****

中国地质大学（北京）

计算机图形学大作业报告

学 院：信息工程学院

专 业：计算机科学与技术

班 级：10041811

学 号：1005183121

姓 名：周子杰

联系方式：18006163783

邮 箱：2476331552@qq.com

指导老师：严红平

日 期：2020年12月13日

**目 录**

[一、 实验内容： 4](#_Toc59280641)

[二、实现方法： 5](#_Toc59280642)

[1、几何建模功能 5](#_Toc59280643)

[1.1 DDA画线 5](#_Toc59280644)

[1.2 中点画线 6](#_Toc59280645)

[1.3 Bresenham画线 8](#_Toc59280646)

[1.4 中点画圆 10](#_Toc59280647)

[1.5 B样条画曲线 12](#_Toc59280648)

[1.6 Liang-Barsky 线段裁剪算法 16](#_Toc59280649)

[1.7 Sutherland-Hodgman多边形裁剪算法 23](#_Toc59280650)

[1.8 多边形扫描填充算法 30](#_Toc59280651)

[2、几何变换功能 32](#_Toc59280652)

[2.1 时钟 32](#_Toc59280653)

[2.2 小球移动缩放 36](#_Toc59280654)

[3、光照、材质和纹理映射功能 38](#_Toc59280655)

[3.1 单光源 38](#_Toc59280656)

[3.2 多光源叠加 40](#_Toc59280657)

[3.3 纹理映射 41](#_Toc59280658)

[4、可视化功能 47](#_Toc59280659)

[4.1 天空盒+几何模型+视点变换+坐标变换+纹理映射（立方体+球体） 47](#_Toc59280660)

[5、菜单功能 66](#_Toc59280661)

[三、结果分析： 75](#_Toc59280662)

[1、几何建模功能 75](#_Toc59280663)

[1.1 DDA画线 75](#_Toc59280664)

[1.2 中点画线 76](#_Toc59280665)

[1.3 Bresenham画线 77](#_Toc59280666)

[1.4 中点画圆 78](#_Toc59280667)

[1.5 B样条画曲线 79](#_Toc59280668)

[1.6 Liang-Barsky 线段裁剪算法 80](#_Toc59280669)

[1.7 Sutherland-Hodgman多边形裁剪算法 81](#_Toc59280670)

[1.8 多边形扫描填充算法 82](#_Toc59280671)

[2、几何变换功能 83](#_Toc59280672)

[2.1 时钟 83](#_Toc59280673)

[2.2 小球移动缩放 85](#_Toc59280674)

[3、光照、材质和纹理映射功能 86](#_Toc59280675)

[3.1 单光源 86](#_Toc59280676)

[3.2 多光源叠加 87](#_Toc59280677)

[3.3 纹理映射 88](#_Toc59280678)

[4、可视化功能 89](#_Toc59280679)

[4.1 天空盒+几何模型+视点变换+坐标变换+纹理映射（立方体+球体） 89](#_Toc59280680)

[5、菜单功能 93](#_Toc59280681)

[四、结论与展望： 94](#_Toc59280682)

# 实验内容：

综合运用所学，开发一个小型图形软件系统，开发的图形软件系统可通过菜单实现以下功能：

1、 几何建模功能：利用简单的动画和交互技术演示基本图元绘制算法（直线、曲线）所学算法、多边形填充算法和裁剪算法（不能用OpenGL 自带函数）的实现过程；

2、 几何变换功能：利用简单的动画和交互技术实现复杂几何模型（两个或两个以上图元的有机合成）的平移、旋转和缩放，要求实现模型的全局运动和局部相对独立运动；

3、 光照、材质和纹理映射功能：利用简单的动画和交互技术实现单个光源的独立运动和多个光源的相互切换和叠加、单个纹理映射和多个纹理映射以及不同纹理的切换，体会光源之间、光源与材质、光源材质与纹理之间的相互作用；

4、可视化功能：绘制观察坐标系，利用简单的动画和交互技术实现任意选择平行投影或透视投影显示几何模型以及改变视点位置。

# 二、实现方法：

## 1、几何建模功能

### 1.1 DDA画线

**思想：**

已知过端点P0 (x0, y0), P1(x1, y1)的直线段L：*y=kx+b*

直线斜率为 

从x的左端点x0开始，向x右端点步进。步长=1(个象素)，计算相应的y坐标y=kx+b；取象素点(x, round(y))作为当前点的坐标。

具体的计算如下：

计算*yi+1 = kxi+1+b*

*= k1xi+b+kΔx*

*= yi+kΔx*

当*Δx =*1*; yi+1 = yi+k*

* 即：当*x*每递增1，*y*递增*k*(即直线斜率)；
* 注意上述分析的算法仅适用于|*k*| ≤1的情形。在这种情况下，*x*每增加1,*y*最多增加1。

当 |*k*| >1时，必须把*x*，*y*地位互换

**实现：**

//DDA

inline *int* round(const *float* *a*)

{

    return *int*(*a* + 0.5);

}

*void* lineDDA(*int* *x0*, *int* *y0*, *int* *xEnd*, *int* *yEnd*)

{

*int* dx = *xEnd* - *x0*;

*int* dy = *yEnd* - *y0*;

*int* steps, k;

*float* xIncrement, yIncrement, x = *x0*, y = *y0*;

    if (fabs(dx) > fabs(dy))

        steps = fabs(dx);

    else

        steps = fabs(dy);

    xIncrement = *float*(dx) / *float*(steps);

    yIncrement = *float*(dy) / *float*(steps);

    setPixel(round(x), round(y));

    for (k = 0; k < steps; ++k)

    {

        x += xIncrement;

        y += yIncrement;

        setPixel(round(x), round(y));

    }

}

### 1.2 中点画线

**思想：**

当前象素点为(xp, yp) ，下一个象素点为P1或P2。设M=(xp+1, yp+0.5)，为p1与p2之中点，Q为理想直线与x=xp+1垂线的交点。将Q与M的y坐标进行比较。

* 当M在Q的下方，则P2应为下一个象素点；
* 当M在Q的上方，应取P1为下一点。



具体计算如下：

* 构造判别式:d=F(M)=F(xp+1,yp+0.5)

=a(xp+1)+b(yp+0.5)+c

其中a=y0-y1, b=x1-x0, c=x0y1-x1y0

* 当d<0，M在L(Q点)下方，取右上方PU为下一个象素；
* 当d>0，M在L(Q点)上方，取右方PD为下一个象素；
* 当d=0，选P1或P2均可，约定取PD为下一个象素；
* d是xp, yp的线性函数，因此可采用增量计算，提高运算效率。初值d0=F(x0+1, y0+0.5)=a+0.5b
* 若当前象素处于d≥0情况，则取正右方象素PD（xp+1, yp）, 要判下一个象素位置，应计算di+1=F(xp+2, yp+0.5)=a(xp+2)+b(yp+0.5)=d+a； 增量为a
* 若d<0时，则取右上方象素PU（xp+1, yp+1）。要判断再下一象素，则要计算di+1= F(xp+2, yp+1.5)=a(xp+2)+b(yp+1.5)+c=d+a+b ；增量为a＋b
* 画线从(x0, y0)开始，d的初值d0=F(x0+1, y0+0.5)=F(x0, y0)+a+0.5b =a+0.5b。
* 可以用2d代替d来摆脱小数，提高效率。

**实现：**

//中点画线

*void* lineMidp(*int* *x0*, *int* *y0*, *int* *x1*, *int* *y1*)

{

*int* a, b, d1, d2, d, x, y;

    a = *y0* - *y1*, b = *x1* - *x0*, d = 2 \* a + b;

    d1 = 2 \* a, d2 = 2 \* (a + b);

    x = *x0*, y = *y0*;

    setPixel(x, y);

    while (x < *x1*)

    {

        if (d < 0)

        {

            x++;

            y++;

            d += d2;

        }

        else

        {

            x++;

            d += d1;

        }

        setPixel(x, y);

    }

    return;

}

### Bresenham画线

**思想：**

过各行各列象素中心构造一组虚拟网格线。按直线从起点到终点的顺序计算直线与各垂直网格线的交点，然后根据误差项的符号确定该列象素中与此交点最近的象素。



具体计算如下：

设直线方程为： （其中*k*=*dy*/*dx*。）



因为直线的起始点在象素中心，所以误差项d的初值d0＝0。

*x*下标每增加1，*d*的值相应递增直线的斜率值*k*，即*d*＝*d*＋*k*。一旦*d*≥1，就把它减去1，这样保证*d*在0、1之间。

* 当*d*≥0.5时，最接近于当前象素的右上方象素（*xi*＋1，*yi*＋1）
* 而当*d*<0.5时，更接近于右方象素（*xi*＋1，*yi*）。

为方便计算，令*e*＝*d*-0.5，

*e*的初值为-0.5，增量为*k*。

* 当*e*≥0时，取当前象素（*xi*，*yi*）的右上方象素（*xi*＋1，*yi*＋1）；
* 而当e<0时，更接近于右方象素（*xi*＋1，*yi*）。

**实现：**

//Bresenham画线

*void* lineBres(*int* *x0*, *int* *y0*, *int* *xEnd*, *int* *yEnd*)

{

*int* dx = fabs(*xEnd* - *x0*), dy = fabs(*yEnd* - *y0*);

*int* p = 2 \* dy - dx;

*int* twoDy = 2 \* dy, twoDyMinusDx = 2 \* (dy - dx);

*int* x, y;

    if (*x0* > *xEnd*)

    {

        x = *xEnd*;

        y = *yEnd*;

*xEnd* = *x0*;

    }

    else

    {

        x = *x0*;

        y = *y0*;

    }

    setPixel(x, y);

    while (x < *xEnd*)

    {

        x++;

        if (p < 0)

            p += twoDy;

        else

        {

            y++;

            p += twoDyMinusDx;

        }

        setPixel(x, y);

    }

}

### 中点画圆

**思想：**

首先我们要考虑圆的特征：八对称性。也就是说，只要扫描转换八分之一圆弧，就可以求出整个圆弧的象素集。

我们考虑中心在原点，半径为R的第二个八分圆，构造如下判别式（圆方程）：





具体计算如下：

* 若 d<0, 则取P1为下一象素，而且下一象素的判别式为
* 若d>=0, 则应取P2为下一象素，而且下一象素的判别式为
* 第一个象素是（0, R），判别式d的初始值为

**实现：**

//中点画圆

*class* *screenPt*

{

*private:*

*GLint* x, y;

*public:*

    screenPt() { x = y = 0; }

*void* setCoords(*GLint* *xCoordValue*, *GLint* *yCoordValue*)

    {

        x = *xCoordValue*;

        y = *yCoordValue*;

    }

*GLint* getx() const { return x; }

*GLint* gety() const { return y; }

*void* incremetx() { x++; }

*void* decrementy() { y--; }

};

*void* circleMidp(*GLint* *xc*, *GLint* *yc*, *GLint* *radius*)

{

*screenPt* circPt;

*GLint* p = 1 - *radius*;

    circPt.setCoords(0, *radius*);

*void* circlePlotPoints(*GLint*, *GLint*, *screenPt*);

    circlePlotPoints(*xc*, *yc*, circPt);

    while (circPt.getx() < circPt.gety())

    {

        circPt.incremetx();

        if (p < 0)

            p += 2 \* circPt.getx() + 1;

        else

        {

            circPt.decrementy();

            p += 2 \* (circPt.getx() - circPt.gety()) + 1;

        }

        circlePlotPoints(*xc*, *yc*, circPt);

    }

}

*void* circlePlotPoints(*GLint* *xc*, *GLint* *yc*, *screenPt* *circPt*)

{

    setPixel(*xc* + *circPt*.getx(), *yc* + *circPt*.gety());

    setPixel(*xc* - *circPt*.getx(), *yc* + *circPt*.gety());

    setPixel(*xc* + *circPt*.getx(), *yc* - *circPt*.gety());

    setPixel(*xc* - *circPt*.getx(), *yc* - *circPt*.gety());

    setPixel(*xc* + *circPt*.gety(), *yc* + *circPt*.getx());

    setPixel(*xc* - *circPt*.gety(), *yc* + *circPt*.getx());

    setPixel(*xc* + *circPt*.gety(), *yc* - *circPt*.getx());

    setPixel(*xc* - *circPt*.gety(), *yc* - *circPt*.getx());

}

### 1.5 B样条画曲线

**思想：**

与Bezier曲线类似，B样条曲线也通过逼近一组控制点来生成。但是B样条曲线

多项式次数可独立于控制点数目，以及B样条允许局部控制曲线或曲面。这样带来的好处就是我们可以实时地去画点。

其公式如下：



* 给定N=m+n+1个控制点（m为最大段号，n为次数），则第i段n次B样条曲线的数学表达式为：



（其中 i=0,1,...,n; 0<=t<=1）

**实现：**

//B样条曲线

*bool* mouseRightIsDown3B = false;

*struct* *Point3B*

{

*int* x, y;

    Point3B(){};

    Point3B(*int* *tx*, *int* *ty*)

    {

        x = *tx*;

        y = *ty*;

    }

};

std::*vector*<*Point3B*> p;

*double* getRatio(*double* *t*, *double* *a*, *double* *b*, *double* *c*, *double* *d*)

{

    return *a* \* pow(*t*, 3) + *b* \* pow(*t*, 2) + *c* \* *t* + *d*;

}

*double* caculateSquarDistance(*Point3B* *a*, *Point3B* *b*)

{

    return (*a*.x - *b*.x) \* (*a*.x - *b*.x) + (*a*.y - *b*.y) \* (*a*.y - *b*.y);

}

*int* getIndexNearByMouse(*int* *x*, *int* *y*)

{

*double* precision = 200; //精确度

*int* index = -1;

*double* Min;

    for (*int* i = 0; i < p.size(); i++)

    {

*double* dis = caculateSquarDistance(p[i], *Point3B*(*x*, *y*));

        if (dis < precision)

        {

            if (index == -1)

            {

                index = i;

                Min = dis;

            }

            else if (dis < Min)

            {

                index = i;

                Min = dis;

            }

        }

    }

    return index;

}

*void* Bspline(*Point3B* *a*, *Point3B* *b*, *Point3B* *c*, *Point3B* *d*)

{

*int* n = 500;

*double* derta = 1.0 / n;

    glPointSize(2);

    glColor3d(0, 0, 0);

    glBegin(GL\_LINE\_STRIP);

    for (*int* i = 0; i <= n; i++)

    {

*double* t = derta \* i;

*double* ratio[4];

        ratio[0] = getRatio(t, -1, 3, -3, 1);

        ratio[1] = getRatio(t, 3, -6, 0, 4);

        ratio[2] = getRatio(t, -3, 3, 3, 1);

        ratio[3] = getRatio(t, 1, 0, 0, 0);

