计算机图形学 HW5 报告

实现一个具有简单光照效果的Teapot模型。

1 开发环境

本作业的开发环境是MinGW，完成作业后使用vs 2017编译并提交release版。 基于GLFW第三方库进行窗口管理。

2 程序实现说明

2.1 概述

由于本程序在OPENGL核心模式下进行编程，因此实现的方法有一些复杂，下面解释一下程序的细节。

整个程序大致分为三部分说明。

1. obj模型文件的读取。
2. 光照模型的实现。
3. 用户操作的实现

2.2 obj模型文件的读取

这一次我在网络上找到了茶壶模型的obj文件，考虑到读取文件的通用型起见，我使用了assimp库来读取obj模型文件，在obj文件中还有每一个定点对应的法向量，以方便计算光照强度。

读取obj模型文件的过程可以概述如下。

1. 使用assimp库读取obj文件，得到 `aiScene` 类型的对象，该对象含有obj模型文件中描述的所有信息。
2. 对该对象中的信息进行处理
    1. 获取顶点信息：主要处理每个节点的 `mVertices` 成员以及 `mNormals` 成员，获得顶点坐标以及对应的法向量
    2. 获取面信息：在每个节点的 `mFaces` 成员中，可以获得顶点索引信息
3. 使用上面的信息，生成一个 `Mesh` 对象，即可使用该对象进行绘制3D模型。

2.2.1 Mesh类的实现

该类提供如下接口。该类最主要的职能是存放一个网格的所有数据（包括顶点数据和顶点索引），并且在主渲染循环中调用 `draw` 函数进行绘制。

```
class Mesh {
    public:
        /*  网格数据  */
        vector<Vertex> vertices;
        vector<unsigned int> indices;l
        Mesh(vector<Vertex> vertices, vector<unsigned int> indices);
        void Draw(Shader shader);
    private:
        /*  渲染数据  */
        unsigned int VAO, VBO, EBO;
        /*  函数  */
        void setupMesh();
};
```

其中Vertex类型在下面加以说明，每个顶点具有对应的坐标和法向量。

```
struct Vertex {
    glm::vec3 Position;
    glm::vec3 Normal;
}
```

2.2.2 读取模型，创建Mesh类对象

从上面的 `Mesh` 类的接口说明即可知道每一个 `Mesh` 对象都能够绘制模型，难点在于我们如何读取模型文件并将其相应的信息转为一个 `Mesh` 类对象。在这里的实现中，我实现了一个 `Model` 类，该类中维护这一个 `Mesh` 类的数组，并且会将模型文件中读取到的数据转为一个个 `Mesh` （可能只有一个）,并放入到该类的 `Mesh数组` 中。此后调用 `Model` 的 `draw` 函数，就可以通过遍历该数组，调用每一个 `Mesh` 对象的 `draw` 方法既可实现一个场景中的多对象的绘制。

`Model` 类提供以下接口，对外的接口为从文件加载模型以及模型的绘制两个功能。为了实现这两个功能，我还是实现了 `loadModel` ， `processNode` ，`processMesh` 。读取文件的过程大体可以分成以下几个步骤。

1. 使用 `Assimp::Importer` 读取文件，得到 `aiScene` 对象
2. 遍历 `aiScene` 中的每一个节点，将每个节点中的数据转成 `Mesh` 对象，并放入 `Model` 中的 `meshes` 数组中。
    1. 这一个转换的过程，由 `processMesh` 函数完成。

```
class Model
{
    public:
        /*  函数   */
        Model(string path)
        {
            loadModel(path);
        }
        void Draw(Shader shader)
        {
```

```
            for(unsigned int i = 0; i < meshes.size(); i++)
                meshes[i].Draw(shader);
        }
    private:
        /*  模型数据  */
        vector<Mesh> meshes;
        string directory;
        /*  函数   */
        void loadModel(string path);
        void processNode(aiNode *node, const aiScene *scene);
        Mesh processMesh(aiMesh *mesh, const aiScene *scene);
};
```

