**附件一：**

**作业报告**

**三维图形变换**

2022年1月14日

# 程序功能及操作简介

**三维图形变换程序命名为3D\_Trans，操作界面为GUI图形显示界面，运行环境为macOS系统，主要操作对象为3D线框立方体。本程序执行后，默认显示3D线框立方体为斜二测投影，可在斜二测投影、正投影和斜等轴投影三个选项间切换，使用键盘方向键可以控制3D线框立方体沿着坐标轴三个方向平移，使用鼠标可以控制3D线框立方体绕着三个坐标轴旋转。**

**本程序主要使用C++语言编写，主要通过加载OpenGL库函数实现图形绘制与窗口显示功能。**

# 核心代码及运行效果截图

## 三维Brehansam直线代码

void setPixel(GLint x, GLint y,GLint z)//在屏幕上画一个点

{

    glPointSize(3.0f);

    glColor3f(1, 0, 0);//设置画点的颜色

    glBegin(GL\_POINTS);

    glVertex3i(x, y,z);

    glEnd();

}

void BresenhamLine(int x0, int y0, int z0,int x1, int y1,int z1,int plane)

{

    //glClear(GL\_COLOR\_BUFFER\_BIT);

 //清除窗口显示内容

    int xx, yy,zz,dxx, dyy,xx11,yy11,e,txx,tyy;

    switch (plane)

    {

        case 1://平面xy中

        {

            xx = x0;

            yy = y0;

            zz = z0;//z坐标不变

            xx11=x1;

            yy11=y1;

            dxx = x1 - x0;

            dyy = y1 - y0;

            //确定步长tx，ty，斜率为正或为负，且只考虑左/右上or左/右下，左/右边的情况不加步长即可

            txx =((x1>x0)?1:(dxx = -dxx, -1));

            tyy =((y1>y0)?1:(dyy = -dyy, -1));//!!!!!!!

        }

            break;

        case 2://平面xz中

        {

            xx = x0;

            yy = z0;

            zz = y0;//y坐标不变

            xx11=x1;

            yy11=z1;

            dxx = x1 - x0;

            dyy = z1 - z0;

            //确定步长tx，ty，斜率为正或为负，且只考虑左/右上or左/右下，左/右边的情况不加步长即可

            txx =((x1>x0)?1:(dxx = -dxx, -1));

            tyy =((z1>z0)?1:(dyy = -dyy, -1));

        }

            break;

        case 3://平面yz中

        {

            xx = y0;

            yy = z0;

            zz = x0;//y坐标不变

            xx11=y1;

            yy11=z1;

            dxx = y1 - y0;

            dyy = z1 - z0;

            //确定步长tx，ty，斜率为正或为负，且只考虑左/右上or左/右下，左/右边的情况不加步长即可

            txx =((y1>y0)?1:(dxx = -dxx, -1));

            tyy =((z1>z0)?1:(dyy = -dyy, -1));//!!!!!!!

        }

            break;

    }

    int flag = 0;  //斜率绝对值标志，0表示斜率绝对值小于1

    if (abs(dyy) > abs(dxx))//斜率绝对值大于1

    {

        flag = 1;

    }

    //glBegin(GL\_POINTS);

    //glVertex3i(xx, yy,zz);

    if (flag == 0) {

        e = -dxx;

        while (xx != xx11)

        {

            xx += txx;

            e += 2 \* dyy;

            if (e >= 0)

            {

                yy += tyy;

                e -= 2 \* dxx;

            }

            //glVertex3i(xx, yy,zz);

            setPixel(xx, yy,zz);

        }

    }

    else {

        e = -dyy;

        while (yy != yy11)

        {

            yy += tyy;

            e += 2 \* dxx;

            if (e >= 0)

            {

                xx += txx;

                e -= 2 \* dyy;

            }

            glVertex3i(xx, yy,zz);

            setPixel(xx, yy,zz);

        }

    }

    //glEnd();

    //glFlush();

}

void DrawLine()//画线回调函数

{

    //glClear(GL\_COLOR\_BUFFER\_BIT);//清除窗口显示内容

    glBegin(GL\_POINTS);

    glColor3f(1, 0, 0);//设置画点的颜色

    BresenhamLine(20, 20, 20,-20,20,20,1);//xy面画线

    BresenhamLine(20, 20, 20,20,-20,20,1);

    BresenhamLine(-20, -20, -20,-20,+20,-20,1);

    BresenhamLine(-20, -20, -20,20,-20,-20,1);

    BresenhamLine(20, 20, -20,20,-20,-20,1);

    BresenhamLine(20, 20, -20,-20,+20,-20,1);

    BresenhamLine(-20, -20, 20,-20,20,20,1);

    BresenhamLine(-20, -20, 20,20,-20,20,1);