*double* x = 0, y = 0;

        x += ratio[0] \* *a*.x + ratio[1] \* *b*.x + ratio[2] \* *c*.x + ratio[3] \* *d*.x;

        y += ratio[0] \* *a*.y + ratio[1] \* *b*.y + ratio[2] \* *c*.y + ratio[3] \* *d*.y;

        x /= 6.0;

        y /= 6.0;

        glVertex2d(x, y);

    }

    glEnd();

}

*void* display3B()

{

    glClear(GL\_COLOR\_BUFFER\_BIT); //清除颜色缓存和深度缓存

    //画点

    glPointSize(5);

    glColor3d(1, 0, 0);

    glBegin(GL\_POINTS);

    for (*int* i = 0; i < p.size(); i++)

        glVertex2d(p[i].x, p[i].y);

    glEnd();

    //画线

    glLineWidth(2);

    glColor3d(0, 1, 0);

    glBegin(GL\_LINE\_STRIP);

    for (*int* i = 0; i < p.size(); i++)

        glVertex2d(p[i].x, p[i].y);

    glEnd();

    if (p.size() >= 4)

        for (*int* i = 0; i < p.size() - 3; i++)

            Bspline(p[i], p[i + 1], p[i + 2], p[i + 3]);

    glFlush();

}

### 1.6 Liang-Barsky 线段裁剪算法

**思想：**

将线段P1P2表示为参数形式：

确定始边和终边

* 左→右，右→左
* 下→上，上→下



对于每条直线，裁剪矩形内的线段部分由参数u1和u2定义

* u1的值由线段的始边边界（即：从外到内遇到的矩形边界）(p<0)所决定。对这些边界计算rk。u1取0和各个rk值之中的最大值
* u2的值由由线段的终边边界（即：线段从内到外遇到的矩形边界）(p>0)所决定。对这些边界计算rk。u2取1和各个rk值之中的最小值
* 如果u1>u2，则线段完全落在裁剪窗口之外，被舍弃
* 否则裁剪线段由参数u的两个值u1,u2计算出来

**实现：**

//Liang-Barsky 线段裁剪算法

*class* *wcPt2D*

{

*public:*

*GLfloat* x, y;

*public:*

    /\*

Default Constructor:initalize position 48(0.0,0.0).\*/

    wcPt2D()

    {

        x = y = 0.0;

    }

    wcPt2D(*GLfloat* *nx*, *GLfloat* *ny*) : x(*nx*), y(*ny*) {}

    wcPt2D(const *wcPt2D* &*pCopy*)

    {

        this->x = *pCopy*.x;

        this->y = *pCopy*.y;

    }

*void* setCoords(*GLfloat* *xCoord*, *GLfloat* *yCoord*)

    {

        x = *xCoord*;

        y = *yCoord*;

        return;

    }

*wcPt2D* &operator=(*wcPt2D* *p2*)

    {

        this->x = *p2*.getx();

        this->y = *p2*.gety();

        return \*this;

    }

*GLfloat* getx() const

    {

        return x;

    }

*GLfloat* gety() const

    {

        return y;

    }

};

inline *GLint* round16(const *GLfloat* *a*)

{

    return *GLint*(*a* + 0.5);

}

*GLint* clipTest(*GLfloat* *p*, *GLfloat* *q*, *GLfloat* \**u1*, *GLfloat* \**u2*)

{

*GLfloat* r;

*GLint* returnValue = true;

    if (*p* < 0.0)

    {

        r = *q* / *p*;

        if (r > \**u2*)

        {

            returnValue = false;

        }

        else if (r > \**u1*)

        {

            \**u1* = r;

        }

    }

    else

    {

        if (*p* > 0.0)

        {

            r = *q* / *p*;

            if (r < \**u1*)

            {

                returnValue = false;

            }

            else if (r < \**u2*)

            {

                \**u2* = r;

            }

        }

        else

        {

            if (*q* < 0.0)

            {

                returnValue = false;

            }

        }

    }

    return (returnValue);

}

*void* lineClipLiangBrask(*wcPt2D* *winMin*, *wcPt2D* *winMax*, *wcPt2D* *p1*, *wcPt2D* *p2*)

{

*GLfloat* u1 = 0.0, u2 = 1.0, dx = *p2*.getx() - *p1*.getx(), dy;

    if (clipTest(-dx, *p1*.getx() - *winMin*.getx(), &u1, &u2))

    {

        if (clipTest(dx, *winMax*.getx() - *p1*.getx(), &u1, &u2))

        {

            dy = *p2*.gety() - *p1*.gety();

            if (clipTest(-dy, *p1*.gety() - *winMin*.gety(), &u1, &u2))

            {

                if (clipTest(dy, *winMax*.gety() - *p1*.gety(), &u1, &u2))

                {

                    if (u2 < 1.0)

                    {

*p2*.setCoords(*p1*.getx() + u2 \* dx, *p1*.gety() + u2 \* dy);

                    }

                    if (u1 > 0.0)

                    {

*p1*.setCoords(*p1*.getx() + u1 \* dx, *p1*.gety() + u1 \* dy);

                    }

                    lineDDA(round16(*p1*.getx()), round16(*p1*.gety()), round16(*p2*.getx()), round16(*p2*.gety()));

                    //                  return 2;

                }

            }

        }

    }

    return;

}

*int* lineClipLiangBrask(*wcPt2D* *winMin*, *wcPt2D* *winMax*, *wcPt2D* *p1*, *wcPt2D* *p2*, *wcPt2D* *pOut*[])

{

*GLfloat* u1 = 0.0, u2 = 1.0, dx = *p2*.getx() - *p1*.getx(), dy;

    if (clipTest(-dx, *p1*.getx() - *winMin*.getx(), &u1, &u2))

    {

        if (clipTest(dx, *winMax*.getx() - *p1*.getx(), &u1, &u2))

        {

            dy = *p2*.gety() - *p1*.gety();

            if (clipTest(-dy, *p1*.gety() - *winMin*.gety(), &u1, &u2))

            {

                if (clipTest(dy, *winMax*.gety() - *p1*.gety(), &u1, &u2))

                {

                    if (u2 < 1.0)

                    {

*p2*.setCoords(*p1*.getx() + u2 \* dx, *p1*.gety() + u2 \* dy);

                    }

                    if (u1 > 0.0)

                    {

*p1*.setCoords(*p1*.getx() + u1 \* dx, *p1*.gety() + u1 \* dy);

                    }

*pOut*[0] = *p1*;

*pOut*[1] = *p2*;

                    return 2;

                }

            }

        }

    }

    return 0;

}

*void* display16()

{

    glClear(GL\_COLOR\_BUFFER\_BIT);

    //可视化边界:

    const *int* minX = 100, minY = 100, maxX = 300, maxY = 300;

    glColor3f(1.0, 0.0, 0.0);

    lineDDA(minX, minY, minX, maxY);

    lineDDA(minX, minY, maxX, minY);

    lineDDA(maxX, maxY, minX, maxY);

    lineDDA(maxX, maxY, maxX, minY);

    glColor3f(0.0, 1.0, 0.0);

*GLint* n = 5;

*wcPt2D* pIn[n];

    pIn[0].setCoords(0, 200);

    pIn[1].setCoords(150, 250);

    pIn[2].setCoords(250, 250);

    pIn[3].setCoords(400, 200);

    pIn[4].setCoords(200, 50);

    for (*int* i = 0; i < n; ++i)

    {

        lineDDA(pIn[i].x, pIn[i].y, pIn[(i + 1) % n].x, pIn[(i + 1) % n].y);

    }

*wcPt2D* pOut[20];

*wcPt2D* tempPOut[2];

*int* outCount = 0;

*int* flag = 0;

    for (*int* i = 0; i < n; ++i)

    {

        flag = lineClipLiangBrask(*wcPt2D*(minX, minY), *wcPt2D*(maxX, maxY), pIn[i], pIn[(i + 1) % n], tempPOut);

        if (flag == 2)

        {

            for (*int* j = 0; j < 2; ++j)

            {

                pOut[outCount++] = tempPOut[j];

            }

        }

    }

    glColor3f(0.0, 0.0, 1.0);

*int* i = 1;

    for (; i <= outCount; ++i)

    {

        lineDDA(pOut[i - 1].x, pOut[i - 1].y, pOut[i % outCount].x, pOut[i % outCount].y);

    }

    glFlush();

    return;

}

### 1.7 Sutherland-Hodgman多边形裁剪算法

**思想：**

* 基本思想是一次用窗口的一条边裁剪多边形
* 考虑窗口的一条边以及延长线构成的裁剪线该线把平面分成两个部分:可见一侧；不可见一侧
* 多边形的各条边的两端点S、P。它们与裁剪线的位置关系只有四种



* 对于情况（1）仅输出顶点P；情况（2）输出0个顶点
* 情况（3）输出线段SP与裁剪线的交点I
* 情况（4）输出线段SP与裁剪线的交点I和终点P
  1. 上述算法仅用一条裁剪边对多边形进行裁剪，得到一个顶点序列，作为下一条裁剪边处理过程的输入
  2. 对于每一条裁剪边，算法框图同上，只是判断点在窗口哪一侧以及求线段SP与裁剪边的交点算法应随之改变

**实现：**

//Sutherland-Hodgman多边形裁剪算法

const *int* Left = 0, Right = 1, Bottom = 2, Top = 3; //左右下上 分别裁剪

const *GLint* nClip = 4;                              //边界边数

//判断点与边界的位置关系

*GLint* inside(*wcPt2D* *p*, *int* *b*, *wcPt2D* *wMin*, *wcPt2D* *wMax*)

{

    switch (*b*)

    {

    case Left:

        if (*p*.getx() < *wMin*.getx())

            return false;

        break;

    case Right:

        if (*p*.getx() > *wMax*.getx())

            return false;

        break;

    case Bottom:

        if (*p*.gety() < *wMin*.gety())

            return false;

        break;

    case Top:

        if (*p*.gety() > *wMax*.gety())

            return false;

        break;

    }

    return true;

}

//判断是否相交

*GLint* cross(*wcPt2D* *p1*, *wcPt2D* *p2*, *int* *winEdge*, *wcPt2D* *wMin*, *wcPt2D* *wMax*)

{

    if (inside(*p1*, *winEdge*, *wMin*, *wMax*) == inside(*p2*, *winEdge*, *wMin*, *wMax*))

        return false;

    else

        return true;

}

//返回交点

*wcPt2D* intersect(*wcPt2D* *p1*, *wcPt2D* *p2*, *int* *winEdge*, *wcPt2D* *wMin*, *wcPt2D* *wMax*)

{

*wcPt2D* iPt;

*GLfloat* m;

    if (*p1*.getx() != *p2*.getx())

    {

        m = (*p1*.gety() - *p2*.gety()) / (*p1*.getx() - *p2*.getx());

    }

    switch (*winEdge*)

    {

    case Left:

        iPt.x = *wMin*.x;

        iPt.y = *p2*.y + (*wMin*.x - *p2*.x) \* m;

        break;

    case Right:

        iPt.x = *wMax*.x;

        iPt.y = *p2*.y + (*wMax*.x - *p2*.x) \* m;

        break;

    case Bottom:

        iPt.y = *wMin*.y;

        if (*p1*.x != *p2*.x)

            iPt.x = *p2*.x + (*wMin*.y - *p2*.y) / m;

        else

            iPt.x = *p2*.x;

        break;

    case Top:

        iPt.y = *wMax*.y;

        if (*p1*.x != *p2*.x)

            iPt.x = *p2*.x + (*wMax*.y - *p2*.y) / m;

        else

            iPt.x = *p2*.x;

        break;

    default:

        break;

    }

    return iPt;

}

//裁剪点

*void* clipPoint(*wcPt2D* *p*, *int* *winEdge*, *wcPt2D* *wMin*, *wcPt2D* *wMax*, *wcPt2D* \**pOut*, *int* \**cnt*, *wcPt2D* *first*[], *wcPt2D* \**s*)

{

*wcPt2D* iPt;

    //无交点则保存

    if (*first*[*winEdge*].x == 0 && *first*[*winEdge*].y == 0)

    {

*first*[*winEdge*] = *p*;

    }

    else

    {

        //有交点则找到交点并保存

        if (cross(*p*, *s*[*winEdge*], *winEdge*, *wMin*, *wMax*))

        {

            iPt = intersect(*p*, *s*[*winEdge*], *winEdge*, *wMin*, *wMax*);

            if (*winEdge* < Top)

            {

                clipPoint(iPt, *winEdge* + 1, *wMin*, *wMax*, *pOut*, *cnt*, *first*, *s*);

            }

            else

            {

*pOut*[\**cnt*] = iPt;

                (\**cnt*)++;

            }

        }

    }

*s*[*winEdge*] = *p*;

    //如果点在边内，换下一条边

    if (inside(*p*, *winEdge*, *wMin*, *wMax*))

    {

        if (*winEdge* < Top)

            clipPoint(*p*, *winEdge* + 1, *wMin*, *wMax*, *pOut*, *cnt*, *first*, *s*);

        else

        {

*pOut*[\**cnt*] = *p*;

            (\**cnt*)++;

        }

    }

}

*void* closeClip(*wcPt2D* *wMin*, *wcPt2D* *wMax*, *wcPt2D* \**pOut*, *GLint* \**cnt*, *wcPt2D* *first*[], *wcPt2D* \**s*)

{

*wcPt2D* pt;

*int* winEdge;

    for (winEdge = Left; winEdge <= Top; winEdge++)

    {

        if (cross(*s*[winEdge], *first*[winEdge], winEdge, *wMin*, *wMax*))

        {

            pt = intersect(*s*[winEdge], *first*[winEdge], winEdge, *wMin*, *wMax*);

            if (winEdge < Top)

            {

                clipPoint(pt, winEdge + 1, *wMin*, *wMax*, *pOut*, *cnt*, *first*, *s*);

            }

            else

            {

*pOut*[\**cnt*] = pt;

                (\**cnt*)++;

            }

        }

    }

}

//多边形裁剪

*GLint* polygonClipSuthHodg(*wcPt2D* *wMin*, *wcPt2D* *wMax*, *GLint* *n*, *wcPt2D* \**pIn*, *wcPt2D* \**pOut*)

{

*wcPt2D* first[nClip], s[nClip];

*GLint* k, cnt = 0;

    for (k = 0; k < *n*; k++)

        clipPoint(*pIn*[k], Left, *wMin*, *wMax*, *pOut*, &cnt, first, s);

    closeClip(*wMin*, *wMax*, *pOut*, &cnt, first, s);

    return cnt;

}

//绘制程序

*void* display17()

{

    glClear(GL\_COLOR\_BUFFER\_BIT); //将屏幕设置为黑色

    //Sutherland-Hodgman多边形裁剪算法

    glClear(GL\_COLOR\_BUFFER\_BIT);