其中，我对 `processMesh` 函数进行详细说明。

该函数实现的是从一个节点中获取一个 `Mesh` 类的数据，并且据此创建一个 `Mesh` 类对象放入该 `Model` 类中的 `meshes` 数组中。

大体的实现流程如下。

1. 通过 `mesh->mNumVertices` 获取顶点数量，并且遍历该数量，分别从 `mVertices` 和 `mNormal` 中获得一个顶点的顶点坐标和对应法向量，并创建顶点数据数组 `vertices` 。
2. 通过 `mesh->mNumFaces` 获取面片数量，并遍历每一个面片，从 `ace.mIndices[j]` 中得到顶点索引数组 `indices` 。
3. 使用 `vertices` 和 `indices` 两个数组创建 `Mesh` 对象并返回。

```
Mesh processMesh(aiMesh *mesh, const aiScene *scene){
    // data to fill
    vector<Vertex> vertices;
    vector<unsigned int> indices;

    // Walk through each of the mesh's vertices
    for(unsigned int i = 0; i < mesh->mNumVertices; i++)
    {
        Vertex vertex;
        glm::vec3 vector;
        // positions
        vector.x = mesh->mVertices[i].x;
        vector.y = mesh->mVertices[i].y;
        vector.z = mesh->mVertices[i].z;
        vertex.Position = vector;
        // cout << vector.x << vector.y << vector.z << endl;
        // cout << mesh->mNumVertices << endl;
        // normals
        vector.x = mesh->mNormals[i].x;
        vector.y = mesh->mNormals[i].y;
        vector.z = mesh->mNormals[i].z;
        // cout << vector.x << vector.y << vector.z << endl;
        // cout << mesh->mNumVertices << " | " << i << endl;
        vertex.Normal = vector;
        // cout << vector.x << vector.y << vector.z;
        vertices.push_back(vertex);
    }

    for(unsigned int i = 0; i < mesh->mNumFaces; i++)
    {
    aiFace face = mesh->mFaces[i];
    // retrieve all indices of the face and store them in the indices vector
```

```
        for(unsigned int j = 0; j < face.mNumIndices; j++){
        indices.push_back(face.mIndices[j]);
        // cout << face.mIndices[j] ;
        }
        // cout << endl;
        }
        // return a mesh object created from the extracted mesh data
        return Mesh(vertices, indices);
    }
```

