青岛大学

计算机科学技术学院

计算机图形学基础大作业

----基于 OpenGL 构建三维场景

课程名称：	计算机图形学基础
姓　　名：	李佳睿
专　　业：	软工创新
年　　级：	2021 级
学　　期：	2023 年秋季学期

2023 年 12 月

一、实验内容：

基于 C/C++语言和 OpenGL 进行程序设计，综合利用计算机图形学基础算法（包括多边形扫描转换、实体建模、几何变换、观察变换、投影变换、消隐、光照、纹理等）构建三维场景，并进行真实感渲染。

二、具体内容：

1. 建模：建立三维模型，如立方体、球体等。模型类型个数不限。（15%）

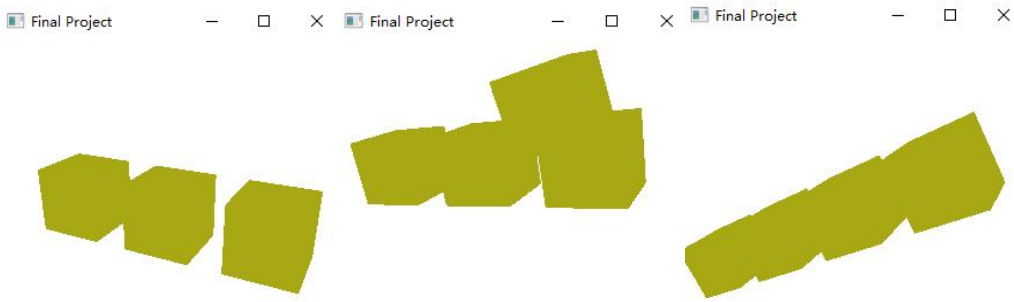
2. 几何变换和三维观察变换：对模型进行几何变换；设置摄像机位置、视线方向、向上方向、焦距，将模型的顶点坐标变换为观察坐标，并进行透视投影变换。（35%）

3. 渲染：根据投影后得到的二维多边形顶点序列，对多边形进行扫描转换，并实现面消隐、光照计算等真实感图形学技术。（35%）

4. 交互：程序需要实现交互操作，可用鼠标或键盘控制模型的几何变换或观察变换参数，也可实现三维漫游。（15%）

三、具体内容实现：

1.场景的设计和显示：



最终场景效果如上图所示。该虚拟场景主要描述了 3 个立方体外加 1 个立方体的场景。在该场景中包含多个虚拟物体，运用了物体绘制，层级建模，纹理贴图，光照效果，阴影效果等多种技术。除此之外还有一系列打印出的详细的操作指南，可以通过键盘进行多种视角切换，物体控制和交互等。

2.建模：

```cpp
// 绘制立方体
void DrawCube(float size) {
    glBegin(GL_QUADS);

    glVertex3f(-size / 2, -size / 2, -size / 2);
    glVertex3f(size / 2, -size / 2, -size / 2);
    glVertex3f(size / 2, size / 2, -size / 2);
    glVertex3f(-size / 2, size / 2, -size / 2);

    glVertex3f(-size / 2, -size / 2, size / 2);
    glVertex3f(size / 2, -size / 2, size / 2);
    glVertex3f(size / 2, size / 2, size / 2);
    glVertex3f(-size / 2, size / 2, size / 2);

    glVertex3f(-size / 2, -size / 2, -size / 2);
    glVertex3f(size / 2, -size / 2, -size / 2);
    glVertex3f(size / 2, -size / 2, size / 2);
    glVertex3f(-size / 2, -size / 2, size / 2);

    glVertex3f(-size / 2, size / 2, -size / 2);
    glVertex3f(size / 2, size / 2, -size / 2);
    glVertex3f(size / 2, size / 2, size / 2);
    glVertex3f(-size / 2, size / 2, size / 2);

    glVertex3f(-size / 2, -size / 2, -size / 2);
    glVertex3f(-size / 2, size / 2, -size / 2);
    glVertex3f(-size / 2, size / 2, size / 2);
    glVertex3f(-size / 2, -size / 2, size / 2);

    glVertex3f(size / 2, -size / 2, -size / 2);
    glVertex3f(size / 2, size / 2, -size / 2);
    glVertex3f(size / 2, size / 2, size / 2);
    glVertex3f(size / 2, -size / 2, size / 2);

    glEnd();
}
```

```cpp
// 绘制球体
void DrawSphere(float radius) {
    glBegin(GL_QUAD_STRIP);
    for (int i = 0; i <= NumStacks; ++i) {
        float phi = PI * i / NumStacks;
        for (int j = 0; j <= NumSlices; ++j) {
            float theta = 2 * PI * j / NumSlices;
            float x = radius * sin(phi) * cos(theta);
            float y = radius * sin(phi) * sin(theta);
            float z = radius * cos(phi);
            glVertex3f(x, y, z);

            phi = PI * (i + 1) / NumStacks;
            x = radius * sin(phi) * cos(theta);
            y = radius * sin(phi) * sin(theta);
            z = radius * cos(phi);
            glVertex3f(x, y, z);
        }
    }
    glEnd();
}
```

```cpp
// 绘制椭球体
void DrawEllipsoid(float radiusX, float radiusY, float radiusZ) {
    glBegin(GL_QUAD_STRIP);
    for (int i = 0; i <= NumStacks; ++i) {
        float phi = PI * i / NumStacks;
        for (int j = 0; j <= NumSlices; ++j) {
            float theta = 2 * PI * j / NumSlices;
            float x = radiusX * sin(phi) * cos(theta);
            float y = radiusY * sin(phi) * sin(theta);
            float z = radiusZ * cos(phi);
            glVertex3f(x, y, z);

            phi = PI * (i + 1) / NumStacks;
            x = radiusX * sin(phi) * cos(theta);
            y = radiusY * sin(phi) * sin(theta);
            z = radiusZ * cos(phi);
            glVertex3f(x, y, z);
        }
    }
    glEnd();
}
```