    //裁剪区域

    const *int* minX = 100, minY = 100, maxX = 300, maxY = 300;

    glColor3f(1.0, 0.0, 0.0);

    lineDDA(minX, minY, minX, maxY);

    lineDDA(minX, minY, maxX, minY);

    lineDDA(maxX, maxY, minX, maxY);

    lineDDA(maxX, maxY, maxX, minY);

    //多边形

    glColor3f(0.0, 1.0, 0.0);

*GLint* n = 4;

*wcPt2D* pIn[n];

    pIn[0].setCoords(50, 200);

    pIn[1].setCoords(200, 350);

    pIn[2].setCoords(350, 200);

    pIn[3].setCoords(200, 50);

    for (*int* i = 0; i < n; ++i)

    {

        lineDDA(pIn[i].x, pIn[i].y, pIn[(i + 1) % n].x, pIn[(i + 1) % n].y);

    }

*wcPt2D* pOut[20];

*int* count = polygonClipSuthHodg(*wcPt2D*(minX, minY), *wcPt2D*(maxX, maxY), n, pIn, pOut);

    glColor3f(0.0, 0.0, 1.0);

*int* i = 1;

    for (; i <= count; ++i)

    {

        lineDDA(pOut[i - 1].x, pOut[i - 1].y, pOut[i % count].x, pOut[i % count].y);

    }

    glFlush();

    return;

}

### 1.8 多边形扫描填充算法

**思想：**

* 求交：计算扫描线与多边形各边的交点；
* 排序：把所有交点按x值递增顺序排序；
* 配对：第一个与第二个，第三个与第四个等等；每对交点代表扫描线与多边形的一个相交区间
* 填色：把相交区间内的象素置成多边形颜色，把相交区间外的象素置成背景色。

这里是改进的扫描线填充算法，所以加入了活动边表：（利用边和扫描线的连贯性）

* 与当前扫描线相交的边称为**活动边**（active edge)，把它们按与扫描线交点x坐标递增的顺序存入一个链表中，称为**活动边表**（AET, Active edge table)
* 只需对当前扫描线的活动边表作更新，即可得到下一条扫描线的活动边表。

**代码：**

//扫描划线填充

*void* boundaryFill(vector <Point> *points*){

    //获取y坐标值最大和最小值

*int* yMin = points[0].y, yMax = points[0].x;

    for(*int* i=1; i<points.size(); ++i){

        if(yMin > points[i].y)

            yMin = points[i].y;

        if(yMax < points[i].y)

            yMax = points[i].y;

    }

    //建立边表edgeTable

    list<Edge> edgeTable[windowHeight];

    for(*int* i=0;i<points.size();++i){

*int* x0 = points[i].x;

*int* y0 = points[i].y;

*int* x1 = points[(i+1)%points.size()].x;

*int* y1 = points[(i+1)%points.size()].y;

        //舍弃与扫描线水平的线

        if(y0==y1)

            continue;

        //edgeTable边的各个参数

*int* yMinTmp = min(y0,y1);

*int* yMaxTmp = max(y0,y1);

*float* x=y0<y1?x0:x1;

*float* dx=(x0-x1)\*1.0/(y0-y1);

        edgeTable[yMinTmp].push\_back (Edge(x,dx,yMaxTmp));

    }

    //建立活动边表activeEdgeTable

    list<Edge> activeEdgeTable;

    //头结点

    activeEdgeTable.push\_back (Edge());

    //建立活动边表

    for(*int* i=yMin;i<yMax;++i){

        //按递增顺序建立活动边表

        for(*auto* j=edgeTable[i].begin(); j!=edgeTable[i].end ();){

*auto* tmp=activeEdgeTable.begin ();

*auto* end=activeEdgeTable.end ();

            for(; tmp!=end; ++tmp){

                if(tmp->x > j->x){

                    break;

                }

                if(j->x == tmp->x && j->dx < tmp->dx){

                    break;

                }

            }

            activeEdgeTable.insert (tmp,\*j);

            j=edgeTable[i].erase (j);

        }

        //删除当前活动边表中y最大的边

        for(*auto* tmp2=activeEdgeTable.begin (); tmp2!=activeEdgeTable.end ();){

            if(tmp2->yMax==i){

                tmp2=activeEdgeTable.erase (tmp2);

            }else{

                tmp2++;

            }

        }

        //填色

        //一次选取两个点组成填色区间

*auto* tmp3 = activeEdgeTable.begin ();

        tmp3++; //跳过头结点

*auto* tmp3\_next=tmp3;

        tmp3\_next++;

*int* count=activeEdgeTable.size ()-1;

        while(count>=2){

            for(*int* x=tmp3->x; x<tmp3\_next->x; ++x){

                setPixel (x,i);

            }

            tmp3++;

            tmp3++;

            tmp3\_next=tmp3;

            tmp3\_next++;

            count-=2;

        }

        //更新activeEdgeTable中x的值

        for(*auto* &tmp:activeEdgeTable){

            tmp.x+=tmp.dx;

        }

    }

    return ;

}

## 2、几何变换功能

### 2.1 时钟

**思想：**

我们先通过中点画圆绘制表盘，同时绘制刻度、时针等东西。然后再加个定时器，每一秒重新绘制图形。而图形采用二维变换里的旋转变换来控制时分秒针的转动。（转动角度这些的比较简单，在这里就不展开了）。同时，需要获取系统当前时间作为初始化。

**实现：**

//时钟

//当前时间，时 分 秒

*float* h = 0.0f;

*float* m = 0.0f;

*float* s = 0.0f;

//画时钟的函数

*void* displayClock(*void*)

{

*int* Width = 600;

*int* Height = 600;

*GLfloat* PI = 3.1415926f;

    //用当前清除色清除颜色缓冲区，即设定窗口的背景色

    glClear(GL\_COLOR\_BUFFER\_BIT);

    glColor3f(0.1f, 0.2f, 0.1f); //设置颜色

    //画表盘

*int* cx = Width / 2; //中心点

*int* cy = Height / 2;

*int* R = 100; // 半径长

*int* n = 100;

*int* i;

    circleMidp(cx, cy, R); //中点画圆法

    //绘制刻度

*int* lines = 60;

    for (i = 0; i < lines; i++)

    {

        //5的倍数的刻度 粗点

        if (i % 5 == 0)

        {

            glLineWidth(3);

            glBegin(GL\_LINES);

            glVertex2f(cx + (R - 10) \* sin(2 \* PI / lines \* i), cy + (R - 10) \* cos(2 \* PI / lines \* i));

            glVertex2f(cx + R \* sin(2 \* PI / lines \* i), cy + R \* cos(2 \* PI / lines \* i));

        }

        else

        {

            glLineWidth(2); //其余刻度

            glBegin(GL\_LINES);

            glVertex2f(cx + (R - 5) \* sin(2 \* PI / lines \* i), cy + (R - 5) \* cos(2 \* PI / lines \* i));

            glVertex2f(cx + R \* sin(2 \* PI / lines \* i), cy + R \* cos(2 \* PI / lines \* i));

        }

    }

    //绘制时 分 秒针

*int* h\_len = 60;  //时针长度

*int* m\_len = 80;  //分针长度

*int* s\_len = 100; //秒针长度

*float* s\_Angle = s / 60.0;

*float* m\_Angle = (m \* 60 + s) / 3600.0;

*float* h2 = h >= 12 ? (h - 12) : h;

*float* h\_Angle = (h2 \* 60 \* 60 + m \* 60 + s) / (12 \* 60 \* 60);

    //时

    glLineWidth(3); //时针宽度

    glBegin(GL\_LINES);

    glVertex2f(cx, cy);

    glVertex2f(cx + h\_len \* sin(2 \* PI \* h\_Angle), cy + h\_len \* cos(2 \* PI \* h\_Angle));

    glEnd();

    //分

    glLineWidth(2); //分针宽度

    glBegin(GL\_LINES);

    glVertex2f(cx, cy);

    glVertex2f(cx + m\_len \* sin(2 \* PI \* m\_Angle), cy + m\_len \* cos(2 \* PI \* m\_Angle));

    glEnd();

    //秒

    glLineWidth(1); //秒针宽度

    glBegin(GL\_LINES);

    glVertex2f(cx - 2 \* 5 \* sin(2 \* PI \* s\_Angle), cy - 2 \* 5 \* cos(2 \* PI \* s\_Angle));

    glVertex2f(cx + (R - 10) \* sin(2 \* PI \* s\_Angle), cy + (R - 10) \* cos(2 \* PI \* s\_Angle));

    glEnd();

    //刷新

    glFlush();

}

//回调函数，配合定时器使用

*void* timerFunc(*int* *value*)

{

    s += 1;

*int* carry1 = 0;

    if (s >= 60)

    {

        s = 0;

        carry1 = 1;

    }

    m += carry1;

*int* carry2 = 0;

    if (m >= 60)

    {

        m = 0;

        carry2 = 1;

    }

    h += carry2;

    if (h >= 24)

        h = 0;

    glutPostRedisplay();               //重画

    glutTimerFunc(1000, timerFunc, 1); //每一秒执行一次

}

//初始化：设置背景，获取时间

*void* initClock(*void*)

{

    // 设置窗口为白色

    glClearColor(1.0f, 1.0f, 1.0f, 1.0f);

    // 获取本地当前时间

*SYSTEMTIME* sys;

    GetLocalTime(&sys);

    h = sys.wHour;

    m = sys.wMinute;

    s = sys.wSecond;

}

### 2.2 小球移动缩放

**思想：**

三维变换类似二维变换，将东西存入矩阵然后不断变换。（要设置变换方式和回调函数等）

**实现：**

//小球移动缩放

*float* angle22 = 0.0f;

*void* Display22()

{

    glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

    //  glutSolidSphere(1.0, 100, 50);

    glPushMatrix();

    glRotatef(angle22, 0.0f, 0.0f, 0.0f); //角度+旋转轴

    glTranslatef(1.0f, 0.0f, 0.0f);       //移动

    glutSolidSphere(1.0, 100, 50);        //半径为1, 100条纬线, 50条经线

    glPopMatrix();

    glFlush();

}

*void* myIdle22()

{

    angle22 += 0.1f;

    if (angle22 >= 360.0f)

    {

        angle22 = 0.0f;

    }

    Display22();

}

*void* Reshape22(*int* *w*, *int* *h*)

{

    //使像素矩阵占据整个新窗口

    glViewport(0, 0, (*GLsizei*)*w*, (*GLsizei*)*h*);

    //置当前矩阵为投影矩阵

    glMatrixMode(GL\_PROJECTION);

    glLoadIdentity();

    //改变窗口大小，图形形状保持不变

    if (*w* <= *h*)

    {

        glOrtho(-1.5, 1.5, -1.5 \* (*GLfloat*)*h* / (*GLfloat*)*w*, 1.5 \* (*GLfloat*)*h* / (*GLfloat*)*w*, -10.0, 10.0);

    }

    else

    {

        glOrtho(-1.5 \* (*GLfloat*)*w* / (*GLfloat*)*h*, 1.5 \* (*GLfloat*)*w* / (*GLfloat*)*h*, -1.5, 1.5, -10.0, 10.0);

    }

    //模型矩阵

    glMatrixMode(GL\_MODELVIEW);

    glLoadIdentity();

}

## 3、光照、材质和纹理映射功能

### 3.1 单光源

**思想：**

设置视角位置、光源位置、颜色等，然后启用灯光。

**实现：**

*void* Init22()

{

*GLfloat* mat\_specular[] = {1.0, 1.0, 1.0, 1.0};

*GLfloat* mat\_shininess[] = {50.0};

    //设置光源位置

*GLfloat* light\_position[] = {-10.0, 1.0, 20.0, 0.0};

*GLfloat* light\_position2[] = {1.0, 1.0, 1.0, 0.0};

*GLfloat* light\_position3[] = {1.0, 10.0, 1.0, 0.0};

    //环境光

*GLfloat* lmodel\_ambient[] = {0.1, 0.1, 0.1, 1.0};

    glLightModelfv(GL\_LIGHT\_MODEL\_AMBIENT, lmodel\_ambient);

    glClearColor(0.0, 0.0, 0.0, 0.0);

    glShadeModel(GL\_SMOOTH);

    glMaterialfv(GL\_FRONT, GL\_SPECULAR, mat\_specular);   //指定材质属性

    glMaterialfv(GL\_FRONT, GL\_SHININESS, mat\_shininess); //镜面反射指数

    //设置light0--white

*GLfloat* white\_light[] = {1.0, 1.0, 1.0, 1.0};      //设置编号、特性、颜色

    glLightfv(GL\_LIGHT0, GL\_POSITION, light\_position); //设置位置

    glLightfv(GL\_LIGHT0, GL\_DIFFUSE, white\_light);

    glLightfv(GL\_LIGHT0, GL\_SPECULAR, white\_light);

    glEnable(GL\_LIGHT0);

    //设置light1--red

*GLfloat* red\_light[] = {1.0, 0.0, 0.0, 1.0};

    glLightfv(GL\_LIGHT1, GL\_POSITION, light\_position);

    glLightfv(GL\_LIGHT1, GL\_DIFFUSE, red\_light);

    glLightfv(GL\_LIGHT1, GL\_SPECULAR, red\_light);

    // glEnable(GL\_LIGHT1);

    //设置light2--green

*GLfloat* green\_light[] = {0.0, 1.0, 0.0, 1.0};

    glLightfv(GL\_LIGHT2, GL\_POSITION, light\_position2);

    glLightfv(GL\_LIGHT2, GL\_DIFFUSE, green\_light);

    glLightfv(GL\_LIGHT2, GL\_SPECULAR, green\_light);

    // glEnable(GL\_LIGHT2);

    //设置light3--blue

*GLfloat* blue\_light[] = {0.0, 0.0, 1.0, 1.0};

    glLightfv(GL\_LIGHT3, GL\_POSITION, light\_position3);

    glLightfv(GL\_LIGHT3, GL\_DIFFUSE, blue\_light);

    glLightfv(GL\_LIGHT3, GL\_SPECULAR, blue\_light);

    // glEnable(GL\_LIGHT3);

    glEnable(GL\_LIGHTING);

    glEnable(GL\_DEPTH\_TEST);

}

### 3.2 多光源叠加

**思想：**

设置多个光源，然后设置视角位置、光源位置、颜色等，最后启用这些灯光即可。

**实现：**

*void* Init22()

{

*GLfloat* mat\_specular[] = {1.0, 1.0, 1.0, 1.0};

*GLfloat* mat\_shininess[] = {50.0};

    //设置光源位置

*GLfloat* light\_position[] = {-10.0, 1.0, 20.0, 0.0};

*GLfloat* light\_position2[] = {1.0, 1.0, 1.0, 0.0};

*GLfloat* light\_position3[] = {1.0, 10.0, 1.0, 0.0};

