图形学实验报告

陈炜 PB16050567

实验一：中点画圆算法

中点画圆算法使用圆函数作为决策函数，来判断(xk,yk)和(xk,yk-1)的中点在圆内还是在圆外，决策函数的值p<0则中点在圆内，绘制(xk,yk)，下一个点是(xk+1,yk)；否则，绘制(xk,yk-1)下一个点是(xk,yk-1)。

决策函数是一个增量函数，在有初值以后，后续的每一个值都可以容易地计算出。

我们只计算x>0且x>y的部分的决策函数的值，其他值可以通过对称性绘制。

中点算法步骤：

1. 输入圆半径 r 和圆心 (x_c, y_c)，并得到圆周（圆心在原点）上的第一个点，

$$(x_0, y_0) = (0, r)$$

2. 计算决策参数的初始值，

$$p_0 = \frac{5}{4} - r$$

3. 在每个 x_k 位置，从 $k=0$ 开始，完成下列测试：假如 $p_k < 0$，圆心在 $(0, 0)$ 的圆的下一点为 (x_{k+1}, y_k)，并且

$$p_{k+1} = p_k + 2x_{k+1} + 1$$

否则，圆的下一个点是 $(x_k + 1, y_k - 1)$，并且

$$p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}$$

其中 $2x_{k+1} = 2x_k + 2$ 且 $2y_{k+1} = 2y_k - 2$。

4. 确定在其他七个八分圆中的对称点。

5. 将每个计算出的像素位置 (x, y) 移动到圆心在 (x_c, y_c) 的圆路径上，并画坐标值：

$$x = x + x_c, \qquad y = y + y_c$$

6. 重复步骤 3～5，直至 $x \geq y$。

由于我们的半径是整数，所以决策函数的初始值取1-r就可以了。

决策函数计算：

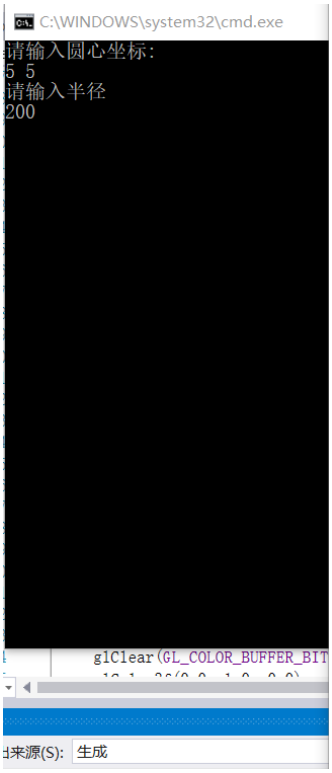
```
x = 0;
    y = r;
```

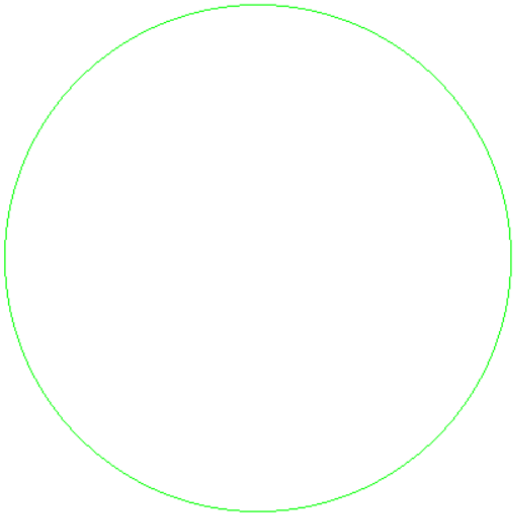
```
    int p;
    p = 1 - r;
    Paint(x, y);
    while (x < y)
    {
        x++;
        if (p >= 0)
        {
            y--;
            p += 2 * x - 2 * y + 1;
        }
        else
        {
            p += 2 * x + 1;
        }
        Paint(x, y);
    }
```

图像绘制：

```
void Paint(int x,int y)
{
    glBegin(GL_POINTS);
    glVertex2d(x, y);
    glVertex2d(-x, y);
    glVertex2d(x, -y);
    glVertex2d(-x, -y);
    glVertex2d(y, x);
    glVertex2d(-y, x);
    glVertex2d(y, -x);
    glVertex2d(-y, -x);
    glEnd();
    glFlush();
}
```

实验结果截图：





实验二：对Sierpinski的三维变换

1、平移：

三维平移矩阵为：

$$\begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

在 OpenGL 中，只需要在绘图之前使用 gltranslatef()函数，将类似的旋转矩阵压到 OpenGL 当前的矩阵堆栈中即相当于乘上了该矩阵。

2、缩放：

三维缩放矩阵为：

$$\begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

与旋转类似的，在 OpenGL 中，只需要在绘图之前使用 glscale()函数，将类似的矩阵压到 OpenGL 当前的矩阵堆栈中，也就相当于乘上了这个矩阵。

3、旋转：

与二维旋转类似，三维旋转可以分为以下几个步骤（假定用户提供的是旋转轴所在直线上的的两个点的坐标）：

①将图形在 xy 平面上平移，直到旋转轴穿过坐标原点，矩阵：

$$= \begin{pmatrix} 1 & 0 & 0 & x_1-x_2 \\ 0 & 1 & 0 & y_1-y_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

②将直线旋转到 z 轴，此过程可分为两步：

（1）绕 x 轴旋转，

$$R(\alpha)=\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & c/d & -b/d & 0 \\ 0 & b/d & c/d & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

（2）绕 y 轴旋转：

$$R(\beta)=\begin{pmatrix} d & 0 & -a & 0 \\ 0 & 1 & 0 & 0 \\ a & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