3.几何变换和三维观察变换：

```
142  // 进行几何变换
143 void GeometryTransformations() {
144     glTranslatef(ModelTranslation[0], ModelTranslation[1], ModelTranslation[2]);
145     glTranslatef(ModelTranslationCube1[0], ModelTranslationCube1[1], ModelTranslationCube1[2]);
146     glTranslatef(ModelTranslationCube2[0], ModelTranslationCube2[1], ModelTranslationCube2[2]);
147     glTranslatef(ModelTranslationCube3[0], ModelTranslationCube3[1], ModelTranslationCube3[2]);
148     glRotatef(ModelRotation[0], 1.0f, 0.0f, 0.0f);
149     glRotatef(ModelRotation[1], 0.0f, 1.0f, 0.0f);
150     glRotatef(ModelRotation[2], 0.0f, 0.0f, 1.0f);
151     glRotatef(ModelRotationCube1[0], 1.0f, 0.0f, 0.0f);
152     glRotatef(ModelRotationCube1[1], 0.0f, 1.0f, 0.0f);
153     glRotatef(ModelRotationCube1[2], 0.0f, 0.0f, 1.0f);
154     glRotatef(ModelRotationCube2[0], 1.0f, 0.0f, 0.0f);
155     glRotatef(ModelRotationCube2[1], 0.0f, 1.0f, 0.0f);
156     glRotatef(ModelRotationCube2[2], 0.0f, 0.0f, 1.0f);
157     glRotatef(ModelRotationCube3[0], 1.0f, 0.0f, 0.0f);
158     glRotatef(ModelRotationCube3[1], 0.0f, 1.0f, 0.0f);
159     glRotatef(ModelRotationCube3[2], 0.0f, 0.0f, 1.0f);
160     glScalef(ModelScale[0], ModelScale[1], ModelScale[2]);
161 }
```

我们使用 glTranslatef 进行平移变换，使用 glRotatef 进行旋转变换，使用 glScalef 进行缩放变换。我们分别对于立方体 1、立方体 2、立方体 3 进行平移变换、旋转变换和缩放变换。

```
163  // 进行视图变换
164 void ApplyViewTransformations() {
165     gluLookAt(CameraPosition[0], CameraPosition[1], CameraPosition[2],
166         LookAtPoint[0], LookAtPoint[1], LookAtPoint[2],
167         UpVector[0], UpVector[1], UpVector[2]);
168 }
169
```

我们使用 LookAtPoint()函数设置一个视图矩阵，其中 CameraPosition[]用来设定观察者的位置坐标，表示相机在三维空间中的位置。LookAtPoint[]函数用来表示相机所看到的目标位置。UpVector[]表示上方向的向量，指定相机的上方向。

```
36  float CameraPosition[3] = { 5.0f, 5.0f, 10.0f }; // 调整摄像机位置
37  float LookAtPoint[3] = { 0.0f, 0.0f, 0.0f };
38  float UpVector[3] = { 0.0f, 1.0f, 0.0f };
39  float FocalLength = 5.0f;
```