    //环境光

*GLfloat* lmodel\_ambient[] = {0.1, 0.1, 0.1, 1.0};

    glLightModelfv(GL\_LIGHT\_MODEL\_AMBIENT, lmodel\_ambient);

    glClearColor(0.0, 0.0, 0.0, 0.0);

    glShadeModel(GL\_SMOOTH);

    glMaterialfv(GL\_FRONT, GL\_SPECULAR, mat\_specular);   //指定材质属性

    glMaterialfv(GL\_FRONT, GL\_SHININESS, mat\_shininess); //镜面反射指数

    //设置light0--white

*GLfloat* white\_light[] = {1.0, 1.0, 1.0, 1.0};      //设置编号、特性、颜色

    glLightfv(GL\_LIGHT0, GL\_POSITION, light\_position); //设置位置

    glLightfv(GL\_LIGHT0, GL\_DIFFUSE, white\_light);

    glLightfv(GL\_LIGHT0, GL\_SPECULAR, white\_light);

    glEnable(GL\_LIGHT0);

    //设置light1--red

*GLfloat* red\_light[] = {1.0, 0.0, 0.0, 1.0};

    glLightfv(GL\_LIGHT1, GL\_POSITION, light\_position);

    glLightfv(GL\_LIGHT1, GL\_DIFFUSE, red\_light);

    glLightfv(GL\_LIGHT1, GL\_SPECULAR, red\_light);

    glEnable(GL\_LIGHT1);

    //设置light2--green

*GLfloat* green\_light[] = {0.0, 1.0, 0.0, 1.0};

    glLightfv(GL\_LIGHT2, GL\_POSITION, light\_position2);

    glLightfv(GL\_LIGHT2, GL\_DIFFUSE, green\_light);

    glLightfv(GL\_LIGHT2, GL\_SPECULAR, green\_light);

    glEnable(GL\_LIGHT2);

    //设置light3--blue

*GLfloat* blue\_light[] = {0.0, 0.0, 1.0, 1.0};

    glLightfv(GL\_LIGHT3, GL\_POSITION, light\_position3);

    glLightfv(GL\_LIGHT3, GL\_DIFFUSE, blue\_light);

    glLightfv(GL\_LIGHT3, GL\_SPECULAR, blue\_light);

    glEnable(GL\_LIGHT3);

    glEnable(GL\_LIGHTING);

    glEnable(GL\_DEPTH\_TEST);

}

### 3.3 纹理映射

**思想：**

我们将一个二维的图片（其格式不一定为BMP，因为我换了个支持格式更多的上传头像的头文件）映射到给出图形的表面来表示其纹理。我们首先建立一个Image类用来上传我们的图片，这里会有一些格式的限制。

具体实现纹理映射的原理和书上的一样，用一个矩阵颜色团来定义表面区域的纹理，其纹理空间的位置用(s,t)这个二维坐标来指定。该纹理空间数组的第一行列出矩图案底部的颜色值，而最后一行列出图案顶部的颜色值。并且，纹理空间(0,0)点对应数组第一行第一个位置，纹理空间(1,1)对应数组最后一行最后一个位置。

而纹理映射就是将纹理图案四角坐标(s,t)赋给场景的n个空间位置，并用线性变换将颜色值赋给指定空间的投影位置。这里我们选择使用纹理扫描，将纹理图案映射到像素坐标，其缺点是边界不匹配。

**实现：**

//纹理映射

//使用stb\_image让其支持的图片格式更多

#define STB\_IMAGE\_IMPLEMENTATION

#include "stb\_image.h"

*void* handleKeypress(*unsigned* *char* *key*, *int* *x*, *int* *y*)

{

    switch (*key*)

    {

    case 27: //Escape key

        exit(0);

    }

}

//image类储存加载到内存中的图片

*struct* *Image*

{

*int* width = 0;

*int* height = 0;

*int* nrChannels = 0;

*unsigned* *char* \*data = nullptr;

};

//图片加载函数

/\*\*

 \* *@brief* loadImage 将图片加载到内存

 \* *@param* *fileName*  图片文件名(相对路径)

 \* *@return*

 \*/

*Image* \*loadImage(const *char* \**fileName*)

{

*Image* \*image = new *Image*();

    //翻转图像, 否则图像是反的

    stbi\_set\_flip\_vertically\_on\_load(true);

    image->data = stbi\_load(*fileName*, &image->width, &image->height,

                            &image->nrChannels, 0);

    return image;

}

//纹理加载函数

/\*\*

 \* *@brief* loadTexture 将图片加载为纹理

 \* *@param* *image* 已经加载到内存中的图片

 \* *@return*

 \*/

*GLuint* loadTexture(*Image* \**image*)

{

*GLuint* textureId;

    //纹理初始化

    glGenTextures(1, &textureId);

    //选定当前要编辑的纹理

    glBindTexture(GL\_TEXTURE\_2D, textureId);

    if (*image*->nrChannels == 3)

    {

        glTexImage2D(GL\_TEXTURE\_2D,               //二维纹理永远是GL\_TEXTURE\_2D

                     0,                           //基本都是用0

                     GL\_RGB,                      //纹理储存格式

*image*->width, *image*->height, //图像宽高

                     0,                           //永远是0

                     GL\_RGB,                      //纹理数据格式, 必须与上头相同

                     GL\_UNSIGNED\_BYTE,            //纹理数据类型, 基本用GL\_UNSIGNED\_BYTE

*image*->data);                //图片data数组指针

    }

    else

    {

        //如果是PNG图像就有alpha通道, 用RGBA

        glTexImage2D(GL\_TEXTURE\_2D,               //二维纹理永远是GL\_TEXTURE\_2D

                     0,                           //基本都是用0

                     GL\_RGBA,                     //纹理储存格式

*image*->width, *image*->height, //图像宽高

                     0,                           //永远是0

                     GL\_RGBA,                     //纹理数据格式, 必须与上头相同

                     GL\_UNSIGNED\_BYTE,            //纹理数据类型, 基本用GL\_UNSIGNED\_BYTE

*image*->data);                //图片data数组指针

    }

    return textureId; //返回加载的纹理ID

}

*GLuint* \_textureId; //The id of the texture

*void* initRendering()

{

    glEnable(GL\_DEPTH\_TEST);

    glEnable(GL\_LIGHTING);

    glEnable(GL\_LIGHT0);

    glEnable(GL\_NORMALIZE);

    glEnable(GL\_COLOR\_MATERIAL);

    //加载纹理用的图片

    //支持绝大多数主流格式

    //注意图像分辨率不要太大(玄学错误), 否则纹理出错

*Image* \*image = nullptr;

    image = loadImage("1.bmp");

    \_textureId = loadTexture(image);

    delete image;

    image = nullptr;

    //  Image\* image = loadTexture("1.bmp");

    //  \_textureId = loadTexture(image);

    //  delete image;

}

*void* handleResize32(*int* *w*, *int* *h*)

{

    glViewport(0, 0, *w*, *h*);

    glMatrixMode(GL\_PROJECTION);

    glLoadIdentity();

    gluPerspective(45.0, (*float*)*w* / (*float*)*h*, 1.0, 200.0);

}

*void* drawScene()

{

    glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

    glMatrixMode(GL\_MODELVIEW);

    glLoadIdentity();

    glTranslatef(0.0f, 1.0f, -6.0f);

*GLfloat* ambientLight[] = {0.2f, 0.2f, 0.2f, 1.0f};

    glLightModelfv(GL\_LIGHT\_MODEL\_AMBIENT, ambientLight);

*GLfloat* directedLight[] = {0.7f, 0.7f, 0.7f, 1.0f};

*GLfloat* directedLightPos[] = {-10.0f, 15.0f, 20.0f, 0.0f};

    glLightfv(GL\_LIGHT0, GL\_DIFFUSE, directedLight);

    glLightfv(GL\_LIGHT0, GL\_POSITION, directedLightPos);

    glEnable(GL\_TEXTURE\_2D);

    glBindTexture(GL\_TEXTURE\_2D, \_textureId);

    //底部图形

    glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_NEAREST);

    glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_NEAREST);

    glColor3f(0.2f, 0.2f, 0.2f); //设置颜色

    glBegin(GL\_QUADS);

    glNormal3f(0.0, 1.0f, 0.0f);    //法向量

    glTexCoord2f(0.0f, 0.0f);       //纹理映射

    glVertex3f(-2.5f, -2.5f, 2.5f); //绘图

    glTexCoord2f(1.0f, 0.0f);

    glVertex3f(2.5f, -2.5f, 2.5f);

    glTexCoord2f(1.0f, 1.0f);

    glVertex3f(2.5f, -2.5f, -2.5f);

    glTexCoord2f(0.0f, 1.0f);

    glVertex3f(-2.5f, -2.5f, -2.5f);

    glEnd();

    //中间的三角形

    glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

    glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

    glColor3f(1.0f, 1.0f, 1.0f);

    glBegin(GL\_TRIANGLES);

    glNormal3f(0.0f, 0.0f, 1.0f);

    glTexCoord2f(0.0f, 0.0f);

    glVertex3f(-2.5f, -2.5f, -2.5f);

    glTexCoord2f(0.5f, 1.0f);

    glVertex3f(0.0f, 2.5f, -2.5f);

    glTexCoord2f(1.0f, 0.0f);

    glVertex3f(2.5f, -2.5f, -2.5f);

    glEnd();

    glutSwapBuffers();

}

## 4、可视化功能

### 4.1 天空盒+几何模型+视点变换+坐标变换+纹理映射（立方体+球体）

**思想：**

**光照**

设置灯光个数、视角位置、光源位置、颜色等一系列参数，然后启用灯光。这个很简单，也有几乎固定的代码，就不展开了。

**三维图形变换**

三维变换类似二维变换，将东西存入矩阵然后不断变换。举个例子，我们可以创建一个堆栈，首先存入旋转量，位移量，然后再创建图形（这里要主义的是，存入堆栈的顺序和实际运行的顺序是反着来的），最后再出栈。这里我们可以讲参数定义为变量，然后将参数的变化放在函数中，最后在main函数中调用glutIdleFunc（）；来让其在程序空闲的时候不断运行。

**纹理映射**

我们将一个二维的图片映射到给出图形的表面来表示其纹理。我们首先建立一个Image类用来上传我们的图片，这里书上的代码会有一些格式的限制，所以我们采用stb\_iamge让其支持的格式变得更多。

具体实现纹理映射的原理和书上的一样，用一个矩阵颜色团来定义表面区域的纹理，其纹理空间的位置用(s,t)这个二维坐标来指定。该纹理空间数组的第一行列出矩图案底部的颜色值，而最后一行列出图案顶部的颜色值。并且，纹理空间(0,0)点对应数组第一行第一个位置，纹理空间(1,1)对应数组最后一行最后一个位置。

而纹理映射就是将纹理图案四角坐标(s,t)赋给场景的n个空间位置，并用线性变换将颜色值赋给指定空间的投影位置。这里我们选择使用纹理扫描，将纹理图案映射到像素坐标，其缺点是边界不匹配。

我们所需要的就是创建一个图片上传函数，一个纹理上传函数，对我们建立的图像进行纹理的映射（这里的坐标不能映射错误，不然图片就会是反的。）

**天空盒**

天空盒的实现其实非常简单。首先先创建一个立方体贴图，这里我的做法是把立方体的六个面都弄上贴图，然后我们将视点放在我们建立的天空盒的内部。这样，只要我们的贴图顺序和位置合理，同时这个立方体足够大，我们所能看见的便是一个类似我们肉眼所见的世界的场景，十分逼真。

**键盘响应**

增加了键盘事件响应，只要按下wasd就会修改响应的视点方向，而按下前进后退键则会改变视点位置，从而能够自由变换视点方向和位置。

**实现：**

//可视化 天空盒+几何模型+视点变换+坐标变换

//视线参数化, 方便更改

static *GLfloat* xequalzero[] = {1.0, 1.0, 1.0, 1.0};

static *double* lookat[] = {0.0, -5.0, 7.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0};

static *double* perspective[] = {90.0, 1.0f, 1.0f, 50.0f};

static *GLfloat* \*currentCoeff;

static *GLenum* currentPlane;

static *GLint* currentGenMode;

static *float* roangles;

const *int* MAX = 20;       //最大纹理数量

*int* textureId[MAX] = {0}; //纹理ID:

//初始化绘制

*void* init41()

{

    glClearColor(0.0, 0.0, 0.0, 0);

    glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT); //清理颜色和深度缓存

    //创建透视效果视图

    glMatrixMode(GL\_PROJECTION);

    glLoadIdentity();

    gluPerspective(perspective[0], perspective[1],

                   perspective[2], perspective[3]);

    //定义视点:

    //头两个不动

    glMatrixMode(GL\_MODELVIEW);

    glLoadIdentity();

    gluLookAt(lookat[0], lookat[1], lookat[2],

              lookat[3], lookat[4], lookat[5],

              lookat[6], lookat[7], lookat[8]);

    //启用深度测试:

    //在绘制3D物体时必要, 否则无法判断消隐关系

    glEnable(GL\_DEPTH\_TEST);

    //定义默认光源:

    {

        //光源位置, 默认在z轴从上到下的光

*GLfloat* sun\_light\_position[] = {0.0f, 0.0f, 30.0f, 1.0f}; //光源的位置在世界坐标系圆心，齐次坐标形式

        //RGBA模式的环境光

*GLfloat* sun\_light\_ambient[] = {0.1f, 0.1f, 0.1f, 1.0f};

        //RGBA模式的漫反射光，全白光

*GLfloat* sun\_light\_diffuse[] = {1.0f, 1.0, 1.0, 1.0f};

        //RGBA模式下的镜面光 ，全白光

*GLfloat* sun\_light\_specular[] = {1.0f, 1.0, 1.0, 1.0f};

        //应用光照:

        glLightfv(GL\_LIGHT0, GL\_POSITION, sun\_light\_position);

        glLightfv(GL\_LIGHT0, GL\_AMBIENT, sun\_light\_ambient);

        glLightfv(GL\_LIGHT0, GL\_DIFFUSE, sun\_light\_diffuse);

        glLightfv(GL\_LIGHT0, GL\_SPECULAR, sun\_light\_specular);

        //开启灯光

        glEnable(GL\_LIGHT0);

        glEnable(GL\_LIGHTING);

    }

    //开启法向量自动重置

    glEnable(GL\_NORMALIZE);

    //开启纹理&颜色叠加

    glEnable(GL\_COLOR\_MATERIAL);

    //加载纹理用的图片

*Image* \*image = nullptr;

    image = loadImage("front.jpg");

    textureId[0] = loadTexture(image);

    delete image;

    image = nullptr;

    image = loadImage("back.jpg");

    textureId[1] = loadTexture(image);

    delete image;

    image = nullptr;

    image = loadImage("bottom.jpg");

    textureId[2] = loadTexture(image);

    delete image;

    image = nullptr;

    image = loadImage("top.jpg");

    textureId[3] = loadTexture(image);

    delete image;

    image = nullptr;

    image = loadImage("right.jpg");

    textureId[4] = loadTexture(image);

    delete image;

    image = nullptr;

    image = loadImage("left.jpg");

    textureId[5] = loadTexture(image);

    delete image;

    image = nullptr;

    image = loadImage("universe.jpg");

    textureId[6] = loadTexture(image);

    delete image;

    image = nullptr;

    return;