    BresenhamLine(20, 20, 20,20,20,-20,2);//xz面画线

    BresenhamLine(-20, 20, 20,-20,20,-20,2);

    BresenhamLine(-20, -20, 20,-20,-20,-20,2);

    BresenhamLine(20, -20, 20,20,-20,-20,2);

    glEnd();

    glFlush();

}

## 三维变换矩阵

 void matrix4x4SetIdentity (Matrix4x4 matIdent4x4)//构建单位矩阵

{

   GLint row, col;

   for (row = 0; row < 4; row++)

      for (col = 0; col < 4; col++)

         matIdent4x4 [row][col] = (row == col);

}

void matrix4x4PreMultiply (Matrix4x4 m1, Matrix4x4 m2)//矩阵计算

{

   GLint row, col;

   Matrix4x4 matTemp;

   for (row = 0; row < 4; row++)

      for (col = 0; col < 4 ; col++)

         matTemp [row][col] = m1 [row][0] \* m2 [0][col] + m1 [row][1] \* m2 [1][col] + m1 [row][2] \* m2 [2][col]+m1 [row][3] \* m2 [3][col];

   for (row = 0; row < 4; row++)

      for (col = 0; col < 4; col++)

         m2 [row][col] = matTemp [row][col];

}

void translate3D (GLfloat tx, GLfloat ty, GLfloat tz)//平移矩阵

{

   Matrix4x4 matTransl;

   //转换矩阵初始化为单位矩阵

   matrix4x4SetIdentity (matTransl);

   matTransl [0][3] = tx;

   matTransl [1][3] = ty;

   matTransl [2][3] = tz;

   //将 matTransl 与复合矩阵连接起来。

   matrix4x4PreMultiply (matTransl, matComposite);

}

void rotate3D (wcPt3D pivotPt, GLfloat theta)//旋转矩阵\*\*\*\*\*

{

   Matrix4x4 matRot;

    //将旋转矩阵初始化为单位矩阵。

   matrix4x4SetIdentity (matRot);

   matRot [0][0] = cos (theta);

   matRot [0][1] = -sin (theta);

   matRot [0][2] = pivotPt.x \* (1 - cos (theta)) +

                        pivotPt.y \* sin (theta);

   matRot [1][0] = sin (theta);

   matRot [1][1] = cos (theta);

   matRot [1][2] = pivotPt.y \* (1 - cos (theta)) -

                        pivotPt.x \* sin (theta);

   //将 matRot 与复合矩阵连接起来。

   matrix4x4PreMultiply (matRot, matComposite);

}

void scale3D (GLfloat sx, GLfloat sy, wcPt3D fixedPt)//缩放矩阵\*\*\*\*\*

{

   Matrix4x4 matScale;

   matrix4x4SetIdentity (matScale);

   matScale [0][0] = sx;

   matScale [0][2] = (1 - sx) \* fixedPt.x;

   matScale [1][1] = sy;

   matScale [1][2] = (1 - sy) \* fixedPt.y;

   matrix4x4PreMultiply (matScale, matComposite);

}

void transformVerts3D (GLint nVerts, wcPt3D \* verts)//使用复合矩阵，计算转换后的坐标。\*\*\*\*\*\*

{

   GLint k;

   GLfloat temp;

   for (k = 0; k < nVerts; k++)

   {

      temp = matComposite [0][0] \* verts [k].x + matComposite [0][1] \*

             verts [k].y + matComposite [0][2];

      verts [k].y = matComposite [1][0] \* verts [k].x + matComposite [1][1] \*

                  verts [k].y + matComposite [1][2];

         verts [k].x = temp;

   }

}

## 计算三维投影

void Calc3DPoint(void)

{

x = (-1)\*x;

xa = cr1\*x - sr1\*z;

za = sr1\*x + cr1\*z;

x = cr2\*xa + sr2\*y;

ya = cr2\*y - sr2\*xa;

z = cr3\*za - sr3\*ya;

y = sr3\*za + cr3\*ya;

x=x+mx; y=y+my; z=z+mz;

sx = d\*x/z;

sy = d\*y/z;

return;

}

// 使用给定的参数逐个元素地构建透视投影矩阵并返回该矩阵。

get\_projection\_matrix(float eye\_fov, float aspect\_ratio,

float zNear, float zFar)

{

Eigen::Matrix4f projection = Eigen::Matrix4f::Identity();

//M persp->ortho,透视投影转换为正交投影

persp2ortho = Eigen::Matrix4f::Identity();

persp2ortho << zNear, 0, 0, 0,

0, zNear, 0, 0,

0, 0, zNear + zFar, -(zNear \* zFar),

0, 0, 1, 0;

//M ortho

Eigen::Matrix4f ortho = Eigen::Matrix4f::Identity();

Eigen::Matrix4f scale = Eigen::Matrix4f::Identity();