③将图形绕 z 轴旋转

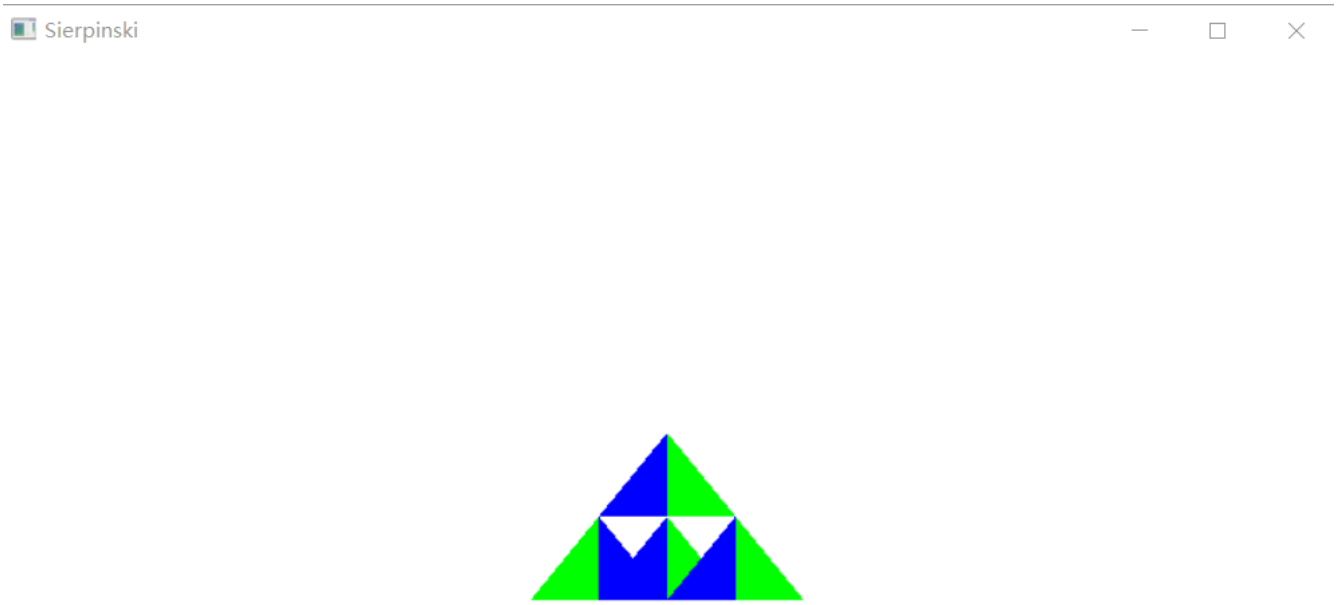
④做②的逆运算

⑤做①的逆运算

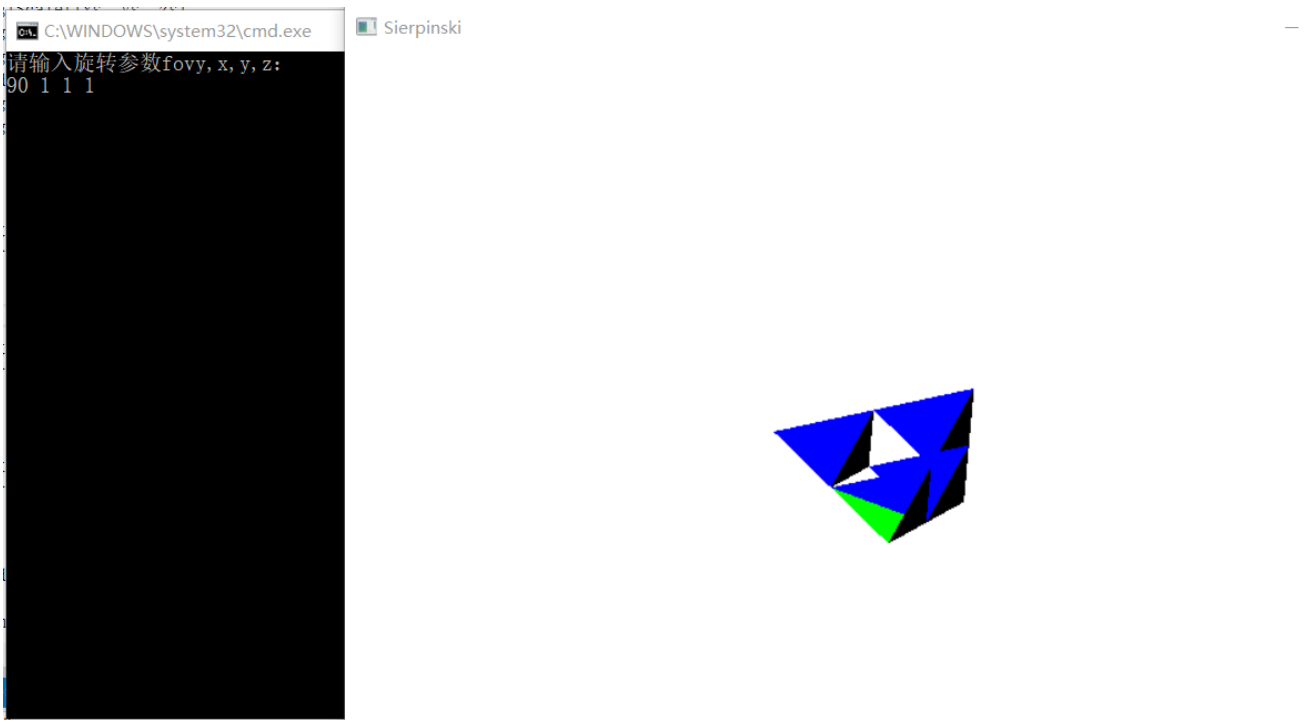
在 OpenGL 中，提供了绕任意向量旋转的函数 glrotate()，可以完成绕经过原点的任意向量的任意角度的旋转，也就是生成②到④运算的复合矩阵并压入矩阵栈中。 我们在实际应用中只需要把旋转轴移动到原点，然后调用 glrotate()函数，然后做移动的逆运算即可。