}

//x,y,z轴旋转角度分量

*GLfloat* xtri = 0;

*GLfloat* ytri = 0;

*GLfloat* ztri = 0;

*GLfloat* xtri2 = 0;

//全局物体旋转方向:

*float* xrotate = 0.0;

*float* yrotate = 0.0;

*float* zrotate = 0.0;

*GLuint* texName;

#define SOLID 1

#define WIRE 2

*void* drawBall(*double* *R*, *double* *x*, *double* *y*, *double* *z*, *double* *angle*, *double* *x2*, *double* *y2*, *double* *z2*, *int* *MODE* = SOLID)

{

    const *int* PREC = 64;

    glPushMatrix();

    glRotatef(*angle*, *x2*, *y2*, *z2*);

    glTranslated(*x*, *y*, *z*);

    glScalef(*R*, *R*, *R*);

    if (*MODE* == SOLID)

    {

        glutSolidSphere(0.5, PREC, PREC);

    }

    else if (*MODE* == WIRE)

    {

        glutWireSphere(0.5, PREC, PREC);

    }

    glPopMatrix();

    return;

}

//显示调用函数

*void* display()

{

    //清除当前颜色&深度缓存

    glClearColor(0.0, 0.0, 0.0, 0);

    glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

    //在display中更新视点:

    glMatrixMode(GL\_MODELVIEW);

    glLoadIdentity();

    //第一组eyex, eyey,eyez 相机在世界坐标的位置

    //第二组centerx,centery,centerz 相机镜头对准的物体在世界坐标的位置

    //第三组upx,upy,upz 相机向上的方向在世界坐标中的方向

    gluLookAt(lookat[0], lookat[1], lookat[2],

              lookat[3], lookat[4], lookat[5],

              lookat[6], lookat[7], lookat[8]);

    //矩阵旋转, 实现物体绕焦点旋转

    //绘制所有物体必须加上这个

    glRotatef(xrotate, 1, 0, 0);

    glRotatef(yrotate, 0, 1, 0);

    glRotatef(zrotate, 0, 0, 1);

    //  添加物体:

    {

        //这是开启glEnable(GL\_COLOR\_MATERIAL)之后设置材质颜色的方法:

        //具体使用glColor4f 还是 glColor3f 根据main中的设置而定

        glColorMaterial(GL\_FRONT, GL\_AMBIENT);

        glColor4f(0.0f, 0.0f, 1.0f, 1.0f);

        glColorMaterial(GL\_FRONT, GL\_DIFFUSE);

        glColor4f(0.0f, 0.0f, 1.0f, 1.0f);

        glColorMaterial(GL\_FRONT, GL\_SPECULAR);

        glColor4f(1.0f, 1.0, 1.0, 1.0f);

        glColorMaterial(GL\_FRONT, GL\_EMISSION);

        glColor4f(0.0f, 0.0f, 0.0f, 1.0f);

        //上头只能设置4个属性

        //材质的镜面指数依然需要使用glMaterialf设置

*GLfloat* mat\_shininess = 32.0f;

        glMaterialf(GL\_FRONT, GL\_SHININESS, mat\_shininess);

        //控制物体三维几何变换 (组合体适用)

        //使用全局变量, 便于在Idle中修改

        glPushMatrix();

        ////        glRotatef(ztri,0.0f,0.0f,1.0f);                // 绕Z轴旋转

        ////        glTranslatef (2.0,0.0,0.0);

        //

        //      //画一个立方体

        ////        glutSolidCube (2);

        //      //glRotatef(xtri,1.0f,0.0f,0.0f);        // 绕X轴旋转

        //

        //      glRotatef(ytri,0.0f,1.0f,0.0f);          // 绕Y轴旋转

        //      glTranslated (3.0,0.0,0.0);

        //      glColor3f(1.0,0.0,0.0); //设置颜色

        //      glBindTexture(GL\_TEXTURE\_2D,textureId[0]);

        //      //画一个球

        //      glutSolidSphere (0.5,20,20);

        //

        //      glRotatef(ytri,0.0f,1.0f,0.0f);          // 绕Y轴旋转

        //      glTranslated (5.0,0.0,0.0);

        //      glColor3f(1.0,0.0,0.0); //设置颜色

        //      //画一个球

        //      glutSolidSphere (0.5,20,20);

        //

        //      glRotatef(ytri,0.0f,1.0f,0.0f);          // 绕Y轴旋转

        //      glTranslated (7.0,0.0,0.0);

        //      glColor3f(1.0,0.0,0.0); //设置颜色

        //      //画一个球

        //      glutSolidSphere (0.5,20,20);

        //

        //

        //      glRotatef(ytri,0.0f,1.0f,0.0f);                // 绕Y轴旋转

        //      glTranslated (3.0,0.0,0.0);

        //      glColor3f(0.0,1.0,0.0); //设置颜色

        //      glutSolidCube (2);

        glPopMatrix();

    }

    //添加贴图

*GLfloat* len = 20.0f;

*GLfloat* \_len = -20.0f;

    //前面

    glEnable(GL\_TEXTURE\_2D);

    {

        //设置当前操作的纹理

        glBindTexture(GL\_TEXTURE\_2D, textureId[0]);

        glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

        glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

        //颜色叠加为白色, 即保留原色彩不变

        glColor3f(1.0f, 1.0f, 1.0f);

        //绘制四边形

        glBegin(GL\_QUADS);

        {

            //确定绘制平面法向量

            glNormal3f(0.0, 1.0f, 0.0f);

            //纹理映射, 0~1, 超过将复制平铺

            glTexCoord2f(0.0f, 0.0f);

            glVertex3f(\_len, \_len, len); // 纹理和四边形的左下

            glTexCoord2f(1.0f, 0.0f);

            glVertex3f(len, \_len, len); // 纹理和四边形的右下

            glTexCoord2f(1.0f, 1.0f);

            glVertex3f(len, len, len); // 纹理和四边形的右上

            glTexCoord2f(0.0f, 1.0f);

            glVertex3f(\_len, len, len); // 纹理和四边形的左上

        }

        glEnd();

    }

    glDisable(GL\_TEXTURE\_2D);

    //后面

    glEnable(GL\_TEXTURE\_2D);

    {

        //设置当前操作的纹理

        glBindTexture(GL\_TEXTURE\_2D, textureId[1]);

        glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

        glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

        //颜色叠加为白色, 即保留原色彩不变

        glColor3f(1.0f, 1.0f, 1.0f);

        //绘制四边形

        glBegin(GL\_QUADS);

        {

            //确定绘制平面法向量

            glNormal3f(0.0, 1.0f, 0.0f);

            //纹理映射, 0~1, 超过将复制平铺

            glTexCoord2f(1.0f, 0.0f);

            glVertex3f(\_len, \_len, \_len); // 纹理和四边形的右下

            glTexCoord2f(1.0f, 1.0f);

            glVertex3f(\_len, len, \_len); // 纹理和四边形的右上

            glTexCoord2f(0.0f, 1.0f);

            glVertex3f(len, len, \_len); // 纹理和四边形的左上

            glTexCoord2f(0.0f, 0.0f);

            glVertex3f(len, \_len, \_len); // 纹理和四边形的左下

        }

        glEnd();

    }

    glDisable(GL\_TEXTURE\_2D);

    //底面

    glEnable(GL\_TEXTURE\_2D);

    {

        //设置当前操作的纹理

        glBindTexture(GL\_TEXTURE\_2D, textureId[3]);

        glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

        glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

        //颜色叠加为白色, 即保留原色彩不变

        glColor3f(1.0f, 1.0f, 1.0f);

        //绘制四边形

        glBegin(GL\_QUADS);

        {

            //确定绘制平面法向量

            glNormal3f(0.0, 1.0f, 0.0f);

            //纹理映射, 0~1, 超过将复制平铺

            glTexCoord2f(1.0f, 0.0f);

            glVertex3f(\_len, len, \_len); // 纹理和四边形的左上

            glTexCoord2f(1.0f, 1.0f);

            glVertex3f(\_len, len, len); // 纹理和四边形的左下

            glTexCoord2f(0.0f, 1.0f);

            glVertex3f(len, len, len); // 纹理和四边形的右下

            glTexCoord2f(0.0f, 0.0f);

            glVertex3f(len, len, -len); // 纹理和四边形的右上

        }

        glEnd();

    }

    glDisable(GL\_TEXTURE\_2D);

    //顶面

    glEnable(GL\_TEXTURE\_2D);

    {

        //设置当前操作的纹理

        glBindTexture(GL\_TEXTURE\_2D, textureId[2]);

        glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

        glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

        //颜色叠加为白色, 即保留原色彩不变

        glColor3f(1.0f, 1.0f, 1.0f);

        //绘制四边形

        glBegin(GL\_QUADS);

        {

            //确定绘制平面法向量

            glNormal3f(0.0, 1.0f, 0.0f);

            //纹理映射, 0~1, 超过将复制平铺

            glTexCoord2f(1.0f, 1.0f);

            glVertex3f(\_len, \_len, \_len); // 纹理和四边形的右上

            glTexCoord2f(0.0f, 1.0f);

            glVertex3f(len, \_len, \_len); // 纹理和四边形的左上

            glTexCoord2f(0.0f, 0.0f);

            glVertex3f(len, \_len, len); // 纹理和四边形的左下

            glTexCoord2f(1.0f, 0.0f);

            glVertex3f(\_len, \_len, len); // 纹理和四边形的右下

        }

        glEnd();

    }

    glDisable(GL\_TEXTURE\_2D);

    //右面

    glEnable(GL\_TEXTURE\_2D);

    {

        //设置当前操作的纹理

        glBindTexture(GL\_TEXTURE\_2D, textureId[5]);

        glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

        glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

        //颜色叠加为白色, 即保留原色彩不变

        glColor3f(1.0f, 1.0f, 1.0f);

        //绘制四边形

        glBegin(GL\_QUADS);

        {

            //确定绘制平面法向量

            glNormal3f(0.0, 1.0f, 0.0f);

            //纹理映射, 0~1, 超过将复制平铺

            glTexCoord2f(1.0f, 0.0f);

            glVertex3f(len, \_len, \_len); // 纹理和四边形的右下

            glTexCoord2f(1.0f, 1.0f);

            glVertex3f(len, len, \_len); // 纹理和四边形的右上

            glTexCoord2f(0.0f, 1.0f);

            glVertex3f(len, len, len); // 纹理和四边形的左上

            glTexCoord2f(0.0f, 0.0f);

            glVertex3f(len, \_len, len); // 纹理和四边形的左下

        }

        glEnd();

    }

    glDisable(GL\_TEXTURE\_2D);

    //左面

    glEnable(GL\_TEXTURE\_2D);

    {

        //设置当前操作的纹理

        glBindTexture(GL\_TEXTURE\_2D, textureId[4]);

        glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

        glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

        //颜色叠加为白色, 即保留原色彩不变

        glColor3f(1.0f, 1.0f, 1.0f);

        //绘制四边形

        glBegin(GL\_QUADS);

        {

            //确定绘制平面法向量

            glNormal3f(0.0, 1.0f, 0.0f);

            //纹理映射, 0~1, 超过将复制平铺

            glTexCoord2f(0.0f, 0.0f);

            glVertex3f(\_len, \_len, \_len); // 纹理和四边形的左下

            glTexCoord2f(1.0f, 0.0f);

            glVertex3f(\_len, \_len, len); // 纹理和四边形的右下

            glTexCoord2f(1.0f, 1.0f);

            glVertex3f(\_len, len, len); // 纹理和四边形的右上

            glTexCoord2f(0.0f, 1.0f);

            glVertex3f(\_len, len, \_len); // 纹理和四边形的左上

        }

        glEnd();

    }

    glDisable(GL\_TEXTURE\_2D);

    //  球体贴图

    //  贴图一定要长, 不然会出现拉伸等情况

    //  启用2D纹理

    glEnable(GL\_TEXTURE\_2D);

    {

        //设置当前操作的纹理

        glBindTexture(GL\_TEXTURE\_2D, textureId[6]);

        //设置环绕方式

        glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT);

        glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT);

        //设置纹理过滤

        glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

        glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

        //颜色叠加为白色, 即保留原色彩不变

        glColor4f(1.0f, 1.0f, 1.0f, 1.0f);

        glTexEnvf(GL\_TEXTURE\_ENV, GL\_TEXTURE\_ENV\_MODE, GL\_REPLACE);

        currentCoeff = xequalzero;

        currentGenMode = GL\_OBJECT\_LINEAR;

        currentPlane = GL\_OBJECT\_PLANE;

        //自动生成纹理坐标, 而不需要使用glTexCoord显式分配

        glTexGeni(GL\_S, GL\_TEXTURE\_GEN\_MODE, currentGenMode);

        glTexGeni(GL\_T, GL\_TEXTURE\_GEN\_MODE, currentGenMode);

        glTexGenfv(GL\_S, currentPlane, currentCoeff);

        //启动自动生成纹理

        //不启用后头的glTexGen无法使用

        glEnable(GL\_TEXTURE\_GEN\_S);

        glEnable(GL\_TEXTURE\_GEN\_T);

        drawBall(3, 0, 0, 0, xtri, 0, 0, 1);

        //关闭自动生成纹理

        //不然影响其他手动贴图

        glDisable(GL\_TEXTURE\_GEN\_S);

        glDisable(GL\_TEXTURE\_GEN\_T);

    }

    glDisable(GL\_TEXTURE\_2D);

    glEnable(GL\_TEXTURE\_2D);

    {

        //设置当前操作的纹理

        glBindTexture(GL\_TEXTURE\_2D, textureId[6]);

        //设置环绕方式

        glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT);

        glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT);

        //设置纹理过滤

        glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

        glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

        //颜色叠加为白色, 即保留原色彩不变

        glColor4f(1.0f, 1.0f, 1.0f, 1.0f);

        glTexEnvf(GL\_TEXTURE\_ENV, GL\_TEXTURE\_ENV\_MODE, GL\_REPLACE);

        currentCoeff = xequalzero;

        currentGenMode = GL\_OBJECT\_LINEAR;

        currentPlane = GL\_OBJECT\_PLANE;

        //自动生成纹理坐标, 而不需要使用glTexCoord显式分配

        glTexGeni(GL\_S, GL\_TEXTURE\_GEN\_MODE, currentGenMode);

        glTexGeni(GL\_T, GL\_TEXTURE\_GEN\_MODE, currentGenMode);

        glTexGenfv(GL\_S, currentPlane, currentCoeff);