Eigen::Matrix4f transformation = Eigen::Matrix4f::Identity();

// t means top, b means bottom,l means left,r means right.

float t = tanf(eye\_fov / 2) \* abs(zNear);

float b = -t;

float r = aspect\_ratio \* t;

float l = -r;

// scale矩阵，将原图形缩放为标准长度为2的立方体

scale << 2 / (r - l), 0, 0, 0,

0, 2 / (t - b), 0, 0,

0, 0, 2 / (zNear - zFar), 0,

0, 0, 0, 1;

// 将立方体的中心移至源点

transformation << 1, 0, 0, -((r + l) / 2),

0, 1, 0, -((t + b) / 2),

0, 0, 1, -((zNear + zFar) / 2),

0, 0, 0, 1;

ortho = scale \* transformation;

projection = ortho \* persp2ortho;

return projection;

}

## 鼠标键盘响应函数

// 鼠标事件

void mouse(int button, int state, int x, int y) {

    if (button == GLUT\_LEFT\_BUTTON && state == GLUT\_DOWN) {

        mouseDown = true;

        xdiff = x - yrot;

        ydiff = -y + xrot;

        std::cout << "xdiff:" << xdiff << "\tydiff" << ydiff << std::endl;

    }

    else

        mouseDown = false;

}

// 鼠标移动事件

void mouseMotion(int x, int y) {

    if (mouseDown) {

        yrot = x - xdiff;

        xrot = y + ydiff;

        std::cout << "yrot:" << yrot << "\txrot" << xrot << std::endl;

        glutPostRedisplay();

    }

}

GLenum fillMode = GL\_SMOOTH;  // Initial polygon fill: color interpolation.

void fillOption (GLint selectedOption)

{

   switch (selectedOption) {

      case 1:  display1();       break;

      case 2:  display2();      break;

       case 3: display3();    break;

   }

   glutPostRedisplay ( );

}

## 菜单函数

//定义主菜单

void Menu(){

    bool opt = true;        //是否循环的一个标志

    while (opt == true) {

        //菜单列表

        std::cout << "\n\t\t\*\*\*\*\*\*\*\*\*\n";

        std::cout << "\t\t\* 1. 斜二测投影  \*\n";

        std::cout << "\t\t\* 2. 正投影     \*\n";

        std::cout << "\t\t\* 3. 斜等轴投影   \*\n";

        std::cout << "\t\t\* 4. 退出       \*\n";

        std::cout << "\t\t\*\*\*\*\*\*\*\*\*\n";

        //接收输入选择

        std::cout << "\t\t选择：";

        char x;

        std::cin >> x;

        //判断用户的选择

        switch (x) {

        case '1':

            std::cout << "\n\t\t你选择了1\n";

            Two-measurement();        //小目录

            break;

        case '2':

            std::cout << "\n\t\t你选择了2\n";

            orthographic();        //小目录

            break;

        case '3':

            std::cout << "\n\t\t你选择了3\n";

            Equiaxial();        //小目录

            break;

        case '4':

            std::cout << "\n\t\t你选择了4\n";

            exit();        //小目录

            break;

        default:

            std::cout << "\n\t\t输入非法，请重新选择!\n";

            break;

