计算机图形学
课程项目

王徐笑风* 凌泽辉†

2020 年 11 月 11 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 **P**，将其平移 **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 **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 **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 **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。

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：

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

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：

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

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：

```
1  class Vector3
2  {
3      friend class Matrix4;
4
5  public:
6      double _x, _y, _z, _w;
7
8  public:
9      Vector3();
10     Vector3(const double &x, const double &y, const
           double &z);
11     Vector3(const Vector3 &copy);
12
13     Vector3 &operator=(const Vector3 &copy);
```

```
14      Vector3 operator+(const Vector3 &b) const;
15      Vector3 operator-(const Vector3 &b) const;
16      Vector3 operator*(const double &a) const;
17      friend Vector3 operator*(const double &a, const
          Vector3 &b);
18      double operator*(const Vector3 &b) const;
19      Vector3 &operator*=(const double &a);
20      Vector3 &operator+=(const Vector3 &b);
21      friend Vector3 operator/(const Vector3 &b, const
          double &a);
22      friend Vector3 &operator/=(Vector3 &b, const double &
          a);
23      double length() const;
24      Vector3 normalize() const;
25      Vector3 &normalized();
26      Vector3 cross(const Vector3 &b) const;
27      Vector3 &crossed(const Vector3 &b);
28      Vector3 &set(const double &x, const double &y, const
          double &z);
29
30      static Vector3 ONE();
31      static Vector3 ZERO();
32      static Vector3 UP();
33      static Vector3 FRONT();
34      static Vector3 RIGHT();
35  };
```

```
1   class Matrix4
2   {
3       friend class Vector3;
4
5       private:
6       double _mat[4][4];
7
```

```
8     public:
9     Matrix4();
10    Matrix4(double copy[4][4]);
11    Matrix4(const Matrix4 &copy);
12
13    Matrix4 quickInvert();
14    Matrix4 &operator=(const Matrix4 &copy);
15    Matrix4 operator*(const Matrix4 &b);
16    Vector3 operator*(const Vector3 &b);
17
18    static Matrix4 PROJECTION(const double &aspect_ratio,
          const double &fov_rad, const double &near_panel,
          const double &far_panel);
19    static Matrix4 ROTATE_X(const double &angle);
20    static Matrix4 ROTATE_Y(const double &angle);
21    static Matrix4 ROTATE_Z(const double &angle);
22    static Matrix4 ROTATE(const double &x, const double &
          y, const double &z);
23    static Matrix4 SCALE(const double &x, const double &y
          , const double &z);
24    static Matrix4 TRANSLATE(const double &x, const
          double &y, const double &z);
25    static Matrix4 POINTAT(Vector3 pos, Vector3 target,
          Vector3 up);
26 };
```

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（*untitled*1*.obj* 文件有省略）：

```
1 # Blender v2.83.4 OBJ File: ''
2 # www.blender.org
3 o Torus
4 v 1.250000 0.000000 0.000000
5 v 1.216506 0.125000 0.000000
6 v 1.125000 0.216506 0.000000
7 v 1.000000 0.250000 0.000000
```

```
 8 ......
 9 s off
10 f 13 2 1
11 f 2 15 3
12 f 15 4 3
13 f 16 5 4
14 ......
```

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。

因此，读取 *.obj* 文件十分简单，设置一个 $std::vector<Vector3>$ 来保存顶点，当读取到一行以“v”开头就构造一个 *Vector*3 顶点，并放入 $std::vector<Vector3>$ 中，当读取到一行以“f”开头就从 $std::vector<Vector3>$ 中依次取出相应下标的顶点，构造三角面，并放入需要的 *Mesh* 实例中的 $std::vector<TriFace>$ 即可。

2 投影

想要实现三维模型的显示，最重要的是找到一种将三维模型“转换”/“映射”/“投影”到二维平面的方法。

当然我们需要规定一下这个投影后的二维平面的格式。一般的，在我们的程序中，屏幕的标准坐标系为：从左到右为 X 轴方向 $[-1,1]$ ，从下到上为 Y 轴方向 $[-1,1]$ 。

自然的，根据上述定义，采用左手系，可以得到 Z 轴方向为从屏幕外指向屏幕内部，这也是我们的摄像机的朝向。

2.1 正交投影

一种简单的投影方式是“正交投影”。它几乎可以简单的理解为：忽略三维点的 z 分量，直接将 x,y 分量和屏幕坐标相关联。因此，正交投影是“忽略透视的”，他没有消失点这一概念，在三维空间中的平行线在投影到屏幕坐标后也是保持平

行的。这当然我我们的现实经验是不同的，见图 (1)。虽然在某些场景下，正交投影的这种性质能更好地表达某些信息，但更为常用的还是“透视投影”。只因正交投影在之后介绍透视投影会有所帮助，因此在这里做简要的介绍。