实验结果：

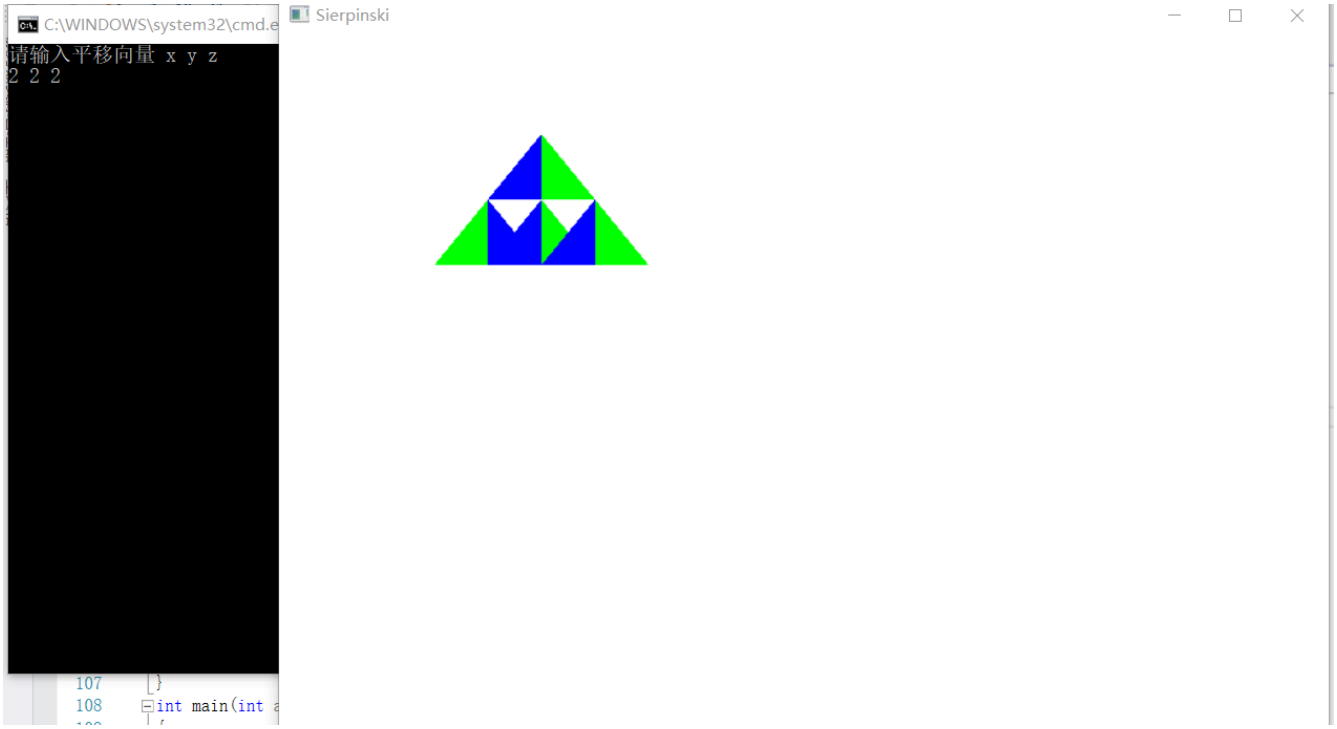
原图：


Sierpinski

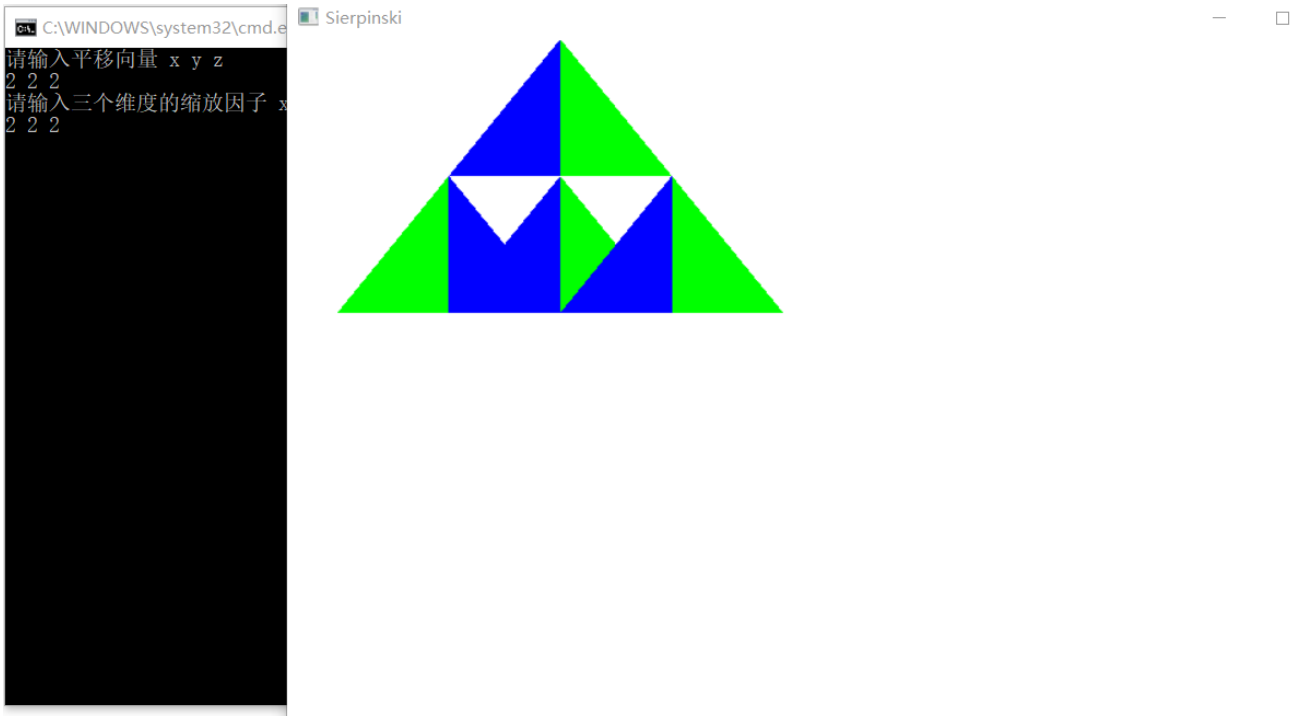
旋转


C:\WINDOWS\system32\cmd.exe
请输入旋转参数fovy,x,y,z:
90 1 1 1
Sierpinski

平移：



缩放：



实验三：

1、实验文档分析：

在luweiqi.txt文件中存储了各种索引表和绘制需要的信息。

文件中前面的所有项目，贴图，顶点，法线等等，都是索引表，真正指挥绘制过程的只有model部分文件。

model：

model共有4个，将整个物体分成了四个部分，每一个部分有不同的材质，对应不同的贴图，拥有不同的三角形。三角形参数分为三个部分，法向量索引，贴图坐标索引（二维）和三个顶点坐标的索引。在绘制的时候，根据每一个三角形的各个索引值绘制即可，主要要先指定法向量和贴图坐标再绘制定点，否则贴图会发生异常。

纹理文件：

纹理文件表指出了各个纹理文件的文件路径，可以被材质表调用。

材质表：

材质表指定了各种材质，和材质绑定的纹理贴图（索引），可以用来对应一个模型。

材质数据包括：

ambient：环境光反射。

diffuse：漫反射光。

specular：镜面光反射。

emission：颜色。

shininess：镜面指数

纹理文件索引

以下三个表直接被模型中的三角形参数引用：

顶点表：

每一行都是三位顶点的坐标。

贴图坐标表：

每一行都是一个二位贴图坐标。

法线表：

每一行都是一个三位向量值。

2、光源设置：

opengl的光照系统有八个光源，这里我们启用光源0。

和材质的三种类型的光相对应的，光照系统也由三种光组成，环境光，漫射光和镜面光，以及一个光源位置坐标。最后使用glEnable启用光源0和光照系统即可。

```
float AmbientColor[] = { 0.2f, 0.0f, 0.0f, 1.0f };
    glLightfv(GL_LIGHT0, GL_AMBIENT, AmbientColor);     //设置环境光颜色（三色光的成分）
```

```
    float DiffuseColor[] = { 0.2f, 0.0f, 0.0f, 1.0f };
    glLightfv(GL_LIGHT0, GL_DIFFUSE, DiffuseColor);     //设置漫射光颜色

    float SpecularColor[] = { 0.2f, 0.0f, 0.0f, 1.0f };
    glLightfv(GL_LIGHT0, GL_SPECULAR, SpecularColor);   //设置镜面光颜色

    float Position[] = { 1.0f, 1.0f, 0.0f, 1.0f };
    glLightfv(GL_LIGHT0, GL_POSITION, Position);        //光源位置

    glEnable(GL_LIGHT0);
    glEnable(GL_LIGHTING);      //开启光照系统
```

3、数据读取与存储：

使用如下数据结构存储从文件中读取数据：

```
int Ntexturefile;       //纹理文件数量
int Nmaterial;      //材质数据数量
int Npoints;        //顶点数量
int Nchartlet;      //贴图坐标数量
int Nnormal;        //法线数量
int Nmodel;         //模型数量

string* texturefile;            //纹理文件名字

struct Material             //材质
{
    GLfloat ambient[4];     //环境光
    GLfloat diffuse[4];     //漫反射
    GLfloat specular[4];        //镜面反射
