLoader(string filename);//构造函数
};
```

其中 Ternary 仅代表一个包含三个数据的结构，它可以使 float 类型的也可以是 string 类型的。

Surface 存储着从文件中读取到的面结构，在之前的介绍中我们简单了解了 obj 文件中

对三角面片结构的描述，对每一个面片我们要存储三个点，每个点都有它的顶点索引、顶点纹理索引、顶点法向量索引，所以我们在 Surface 中定义了三个 Ternary<int>类型的变量来分别存储这些信息。

ObjLoader 存储着我们从 obj 文件中读取出来的所有信息，包含顶点坐标 obj_vertexe、顶点纹理坐标 obj_texture、顶点法向坐标 obj_normal 和三角面片信息 obj_surface。知道了这些信息我们就能使用 OpenGL 重绘读取的模型了。

读取文件并存储的函数实现是在 ObjLoader 的重构函数中，我们期望能够在模型定义的过程中完成 obj 文件的读取并存储操作，ObjLoader 的重构函数实现如下：

```
1.  ObjLoader::ObjLoader(string filename)
2.  {
3.      string line;
4.      fstream f;
5.      f.open(filename, ios::in);
6.      if (!f.is_open()) {
7.          cout << "Something Went Wrong When Opening Objfiles" << endl;
8.      }
9.
10.     while (!f.eof()) {
11.         getline(f, line);//拿到 obj 文件中一行，作为一个字符串
12.         vector<string> data;
13.         split(line, data);
14.         if (data.size() < 4) {
15.             continue;
16.         }
17.         if (data[0] == "v") {//读入的一行是点的坐标信息
18.             obj_vertexe.push_back(Ternary<float>(stringToNum<float>(data[2])
    , stringToNum<float>(data[3]), stringToNum<float>(data[4])));
19.         }
20.         else if(data[0] == "vn"){//读入的一行是点的法向量
21.             obj_normal.push_back(Ternary<float>(stringToNum<float>(data[1]),
     stringToNum<float>(data[2]), stringToNum<float>(data[3])));
22.         }
23.         else if (data[0] == "vt") {//读入的一行是点的纹理坐标
24.             obj_texture.push_back(Ternary<float>(stringToNum<float>(data[1])
    , stringToNum<float>(data[2]), stringToNum<float>(data[3])));
25.         }
26.         else if (data[0] == "f") {//读入的一行是面的信息
27.             vector<string> tmp;
28.             Ternary<int> tmp_surface;
29.             Ternary<int> tmp_texture;
30.             Ternary<int> tmp_vn;
31.             split(data[1], tmp, '/');
32.             tmp_surface.x = stringToNum<int>(tmp[0]);
33.             tmp_texture.x = stringToNum<int>(tmp[1]);
```

```
            tmp_vn.x = stringToNum<int>(tmp[2]);

            split(data[2], tmp, '/');
            tmp_surface.y = stringToNum<int>(tmp[0]);
            tmp_texture.y = stringToNum<int>(tmp[1]);
            tmp_vn.y = stringToNum<int>(tmp[2]);

            split(data[3], tmp, '/');
            tmp_surface.z = stringToNum<int>(tmp[0]);
            tmp_texture.z = stringToNum<int>(tmp[1]);
            tmp_vn.z = stringToNum<int>(tmp[2]);

            obj_surface.push_back(Surface(tmp_surface, tmp_texture, tmp_vn))
    ;
        }
        //cout << "ok" << endl;
    }
    f.close();
}
```

将读取的模型通过 OpenGL 重新绘制

在这个阶段我们主要需要做的事如下：定义相机顶点坐标、创建窗口并初始化 OpenGl 窗口参数、对模型进行绘制。

初始化

首先相机坐标的定义我们可以用到之前的 Ternary 结构，毕竟三维空间中说明一个点也是用到三个数字，只不过我们这里需要的坐标是 float 类型。

之后我们需要进行初始化操作，在这里我们把初始化操作封装成函数 void init():

```
void init() {
    camera.x = -20;
    camera.y = 30;
    camera.z = 20;
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("ObjLoader");

