计算机图形学
课程项目

王徐笑风[*]　　　凌泽辉[†]

2020 年 11 月 5 日

[*]学号:18120193 E-mail:2208740924@qq.com
[†]学号:18120193 E-mail:785896610@qq.com

目录

摘要

查找资料，学习了解三维网格模型的相关知识。完成一个三维网格模型的显示系统。

数据输入：通过文件读取模型数据

数据存储：设计程序内用于存储模型数据的数据结构

数据输出：在窗口界面进行模型显示

编程实现三维到二维的投影变换计算

编程实现通过键盘或鼠标驱动模型的平移、缩放及旋转变换

可以使用开发工具中提供光照函数，若自己编程实现光照计算，则可获得额外加分

1 三维图形的变换与表示

1.1 向量与点

在三维空间中我们常用一个三维向量表示一个点，虽然向量本身只表达长度和方向，他是无关坐标系的，而点显然是与选取的坐标系是相关的。因此在这里将点理解为在一个给定的坐标系下，原点按某一向量移动后的位置，它写作式 (1)：

$$\mathbf{P} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{1}$$

1.2 变换

对于三维空间中的点，常用到的仿射变换和二维中的类似：平移、旋转和缩放。

1.2.1 平移

平移变换是将一个点按一个方向，移动一段距离。考虑到上面我们的点的定义即为在给点的坐标系下，原点按一个向量移动的距离。那么显然平移一个点即将原点平移两次，即点所代表的“向量”和平移向量的共同作用。

对于点 $\mathbf{P}$，将其平移 $\mathbf{V}$：

$$\mathbf{P} = \begin{bmatrix} \mathbf{P}_x \\ \mathbf{P}_y \\ \mathbf{P}_z \end{bmatrix}$$

$$\mathbf{V} = \begin{bmatrix} \mathbf{V}_x \\ \mathbf{V}_y \\ \mathbf{V}_z \end{bmatrix}$$

则有平移后的点 $\mathbf{N}$：

$$\mathbf{N} = \begin{bmatrix} \mathbf{P}_x \\ \mathbf{P}_y \\ \mathbf{P}_z \end{bmatrix} + \begin{bmatrix} \mathbf{V}_x \\ \mathbf{V}_y \\ \mathbf{V}_z \end{bmatrix} = \begin{bmatrix} \mathbf{P}_x + \mathbf{V}_x \\ \mathbf{P}_y + \mathbf{V}_y \\ \mathbf{P}_z + \mathbf{V}_z \end{bmatrix}$$

1.2.2 旋转

旋转指的是：点以三维空间中的某点为旋转中心，进行旋转，这里简略的认为旋转中心为坐标系原点，即此时的旋转变换是一个特殊的线性变换。在三维坐标系中对点做按原点的旋转，即是对一个向量进行旋转。只需要求取原基向量 $\hat{i}$、$\hat{j}$、$\hat{k}$ ，在旋转后的 $\hat{i'}$、$\hat{j'}$、$\hat{k'}$，可以得到旋转矩阵：

$$\mathbf{RotateMatrix} = \begin{bmatrix} \hat{i'_x} & \hat{j'_x} & \hat{k'_x} \\ \hat{i'_y} & \hat{j'_y} & \hat{k'_y} \\ \hat{i'_z} & \hat{j'_z} & \hat{k'_z} \end{bmatrix}$$

则有旋转后的点 $\mathbf{N}$：

$$\mathbf{N} = \begin{bmatrix} \hat{i'_x} & \hat{j'_x} & \hat{k'_x} \\ \hat{i'_y} & \hat{j'_y} & \hat{k'_y} \\ \hat{i'_z} & \hat{j'_z} & \hat{k'_z} \end{bmatrix} \times \begin{bmatrix} \mathbf{P}_x \\ \mathbf{P}_y \\ \mathbf{P}_z \end{bmatrix}$$

