

# 计算机图形学第一次作业报告

姓名：江金昱 学号：17341067

## 0. 前言

文件名	文件描述
myfunction.h	实现了报告中提到的函数  诸如平移函数，旋转函数，矩阵乘法，直线 绘制算法等
myglwidget.cpp	在 scene_1 中实现了使用自己的方法绘制了  scene_0 同样的场景

## 1. 平移变换函数 Translate 原理：

代码位置：myfunction.h 的 myTranslatef

opengl 内置的函数 glTranslate 接受三个参数，分别表示沿着 x 轴，y 轴，z 轴平移的距离。我使用了**矩阵乘法**实现了自己的平移变换函数 **myTranslatef(GLfloat x,GLfloat y,GLfloat z)**。

在 opengl 中维护一个 modelview 矩阵，用以实现对屏幕中的物体的各种变换操作，如空间变换（平移、旋转等）。modelview 矩阵是一个 4X4 的矩阵，是齐次坐标的表现形式。假设我们有一个点  $\vec{x} = (x,y,z)$ 需要进行平移变换，我们用齐次坐标表示(x, y, z, 1)表示，我们想要将其沿着 x, y, z 轴分别平移 $d_x, d_y, d_z$ 个大小。那么我们需要构造一个平移变换矩阵 T

$$T = T(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

我们求 $T \cdot \vec{x}$ 可得结果：

$$\begin{bmatrix} x + d_x \\ y + d_y \\ z + d_z \\ 1 \end{bmatrix}$$

可以看到已经平移成功

在 opengl 中，使用 `glGetFloatv(GL_MODELVIEW_MATRIX,mat)` 可从 modelview 栈中获取栈顶矩阵，并保存在一维大小为 16 的数组 mat 中，且 mat 是以 **column-major** 的形式保存的。我们将 mat 右乘平移变换矩阵 T，即  $mat \cdot T$  然后再使用 `glLoadMatrixf` 将结果矩阵放入栈顶即可完成

## 2. 矩阵乘法的实现

代码位置：myfunction.h 里的 myMultiMatrix

由于 opengl 里面是以 column-major 的方法存储矩阵的，也就是说，假如我们有矩阵：

$$A = \begin{bmatrix} a & e & i & m \\ b & f & j & n \\ c & g & k & o \\ d & h & l & p \end{bmatrix}$$

那么我们使用函数 `glGetFloatv` 取到并存储在一维数组 mat 里的顺序将会是这样的：

$$mat = [a \ b \ c \ d \ e \ f \ g \ h \ i \ j \ k \ l \ m \ n \ o \ p]$$

因此我们最后使用 `glLoadMatrix` 上传的时候也要注意是以 columnmajor 的形式上传，代码实现如下：

```
/* 函数：myMultiMatrix
 * 函数描述：实现将 mat 矩阵右乘变换矩阵 T
 * 参数描述：
 * result：结果矩阵
 * mat：栈顶矩阵
 * T：变换矩阵
 */
void myMultiMatrix(GLfloat* result, GLfloat* mat, GLfloat* T) {
    for(int i=0; i<16; i++) {
        int row=i%4;
        int col=i/4;

        result[i]=0;
        for(int j=0; j<4; j++) {
            result[i]+=mat[row+j*4]*T[col*4+j];
        }
    }
}
```

## 3. 旋转变换函数的原理

### (1) 使用矩阵乘法实现旋转变换

代码位置: myfunction.h 里的 myTranslatef

基本思想: 给定一个旋转轴  $\vec{n} = (a, b, c)$  (这里  $\vec{n}$  需要为单位向量) 我们可以将  $x, y, z$  轴的基向量绕着该轴旋转, 得到新的一组基向量, 然后计算在新的基向量下点的坐标。

比如基向量  $\vec{v} = [1\ 0\ 0]$  绕  $\vec{n}$  进行旋转  $\theta^\circ$ , 可得旋转后的基向量:

$$\begin{bmatrix} a^2(1 - \cos\theta) + \cos\theta \\ ab(1 - \cos\theta) + c \cdot \sin\theta \\ ac(1 - \cos\theta) - b \cdot \sin\theta \end{bmatrix}$$