    GLfloat emission[4];        //辐射
    GLfloat shiness;            //光斑
    GLint index;                //纹理文件索引
};
Material* materials;

struct Point            //顶点索引表
{
    float p[3];
};
Point* points;

struct Chatlet      //贴图
{
    float c[2];
};
Chatlet* chatlets;

struct Normal       //法线
{
```

```
    float n[3];
};
Normal* normals;

float scale[3];     //缩放系数

struct Triangle     //三角形
{
    int  t[9];
};
struct Model
{
    int Ntriangle;      //三角形数量
    Triangle* T;        //三角形数组
    int index;  //材质索引

};
Model* models;
int* texture;   //各个纹理图
```

4、模型绘制：

共有四个子模型，分别绘制即可。

①材质设置：

每一种模型对应一种材质，使用模型的材质索引在材质表中找到对应的材质数据设置即可。

```
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, materials[models[i].index].ambient);
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, materials[models[i].index].diffuse);
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, materials[models[i].index].specular);
glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, materials[models[i].index].emission);
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, materials[models[i].index].shiness);
```

②纹理加载：

使用了SOIL库来加载图像：

```
index = materials[models[i].index].index;
texture[i] = SOIL_load_OGL_texture(texturefile[index].c_str(), SOIL_LOAD_AUTO,SOIL_CREATE_NEW_ID,
SOIL_FLAG_INVERT_Y);
if (!texture[i])
{
    printf("cannot load the image\n");
    exit(0);
}
```

打开的文件用texture[i]作ID。

打开纹理文件以后，需要将该TEXTURE_2D绑定到到该纹理文件，才能正常使用。

```
glBindTexture(GL_TEXTURE_2D, texture[i]);   //texture2D绑定到textre[i]，即是选择了这个纹理
```

绑定纹理文件以后，我们还需要指定从纹理坐标映射到帧缓存的方式，这里涉及到两个问题：

1、纹理坐标的中心并不一定对准纹理元素的中心，我们选择在这个坐标的附近做线性的加权平均算出输出的颜色值（线性滤波）。

2、从纹理元素到像素的转换：

下面代码中的MIN和MAG分别对应从纹理元素到像素转换时，一个纹理元素对应多个像素和一个纹理元素对应少于一个像素的情况，分别要对纹理元素进行放大和缩小。不管是要放大的纹理元素还是要缩小的纹理元素，我们都使用线性滤波的方法进行处理。