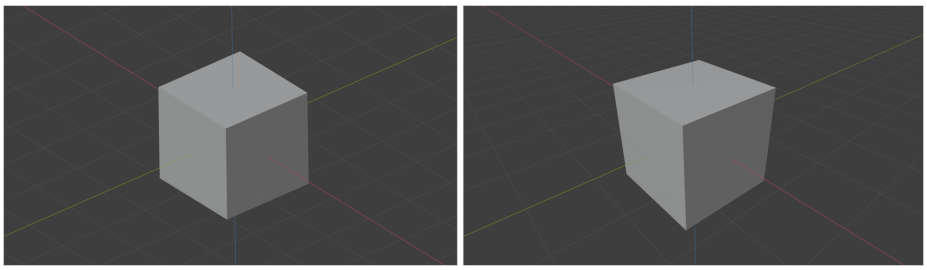


图 1: 正交投影 *vs* 透视投影

在正交投影中，我们需要把某个区域内的点映射到 $[-1,1]$ 的屏幕空间中，这个区域是一个长方体，称为“视域”。它由上下左右和远近 6 个平面组成。

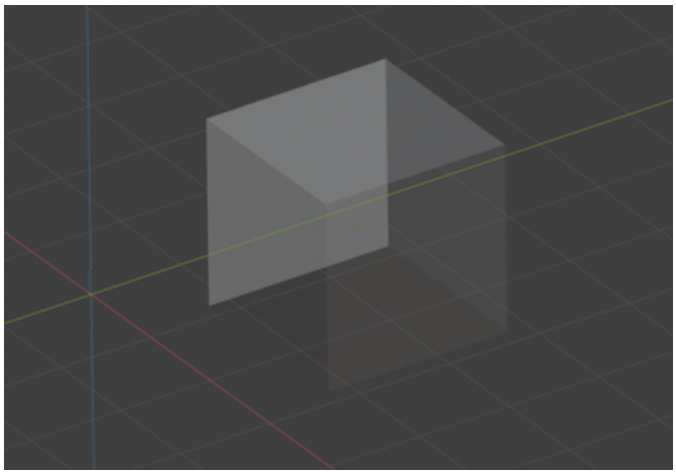


图 2: 视域

对于每个面有如下参数：

1. $l =$ 左平面 x 分量
2. $r =$ 右平面 x 分量
3. $t =$ 上平面 y 分量
4. $b =$ 下平面 y 分量
5. $f =$ 远平面 z 分量

6. $n=$ 近平面 z 分量

在这个视域中的三维点 $\mathbf{P}$，其 x 分量需要按照左右平面的位置映射近 $[-1,1]$ 区间，在数轴上可以如下表示：

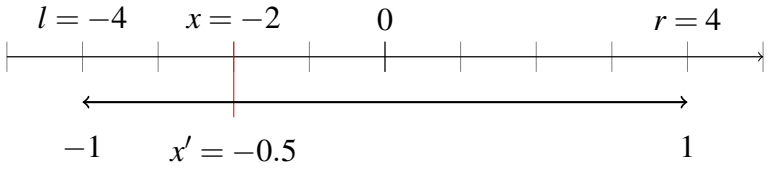


图 3: x 分量，与 l,r 的映射关系

显然的，x 分量的映射公式为：

$$x' = \frac{2x}{r-l} - \frac{r+l}{r-l}$$

同理，y 分量的映射公式为：

$$y' = \frac{2y}{t-b} - \frac{t+b}{t-b}$$

最后，z 分量的映射公式回稍有不同，因为这次我们将他映射到 $[0,1]$ 之间，而非 $[-1,1]$。

$$z' = \frac{z}{f-n} - \frac{n}{f-n}$$

不妨将上述三个公式整合入一个变换矩阵，即为正交投影矩阵：

$$\textbf{Orthogonal} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{1}{f-n} & -\frac{n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{8}$$

$$\mathbf{P}' = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{1}{f-n} & -\frac{n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{2x}{r-l} - \frac{r+l}{r-l} \\ \frac{2y}{t-b} - \frac{t+b}{t-b} \\ \frac{z}{f-n} - \frac{n}{f-n} \\ 1 \end{bmatrix}$$

特别的，若取 $l=-r, t=-b$ 则 **Orthogonal** 还可简化为：

$$\textbf{Orthogonal} = \begin{bmatrix} \frac{2}{w} & 0 & 0 & 0 \\ 0 & \frac{2}{h} & 0 & 0 \\ 0 & 0 & \frac{1}{f-n} & -\frac{n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{9}$$

其中 w 和 h 分别是视域的宽度和高度。

2.2 透视投影

透视投影能展现出人眼近大远小的视觉经验。它与正交透视的主要区别就是视域不同，不再是一个长方体，而是一个类似截去了一块的金字塔：

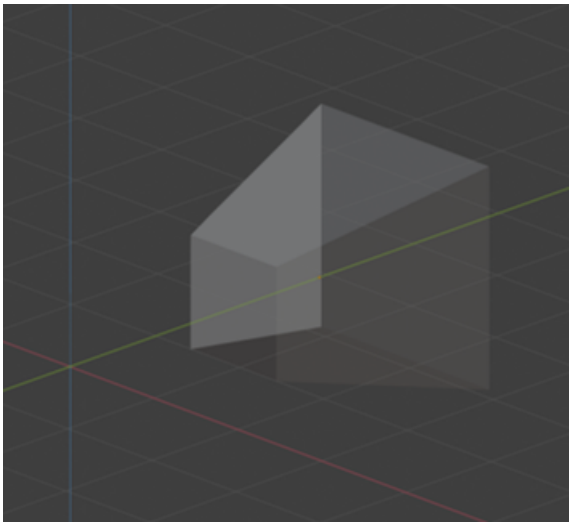


图 4: 透视投影视域

在上图中，4 根侧边的棱的延长线会交于坐标原点。实际上，在视域中的任意一点与原点构成的直线上的任意一点都会被映射到屏幕空间上的同一点。这一性质为我们实现透视投影提供了线索。