特别的，绕 Y 轴旋转 θ 弧度的旋转矩阵，可以这么考虑：首先 Y 轴显然是不变的，在左手系下从 Y 轴逆方向向下看，Z 轴 X 轴正好组成一个平面直角坐标系。则 $\hat{i}$ 顺时针旋转过 θ 弧度后的向量为：

$$\hat{i'} = \begin{bmatrix} cos(\theta) \\ 0 \\ -sin(\theta) \end{bmatrix}$$

同样的 $\hat{k}$ 旋转过 θ 弧度后的向量为：

$$\hat{k}' = \begin{bmatrix} sin(\theta) \\ 0 \\ cos(\theta) \end{bmatrix}$$

综上，Y 轴旋转矩阵为：

$$\mathbf{RotateY}(\theta) = \begin{bmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{bmatrix} \tag{2}$$

类似的我们也可以得到 X 轴旋转和 Z 轴旋转矩阵：

$$\mathbf{RotateX}(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\theta) & -sin(\theta) \\ 0 & sin(\theta) & cos(\theta) \end{bmatrix} \tag{3}$$

$$\mathbf{RotateZ}(\theta) = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4}$$

最后，通过上述 (3)、(2)、(4) 的旋转矩阵的复合矩阵即可实现任意的旋转，即：

$$\mathbf{Rotate}(\alpha, \beta, \gamma) = \mathbf{RotateZ}(\gamma) \times \mathbf{RotateY}(\beta) \times \mathbf{RotateX}(\gamma)$$

1.2.3 缩放

缩放的实现和旋转矩阵类似，计算出新的 $\hat{i}'$、$\hat{j}'$、$\hat{k}'$ 即可：

$$\mathbf{Scale}(\alpha, \beta, \gamma) = \begin{bmatrix} \alpha & 0 & 0 \\ 0 & \beta & 0 \\ 0 & 0 & \gamma \end{bmatrix}$$

则缩放后的点为：

$$\mathbf{N} = \begin{bmatrix} \alpha & 0 & 0 \\ 0 & \beta & 0 \\ 0 & 0 & \gamma \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \alpha x \\ \beta y \\ \gamma z \end{bmatrix}$$

1.2.4 复合变换

不难看出上述的变换中，旋转和缩放的变换可以简单地实现符合，将旋转矩阵和缩放矩阵进行矩阵乘法复合即可，同时这里两者是可交换的。

但平移变换就无法使用矩阵表达（矩阵变换后的点 $\mathbf{N}$ 的一个分量可以表达为 $\mathbf{N}_x = a\mathbf{P}_x + b\mathbf{P}_y + c\mathbf{P}_z$ 而平移变换可表达为 $\mathbf{N}_x = \mathbf{P}_x + d$ 显然上述公式中 $a = 1, \mathbf{P}_y + c\mathbf{P}_z = d$ 显然无法使用一个静态的矩阵实现），因此在描述一个点的平移、旋转和缩放变换时，我们使用下述公式 (5)

$$\mathbf{N} = (\mathbf{Scale}(a, b, c) \times \mathbf{Rotate}(\alpha, \beta, \gamma)) \times \mathbf{P} + \mathbf{V} \tag{5}$$

但使用上述公式计算是不“简单”的，我们希望能有一种方法通过一次一种计算即可表达上述三种变换，同时又能提高运算的效率。

1.3 三维的齐次坐标

在三维中我们无法将平移、旋转和缩放变换由一个矩阵描述。不妨假设我们在四维中，并规定点 $\mathbf{P}$ 在四维中的坐标为：

$$\mathbf{P} = \begin{bmatrix} x \\ y \\ z \\ w = 1 \end{bmatrix}$$

即对于 $\forall \mathbf{P}$ 他们位于四维空间中分量 $w = 1$ 的一个三维“切片”空间中。而对于 $w \neq 1$ 的点，可以通过计算 $x' = \frac{x}{w}, y' = \frac{y}{w}, z' = \frac{z}{w}$ 来得到对应单位空间中的坐标。