        }

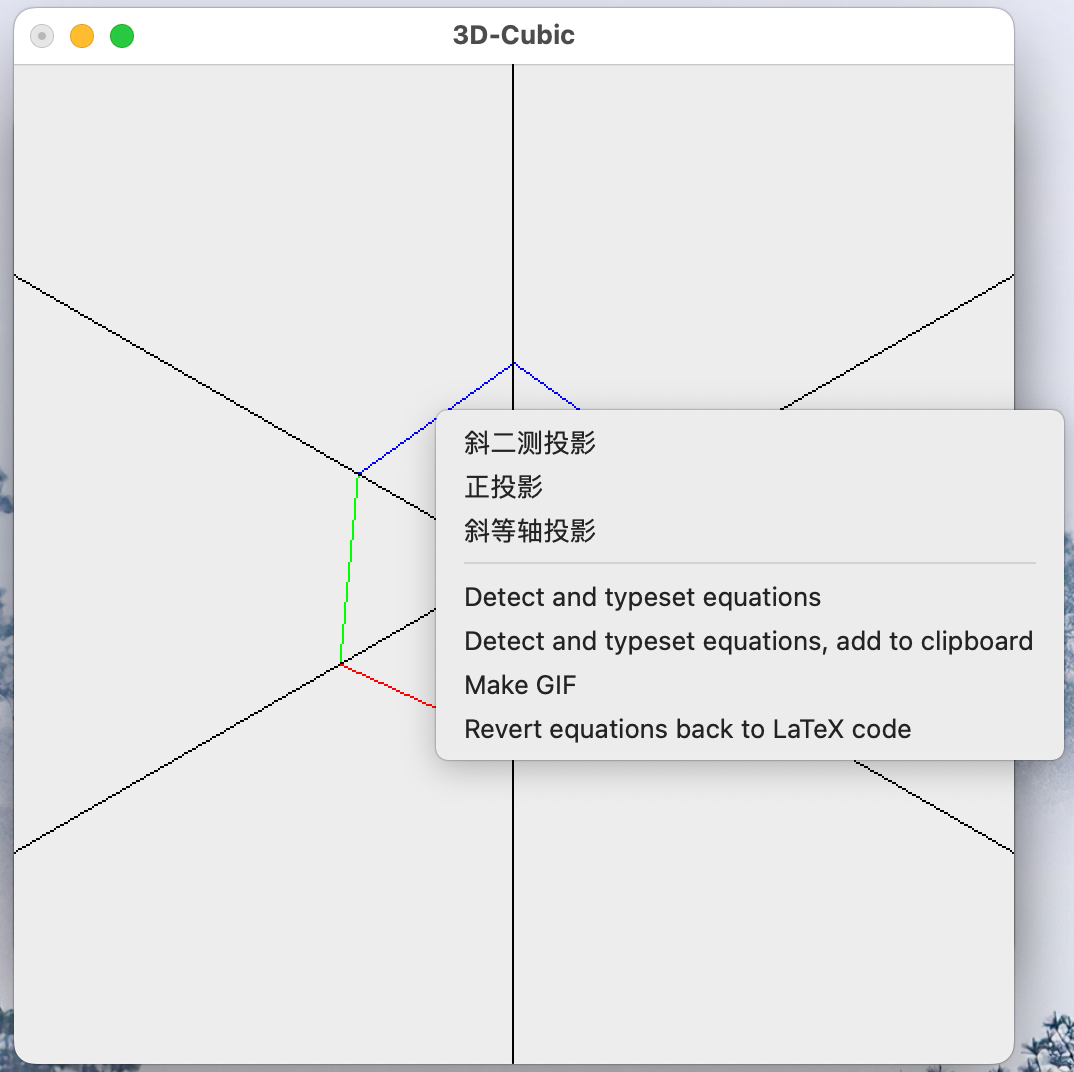
    }

    std::cout << "\n\t\t菜单已退出!\n";