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR); // Filter参数，处理映射到坐标非对齐的情况，Linear，使用附近的纹素加权
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR); // 如果是near的话，会选择最近的坐标直接使用
```

③三角形绘制

在这个实验中，我们使用三角形来构建模型，模型的每一个三角形都有三个顶点，三个法线和三个纹理坐标，都通过索引的方式给出，我们只需要通过索引来找到对应的值即可：

```
glBegin(GL_TRIANGLES);
        for (size_t j = 0; j < models[i].Ntriangle; j++)

        {
            //点1
            glNormal3fv(normals[models[i].T[j].t[2] - 1].n);
            glTexCoord2f(chatlets[models[i].T[j].t[1] - 1].c[0], 1 - chatlets[models[i].T[j].t[1] - 1].c[1]);       //纹理索引
            glVertex3fv(points[models[i].T[j].t[0] - 1].p);
                        //法线（默认索引-1），从0开始
            //点2
            glNormal3fv(normals[models[i].T[j].t[5] - 1].n);    //法线（默认索引-1），从0开始
            glTexCoord2f(chatlets[models[i].T[j].t[4] - 1].c[0], 1 - chatlets[models[i].T[j].t[4] - 1].c[1]);       //纹理索引
            glVertex3fv(points[models[i].T[j].t[3] - 1].p);
            //点3
            glNormal3fv(normals[models[i].T[j].t[8] - 1].n);//法线（默认索引-1），从0开始
            glTexCoord2f(chatlets[models[i].T[j].t[7] - 1].c[0], 1 - chatlets[models[i].T[j].t[7] - 1].c[1]);       //纹理索引
            glVertex3fv(points[models[i].T[j].t[6] - 1].p);

        }
        glEnd();
    }
```

要注意的是，通过SOIL库打开的纹理文件中，y坐标是反向的，要使用 1 - chatlets[models[i].T[j].t[7] - 1].c[1]

才能得到正确的值。

至此，完整的模型已经构建起来了，接下来我们需要进行三维观察和三维变换。

5、显示模型的点和线：

这个很简单，只需要上面的三角形绘制部分，分别改成GL_LINES和GL_POINTS就可以了。

线框图：

```
void displayline()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(xs, ys, zs, xref, yref, zref, Vx, Vy, Vz);
    glTranslatef(translatex, translatey, translatez);
    glScalef(scalex, scaley, scalez);
    glRotatef(rotateA, rotatex, rotatey, rotatez);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity(); if (!pers)    //正交投影
        glOrtho(xwMin, xwMax, ywMin, ywMax, dnear, dfar);
    else
        gluPerspective(angle, aspect, Pnear, Pfar);
    for (int i = 0; i < Nmodel; i++)        //分模型绘制，每一个模型有不同的材质
    {                                                                   //绘制该模型中每一个三角形
        glBegin(GL_LINES);
        for (size_t j = 0; j < models[i].Ntriangle; j++)        //必须是先法向，再纹理，最后三角，否
则画出来很奇怪

        {
            //点1
            glVertex3fv(points[models[i].T[j].t[0] - 1].p);

            //点2
            glVertex3fv(points[models[i].T[j].t[3] - 1].p);

            //点3
            glVertex3fv(points[models[i].T[j].t[6] - 1].p);
        }
        glEnd();
    }
    glFlush();
}
```

点图

```
for (int i = 0; i < Nmodel; i++)        //分模型绘制，每一个模型有不同的材质
    {                                                                   //绘制该模型中每一个三角形
        glBegin(GL_POINTS);
        for (size_t j = 0; j < models[i].Ntriangle; j++)        //必须是先法向，再纹理，最后三角，否
则画出来很奇怪

        {
```

```
            //点1
            glVertex3fv(points[models[i].T[j].t[0] - 1].p);

            //点2
            glVertex3fv(points[models[i].T[j].t[3] - 1].p);

            //点3
            glVertex3fv(points[models[i].T[j].t[6] - 1].p);
        }
        glEnd();
    }
```

6、三维变换：

三维变换代码如下：

```
glTranslatef(translatex, translatey, translatez);
glScalef(scalex, scaley, scalez);
glRotatef(rotateA, rotatex, rotatey, rotatez);
```

三个参数默认都是不平移，不缩放，不选择的。

用户通过键盘事件T,S,R来设定参数：

```
if (value == 'T')   //translate
    {
        printf("目前的translate参数为:x:%f,y:%f,z:%f\n", translatex, translatey, translatez);
        printf("请在命令行输入新的translate参数(x,y,z)\n");
        scanf("%f %f %f", &translatex, &translatey, &translatez);
    }
    if (value == 'S')   //scale
    {
        printf("目前的scale参数为:x:%f,y:%f,z:%f\n", scalex, scaley, scalez);
        printf("请在命令行输入新的scale参数(x,y,z)\n");
        scanf("%f %f %f", &scalex, &scaley, &scalez);
    }
    if (value == 'R')
    {
        printf("目前的rotate参数为:angle:%f, x:%f, y:%f, z:%f\n", rotateA, rotatex, rotatey,
rotatez);
        printf("请在命令行输入新的rotate参数(Angle,x,y,z)\n");
        scanf("%f %f %f %f", &rotateA, &rotatex, &rotatey, &rotatez);
    }
```

7、三维观察：

初始的三维观察参数如下：

```
pers = false;
    //观察参数
    xs = 0;
    ys = 0;
    zs = 0;
    xref = 1;
    yref = 1;
    zref = 0;
    Vx = 0;
    Vy = 0;
    Vz = 1;
    //正交投影参数
    xwMin = -2;
    xwMax = 2;
    ywMin = -2;
    ywMax = 2;
    dnear = 0;
    dfar = 12;

    //透视投影参数
    angle = 120;
    aspect = 1;
    Pnear = 0.1;
    Pfar = 5;
```

默认从(1,1,0)的地方向(0,0,0)的地方投去视线，默认使用正交投影。

用户可以通过鼠标和键盘时间来更改投影参数的值：

投影方式的变换：

使用鼠标左键一键切换透视投影和正交投影：

```
void mouse(int btn, int state, int x, int y) {
    if (btn == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
        pers = !pers;