有了三维其次坐标后，我们可以改写公式 (5) 为：

$$\mathbf{N} = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ 0 & 0 & 0 & 1 \end{bmatrix}_{\mathbf{Transform}} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}_{\mathbf{P}} = \begin{bmatrix} ax + by + cz + d \\ ex + fy + gz + h \\ ix + jy + kz + l \\ 1 \end{bmatrix} \tag{6}$$

观察公式 (6)，其中 d, h, l 显然对应了平移向量：

$$\mathbf{V} = \begin{bmatrix} d \\ h \\ l \\ 1 \end{bmatrix}$$

它可以写成一个平移矩阵：

$$\textbf{Translate} = \begin{bmatrix} 1 & 0 & 0 & d \\ 0 & 1 & 0 & h \\ 0 & 0 & 1 & l \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7}$$

而将平移矩阵和旋转缩放矩阵复合即可得到公式 (6) 中的变换矩阵：

$$\begin{aligned} \textbf{Translate} \times \textbf{Scale} \times \textbf{Rotation} &= \begin{bmatrix} 1 & 0 & 0 & d \\ 0 & 1 & 0 & h \\ 0 & 0 & 1 & l \\ 0 & 0 & 0 & 1 \end{bmatrix}_{\textbf{Translate}} \times \begin{bmatrix} a & b & c & 0 \\ e & f & g & 0 \\ i & j & k & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}_{\textbf{Scale}\times\textbf{Rotation}} \\ &= \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ 0 & 0 & 0 & 1 \end{bmatrix}_{\textbf{Transform}} \end{aligned}$$

综上我们通过三维空间中的齐次坐标实现了 3 种变换的复合。当然 $Transform$ 矩阵也能表达切变等仿射变换，由于在三维投影的过程中并不常用，这里就不详细叙述。

1.3.1 $sunMatLib.h$ 矩阵运算库

在本项目中，除了窗口创建和像素绘制使用到了第三方软件库，所有的数据结构、矩阵矩阵运算、投影、光栅化、光照等都有我们小组自己实现。这里简单介绍一下整个项目中频繁使用到的矩阵运算类库，见源码：

```
class Vector3
{
    friend class Matrix4;

public:
    double _x, _y, _z, _w;

public:
    Vector3();
    Vector3(const double &x, const double &y, const
        double &z);
    Vector3(const Vector3 &copy);

    Vector3 &operator=(const Vector3 &copy);
    Vector3 operator+(const Vector3 &b) const;
    Vector3 operator-(const Vector3 &b) const;
    Vector3 operator*(const double &a) const;
    friend Vector3 operator*(const double &a, const
        Vector3 &b);
    double operator*(const Vector3 &b) const;
    Vector3 &operator*=(const double &a);
    Vector3 &operator+=(const Vector3 &b);
    friend Vector3 operator/(const Vector3 &b, const
        double &a);
    friend Vector3 &operator/=(Vector3 &b, const
        double &a);
    double length() const;
    Vector3 normalize() const;
    Vector3 &normalized();
    Vector3 cross(const Vector3 &b) const;
    Vector3 &crossed(const Vector3 &b);
```

```
    Vector3 &set(const double &x, const double &y,
       const double &z);

    static Vector3 ONE();
    static Vector3 ZERO();
    static Vector3 UP();
    static Vector3 FRONT();
    static Vector3 RIGHT();
};
```

```
class Matrix4
{
    friend class Vector3;

    private:
    double _mat[4][4];

    public:
    Matrix4();
    Matrix4(double copy[4][4]);
    Matrix4(const Matrix4 &copy);

    Matrix4 quickInvert();
    Matrix4 &operator=(const Matrix4 &copy);
    Matrix4 operator*(const Matrix4 &b);
    Vector3 operator*(const Vector3 &b);

    static Matrix4 PROJECTION(const double &
       aspect_ratio, const double &fov_rad, const
       double &near_panel, const double &far_panel);
    static Matrix4 ROTATE_X(const double &angle);
    static Matrix4 ROTATE_Y(const double &angle);
```

```
    static Matrix4 ROTATE_Z(const double &angle);
    static Matrix4 ROTATE(const double &x, const double &y, const double &z);
    static Matrix4 SCALE(const double &x, const double &y, const double &z);
    static Matrix4 TRANSLATE(const double &x, const double &y, const double &z);
    static Matrix4 POINTAT(Vector3 pos, Vector3 target, Vector3 up);
};
```