由此可得  $[0\ 1\ 0]$  和  $[0\ 0\ 1]$  旋转后的基向量, 组装成矩阵可得

$$T = \begin{bmatrix} a^2(1 - \cos\theta) + \cos\theta & ab(1 - \cos\theta) - c \cdot \sin\theta & ac(1 - \cos\theta) + b \cdot \sin\theta \\ ab(1 - \cos\theta) + c \cdot \sin\theta & b^2(1 - \cos\theta) + \cos\theta & bc(1 - \cos\theta) - a \cdot \sin\theta \\ ac(1 - \cos\theta) - b \cdot \sin\theta & bc(1 - \cos\theta) + a \cdot \sin\theta & c^2(1 - \cos\theta) + \cos\theta \end{bmatrix}$$

### (2) 使用四元数实现旋转变换

代码位置: myfunction.h 里的 quaternionRotate

四元数是指这样的数:

$$Q = xI + yJ + zK + w = [\vec{v}, w]$$

给定旋转轴  $\vec{n}$ , 需要旋转的角度为  $2\theta$ , 构造四元数

$$Q = [\vec{n}\sin\theta, \cos\theta]$$

于是可求得对应的  $x, y, z, w$ , 那么我们有对应的旋转矩阵:

$$\begin{bmatrix} 1 - 2y^2 - 2z^2 & 2(xy - wz) & 2(xz + wy) \\ 2(xy + wz) & 1 - 2x^2 - 2z^2 & 2(yz - wx) \\ 2(xz - wy) & 2(yz + wx) & 1 - 2x^2 - 2y^2 \end{bmatrix}$$

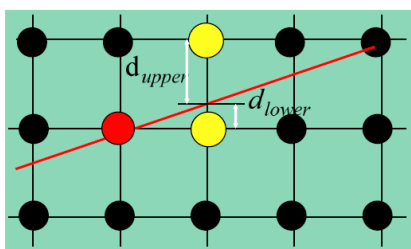
右乘栈顶矩阵即可

## 4. 画线算法

给定两个坐标  $(x_0, y_0)$  以及  $(x_1, y_1)$  我们目标是要在这两点间画一条直线

代码位置: myfunction.h 里的 drawline

使用了 **Bresenham 算法**



如图，基本思想是，对于每一个 $(x_i + 1, \overline{y_{i+1}})$ ，假如 $d_{lower} < d_{upper}$ ，那么 $\overline{y_{i+1}} = \overline{y_i}$ 否

则 $\overline{y_{i+1}} = \overline{y_i} + 1$

记 $p_i = \Delta x \cdot (d_{lower_i} - d_{upper_i})$

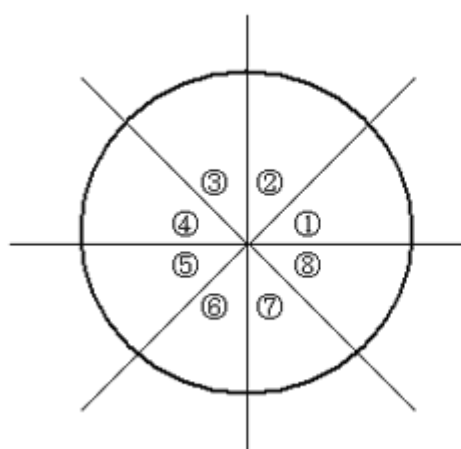
可通过公式 $p_{i+1} - p_i = 2\Delta y - 2\Delta x \cdot (\overline{y_{i+1}} - \overline{y_i})$ 迭代计算 $p_i$

当 $p_i \leq 0$ 时，可知 $d_{lower} \leq d_{upper}$ ，取下方像素，因此有 $p_{i+1} = p_i + 2\Delta y$

当 $p_i > 0$ 时，可知 $d_{lower} > d_{upper}$ ，取上方像素，因此 $p_{i+1} = p_i + 2\Delta y - 2\Delta x$