}
```

投影参数变换：

可以通过键盘事件来方便地小幅度修改观察参数，也可以自行输入：

```
if (value == 'w')
    {
        yref -= 0.1;
    }
    if (value == 's')
    {
        yref += 0.1;
    }
    if (value == 'a')
    {
```

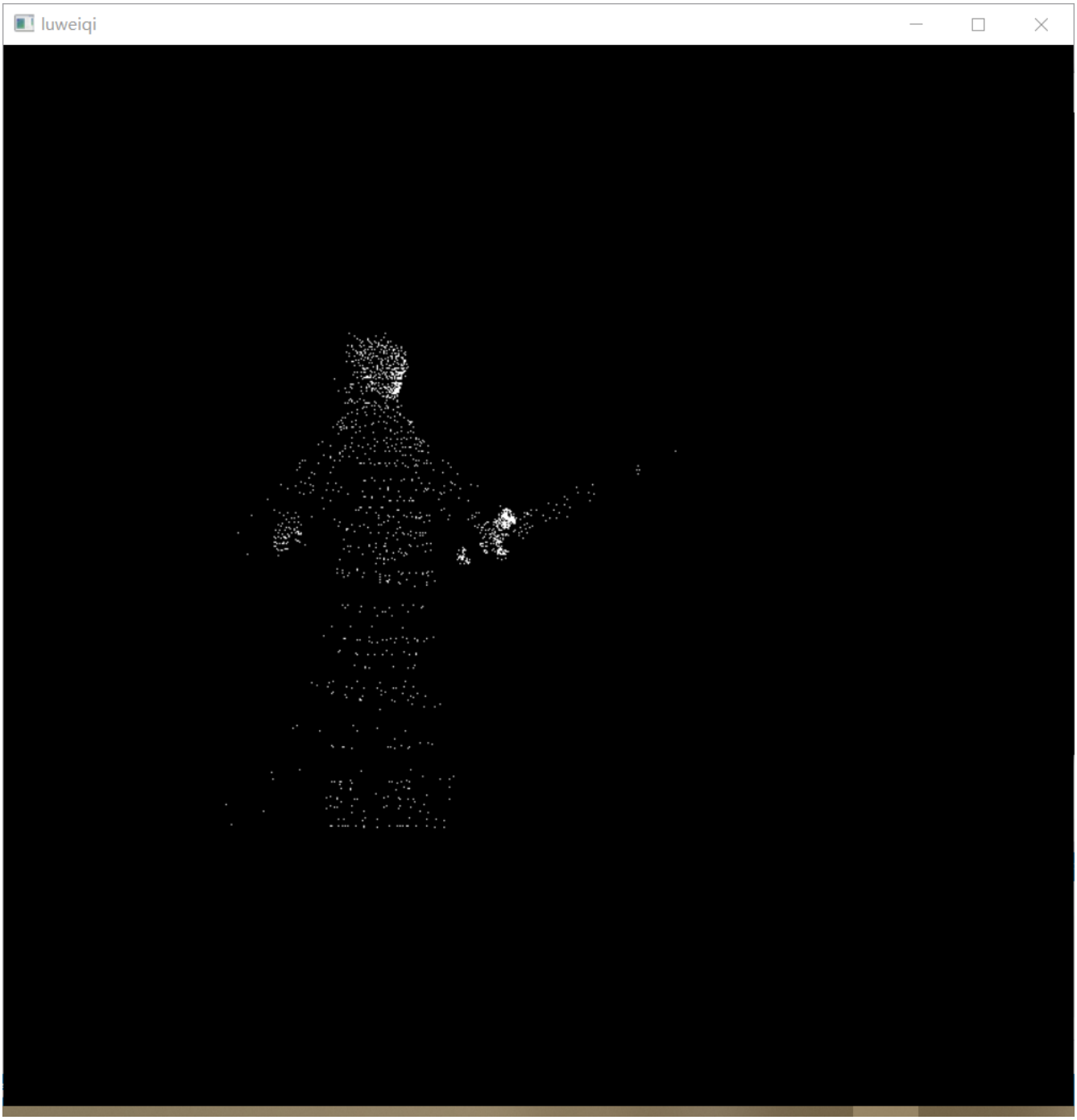
```
        xref -= 0.1;
    }
    if (value == 'd')
    {
        xref += 0.1;
    }if (value == 'q')
    {
        zref -= 0.1;
    }
    if (value == 'e')
    {
        zref += 0.1;
    }