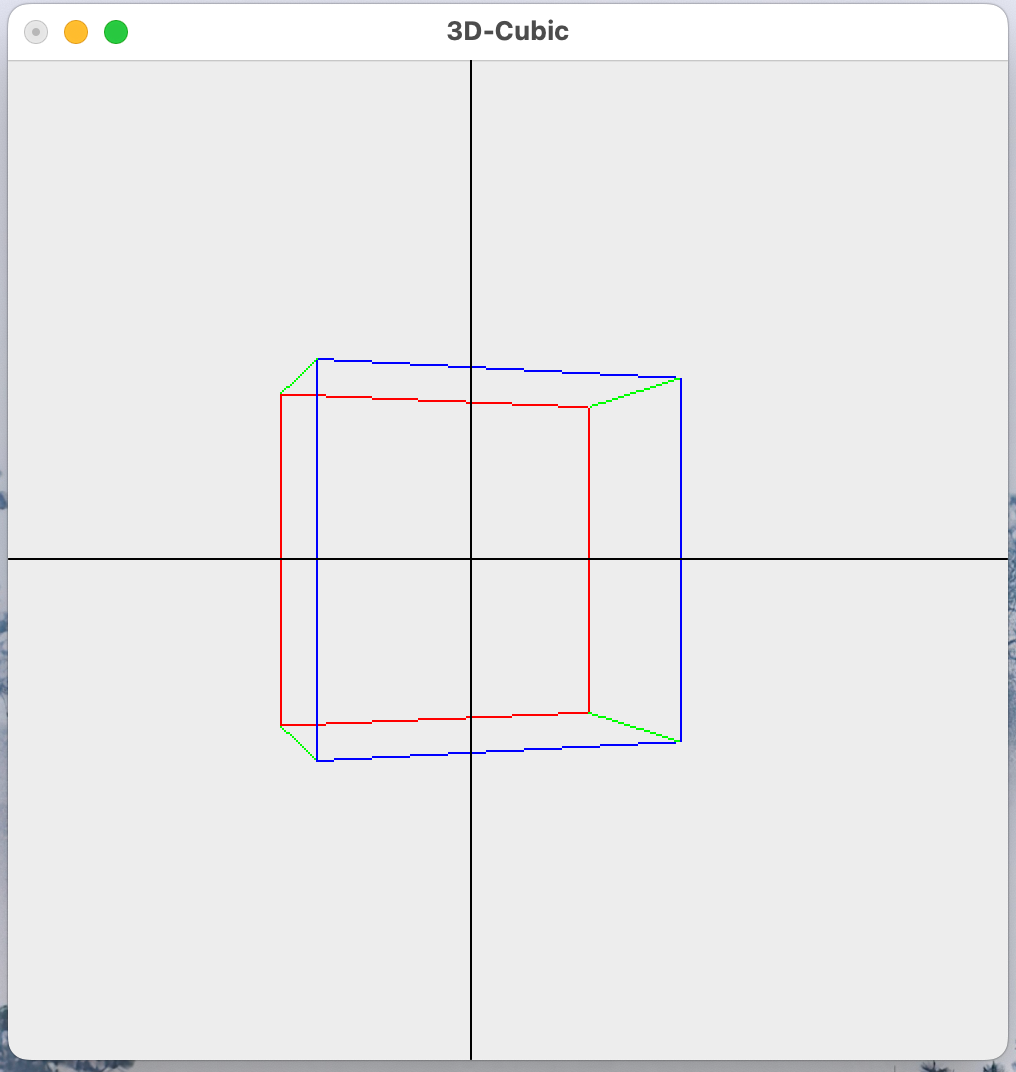
}

## 运行效果截图

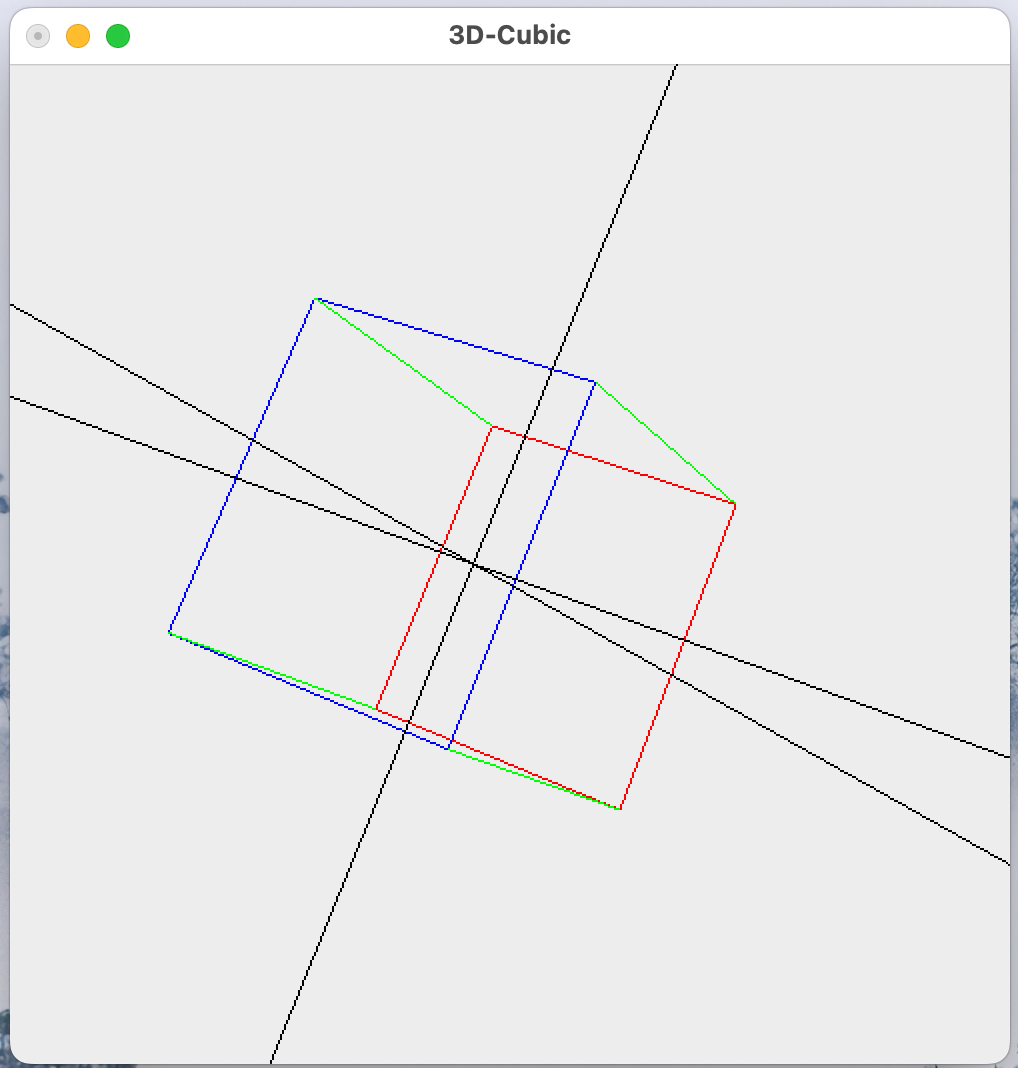
菜单界面：



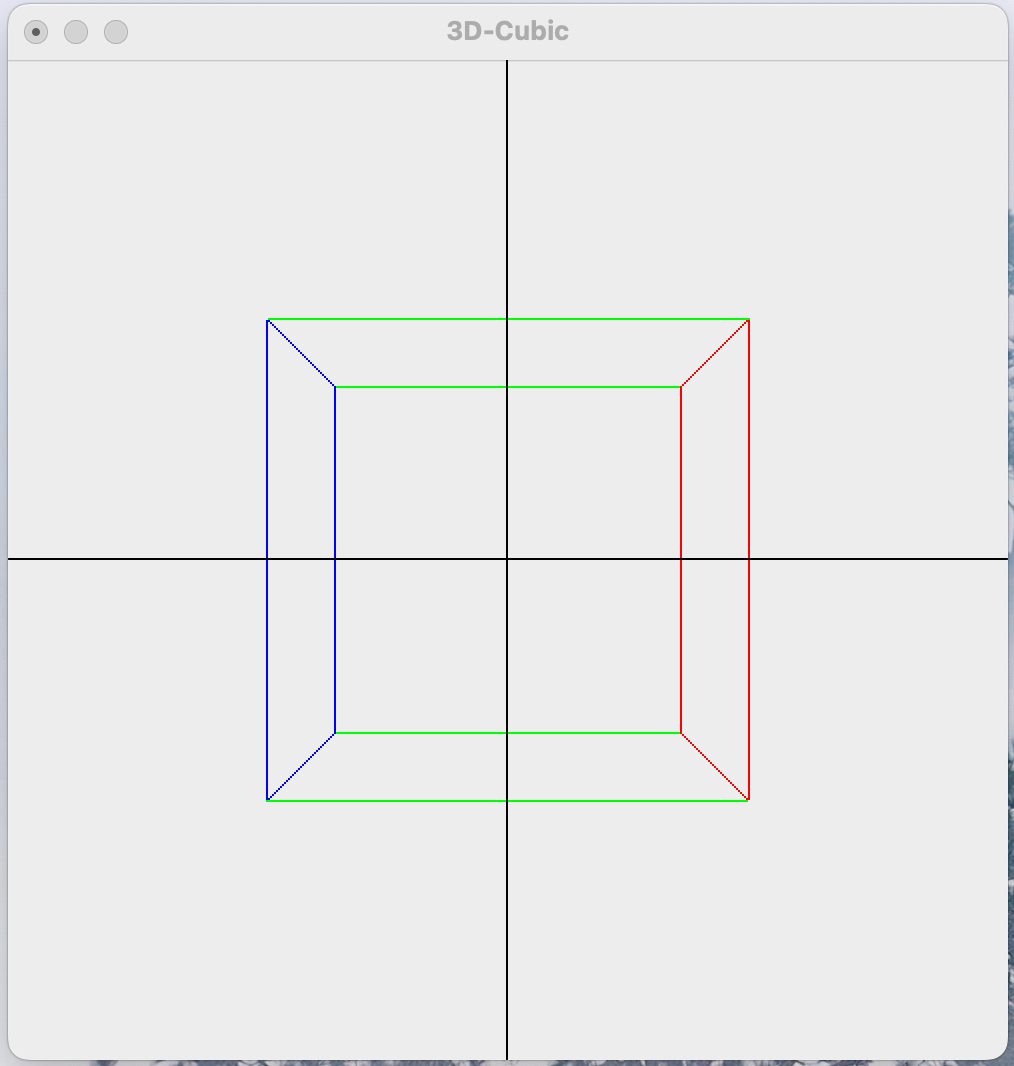
斜二测投影：



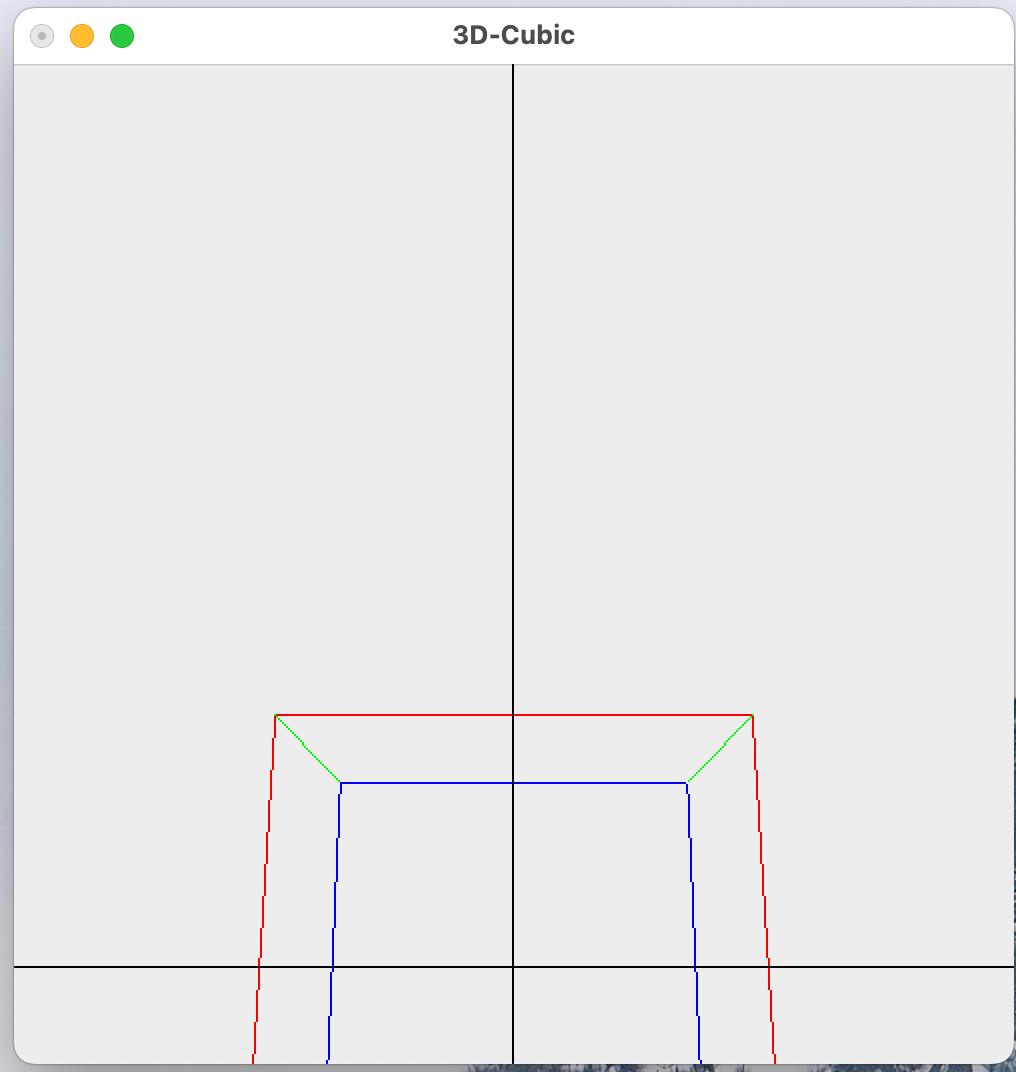
旋转：



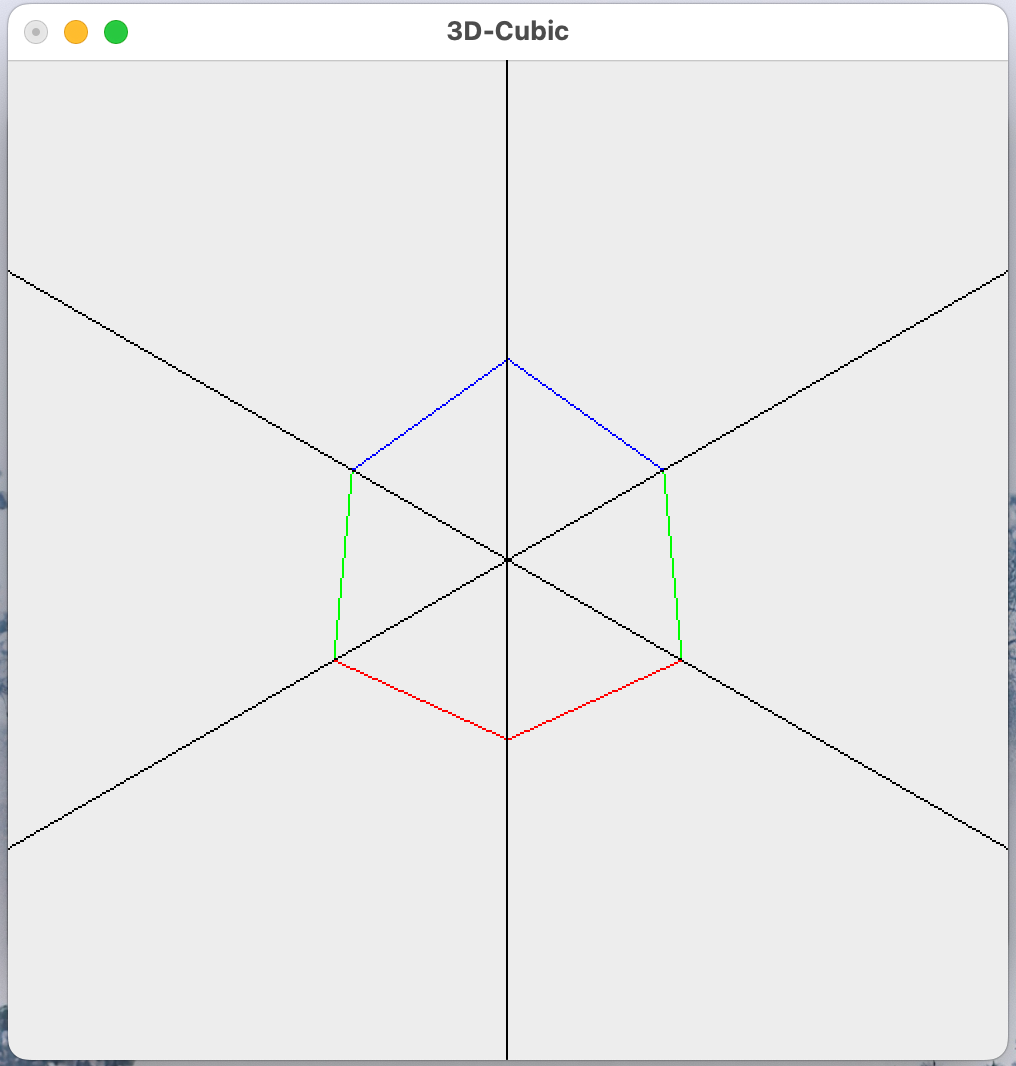
正投影：



平移：



斜等轴投影：



# 三、技术难点分析

## 1. 绘制三维Brehansam直线

二维Brehansam直线所用完整普适性算法教材已经给出，通过查找相关资料，发现并没有现成的算法推广可供借鉴，于是分析三维对象成像原理，发现对于简单的立方体来说，完全可以通过设置三个坐标其中一个参数不变，其余两个坐标参数依