        //启动自动生成纹理

        //不启用后头的glTexGen无法使用

        glEnable(GL\_TEXTURE\_GEN\_S);

        glEnable(GL\_TEXTURE\_GEN\_T);

        drawBall(3, 5.0, 0, 0, ytri, 0, 1.0f, 0);

        //关闭自动生成纹理

        //不然影响其他手动贴图

        glDisable(GL\_TEXTURE\_GEN\_S);

        glDisable(GL\_TEXTURE\_GEN\_T);

    }

    glDisable(GL\_TEXTURE\_2D);

    //刷新缓冲区

    glutSwapBuffers();

    return;

}

//空闲调用函数

*void* myIdle(*void*)

{

    //执行画面更新任务

    //  xtri+=0.5f;

    //  ytri+=1;

    //  ztri+=1;

    xtri += 1;

    ytri += 1;

    ztri += 0;

    xtri2 += 1;

    if (xtri2 >= 10)

    {

        xtri2 = 0;

    }

    //窗口重绘函数:

    //通常用glutPostRedisplay, 其更为智能一点

    glutPostRedisplay();

    //通常需要一个sleep来防止帧率过高, 这里sleep 16ms, 约等于60fps

    Sleep(16);

    return;

}

//窗口尺寸重置函数:

//当窗口大小变化时调用, 保证图形不变形

*void* handleResize(*int* *w*, *int* *h*)

{

    glViewport(0, 0, *w*, *h*);

    glMatrixMode(GL\_PROJECTION);

    glLoadIdentity();

    perspective[1] = (*float*)*w* / (*float*)*h*;

    gluPerspective(perspective[0], perspective[1],

                   perspective[2], perspective[3]);

    return;

}

//键盘响应函数:

*void* keyboradAck(*unsigned* *char* *key*, *int* *x*, *int* *y*)

{

    printf("%d=%c, x=%d, y=%d\n", *key*, *key*, *x*, *y*);

    static *double* flag = 1.0;

    switch (*key*)

    {

        //控制视角以当前视点为中心旋转

    case 'w':

    {

        printf("up\n");

        xrotate += 2;

        break;

    }

    case 's':

    {

        printf("down\n");

        xrotate -= 2;

        break;

    }

    case 'a':

    {

        printf("left\n");

        yrotate += 2;

        break;

    }

    case 'd':

    {

        printf("right\n");

        yrotate -= 2;

        break;

    }

    case 'q':

    {

        printf("q\n");

        zrotate += 2;

        break;

    }

    case 'e':

    {

        printf("e\n");

        zrotate -= 2;

        break;

    }

    default:

    {

    }

    }

    return;

}

*void* specialKeyAck(*int* *key*, *int* *x*, *int* *y*)

{

    printf("%d=%c, x=%d, y=%d\n", *key*, *key*, *x*, *y*);

    //控制前进后退

    if (*key* == GLUT\_KEY\_UP)

    {

        printf("UP\n");

        //      lookat[2] += 2.0f;

        //      lookat[0] += 2.0f;

        lookat[2] -= 0.5f;

    }

    else if (*key* == GLUT\_KEY\_DOWN)

    {

        printf("DW\n");

        //      lookat[2] -= 2.0f;

        //      lookat[0] -= 2.0f;

        lookat[2] += 0.5f;

    }

    return;

}

## 5、菜单功能

**思想：**

这个就蛮简单的，首先设置一个主菜单，里面写上“请选择”之类的话，然后添加子菜单，每个菜单里放上我们需要的内容。

在我这个作业中，因为我的设计是一个多窗口的程序，所以每个子菜单选择后会弹出一个新窗口，这时候我们就需要事先创建一个操作界面用来右键加载菜单界面供我们选择。

**实现：**

//----------------------------------------------------------------------------------

//菜单

//操作界面

*void* draw()

{

    glClearColor(0, 0, 0, 0);

    glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

    glColor3f(0, 1, 0);

    glOrtho(-1, 1, -1, 1, -1, 1);

    glutSwapBuffers();

}

//点击某个条目时，输出该条目属于哪个菜单

*void* getcurrentmenu()

{

*int* nmenu;

    nmenu = glutGetMenu();

    if (nmenu == menu)

        printf("请选择.\n");

}

//第一个子菜单    几何建模功能

*void* submenufunc1(*int* *data*)

{

    getcurrentmenu();

    switch (*data*)

    {

    case 1:

        printf("DDA画线.\n");

        setNewWindow();

        glutCreateWindow("line\_DDA"); //设置窗口标题

        init();

        glutDisplayFunc(displayLineDDA);

        glutMainLoop();

        break;

    case 2:

        printf("中点画线.\n");

        setNewWindow();

        glutCreateWindow("line\_Midpoint"); //设置窗口标题

        init();

        glutDisplayFunc(displayLineMidpoint);

        glutMainLoop();

        break;

    case 3:

        printf("Bresenham画线.\n");

        setNewWindow();

        glutCreateWindow("line\_Bresenham"); //设置窗口标题

        init();

        glutDisplayFunc(displayLineBres);

        glutMainLoop();

        break;

    case 4:

        printf("中点画圆.\n");

        setNewWindow();

        glutCreateWindow("circle\_Midpoint"); //设置窗口标题

        init();

        glutDisplayFunc(displayCircleMidpoint);

        glutMainLoop();

        break;

    case 5:

        printf("B样条曲线.\n");

        //初始化并显示到屏幕中央

        glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

        glutInitWindowPosition(600, 100); //指定窗口位置

        glutInitWindowSize(600, 600);     //指定窗口大小

        glutCreateWindow("鼠标单机画点，多个点自动拟合曲线");

        glClearColor(1, 1, 1, 0);

        glShadeModel(GL\_FLAT);

        glutDisplayFunc(display3B);

        glutReshapeFunc(Reshape3B);

        glutMouseFunc(mouse3B);

        glutKeyboardFunc(keyboard3B);

        glutMainLoop();

        break;

    case 6:

        printf("Liang-Barsky 线段裁剪算法.\n");

        glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);   //设置显示模式为单缓冲，RGB模式

        glutInitWindowPosition(600, 100);              //设置窗口位置

        glutInitWindowSize(windowWidge, windowHeight); //设置窗口大小

        glutCreateWindow("Liang-Barsky 线段裁剪算法"); //设置窗口标题

        init16();

        glutDisplayFunc(display16);

        glutMainLoop();

        break;

    case 7:

        printf("Sutherland-Hodgman多边形裁剪算法.\n");

        glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);          //设置显示模式为单缓冲，RGB模式

        glutInitWindowPosition(600, 100);                     //设置窗口位置

        glutInitWindowSize(windowWidge, windowHeight);        //设置窗口大小

        glutCreateWindow("Sutherland-Hodgman多边形裁剪算法"); //设置窗口标题

        init17();

        glutDisplayFunc(display17);

        glutMainLoop();

        break;

    }

}

//第二个子菜单    几何变换功能

*void* submenufunc2(*int* *data*)

{

    getcurrentmenu();

    switch (*data*)

    {

    case 1:

        printf("时钟.\n");

        setNewWindow();

        glutInitWindowSize(600, 600); //设置窗口的宽高

        glutCreateWindow("时钟");

        gluOrtho2D(0, 600.0, 0.0, 600.0); //设置坐标系

        initClock();                      //初始化

        glutDisplayFunc(displayClock);

        glutTimerFunc(1000, timerFunc, 1); //每一秒执行一次

        glutMainLoop();

        break;

    case 2:

        printf("小球移动缩放.\n");

        setNewWindow();

        glutCreateWindow("小球移动缩放");

        Init22();

        glutDisplayFunc(&Display22);

        glutReshapeFunc(&Reshape22);

        glutIdleFunc(&myIdle22);

        glutMainLoop();

        break;

    }

}

//第三个子菜单    光源、材质和纹理映射功能

*void* submenufunc3(*int* *data*)

{

    getcurrentmenu();

    switch (*data*)

    {

    case 1:

        printf("单光源.\n");

        setNewWindow();

        glutCreateWindow("单光源");

        Initsinglelight();

        glutDisplayFunc(&Display22);

        glutReshapeFunc(&Reshape22);

        glutIdleFunc(&myIdle22);

        glutMainLoop();

        break;

    case 2:

        printf("多光源叠加.\n");

        setNewWindow();

        glutCreateWindow("多光源叠加");

        Init22();

        glutDisplayFunc(&Display22);

        glutReshapeFunc(&Reshape22);

        glutIdleFunc(&myIdle22);

        glutMainLoop();

        break;

    case 3:

        printf("纹理映射.\n");

        glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGB | GLUT\_DEPTH);

        glutInitWindowPosition(600, 100); //设置窗口位置

        glutInitWindowSize(400, 400);

        glutCreateWindow("纹理映射");

        initRendering();

        glutDisplayFunc(drawScene);

        glutKeyboardFunc(handleKeypress);

        glutReshapeFunc(handleResize32);

        glutMainLoop();

        break;

    }

}

//第四个子菜单    可视化功能

*void* submenufunc4(*int* *data*)

{

    getcurrentmenu();

    switch (*data*)

    {

    case 1:

        printf("天空盒+几何模型+视点变换+坐标变换+纹理映射（立方体+球体）.\n");

        glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGBA | GLUT\_DEPTH); //设置显示模式为单缓冲，RGB模式

        //设置窗口位置

        glutInitWindowPosition(600, 100);

        //设置窗口大小

        glutInitWindowSize(windowWidge, windowHeight);

        //设置窗口标题

        glutCreateWindow("上下控制移动，wasd控制方向");

        //初始化glut并绘制画面:

        init41();

        glutDisplayFunc(display);

        //注册键盘响应函数

        glutKeyboardFunc(keyboradAck);

        glutSpecialFunc(specialKeyAck);

        //空闲时自动调用函数:

        glutIdleFunc(myIdle);

        //窗口尺寸变化函数:

        glutReshapeFunc(handleResize);

        glutMainLoop();

        break;

    }

}

//主菜单

*void* menufunc(*int* *data*)

{

    getcurrentmenu();

    switch (*data*)

    {

    case 1:

        return;

        break;

    }

}

//----------------------------------------------------------------------------------

*int* main(*int* *argc*, *char* \**argv*[])

{

    glutInit(&*argc*, *argv*);

    glutInitDisplayMode(GLUT\_RGB | GLUT\_DOUBLE);

    glutInitWindowPosition(0, 100);

    glutInitWindowSize(512, 512);

    glutCreateWindow("需要菜单请在此黑色区域右键！！！！！！弹窗请勿关闭！！！！！！！");

    glutDisplayFunc(draw);

    //构建子菜单1的内容

    submenu1 = glutCreateMenu(submenufunc1);

    glutAddMenuEntry("DDA画线", 1);

    glutAddMenuEntry("中点画线", 2);

    glutAddMenuEntry("Bresenham画线", 3);

    glutAddMenuEntry("中点画圆", 4);

    glutAddMenuEntry("B样条画曲线", 5);

    glutAddMenuEntry("Liang-Barsky 线段裁剪算法", 6);

    glutAddMenuEntry("Sutherland-Hodgman多边形裁剪算法", 7);

    //构建子菜单2的内容

    submenu2 = glutCreateMenu(submenufunc2);

    glutAddMenuEntry("时钟", 1);

    glutAddMenuEntry("小球移动缩放", 2);

    //构建子菜单3的内容

    submenu3 = glutCreateMenu(submenufunc3);

    glutAddMenuEntry("单光源", 1);

    glutAddMenuEntry("多光源叠加", 2);

    glutAddMenuEntry("纹理映射", 3);

    //构建子菜单4的内容

    submenu4 = glutCreateMenu(submenufunc4);

    glutAddMenuEntry("天空盒+几何模型+视点变换+坐标变换+纹理映射（立方体+球体）", 1);

    //构建主菜单的内容

    menu = glutCreateMenu(menufunc);

    glutAddMenuEntry("请选择-注意：弹窗请勿关闭", 1);

    //将两个菜单变为另一个菜单的子菜单

    glutAddSubMenu("几何建模功能", submenu1);

    glutAddSubMenu("几何变换功能", submenu2);

    glutAddSubMenu("光源、材质和纹理映射功能", submenu3);

    glutAddSubMenu("可视化功能", submenu4);

    //点击鼠标右键时显示菜单

    glutAttachMenu(GLUT\_RIGHT\_BUTTON);

    glutMainLoop();

    return 0;