    //材质反光性设置
    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };  //镜面反射参数
    GLfloat mat_shininess[] = { 50.0 };               //高光指数
```

```
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
    GLfloat white_light[] = { 1.0, 1.0, 1.0, 1.0 };   //灯位置(1,1,1), 最后1-开关
    GLfloat Light_Model_Ambient[] = { 0.2, 0.2, 0.2, 1.0 }; //环境光参数

    glClearColor(0.0, 0.0, 0.0, 0.0);  //背景色
    glShadeModel(GL_SMOOTH);           //多变性填充模式

                                       //材质属性
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);

    //灯光设置
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, white_light);   //散射光属性
    glLightfv(GL_LIGHT0, GL_SPECULAR, white_light);  //镜面反射光
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, Light_Model_Ambient);  //环境光参数

    glEnable(GL_LIGHTING);   //开关:使用光
    glEnable(GL_LIGHT0);     //打开0#灯
    glEnable(GL_DEPTH_TEST); //打开深度测试
}
```

绘制函数

Glut 运行过程的绘制函数原型为 void glutDisplayFunc(void (*func)(void));这里需要我们写的是绘制函数 void(*func)(void),在我们函数中的体现就是绘制函数 void Draw()。我们来看一下代码：

```
void Draw() {

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    //开始绘制
    for (int i = 0; i < objModel.obj_surface.size(); i++) {
        glBegin(GL_TRIANGLES);

        Ternary<float> normal = objModel.obj_normal[objModel.obj_surface[i].normal.x - 1];
        glNormal3f(normal.x, normal.y, normal.z);
        Ternary<float> texture = objModel.obj_texture[objModel.obj_surface[i].texture.x - 1];
        glTexCoord3f(texture.x, texture.y, texture.z);
```

```
        Ternary<float> vertex = objModel.obj_vertexe[objModel.obj_surface[i].vertex.x - 1];
        glVertex3f(vertex.x, vertex.y, vertex.z);

        normal = objModel.obj_normal[objModel.obj_surface[i].normal.y - 1];

        glNormal3f(normal.x, normal.y, normal.z);
        texture = objModel.obj_texture[objModel.obj_surface[i].texture.y - 1];
        glTexCoord3f(texture.x, texture.y, texture.z);
        vertex = objModel.obj_vertexe[objModel.obj_surface[i].vertex.y - 1];

        glVertex3f(vertex.x, vertex.y, vertex.z);

        normal = objModel.obj_normal[objModel.obj_surface[i].normal.z - 1];

        glNormal3f(normal.x, normal.y, normal.z);
        texture = objModel.obj_texture[objModel.obj_surface[i].texture.z - 1];
        glTexCoord3f(texture.x, texture.y, texture.z);
        vertex = objModel.obj_vertexe[objModel.obj_surface[i].vertex.z - 1];

        glVertex3f(vertex.x, vertex.y, vertex.z);

        glEnd();
    }
    glFlush();
}
```

适应窗口改变函数

因为在本次试验中我们可能会改变观察视角和窗口大小，所以重绘函数 void reshape() 是很重要的，本函数主要完成的事情包括判断窗口大小是否在规定范围内、在视角改变时重新计算相机坐标和视角开度，话不多说，直接上代码：

```
void reshape(int w, int h)
{
    float size = 80;
    glViewport(0, 0, (GLsizei)w, (GLsizei)h);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
```

```
        glOrtho(-size, size, -
size * (GLfloat)h / (GLfloat)w, size*(GLfloat)h / (GLfloat)w, -
size, size);
    else
        glOrtho(-
size * (GLfloat)w / (GLfloat)h, size*(GLfloat)w / (GLfloat)h, -size, size, -
size, size);

    //设置模型参数--几何体参数
    gluLookAt(camera.x, camera.y, camera.z, 0, 20, 0, 0, 1, 0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
```

实现模型颜色的改变

这里我们使用鼠标右键点击控制模型颜色改变，主要做法是右击是用随机数产生三个 0~1 数字，分别替代原画笔颜色重的 RGB 值，本部分用到的代码如下：