在正交投影中，Z 轴方向的直线上的点映射到同一屏幕上的同一点。

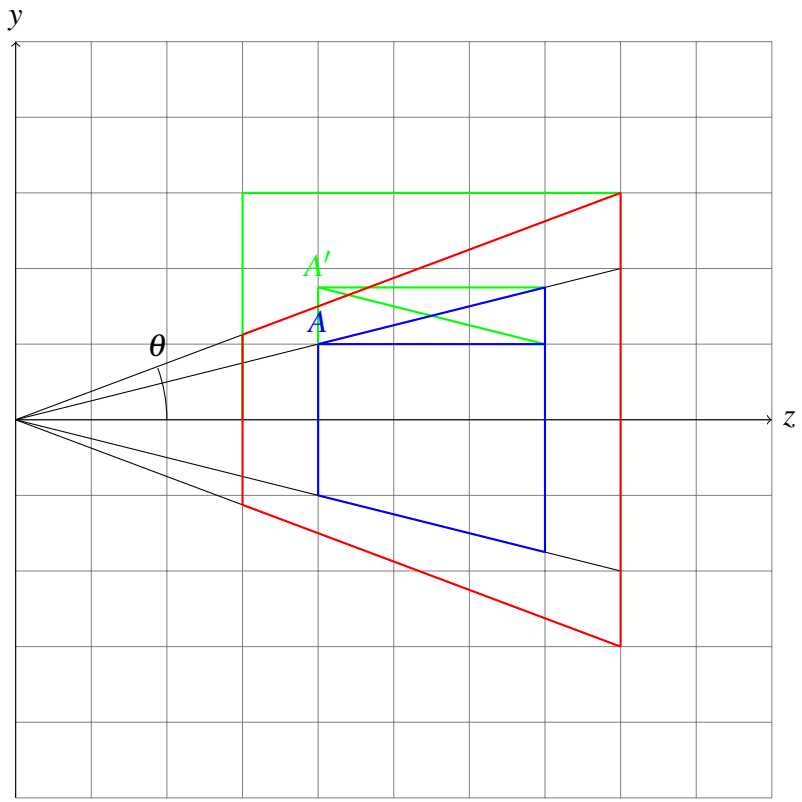


图 5: x 分量，与 l,r 的映射关系

在上图中，红线的是视域，蓝线是三维中的点组成的形体。现在在近平面上，我们将它缩放到和远平面一样大，此时近平面显示为绿色。类似的对于每个蓝色的点，将它所处的那个视域 Z 平面都缩放到和远平面一样大，显示为绿色。

可以看到处于与原点连线同一直线上的点在操作后被映射到了同一 Z 方向的直线上，而原本在同一 Z 方向的直线上的点在操作后则变为一根近大远小的绿线。可见只要对每个点做上述操作后在进行正交投影，就是我们需要的透视投影的结果。

以 A 为例，在 y 份量上的映射关系为：

$$\tan(\theta) = \frac{A_y}{A_z}$$

A 所在 Z 平面的上端坐标 Z_y:

$$Z_y = A_z \tan(\theta)$$

远平面的上端坐标 F_y:

$$F_y = f \tan(\theta)$$

易得：

$$\frac{A'_y}{F_y} = \frac{A_y}{Z_y}$$

则有：

$$\begin{aligned} A'_y &= \frac{A_y}{Z_y} F_y \\ &= \frac{A_y}{A_z \tan(\theta)} f \tan(\theta) \\ &= f \frac{A_y}{A_z} \end{aligned}$$

同理有：

$$\begin{aligned} A'_x &= \frac{A_x}{Z_x} F_x \\ &= \frac{A_x}{A_z \tan(\beta)} f \tan(\beta) \\ &= f \frac{A_x}{A_z} \end{aligned}$$

最后完整的映射为：

$$\begin{cases} x' = f\frac{x}{z} \\ y' = f\frac{y}{z} \\ z' = fz \end{cases} \rightarrow \begin{cases} x'z = fx \\ y'z = fy \\ z' = z \end{cases}$$

尝试将上述的映射变为矩阵形式：

$$\mathbf{A}' = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}_{\textbf{Persp}\rightarrow\textbf{Orth}} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x'z \\ y'z \\ z' \\ z \end{bmatrix} \tag{10}$$

将该矩阵与正交投影矩阵相乘：

$$\begin{aligned}\textbf{Perspective} &= \begin{bmatrix} \frac{2}{w} & 0 & 0 & 0 \\ 0 & \frac{2}{h} & 0 & 0 \\ 0 & 0 & \frac{1}{f-n} & -\frac{n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \\ &= \begin{bmatrix} \frac{2f}{w} & 0 & 0 & 0 \\ 0 & \frac{2f}{h} & 0 & 0 \\ 0 & 0 & \frac{1}{f-n} & -\frac{n}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix}\end{aligned}$$