用来设置相机的位置、实现方向、上方向和焦距。

```
354  // 改变形状
355 void Reshape(int width, int height) {
356     glViewport(0, 0, width, height);
357     glMatrixMode(GL_PROJECTION);
358     glLoadIdentity();
359     gluPerspective(45.0, (double)width / height, 1.0, 100.0);
360     glMatrixMode(GL_MODELVIEW);
361 }
```

将模型的顶点坐标变换为观察坐标，并进行透视投影变换。其中 glViewport(0, 0, width, height)用来设置视口的位置和大小。视口定义了窗口中能够显示图形的区域，通常与窗口大小一致。glMatrixMode(GL_PROJECTION)用来设置当前矩阵为投影矩阵。OpenGL 中有两

个主要的矩阵堆栈，一个用于模型视图变换，另一个用于投影变换。这里切换到投影矩阵堆栈。glLoadIdentity()用来将当前矩阵重置为单位矩阵。这是为了确保接下来的变换是基于一个干净的状态进行的。gluPerspective(45.0, (double)width / height, 1.0, 100.0)使用透视投影矩阵。gluPerspective 函数定义了一个透视投影矩阵，它会根据参数设置创建透视效果。参数包括视场角（在垂直方向上的），宽高比，以及近裁剪面和远裁剪面的距离。这个函数将透视投影矩阵乘到当前投影矩阵堆栈的栈顶。glMatrixMode(GL_MODELVIEW)用来切换回模型视图矩阵堆栈。这样之后的变换操作将影响模型视图矩阵。

4. 添加光照、材质：

```
298     // 设置光源0的属性
299     GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
300     GLfloat light_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
301     GLfloat light_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
302     GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
303
304     glLightfv(GL_LIGHT0, GL_POSITION, light_position);
305     glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
306     glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
307     glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
```

GLfloat light_position 用来定义光源的位置，GLfloat light_ambient 用来定义光源的环境光的颜色，GLfloat light_diffuse 用来定义光源漫反射光的颜色，GLfloat light_specular 用来定义光源的镜面光的颜色。

```
309     // 设置立方体的材质属性（黄色）
310     GLfloat material_diffuse_cube[] = { 1.0, 1.0, 0.0, 1.0 };
311     GLfloat material_specular_cube[] = { 1.0, 1.0, 1.0, 1.0 };
312     GLfloat material_shininess_cube[] = { 50.0 };
313
314     glMaterialfv(GL_FRONT, GL_DIFFUSE, material_diffuse_cube);
315     glMaterialfv(GL_FRONT, GL_SPECULAR, material_specular_cube);
316     glMaterialfv(GL_FRONT, GL_SHININESS, material_shininess_cube);
```

这段代码用来设置物体的材质属性。GLfloat material_diffuse_cube[]用来定义物体漫反射光的颜色，GLfloat material_specular_cube[]用来定义物体的镜面光的颜色，GLfloat material_shininess_cube[]用来定义物体的高光系数。

5.渲染：

```
// 区间扫描线算法
void ScanlineAlgorithm() {
    for (const auto& vertices : PolygonVertices) {
        for (size_t i = 1; i < vertices.size() - 1; ++i) {
            float x1 = vertices[0].first;
            float y1 = vertices[0].second;
            float z1 = InterpolateZ2(x1, y1, vertices[i].first, vertices[i].second,
                vertices[i + 1].first, vertices[i + 1].second);

            float x2 = vertices[i].first;
            float y2 = vertices[i].second;
            float z2 = InterpolateZ2(x2, y2, vertices[i + 1].first, vertices[i + 1].second,
                x1, y1);

            float x3 = vertices[i + 1].first;
            float y3 = vertices[i + 1].second;
            float z3 = InterpolateZ2(x3, y3, x1, y1, x2, y2);

            FillTriangle(x1, y1, z1, x2, y2, z2, x3, y3, z3);
        }
    }
}
```