```
default_random_engine e;
uniform_real_distribution<double> u(0, 1);
if (button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
{
    glColor3f(u(e), u(e), u(e));
    Draw();
}
```

在改变画笔颜色之后，再重绘模型时就会导致模型颜色的改变。

实现控制模型移动（旋转）

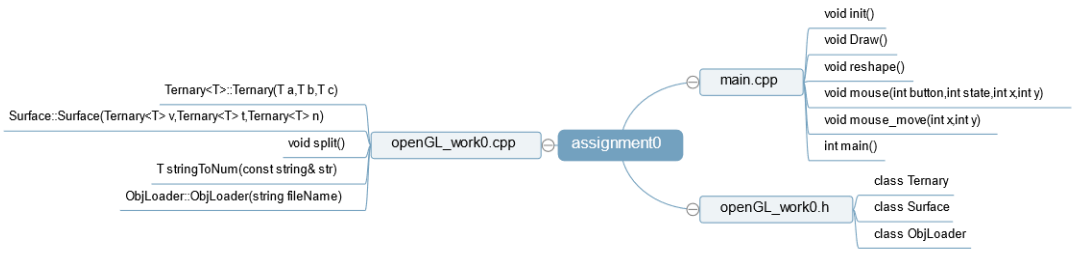
在本实验中，我们控制模型旋转（移动）的本质实际上是对相机坐标和角度的变化来实现的，在这里我们使用一组坐标 past_x 和 past_y 来记录鼠标左键按下时的坐标，然后在鼠标移动事件检测函数 void mouse_move(x,y)中根据当前鼠标坐标 x,y 和之前坐标 past_x, past_y 的差来判断模型应该旋转的角度。Void mouse_move()函数的实现如下：

```
void mouse_move(int x, int y)
{
    float speed = 0.3;
    if (get_pos == 1)
    {
        glRotatef(speed*(x - past_x), 0, 1, 0);
        glRotatef(speed*(y - past_y), 1, 0, 0);
```

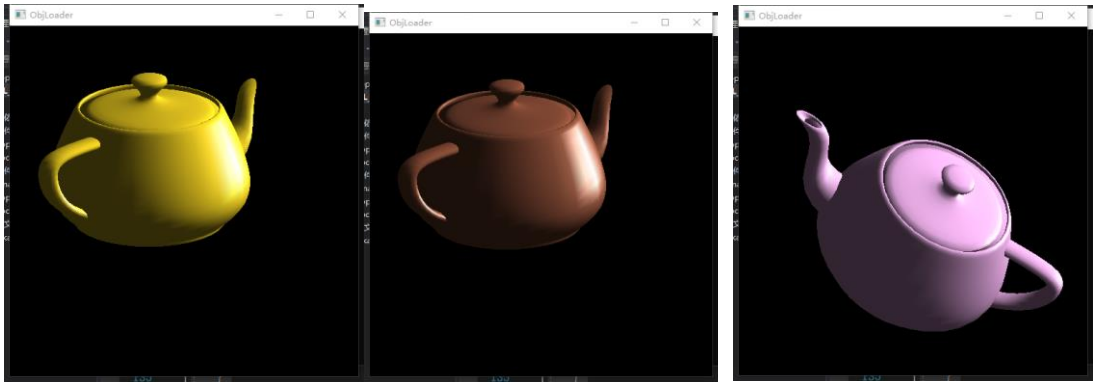
```
8.          Draw();
9.          past_x = x;
10.         past_y = y;
11.     }
12. }
```

至此，我们所有得小目标都已实现，将所有的代码组合起来我们就能得到可以运行的代码。

实验代码结构



实验结果和收获



上面三张图片从左至右依次展示了模型显示、颜色变化、模型变换的功能。

通过本次实验，我对上课屠老师讲解的图形学基本知识有了初步的了解，了解了模型文件 obj 的存储方式，复习了鼠标事件的使用方法，为接下来的几个大实验起到了抛砖引玉的作用。

实验源代码

https://github.com/yuemos/Computer-Graphics.git