旧按照一般二维Brehensam算法执行，于是设置switch循环语句，通过switch选择参数来判断在xOy、xOz或yOz平面，从而实现在该平面画线。

此外，setPixel画点函数在C++原生库中并没有相关函数，于是通过配置OpenGL环境实现对其库函数的使用，查阅发现OpenGL中的

glBegin(GL\_POINTS);

glVertex3i(x, y,z);

glEnd();

相关代码可以基本实现画点功能，通过设置其大小和颜色，成功应用于程序。

## 实现平移、旋转的三维变换矩阵

图形学课程讲解了三维变换矩阵的数学原理，但是如何将其应用于可实现的代码存在困难。通过查阅资料、上网学习，利用数组成功实现相关变换矩阵的存储及其矩阵乘法运算。

此外，OpenGL库中提供的glLoadIdentity()具有初始化变换矩阵为单位矩阵的功能，而glPopMatrix()能够从堆栈中恢复已压栈的变换矩阵。绘制完成基本图形元素后，即可利用定义的矩阵函数和OpenGL矩阵处理函数来实现图形的平移和旋转。

## 实现投影

本题目所要求的投影均属于平行投影。正投影是一种特殊的平行投影，投射方向垂直于投影面。斜二测投影使得垂直于投影面的线段长度缩短为原来的一半，投射直线与投影面所成的角a满足tan a=2，即a=63.4°。斜等轴投影使得垂直于投影面的线段仍保持原来长度，直线与投影面所成夹角a满足tan a=1，即a=45°。于是通过查阅相关资料及互联网，构建出投影函数，给出的参数：float eye\_fov, float aspect\_ratio,float zNear, float zFar。其中，eve\_fov（对应forY）、aspect\_ratio（对应aspect）、zNear（对应n）给的这些参数是lecture5开头计算l,r,b,t所需的条件。而zNear（对应n）和zFar（对应f）是进行透视投影转正交投影的条件。

## 建立鼠标响应函数和键盘响应函数

OpenGL提供了相关响应函数和回调函数。对GLUT鼠标函数，指定鼠标指针在窗口内并且按下或松开，调用了函数glutMouseFunc（mouseFcn），对于鼠标回调函数mouseFcn，允许选择BUTTON参数为GLUT\_LEFT\_BUTTON为鼠标左键，激活拖拽函数。鼠标在窗口内移动并且按下鼠标按钮，调用鼠标回调函数fcnDoSomething（x，y），这坐标被传递给相关函数激活。

对GLUT键盘函数，使用glutKeyboardFunc（keyFcn），使用glutSpecialFunc（SpecialKeyFcn）指定功能键、方向键等。

同时，GLUT提供了菜单函数，并与鼠标结合。glutCreateMenu（menuFcn）创建一个弹出式菜单，它被创建时可以设置menuFcn函数关联到当前某些窗口。光标置于窗口内时，左键鼠标即可显示菜单。