注意上述矩阵中的 w 和 h 是远平面的宽高。实际上 $\frac{2f}{h}$ 即为图 (5) 中的 $\cot(\theta)$ ，常用 $fov = 2\theta$ ， $aspect = \frac{w}{h}$ ，则透视投影矩阵变为：

$$\textbf{Perspective} = \begin{bmatrix} \frac{\cot(\frac{\theta}{2})}{aspect} & 0 & 0 & 0 \\ 0 & \cot(\frac{\theta}{2}) & 0 & 0 \\ 0 & 0 & \frac{1}{f-n} & -\frac{n}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

最后，注意到公式 (10) 中的矩阵 $\textbf{Persp} \to \textbf{Orth}$ 的变换结果为 x'_z, y'_z ，因此在计算完 $\mathbf{N} = \textbf{Perspective} \times \mathbf{P}$ 后， $\frac{N_x}{N_w}, \frac{N_y}{N_w}$ 才是最终的结果。

3 相机视口变换

3.1 世界坐标系

在第二章中讲到的两种投影变换都是按照相机位于三维空间原点，朝向 Z 轴方向，Y 轴向上，X 轴向右的情况下得到的。但在实际的实现中，我们往往希望相机也能作为一个三维物体进行移动、旋转等变换。这就使得我们不能直接按照三维点关于原三维空间的坐标进行投影变换。

3.2 视口坐标系

换而言之，需要进行投影操作的点的坐标实际上是该点在以相机为原点，按相机朝向旋转后的基向量组成的坐标系，这里称为“视口坐标系”。我了能够正确的实现从相机获得的画面，我们需要对点的世界坐标进行变换。

3.3 坐标系变换

可知相机有旋转和平移变换，将这一变换记作 $\mathbf{C}$ ，则有视口坐标系基向量组成的矩阵 B:

$$\mathbf{CI} = \mathbf{B}$$

$$\mathbf{C} = \mathbf{B}$$

而对于三维世界坐标系中的一点 $\mathbf{P}$ ，其视口坐标为 $\mathbf{P}'$ ，则有：

$$\mathbf{BP}' = \mathbf{P}$$

两边同左乘 $\mathbf{B}^{-1}$

$$\begin{aligned} \mathbf{B}^{-1}\mathbf{BP}' &= \mathbf{B}^{-1}\mathbf{P} \\ \mathbf{P}' &= \mathbf{B}^{-1}\mathbf{P} \\ &= \mathbf{C}^{-1}\mathbf{P} \end{aligned} \tag{11}$$

其中称式 (11) 中的 $\mathbf{C}^{-1}$ 为视口变换矩阵。

综上，在将点进行投影之前，首先要计算 $\mathbf{P}' = \mathbf{C}^{-1}\mathbf{P}$ ，后将 $\mathbf{P}'$ 作为需要变换的点即可。矩阵的逆的算法不是本次项目的重点，这里不做介绍，可见 *sunMatLib.h* 文件。

4 绘制/光栅化

4.1 三角填充

标准的三角填充算法主要考虑了底部平行 X 轴和顶部平行 X 轴这两种情况，如图所示:

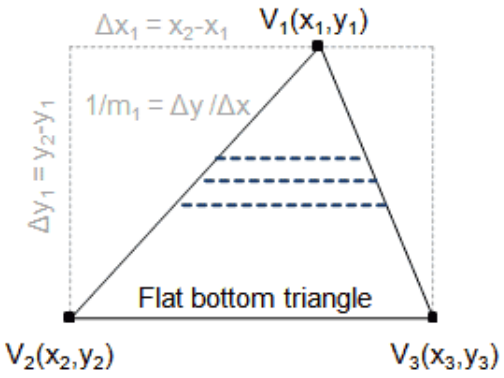


图 6: 底部平行 X 轴

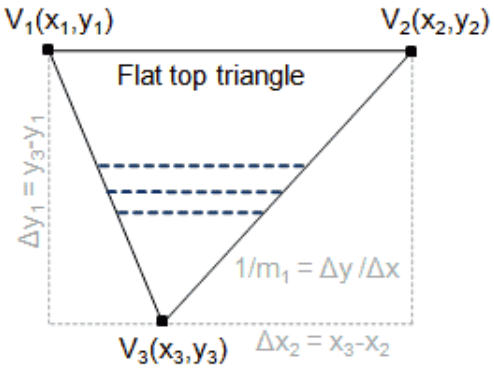


图 7: 顶部平行 X 轴

如图 (6) 所示，我们从 $V1$ 开始平行于底部的 $V2V3$ 开始绘制一条条平行线，他们的交点分别为 $p1$ ， $p2$ ， 显然 $p1.y = p2.y$

同理，$V1.y - p1.y = V1.y - p2.y$, 从 $V1$ 开始一条一条往下绘制直线来填充整个三角形。

我们将 $\triangle y$ 设置为 1，那么 $\triangle x = \frac{\triangle y}{k}$ k 为 $V1V2$ 与 $V1V3$ 的斜率。

那么我们可以得到如下公式

$$K_{V1V2} = \frac{V2.y - V1.y}{V2.x - V1.x}$$

$$K_{V1V3} = \frac{V3.y - V1.y}{V2.x - V1.x}$$

$$\triangle x_{V1V2} = \frac{\triangle y}{K_{V1V2}}$$

$$\triangle x_{V1V3} = \frac{\triangle y}{K_{V1V3}}$$

其代码大致如下:

```
for(int y = v1; y<V2.y,y++)
{
    drawline(V2.x,y,V3.x,y)
    V2.x+=dx
    V3.x+=dx
}
```