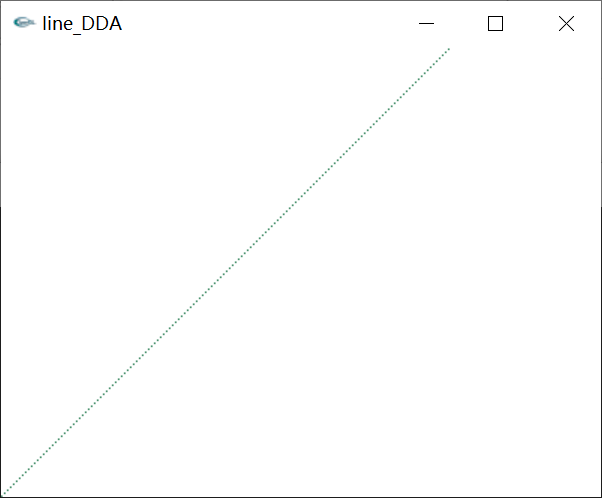
}

# 三、结果分析：

## 1、几何建模功能

### 1.1 DDA画线

**结果：**

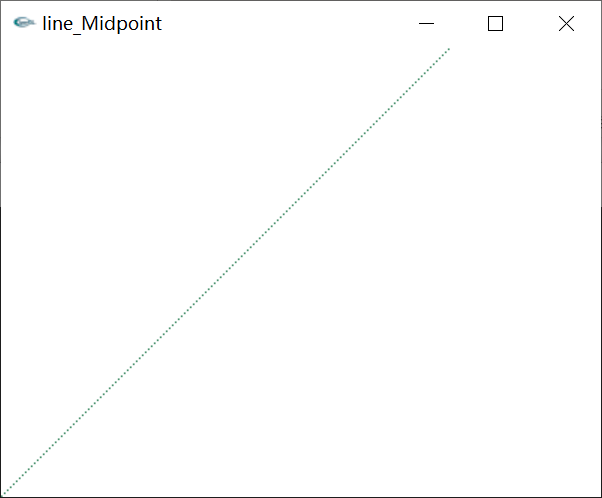


**分析：**

显示正常

### 1.2 中点画线

**结果：**

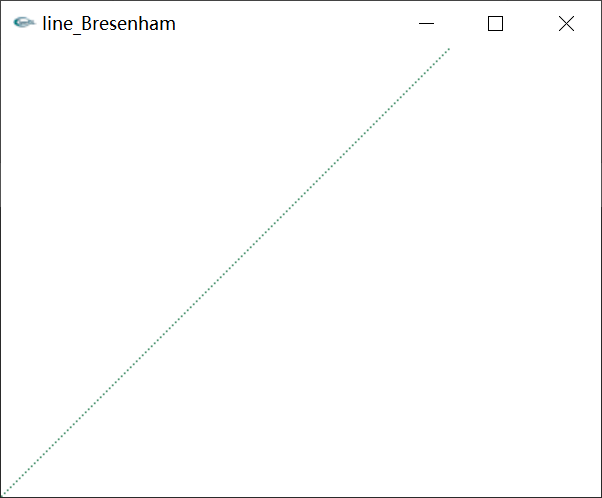


**分析：**

显示正常

### 1.3 Bresenham画线

**结果：**

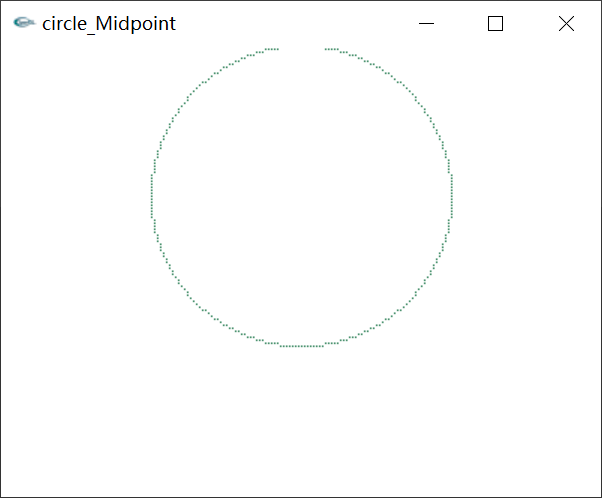


**分析：**

显示正常

### 1.4 中点画圆

**结果：**

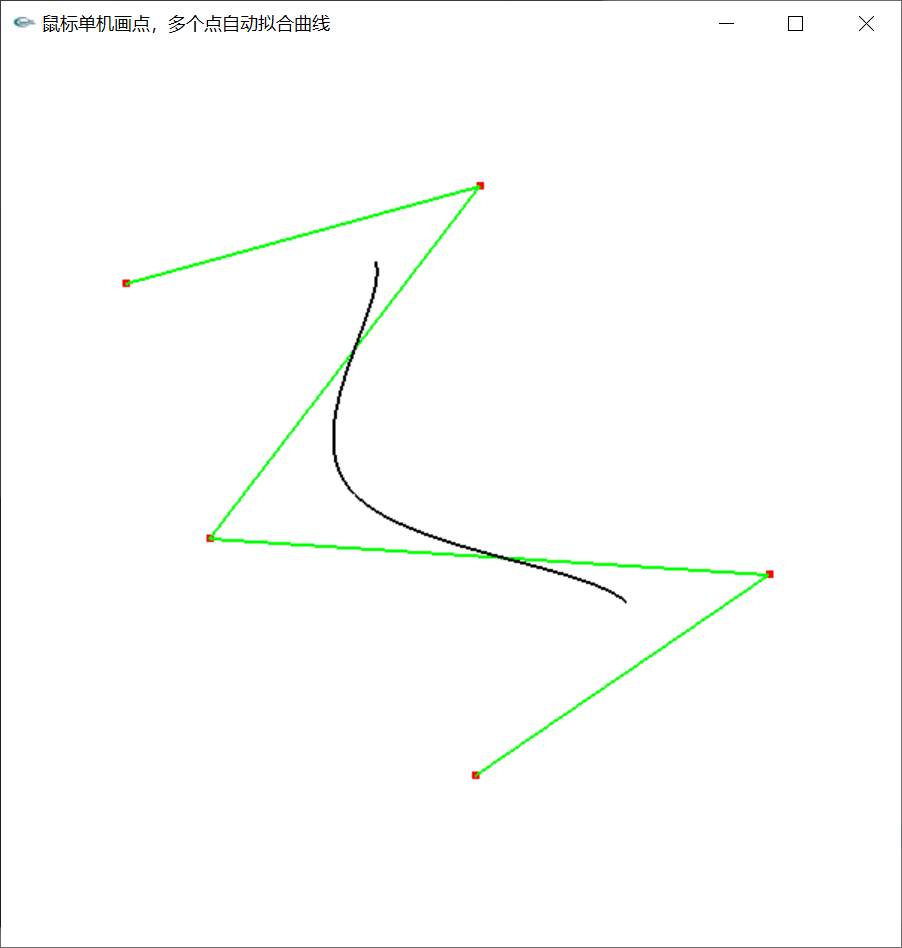


**分析：**

显示正常，不过我给的圆心坐标有点偏上。

### 1.5 B样条画曲线

**结果：**

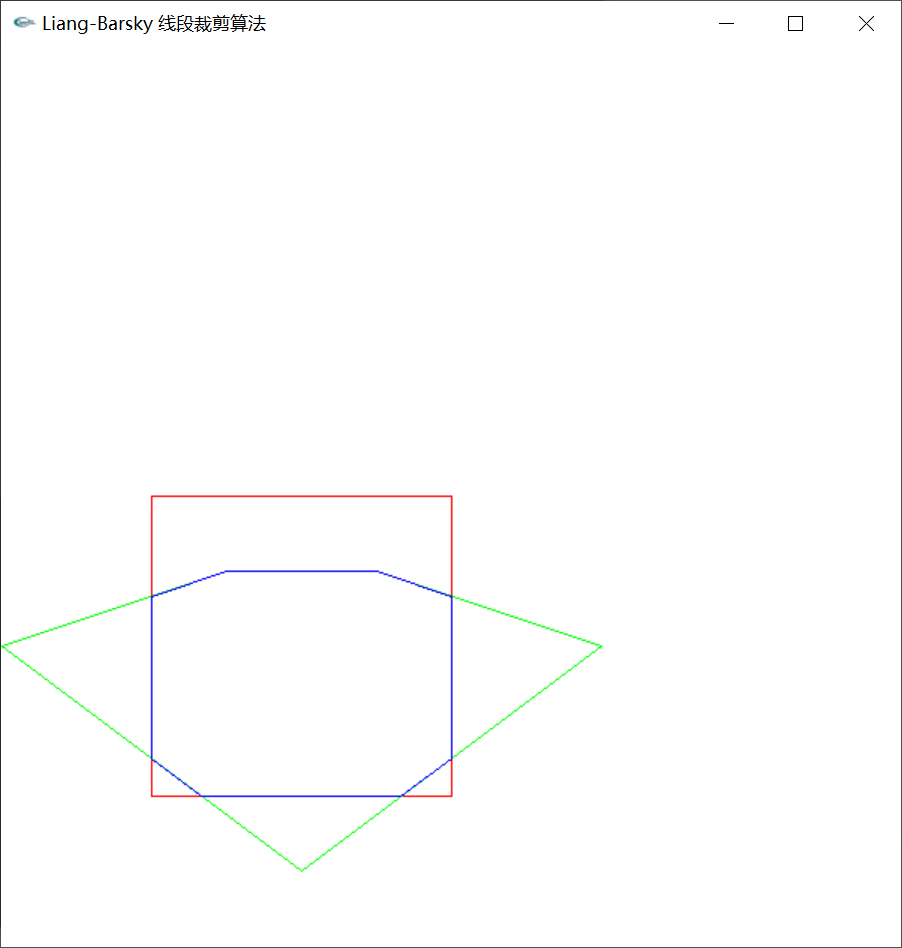


**分析：**

显示正常，根据给出的点会自动拟合B样条曲线。

### 1.6 Liang-Barsky 线段裁剪算法

**结果：**

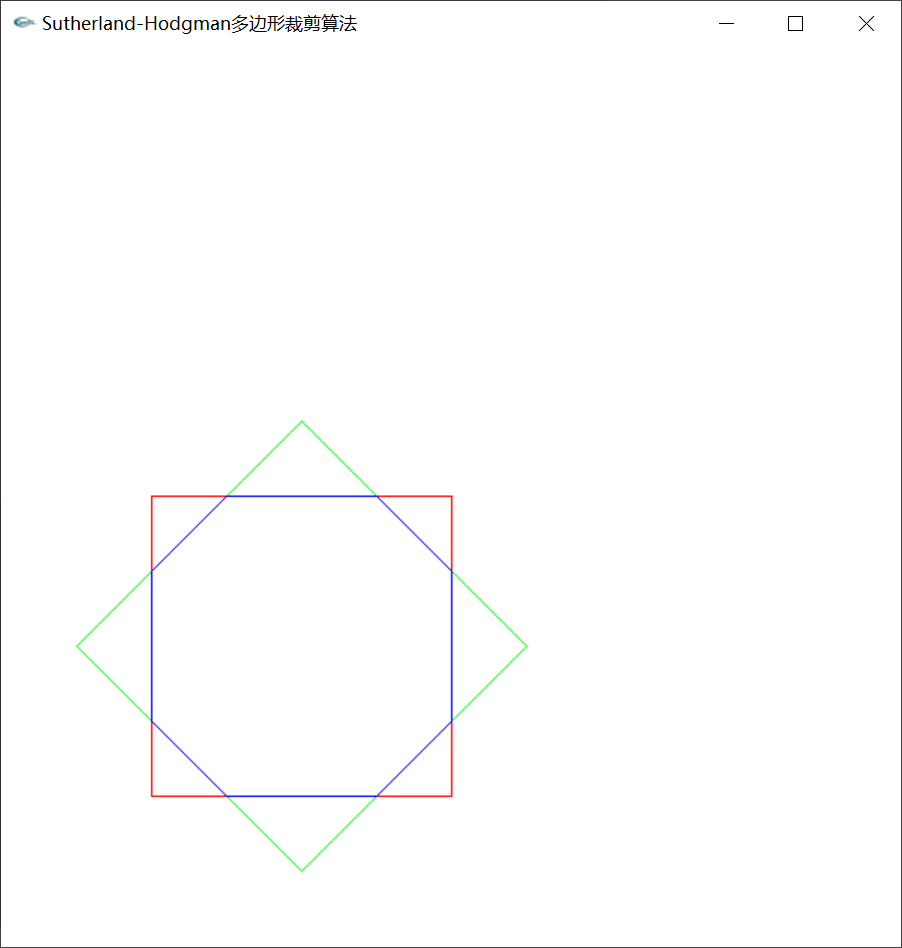


**分析：**

显示正常，裁剪完美，不过图像位置有那么一点点偏，问题不大。

### 1.7 Sutherland-Hodgman多边形裁剪算法

**结果：**

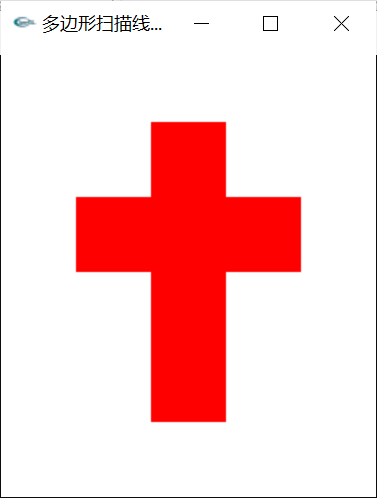


**分析：**

显示正常，裁剪完美，图像位置有点小偏。

### 1.8 多边形扫描填充算法

**结果：**



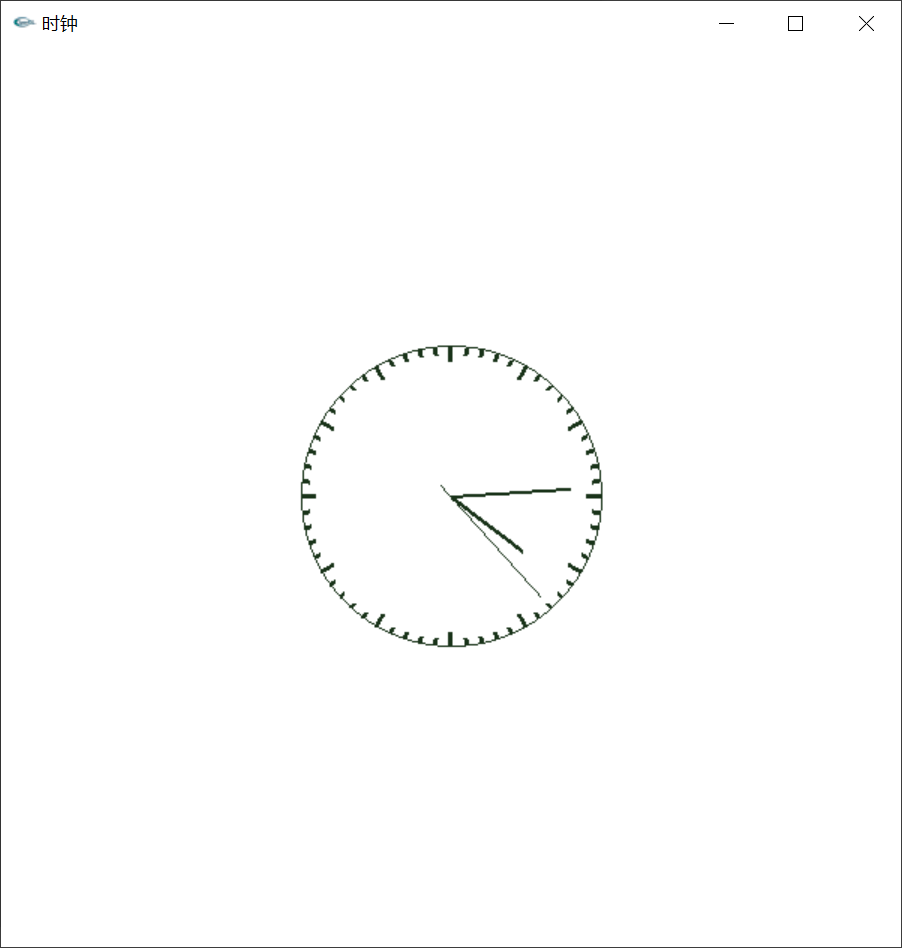