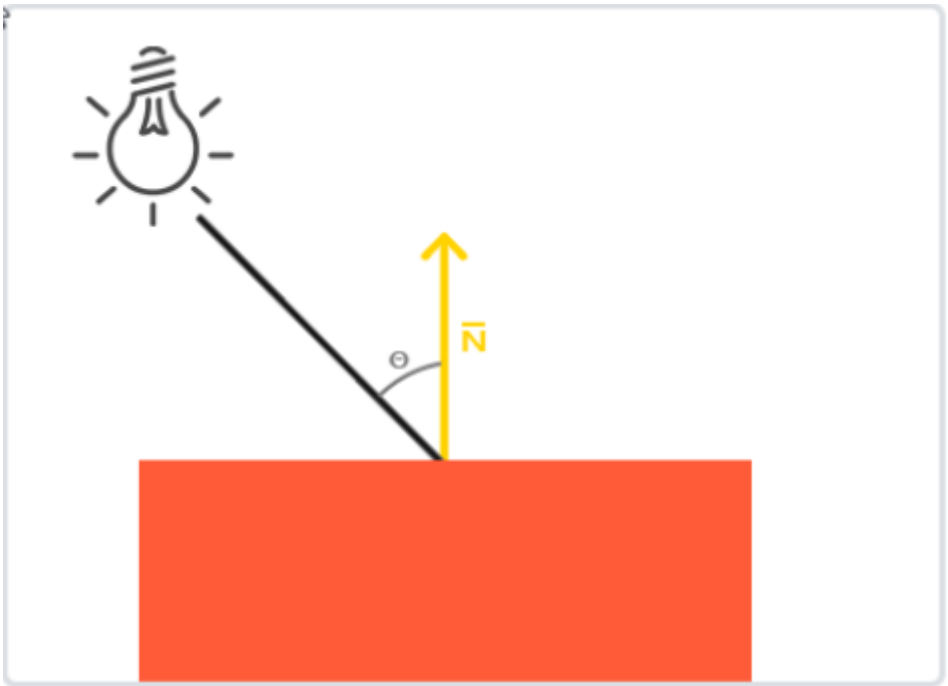
2.3 光照模型的实现

本作业实现了漫反射的光照模型。

在漫反射的模型中，以下几个量与光照的形成密切相关。

关键量 变量名	作用说明
objectCol	模型本身颜色
lightCol	光源颜色
FragPos	模型上的点的颜色
lightPos	光源位置
normal	模型上的每个点对应的法向量

2.3.1 漫反射实现原理



如上图所示，在计算漫反射的光照，我们关键在于计算光线与物体表面法向量的夹角（或者说算出夹角的余弦值）。

若夹角越小，则夹角余弦值越大，反射光越强，若夹角越小，则夹角余弦值越小，反射光越弱。

若认为计算得到的向量都是单位向量， 余弦值的计算可以使用下面的计算方法得到。

$cos(\theta) = (lightPos - FragPos) * normal$

2.3.2 着色器的编写

顶点着色器

该着色器具有以下两个职能。

1. 计算模型的世界坐标。
2. 将模型的世界坐标，法向量传递给片段着色器。

代码如下

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;


uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

out vec3 normal;
out vec3 FragPos;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    FragPos = vec3(model * vec4(aPos, 1.0));
    normal = aNormal;
}
```

片段着色器

该着色器具有以下的功能：计算某一点应有的颜色。

在代码中，前面经过了单位化的处理后，余弦值的计算由 `dot(norm, lightDir)` 得到，赋值给diff变量。

然后在后面 `result = diffuse * objectCol` 完成光源颜色与物体颜色的叠加，在中间有一些常数控制物体的反光程度。

最后返回 `vec4(result, 1.0)` 作为该点的真正颜色。

代码如下:

```
#version 330 core
out vec4 FragColor;

in vec3 normal;
in vec3 FragPos;