使用 ScanlineAlgorithm()函数进行区间扫描线算法实现多边形扫描转换

```
// 区间扫描线填充一个三角形
void FillTriangle(float x1, float y1, float z1,
    float x2, float y2, float z2,
    float x3, float y3, float z3) {

    // 计算三角形的包围盒
    float minX = std::min({ x1, x2, x3 });
    float minY = std::min({ y1, y2, y3 });
    float maxX = std::max({ x1, x2, x3 });
    float maxY = std::max({ y1, y2, y3 });

    // 对包围盒进行扫描线填充
    for (int y = static_cast<int>(minY); y <= static_cast<int>(maxY); ++y) {
        for (int x = static_cast<int>(minX); x <= static_cast<int>(maxX); ++x) {
            float AreaTri = Area(x1, y1, x2, y2, x3, y3);
            float Area1 = Area(x, y, x2, y2, x3, y3);
            float Area2 = Area(x, y, x3, y3, x1, y1);
            float Area3 = Area(x, y, x1, y1, x2, y2);

            if (Area1 + Area2 + Area3 <= AreaTri) {
                float zInterpolated = InterpolateZ(x, y, x1, y1, z1, x2, y2, z2, x3, y3, z3)

                // 执行深度测试
                if (zInterpolated < DepthBuffer[y][x]) {
                    DepthBuffer[y][x] = zInterpolated;

                    // 进行颜色填充，这里简单使用红色
                    glColor3f(1.0, 0.0, 0.0);
                    glBegin(GL_POINTS);
                    glVertex2f(x, y);
                    glEnd();
                }
            }
        }
    }
}
```

使用了深度缓冲（Depth Buffer）来进行面消隐。在 FillTriangle()函数中，通过比较深度值来确定每个像素是否被三角形覆盖，以达到隐藏被遮挡的部分的目的。但是，并没有使用如 Z 缓冲缓存冲突、透明物体的处理这样更加高级的面消隐算法。

对于光照的计算，在 Display() 函数中启用了光照 (glEnable(GL_LIGHTING))，并设置了光源 (GL_LIGHT0) 的属性，包括环境光（ambient）、漫反射光（diffuse）、镜面反射光（specular）等。同时为立方体设置了材质属性（material_diffuse_cube、material_specular_cube、material_shininess_cube）以影响光照计算。这些属性在 glMaterialfv 中进行了设置。

```
329     glTranslatef(3.0f, 0.0f, 0.0f);  // 调整平移，使得两个立方体不重叠
330     glColor3f(0, 0, 1); // 渲染蓝色的立方体
331     DrawCube(2);
```

同时使用 glColor3f 对于绘制的图形，如立方体等进行渲染。

6.交互：

```
// 键盘回调函数
void Keyboard(unsigned char key, int x, int y) {
    float TranslationSpeed = 0.1f;

    switch (key) {
    case 'w':
        if (SelectedCube == 1) ModelTranslationCube1[1] += TranslationSpeed;
        else if (SelectedCube == 2) ModelTranslationCube2[1] += TranslationSpeed;
        else if (SelectedCube == 3) ModelTranslationCube3[1] += TranslationSpeed;
        break;
    case 's':
        if (SelectedCube == 1) ModelTranslationCube1[1] -= TranslationSpeed;
        else if (SelectedCube == 2) ModelTranslationCube2[1] -= TranslationSpeed;
        else if (SelectedCube == 3) ModelTranslationCube3[1] -= TranslationSpeed;
        break;
    case 'a':
        if (SelectedCube == 1) ModelTranslationCube1[0] -= TranslationSpeed;
        else if (SelectedCube == 2) ModelTranslationCube2[0] -= TranslationSpeed;
        else if (SelectedCube == 3) ModelTranslationCube3[0] -= TranslationSpeed;
        break;
    case 'd':
        if (SelectedCube == 1) ModelTranslationCube1[0] += TranslationSpeed;
        else if (SelectedCube == 2) ModelTranslationCube2[0] += TranslationSpeed;
        else if (SelectedCube == 3) ModelTranslationCube3[0] += TranslationSpeed;
        break;
    case 'q':
        if (SelectedCube == 1) ModelRotationCube1[2] += 5.0f;
        else if (SelectedCube == 2) ModelRotationCube2[2] += 5.0f;
        else if (SelectedCube == 3) ModelRotationCube3[2] += 5.0f;
        break;
    case 'e':
        if (SelectedCube == 1) ModelRotationCube1[2] -= 5.0f;
        else if (SelectedCube == 2) ModelRotationCube2[2] -= 5.0f;
        else if (SelectedCube == 3) ModelRotationCube3[2] -= 5.0f;
        break;
    case '1':
        SelectedCube = 1;
        break;
    case '2':
        SelectedCube = 2;
        break;
    case '3':
        SelectedCube = 3;
        break;
    case 'x':
        SelectedCube = 0;
        break;
    }
    glutPostRedisplay();
}
```

使用键盘回调函数进行基本的交互。

7.使用说明：

本项目是基于 Visual Studio2022+OpenGL 进行开发的，实现了利用计算机图形学基础算法（包括多边形扫描转换、实体建模、几何变换、观察变换、投影变换、消隐、光照、纹理