对于图 (7) 的处理方法，与图 1 的处理方法是一样的，只是图二是由下向上绘制一条条的平行线，在此不多赘述。

有了这两个三角形的绘制基础，接下来我们考虑一般三角形的情况

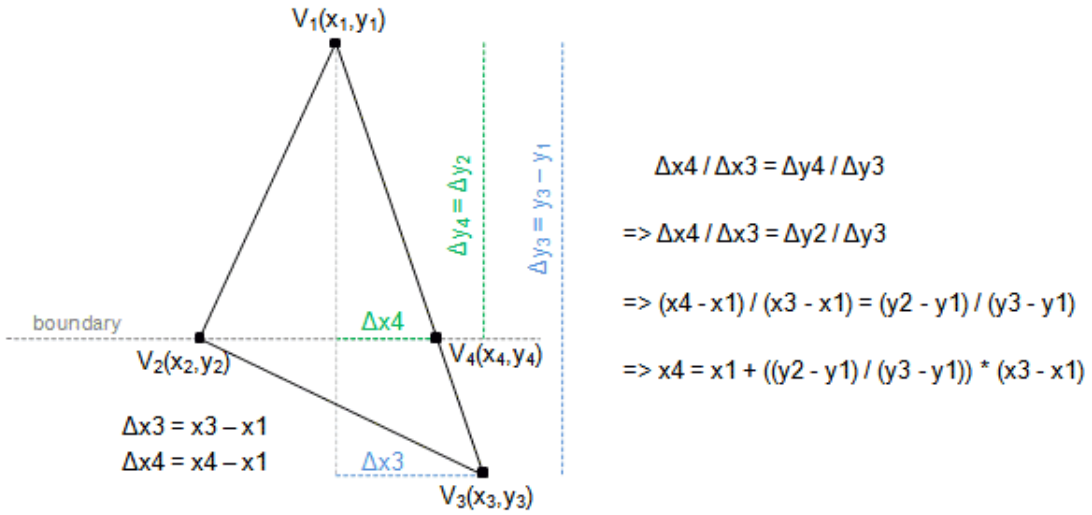


图 8: 一般三角形

如图 (8)，我们基本的思路是将该三角形分解为上面类型一样的两个三角形，一个平底的，一个平顶的。这样我们就能将一个新问题转化为已经解决了的问题。现在的问题关键点是我们应该如何切分这个三角形。

在图 3 中，我们可以穿过 $V2$ 沿着 X 轴构建一条直线，假设这条线的与三角形的交点为 $V4$ 并且 $V4.y = V2.y$ $V4.x$ 也可以通过计算得出。最终我们获得了两个三角形 $\triangle 1 = (V1, V2, V4), \triangle 2 = (V2, V4, V3)$

最终代码如下:

```
1  drawTriangle()
2  {
3      sorty();          //将三个顶点按y值大小排序，确定顶部点
4      if(v2.y ==v3.y)           //平底三角形
5      {
6          fillBottomFlatTriangle(v1,v2,v3);
7      }
8      else if(v1.y == v2.y)    //平顶三角形
9      {
```

```
10          fillTopFlatTriangle(v1,v2,v3)
11      }
12      else                        //一般情况
13      {
14          Vertice v4 = new Vertice
15          fillBottomFlatTriangle(v1,v2,v4)
16          fillTopFlatTriangle(v2,v4,v3)
17      }
18  }
```

4.2 绘制顺序问题

我们要做的是将一个三维的物体投影至二维平面。那么为了让三维的物体能够准确的显示，我们必须确定哪些多边形是可见的，哪些多边形是不可见的。当我们直视一个三维的物体时，其背面我们是无法看见的，因此投影时，背面的多边形是不可见的。因此，绘制时应该由远及近得绘制多边形，这样近的多边形会将远的多边形所覆盖从而变得不可见。

4.3 画家算法

“画家算法”表示头脑简单的画家首先绘制距离较远的场景，然后用绘专制距离较属近的场景覆盖较远的部分。画家算法首先将场景中的多边形根据深度进行排序，然后按照顺序进行描绘。这种方法通常会将不可见的部分覆盖，这样就可以解决可见性问题。

其算法的大致思路如下：

1. 将屏幕设成背景色
2. 把要画的物体（多边形）按其离开视点的从远到近排序由此构成深度优先级表。然后从远到近画物体（多边形），近的就因为优先级高而覆盖远的多边形。由此可消隐。

伪代码如下:

```
sort(polygon.distance)              //对多边形的距离行排序
for(int i = n; i > 0; i--)
{
    draw_polygon(polygon[i].distance);  //由远及近画多边形
}
```

4.3.1 画家算法的问题

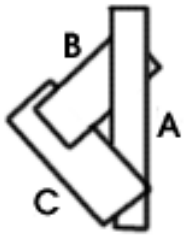


图 9: 画家算法的问题

在有些场合下，画家算法可能无法解决可见性问题。如图 (9)，在这个例子中，多边形 A、B、C 互相重叠，我们无法确定哪一个多边形在上面，哪一个在下面，我们也无法确定两个多边形什么时候在三维空间中交叉。