1.4 三角面

虽然三维形体有很多表示方法，但在三维的渲染引擎或是游戏引擎中，最常使用到的还是三角网格模型。

在这种网格模型中一个面一个三维空间中的三角面，它由 3 个顶点 $\mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3$ 即可表达。

保留法向信息，三角化的 $.obj$ 文件就是这样一种三角网格模型，在本项目中也该文件格式作为标准。

特别的，在三维渲染中，单个三角面往往还有“朝向”这一属性。背对我们的三角面是不显示的，只有面对我们的才会渲染。这种规定在处理封闭模型和时候会更加有效：首先我们是无法看到封闭模型的内部的，自然无需渲染其三角面的背面；第二，背对我们的三角面不进行渲染也能降低对于性能的需求。因此三角面还存在一个法向量，它垂直于三角面所在的平面，并与三角面朝向同向。

但这不代表我们需要使用除 3 个三维点以外的数据保存法向信息。在 $.obj$ 中一个三角面 $face$ 由 3 个顶点 $vertex$ 组成。这三个顶点按原三角面的逆时针排列，其法向满足右手螺旋定则，因此对于 1 个三角面的有序顶点 $\mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3$ ，该三角面的单位法向量 $\mathbf{U}$ 为：

$$\mathbf{U} = \frac{(\mathbf{P}_2 - \mathbf{P}_1) \times (\mathbf{P}_3 - \mathbf{P}_1)}{|(\mathbf{P}_2 - \mathbf{P}_1) \times (\mathbf{P}_3 - \mathbf{P}_1)|}$$

在 $sunMatLib.h$ 也有三角面类 $TriFace$ ，它实现了法向的自动计算，在通过顶点构造了三角面后，即可通过 $get_Normal()$ 方法获得单位法向量。具体实现可见源程序代码。

1.5 网格

在有了三角面的数据结构后，三角网格模型的最简单的实现就是使用变长数组保存每一个三角面。

在 $sun3D.h$ 中的 $Mesh$ 类就是使用了 $std::vector<TriFace>$ 来保存三角网格模型。

1.5.1 $.obj$ 文件的加载

$.obj$ 文件的构成或是说格式，基本上一一对应了上述的三角面、三角网格模型的数据结构。我们先来看一个 $.obj$ 文件实例（$untitled1.obj$ 文件有省略）：

```
# Blender v2.83.4 OBJ File: ''
# www.blender.org
o Torus
v 1.250000 0.000000 0.000000
v 1.216506 0.125000 0.000000
v 1.125000 0.216506 0.000000
v 1.000000 0.250000 0.000000
......
s off
f 13 2 1
f 2 15 3
f 15 4 3
f 16 5 4
......
```

其中“$\#$”开头的是注释，忽略即可。

“v” 开头的一行定义了一个顶点，后跟三个数值分别表示该顶点 $\mathbf{P}$ 的 x, y, z 分量

“f” 开头的一行定义了一个三角面，后跟三个整数分别表示该三角面的 3 个顶点 $\mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3$ 所对应的顶点 “v” 的序号。例如：上述文件中的第 10 行 “f 13 2 1”，其中 “2 1” 表明 $\mathbf{P}_2$ 为第 5 行的顶点，$\mathbf{P}_3$ 为第 4 行的顶点。

2 投影

2.1 正交投影

2.2 透视投影

3 相机变换

3.1 世界坐标系

3.2 视口坐标系

3.3 坐标系变换

4 绘制/光栅化

4.1 三角填充

4.2 绘制顺序问题

4.3 画家算法

4.3.1 画家算法的问题

4.3.2 改进方向

5 三维裁剪

5.1 性能问题与原因

5.2 三角面裁剪

5.2.1 近平面裁剪

5.2.2 视口裁剪

6 基础光照

6.1 全局光照

6.2 方向光