等）构建三维场景，并进行真实感渲染。虽然构建的三维场景较为简单，但是也是将所学知识进行了实际的学习和应用。

我们的三维场景设计了 3+1 个立方体，其中 3 个立方体是始终存在于三维场景当中的。另一个立方体会根据我们的按键出现在不同的位置上。若我们按下 1，则这一个立方体会出现在中间的立方体附近；若我们按下 2，则这一个立方体会出现在右边的这一个立方体附近；若我们按下 3，则这一个立方体会出现在左边的这一个立方体附近。这一个立方体会较其他三个立方体有更快的移动速度。

- `'w'`: 向上移动所选的立方体。
- `'s'`: 向下移动所选的立方体。
- `'a'`: 向左移动所选的立方体。
- `'d'`: 向右移动所选的立方体。
- `'q'`: 绕垂直轴逆时针旋转所选的立方体。
- `'e'`: 绕垂直轴顺时针旋转所选的立方体。
- `'1'`: 选择第1个立方体进行操作。
- `'2'`: 选择第2个立方体进行操作。
- `'3'`: 选择第3个立方体进行操作。
- `'x'`: 取消选择任何立方体。

上图是其他的一些基本操作。

- `GLUT_KEY_UP`: 当用户按下上箭头键时，使所选的立方体向屏幕内部移动（沿 Z 轴负方向）。
- `GLUT_KEY_DOWN`: 当用户按下下箭头键时，使所选的立方体向屏幕外部移动（沿 Z 轴正方向）。

上图是一些特殊的操作。

代码当中的算法及其原理：

光照计算：

光源设置：通过设置光源的位置、环境光、漫反射光、镜面光等属性，在代码当中使用了一个白色光源 GL_LIGHT0。

材质属性：通过设置物体表面的材质属性，包括漫反射、镜面反射等。在代码当中通过 glMaterialfv 函数设置了一个黄色立方体的漫反射和镜面反射属性。

深度缓冲（Depth Buffer）：

深度测试：通过启用深度测试 glEnable(GL_DEPTH_TEST)，在代码当中使用深度测试来确保在绘制多个物体时正确处理遮挡关系。

深度缓冲清除：在每一帧绘制之前，通过 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT) 清除深度缓冲。

视图变换和几何变换：

视图变换：通过 gluLookAt 函数实现视图变换，调整摄像机位置、观察点和上方向。

几何变换：通过 glTranslatef、glRotatef、glScalef 函数对模型进行平移、旋转和缩放，实现几何变换。

图元绘制：

绘制球体：通过使用 glBegin(GL_QUAD_STRIP) 和 glVertex3f 绘制球体的顶点。

绘制立方体：通过 glBegin(GL_QUADS) 和 glVertex3f 绘制立方体的顶点。

区间扫描线填充算法：

三角形面积计算：通过 Area 函数计算三角形的面积。

插值计算深度：通过 InterpolateZ 函数计算在三角形内插值得到每个像素的深度。

扫描线填充：通过 FillTriangle 函数对三角形进行扫描线填充。

阴影的添加：

深度缓冲阴影：

通过在绘制物体之前，关闭深度缓冲的写入，绘制场景的光源部分，生成深度图。然后，再次打开深度缓冲写入，绘制实际物体时，根据深度图进行阴影计算。在你的代码中，通过 glDepthMask(GL_FALSE) 关闭深度缓冲写入，绘制光源和立方体，然后通过 glDepthMask(GL_TRUE) 打开深度缓冲写入，绘制其他物体。

在做这个实验项目的过程当中，自己遇到了一些问题，例如绘制的图元无法在三维场景当中显示、无法设置有效的阴影、纹理、无法通过键盘回调函数进行有效的控制等等，包括基本环境的配置，都花费了自己较多的时间进行配置和调试，但是正是在解决这些问题的过程当中，自己对于计算机图形学的一些基本的算法，以及这些算法的实现有了一个更为清晰的认识。虽然自己构建的三维场景较为简单，但是更为重要的是，自己在做这个的过程当中，真正地锻炼提升了自己的能力。

致谢：

感谢 github 上面的一些开源的项目对自己的启发和思考，尤其是深圳大学石弋川同学的大作业（https://github.com/CarrotSwordsman/ComputerGraphics_SZU/tree/main）带给了自己很大的启发，自己有感于该同学硬核以及高质量的大作业，该大作业构建了非常复杂的三维场景，而且有很多自己的思考，我觉得这一点品质是尤为可贵的，也是自己应该学习的品质。

说明：

本项目的代码完全公开，供交流学习和进一步的完善，开源地址为：https://github.com/Jerry-woodson/Computer-Graphics-Project