一些基本的画家算法实现方法也可能效率很低，因为这将使得系统将可见多边形集合中的每个点都进行渲染，而没有考虑这些多边形在最终场景中可能被其它部分遮挡。这也就是说，对于细致的场景来说，画家算法可能会过度地消耗计算机资源。

4.3.2 改进方向，$Z-buffer$ 算法

深度缓存（$Z-buffer$）算法是一种典型的、简单的图象空间面消隐算法。该算法需要一个深度缓存数组 ZB，此外还需要一个颜色属性数组 CB，它们的大小与屏幕上像素点的个数相同。$Z-buffer$ 算法的步骤如下：

1. 初始化 ZB 和 CB，使得 $ZB(i,j)=Zmax, CB(i,j)=backgroundcolor, i=1,\cdots,m; j=1,\cdots,n$ 。
2. 对多边形 P，计算它在点 (i,j) 处的深度值 $z(i,j)$ 。

3. 若 $z(i,j) < ZB(i,j)$ ，则 $ZB(i,j) = z(i,j), CB(i,j) = colorofP$。

4. 对每个多边形重复（2）、（3）两步，最终在 CB 中存放的就是消隐后的图形。这个算法的关键在第（2）步，要尽快判断出哪些点落在一个多边形内，并尽快求出一个点的深度值。这里需要应用多边形点与点之间的相关性，包括水平相关性和垂直相关性。

5 三维裁剪

5.1 性能问题与原因

在上述的功能实现后，程序已经能够正确地绘制三维网格模型了，但在我们将摄像机靠近模型时，我们会发现性能的快速下降。在十分靠近模型的时候还会出现“卡死”的情况。

该问题的原因十分简单，虽然在投影一章中我们设置了远近两个平面，但在这两个平面以外的点实际上只是 z 分量不在 $[0,1]$ 的区间中，他们仍会参与绘制。在正交投影中还不能看出这有什么问题，但在透视投影中，在视口坐标系 Z 轴正半轴区域内的点，越靠近原点，由于近大远小的原因，投影后的坐标在屏幕坐标系中将变得很大，此时三角面的光栅化将难以运行。

因此我们只希望渲染近平面以外的视域中的三角面（不考虑远平面），但在光栅化的过程中时以三角面为单位，单纯判断整个三角面是否在视域外无法解决上述的性能问题，而单纯判断三角面是否有顶点在视域外又会造成整个三角面不被渲染，从而导致模型“破损”，因此我们需要对三角面进行裁剪，对于每个面只保留在视域中的部分。

5.2 三角面裁剪

在三维空间中对三角面的裁剪如下图 (10)，是一个三角面关于一张三维空间中平面的裁剪。

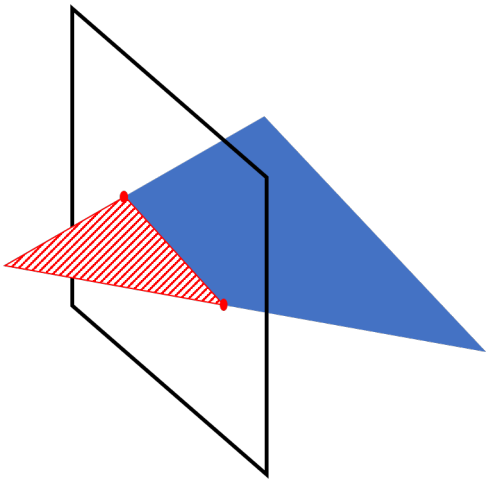


图 10: 三角面裁剪

以平面法向的正方向表示要保留的区域，则会出现四种情况，如图 (11)：

1. 整个三角面都在保留区内
2. 整个三角面都在保留区外
3. 有 1 个顶点在保留区外，此时要保留的部分是 2 个三角面
4. 有 2 个顶点在保留区外，此时要保留的部分是 1 个三角面

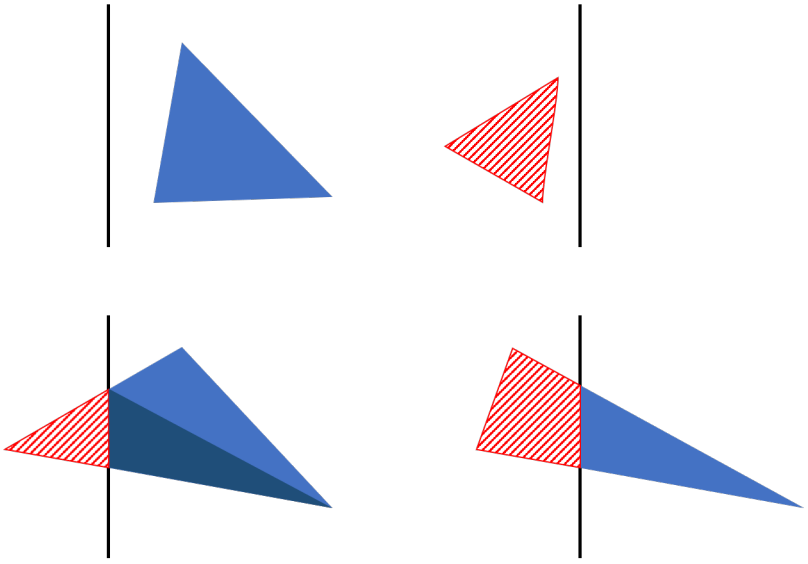


图 11: 三角面裁剪

首先引入“有向距离”$distance(\mathbf{P_n})$ 是空间中一点 $\mathbf{P_n}$ 到一张平面的垂直距离，在保留区内为正，否则为负。