迭代计算以上步骤即可得到 $(x_0, y_0)$ 与 $(x_1, y_1)$ 之间的直线

上述算法只阐述了从左下角画到右上角的直线算法，为了能够使得算法适用下图八个方向的直线绘制



- ①④:  $x_{i+1}=x_i+1$ 、 $x_i-1$   
 $y_{i+1}=y_i$  或  $y_i+1$
- ⑧⑤:  $x_{i+1}=x_i+1$ 、 $x_i-1$   
 $y_{i+1}=y_i$  或  $y_i-1$
- ②⑦:  $y_{i+1}=y_i+1$ 、 $y_i-1$   
 $x_{i+1}=x_i$  或  $x_i+1$
- ③⑥:  $y_{i+1}=y_i+1$ 、 $y_i-1$   
 $x_{i+1}=x_i$  或  $x_i-1$

需要对算法进行微小的改动。

其中方向①⑧⑤④可分为一组，上述算法仍然适用，除了要将 $x$ 和 $y$ 递增/递减的方向更改一下，而且 $\Delta x = |x_1 - x_0|$ ， $\Delta y = |y_1 - y_0|$ 。

②③⑥⑦分为一组，此时需要将算法中的 $\Delta x$ 与 $\Delta y$ 进行对换

算法的具体实现见 myfunction.h 里的 drawline

## 5. 多个变换之间的顺序关系对结果的影响

opengl 将“最后写的函数最先应用”，比如你先写了 translate 再写了 rotate，其实最终是先进行 rotate 再进行 translate

## 6. 解决显示过于稀疏的问题

由于 Bresenham 算法是基于整数进行画线，因此为了让显示不过于稀疏，需要调整 `glOrtho` 中的宽和高，使得显示在窗口中的像素密度变大，同时也要等比缩放 `glTranslate`，以及画线的位置的 `x`, `y`，即乘上放大的比例。

比如，`scene0` 中使用

```
glOrtho(0.0f, 100.0f, 0.0f, 100.0f, -1000.0f, 1000.0f);
```

`scene1` 中使用

```
glOrtho(0.0f, 5000.0f, 0.0f, 5000.0f, -1000.0f, 1000.0f);
```

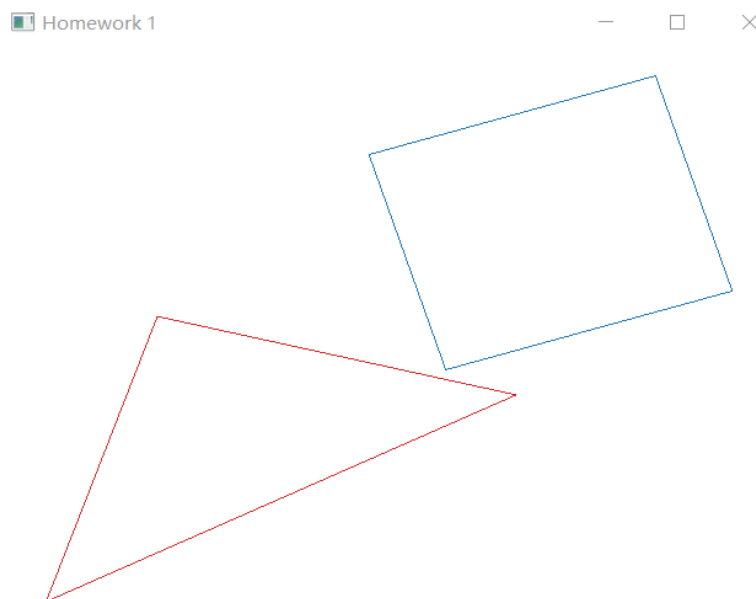
则在 `glTranslate` 中宽和高要分别乘上一个系数：

```
GLfloat w_ratio=5000.0f/100.0f;
```


```
GLfloat h_ratio=5000.0f/100.0f;
```

## 7. 运行结果截图

**scene0:**



scene1:

 Homework 1

— □ ×