uniform vec3 lightPos;
uniform vec3 lightCol;
uniform vec3 objectCol;
```

```
void main()
{
    float add = 0.7;
    vec3 e = normal;
    vec3 norm = normalize(e);
    vec3 lightDir = normalize(lightPos - FragPos);
    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = (diff + add) * lightCol;
    vec3 result = diffuse * objectCol;
    FragColor = vec4(result,1.0);
}
```

2.4 用户操作的实现

本作业支持两种交互模式：操作模式和探索模式。

这两个模式的实现复用了在HW4-2-meshlab中的代码，因此与作业HW4-2-meshlab的实现相同，不做重复讲解。

具体操作可见第三部分使用说明，也可见demo视频。

3 程序运行方法

3.1 编译方法

在代码目录中，可使用cmake工具进行编译。

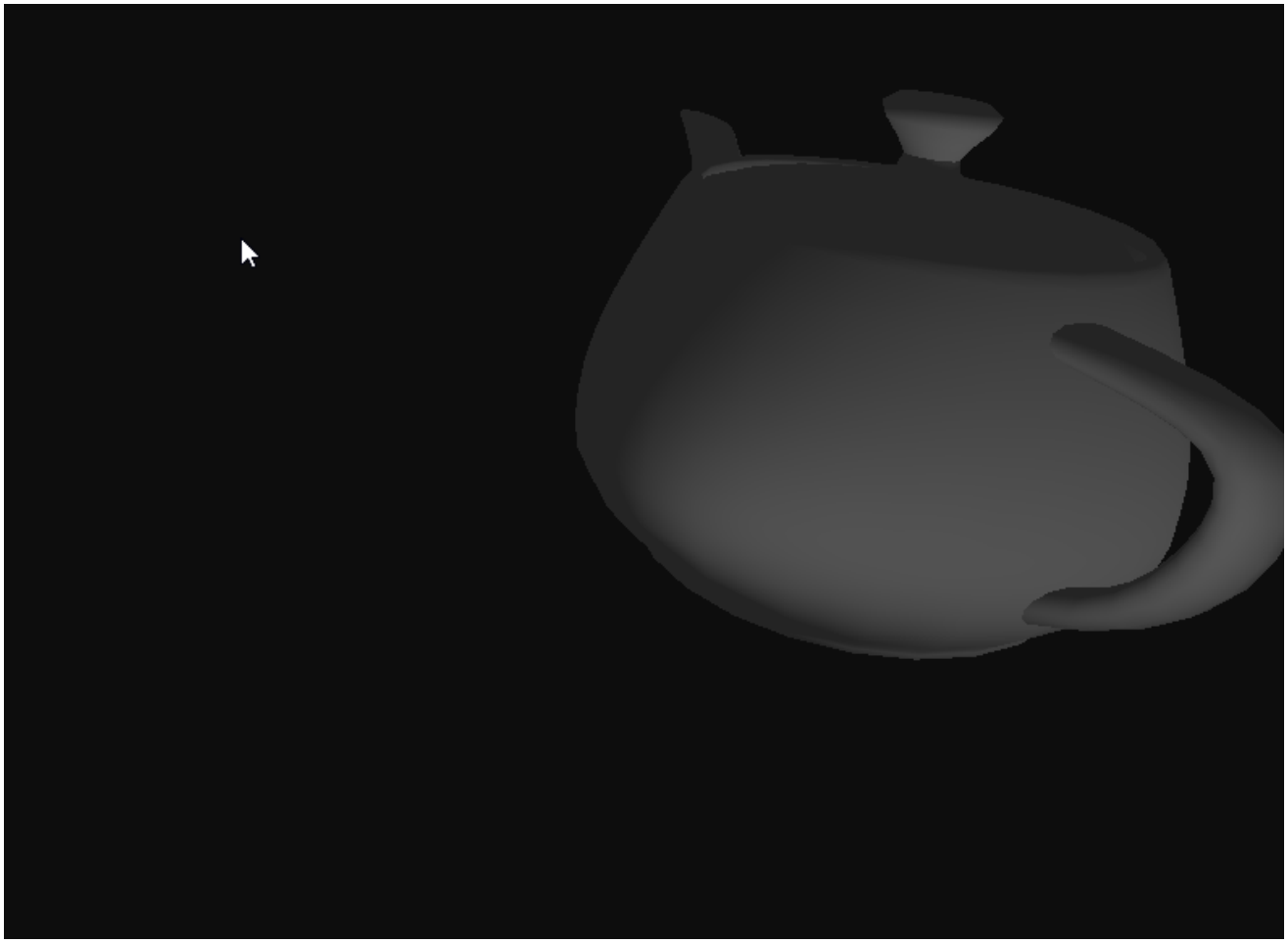
依赖的第三方库有

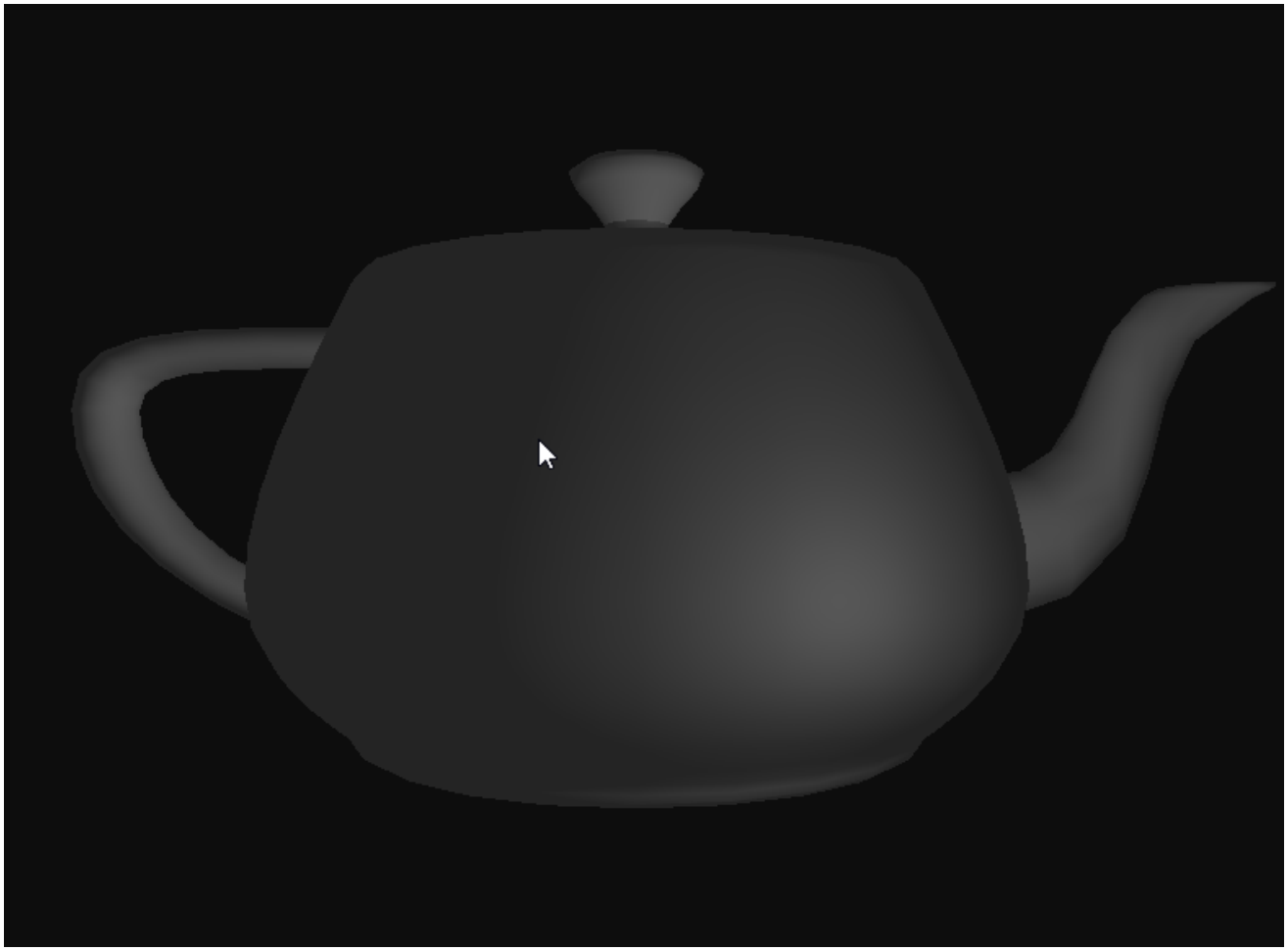
库名称	库作用
assimp	读取模型文件

3.2 运行方法

1. 通用按键
    1. 按下t键，可进行模式的切换，具体切换到的模式可以在控制台输出查看
        1. 处理了按键抖动的问题。
    2. 按下c键，可清空之前的所有操作，回到最初的状态
2. 操作模式（operating mode）
    1. 可以使用小键盘的方向键，对物体进行上下平移。
    2. 键盘的 `j` ， `k` 键，可以让物体前后平移
    3. 可以使用鼠标拖动，实现物体的旋转
3. 探索模式（exploring mode）
    1. 鼠标移动直接带来视角的改变
    2. 键盘wasd四个键可以改变摄像机的水平位置。
    3. 键盘ui两键可以改变摄像机的高度。u键上升，i键下降。

4 程序运行截图





5 程序运行录屏

录屏展示键盘鼠标操作对模型的影响，以及光源的影响。

可见作业目录下的 `teapot.mp4`