显然，$distance < 0$ 的点在保留区外。对于三角面的每个顶点 $\mathbf{P_n}$（按顺时针顺序处理），当 $distance(\mathbf{P_n}) < 0$，将 $\mathbf{P_n}$ 放入 *outside* 数组，否则放入 *inside* 数组。

当 *inside* 数组长度为 3 时对应情况 1，返回原三角面。

当 *inside* 数组长度为 0 时对应情况 2，返回空。

当 *inside* 数组长度为 1，*outside* 数组长度为 2 时对应情况 4，此时计算 $outside[0]$ 与 $inside[0]$ 连线与裁剪面的交点 N_0，计算 $outside[1]$ 与 $inside[0]$ 连线与裁剪面的交点 N_1。显然 $outside[0], outside[1], inside[0]$ 是逆时针排列的，则 $N_0, N_1, inside[0]$ 也是逆时针排列的，即裁剪后的新三角面 $TriFace(N_0, N_1, inside[0])$ 保持了原三角面的朝向。

当 *inside* 数组长度为 2，*outside* 数组长度为 1 时对应情况 3，此时计算 $outside[0]$ 与 $inside[0]$ 连线与裁剪面的交点 N_0，计算 $outside[0]$ 与 $inside[1]$ 连线与裁剪面的交点 N_1。显然 $inside[0], inside[1], outside[0]$ 是逆时针排列的，则 $inside[0], inside[1], N_0$ 也是逆时针排列的，即裁剪后的新三角面 $TriFace(inside[0], inside[1], N_0)$ 保持了原三角面的朝向。而 $inside[1], N_0, N_1$ 也是逆时针排列的，即裁剪后的第二个三角面 $TriFace(inside[0], N_0, N_1)$ 也保持了原三角面的朝向。

sunMatLib.h 中的 *Plane* 类实现了上述裁剪算法，代码如下：

```
1  int clip_TriFace(TriFace &in, TriFace &out1, TriFace &
       out2) const
2  {
3      Vector3 *inside_point[3];
4      int inside_count = 0;
5      Vector3 *outside_point[3];
6      int outside_count = 0;
7      double d0 = distance(in.get_Point1());
8      double d1 = distance(in.get_Point2());
9      double d2 = distance(in.get_Point3());
10     if (d0 >= 0)
11     {
12         inside_point[inside_count++] = &in.get_Point1();
13     }
14     else
15     {
```

```
16          outside_point[outside_count++] = &in.get_Point1()
                ;
17      }
18      if (d1 >= 0)
19      {
20          inside_point[inside_count++] = &in.get_Point2();
21      }
22      else
23      {
24          outside_point[outside_count++] = &in.get_Point2()
                ;
25      }
26      if (d2 >= 0)
27      {
28          inside_point[inside_count++] = &in.get_Point3();
29      }
30      else
31      {
32          outside_point[outside_count++] = &in.get_Point3()
                ;
33      }
34      if (inside_count == 0)
35      {
36          // No remain triface
37          return 0;
38      }
39      if (inside_count == 3)
40      {
41          // origin triface
42          out1 = in;
43          return 1;
44      }
45      if (inside_count == 1 && outside_count == 2)
```

```
46      {
47          // one remain triface
48          Vector3 p2 = intersect_Line(*inside_point[0], *
               outside_point[0]);
49          Vector3 p3 = intersect_Line(*inside_point[0], *
               outside_point[1]);
50          out1.set(*inside_point[0], p2, p3);
51          return 1;
52      }
53      if (inside_count == 2 && outside_count == 1)
54      {
55          // two new triface
56          Vector3 p12 = intersect_Line(*inside_point[0], *
               outside_point[0]);
57          out1.set(*inside_point[0], *inside_point[1], p12)
               ;
58          Vector3 p22 = intersect_Line(*inside_point[1], *
               outside_point[0]);
59          out2.set(*inside_point[1], p12, p22);
60          return 2;
61      }
62  }
```

5.2.1 近平面裁剪

在有了上述的平面裁剪算法后实现近平面算法实现就很简单了。首先实例化一个 *Plane* 类，其坐标为 $(0,0,n)$ ，法向为 Z 轴方向 $(0,0,1)$ 。在一个三角面执行了视口变换后，将变换后的三角面执行关于近平面的裁剪算法，将得到的结果放入投影数组即可。