    if (value == '8')
    {
        angle -= 5;
    }
    if (value == '2')
    {
        angle += 5;
    }
    if (value == 'l')
    {
        printf("当前LookAt参数是：xs:%f,ys:%f,zs:%f,xref:%f,yref:%f,zref:%f,Vx:%f,Vy:%f,Vz:%f\n",
xs, ys, zs, xref, yref, zref, Vx, Vy, Vz);
        printf("请输入LookAt参数，xs ys zs xref  yref  zref, Vx  Vy  Vz\n");
        scanf("%f %f %f%f%f%f%f%f%f", &xs, &ys, &zs, &xref, &yref, &zref, &Vx, &Vy, &Vz);
    }
    if (value == 'p')
    {
        printf("当前perspective参数是：fovy:%f,aspect:%f,near:%f,far:%f\n", angle, aspect, Pnear,
Pfar);
        printf("请输入Perspective参数，fovy aspect near far \n");
        scanf("%f %f %f %f", &angle, &aspect, &Pnear, &Pfar);
    }
    if (value == 'o')
    {
        printf("当前glOthor参数是：xwMin:%f,xwMax:%f,ywMin:%f,ywMax:%f,near:%f,far:%f\n", xwMin,
xwMax, ywMin, ywMax, dnear, dfar);
        printf("请输入glOthor正交投影参数，xwMin xwMax ywMin ywMax dnear dfar\n");
        scanf("%f %f %f%f%f%f%f%f%f", &xwMin, &xwMax, &ywMin, &ywMax, &dnear, &dfar);
    }
```

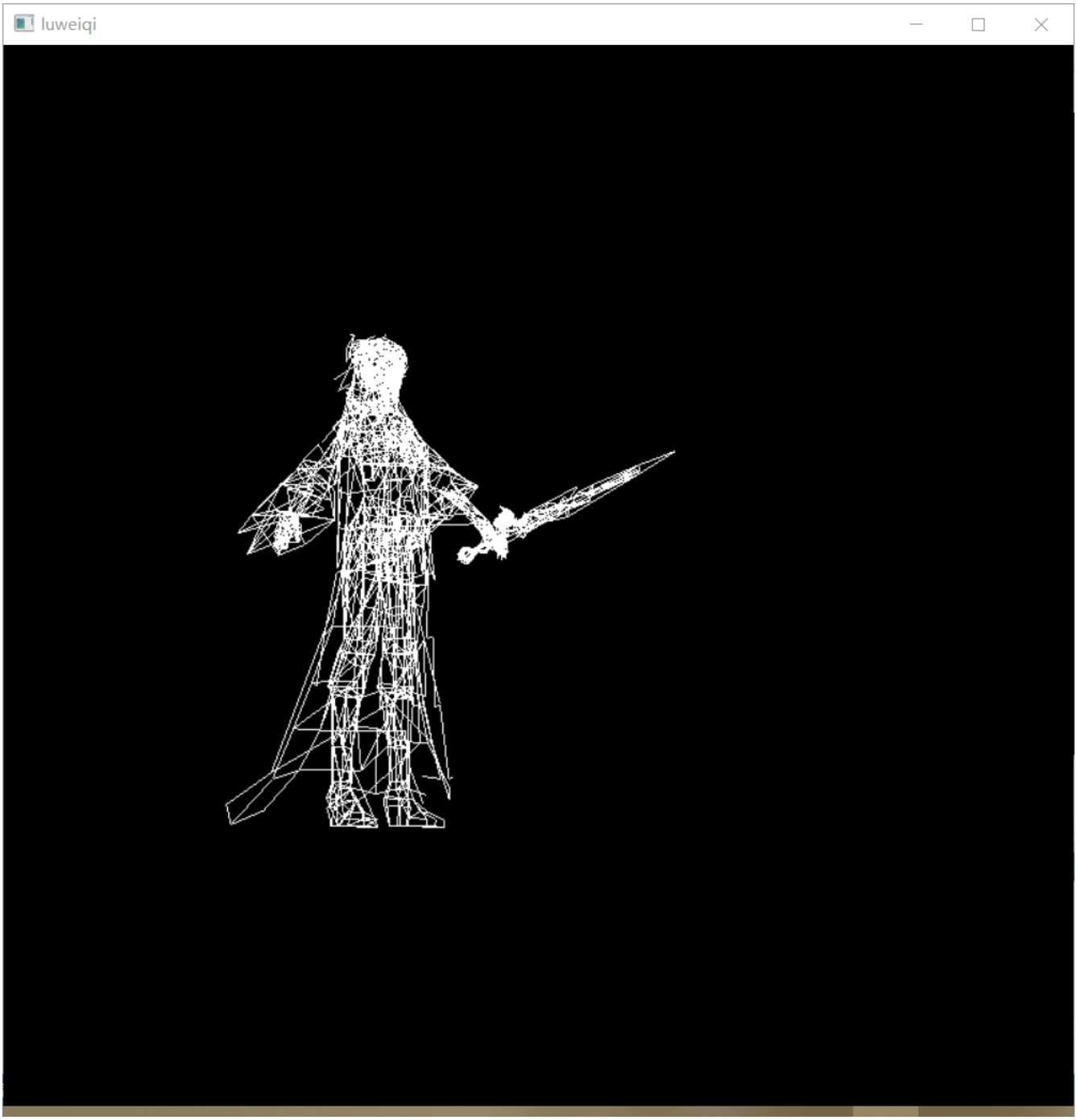
综上，实验三的目标已经全部实现了，我们对luweiqi进行了建模，贴图和三位观察。

最终结果：

点图：

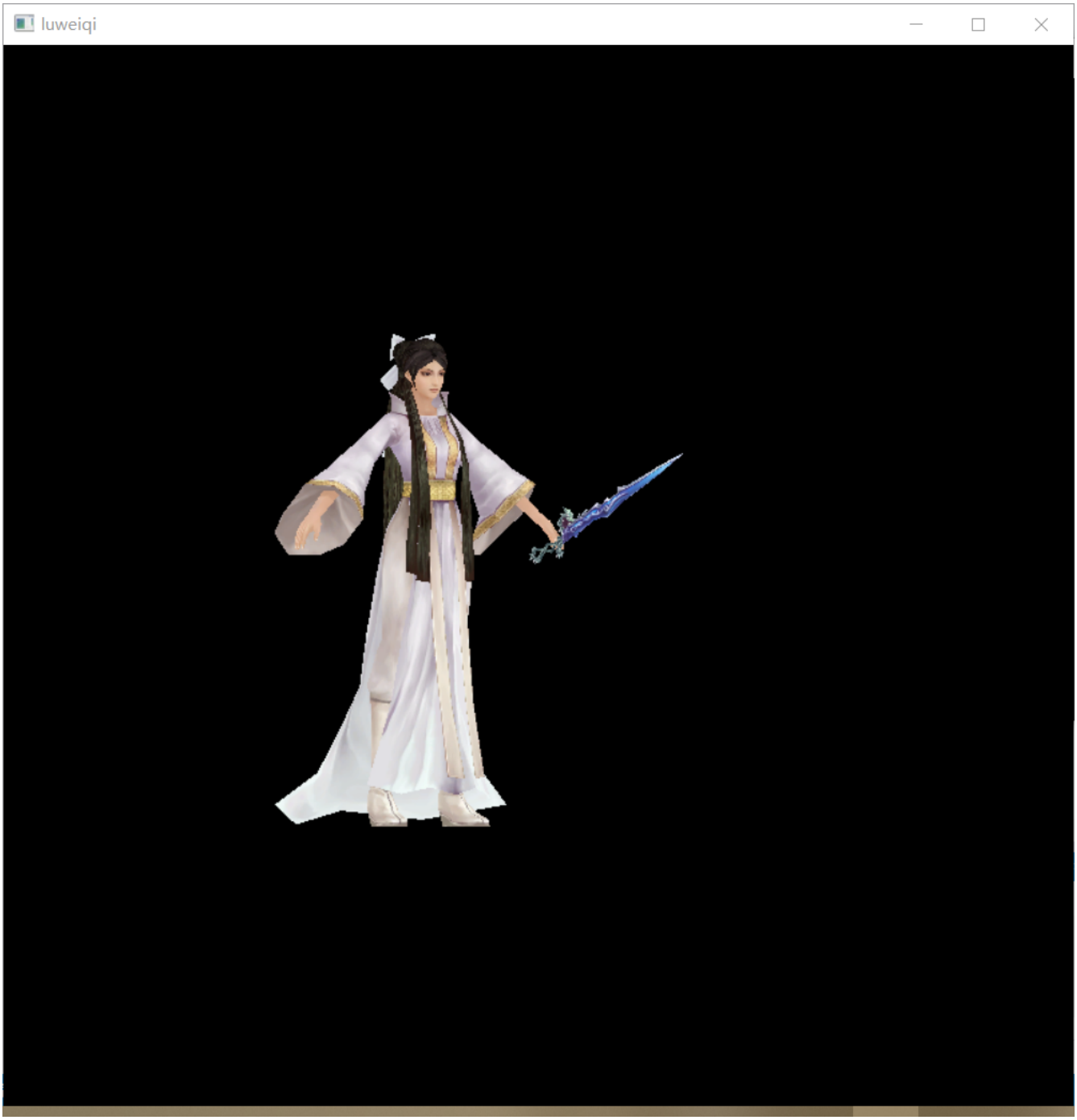


线框图：

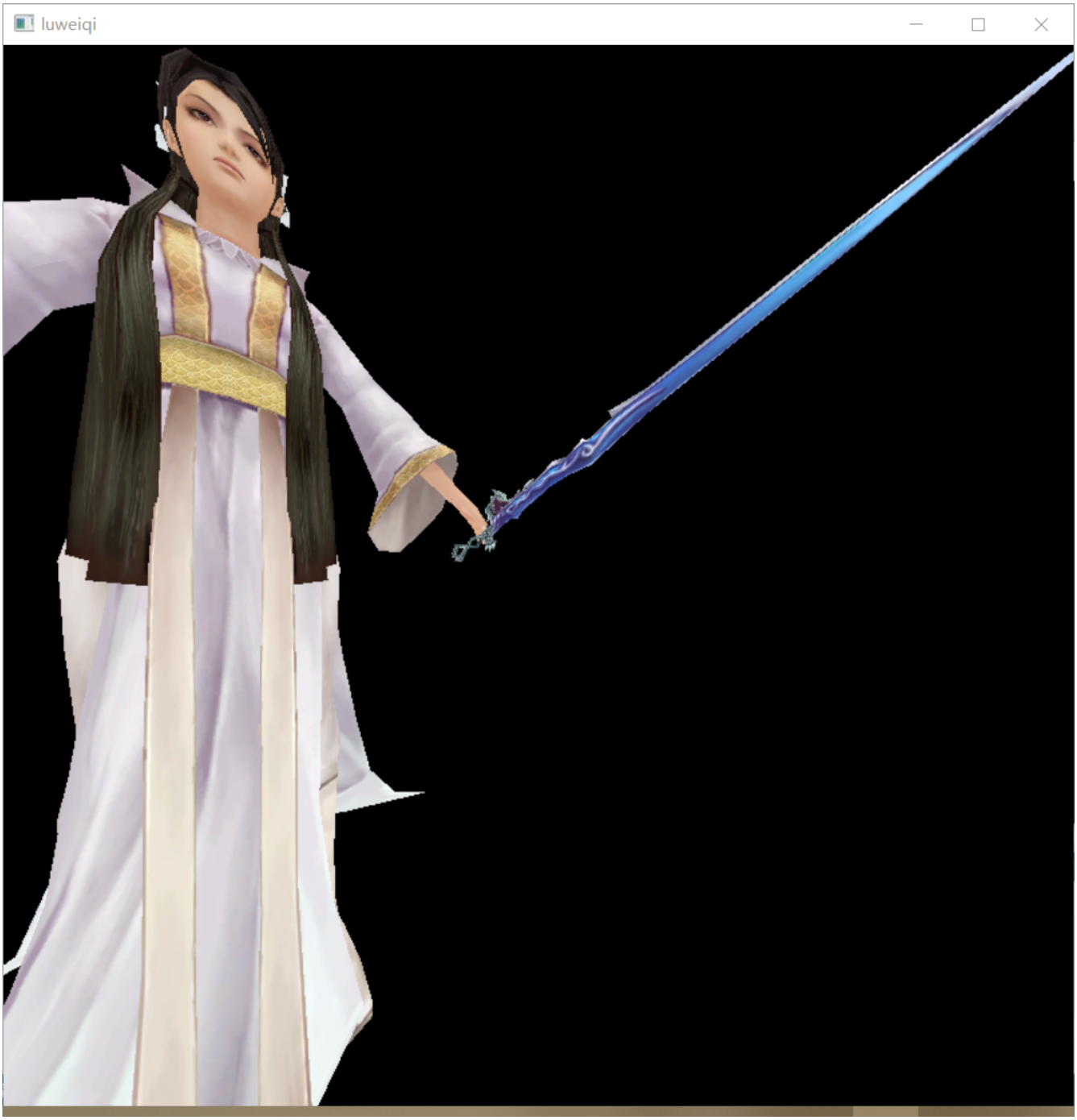


经过渲染的模型：

正交投影：



透视投影：


luweiqi