5.2.2 视口裁剪

视口裁剪在投影前和投影后都可以实现。投影前的裁剪和近平面裁剪类似，将近平面裁剪的结果放入裁剪队列，然后对于上下左右四个视域侧面的裁剪面，进行

裁剪操作，每次都将上次的裁剪结果作为带裁剪数据。从而实现视域的裁剪。这种裁剪由于视域是一个锥体，所以上下左右裁剪面的法向较难计算。

在投影后相当于在正交投影的视域内裁剪，步骤和上述裁剪一致，只是上下左右裁剪面是固定值，计算较为便利。

在我们的程序使用后者进行视口裁剪：

```
1  TriFace clipped[2];
2  int new_count = 1;
3  std::list<TriFace> trifacelist;
4  trifacelist.push_back(face);
5  for (int p = 0; p < 4; p++)
6  {
7      int new_tri = 0;
8      while (new_count > 0)
9      {
10         TriFace test = trifacelist.front();
11         trifacelist.pop_front();
12         new_count--;
13         if (p == 0) //top
14         {
15             const Vector3 pos(0, -1, 0);
16             const Vector3 normal(0, 1, 0);
17             Plane clip_plane(pos, normal);
18             new_tri = clip_plane.clip_TriFace(test,
                   clipped[0],clipped[1]);
19         }
20         else if (p == 1) //bottom
21         {
22             const Vector3 pos(0, 1, 0);
23             const Vector3 normal(0, -1, 0);
24             Plane clip_plane(pos, normal);
25             new_tri = clip_plane.clip_TriFace(test,
                   clipped[0],clipped[1]);
```

```
26          }
27          else if (p == 2) //left
28          {
29              const Vector3 pos(-1, 0, 0);
30              const Vector3 normal(1, 0, 0);
31              Plane clip_plane(pos, normal);
32              new_tri = clip_plane.clip_TriFace(test,
                    clipped[0],clipped[1]);
33          }
34          else if (p == 3) //right
35          {
36              const Vector3 pos(1, 0, 0);
37              const Vector3 normal(-1, 0, 0);
38              Plane clip_plane(pos, normal);
39              new_tri = clip_plane.clip_TriFace(test,
                    clipped[0],clipped[1]);
40          }
41          for (int i = 0; i < new_tri; i++)
42          {
43              trifacelist.push_back(clipped[i]);
44          }
45      }
46      new_count = trifacelist.size();
47  }
```

6 基础光照

6.1 环境光

环境光是指光源间接对物体的影响，是在物体和环境之间多次反射，最终平衡时的一种光。我们近似的认为同一环境的环境光，其光强分布是均匀的，他在任何一个方向上的分布都相同，在简单的光照模型中，我们用一个常数来模拟环境光，

用式子表达如下：

$$I_e = I_a \cdot K_a$$

其中，I_a 为环境光光强，K_a 为反射系数。

而如果环境光是彩色光源，则环境光用矩阵表示如下

$$I_a = \begin{bmatrix} I_{ar} \\ I_{ag} \\ I_{ab} \end{bmatrix}$$

6.2 方向光

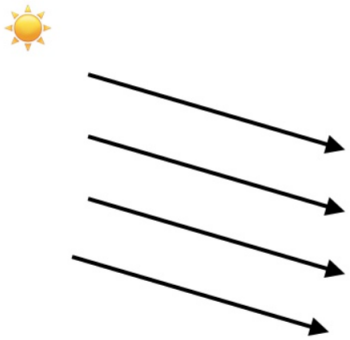
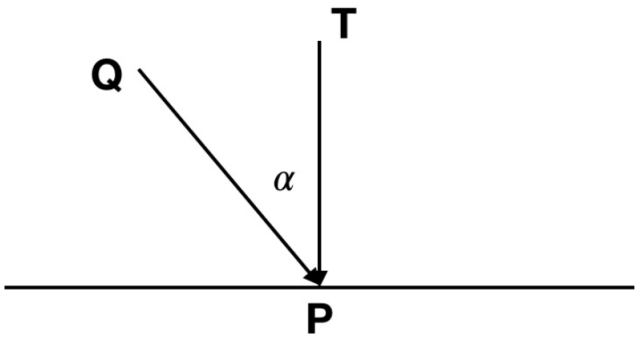


图 12: 平行光

方向光又称为平行光，他是一组没有衰减的平行光线，类似于太阳光的效果。在方向光中，我们需要知道的是方向 $\vec{L}$ 与光强 I_a



利用方向光可以实现对物体的漫反射。当方向光射至物体时，到达物体表面的光实际上只有 $\vec{QP}$ 在 $\vec{TP}$ 上的投影，即 $\vec{QP} \cdot \cos\alpha$

在此仅考虑 $\cos\alpha$ 大于零的情况，如果小于零，则说明方向光射至物体的背面。

我们可以得到到如下公式：

$$\cos\alpha = \frac{\vec{QP}\cdot\vec{TP}}{\left|\vec{QP}\right|\cdot\left|\vec{TP}\right|}$$

得到了 $\cos\alpha$ 以后，便可以通过方向光的各个投影的叠加实现漫反射，最后加上环境光的常量，就得到了最终的效果。

$$I = I_A + \sum_{i=1}^{n} I_i \frac{\vec{QP_i}\cdot\vec{TP_i}}{\left|\vec{QP_i}\right|\cdot\left|\vec{TP_i}\right|}$$

给出漫反射的伪代码

```
for(int i = 0; i < n; i++)
{
    ca = caculate(P[i].N,L.N);
    //计算物体法向量与方向光的余弦
    if(ca<0)              //小于0，在背面，舍弃
    {
        continue;
    }
    else
    {
        P[i].L=Ia+L*ca
        //最终光照是方向光的投影与环境光相加的结果
    }
}
```