**分析：**

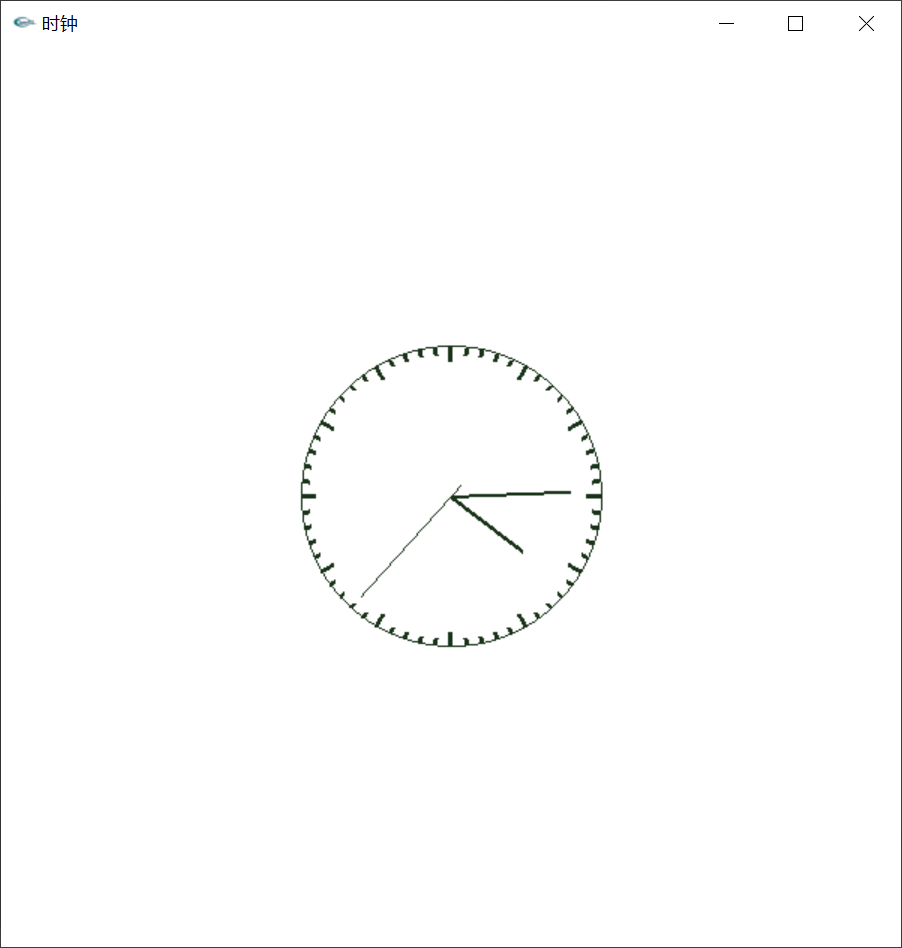
显示正常，结果正常。但是要注意画点的顺序（在这里我之前耗费了很多时间）。

## 2、几何变换功能

### 2.1 时钟

**结果：**



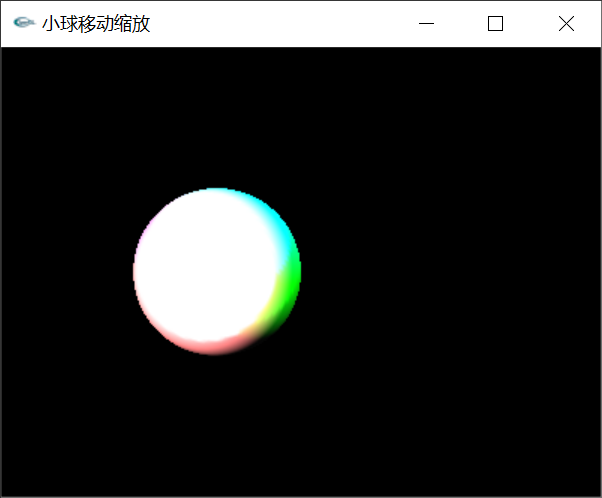


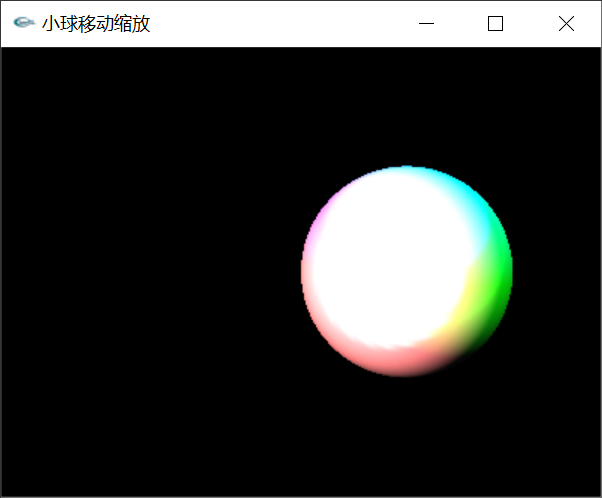
**分析：**

显示正常，会每秒变换时间，时间和系统当前时间一致。可惜在多窗口切换到别的窗口的时候暂停时间。

### 2.2 小球移动缩放

**结果：**





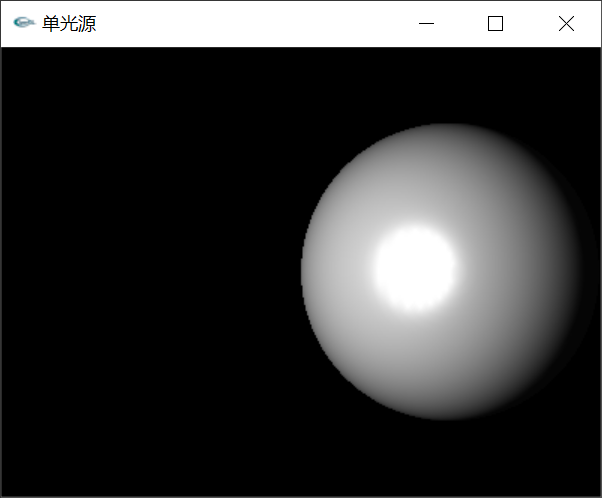
**分析：**

显示正常，缩放正常。平移的过程中，会从左边一直到中间不断缩小再从中间到右边不断放大，然后返回，从右往左亦如此。

## 3、光照、材质和纹理映射功能

### 3.1 单光源

**结果：**

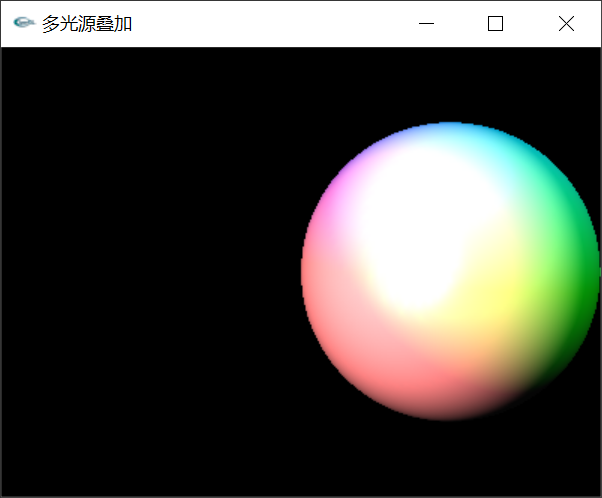


**分析：**

显示正常，灯效符合。

### 3.2 多光源叠加

**结果：**



**分析：**

显示正常，灯效符合（还蛮漂亮的）。

### 3.3 纹理映射

**结果：**



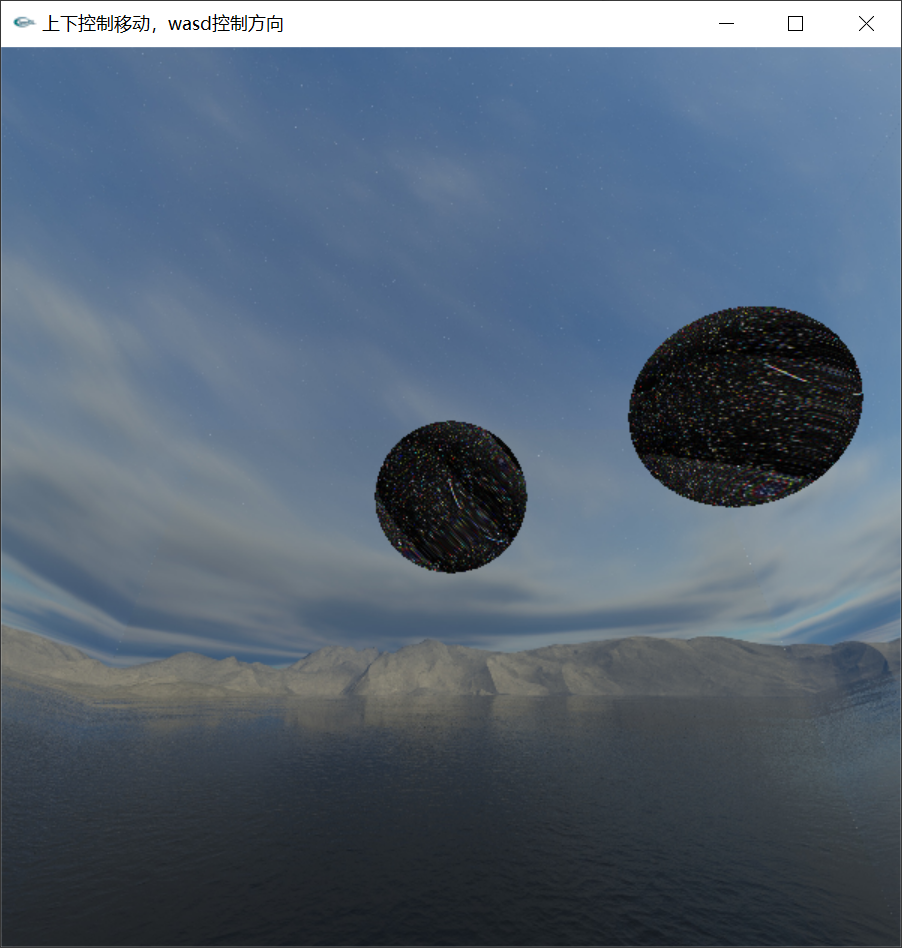
**分析：**

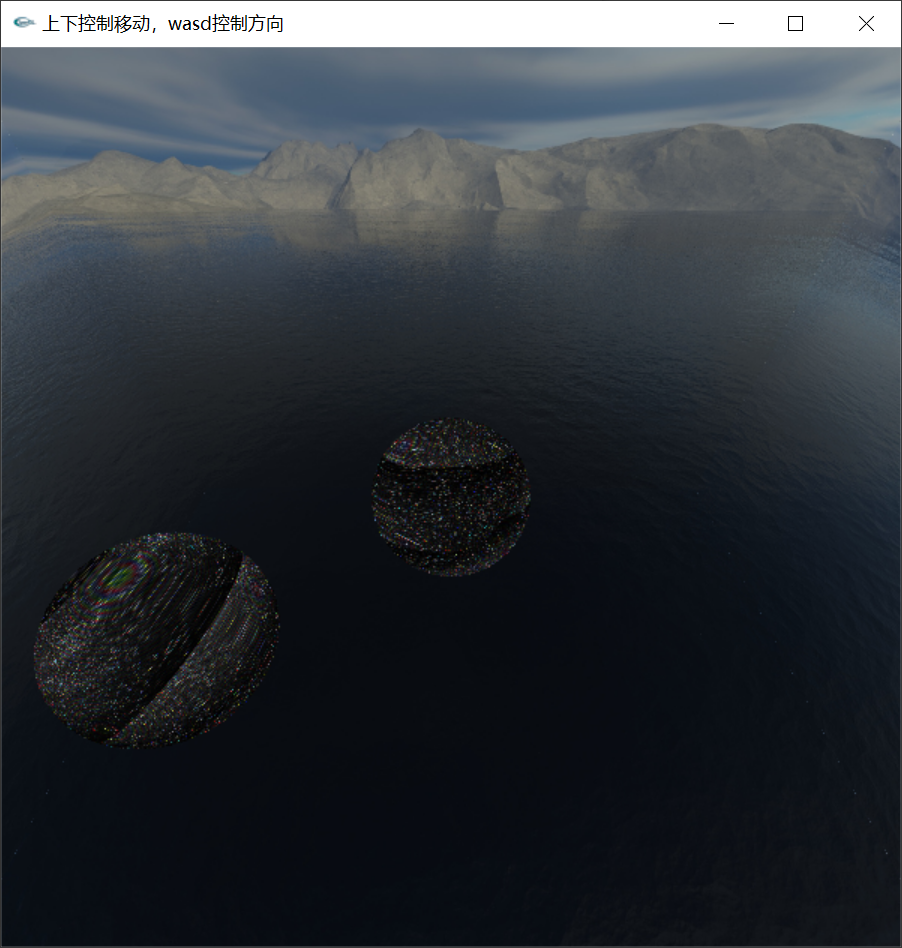
显示正常，结果一致。阴影效果完美，不过我没有调整阴影上图像的方向。

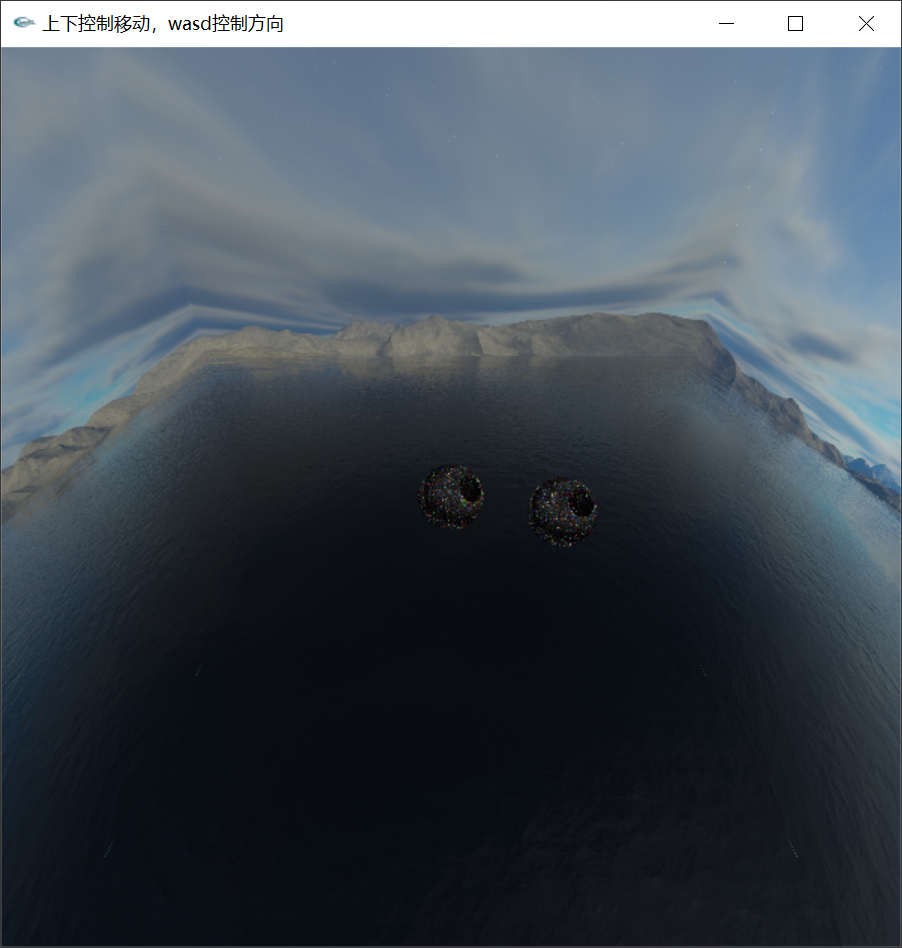
## 4、可视化功能

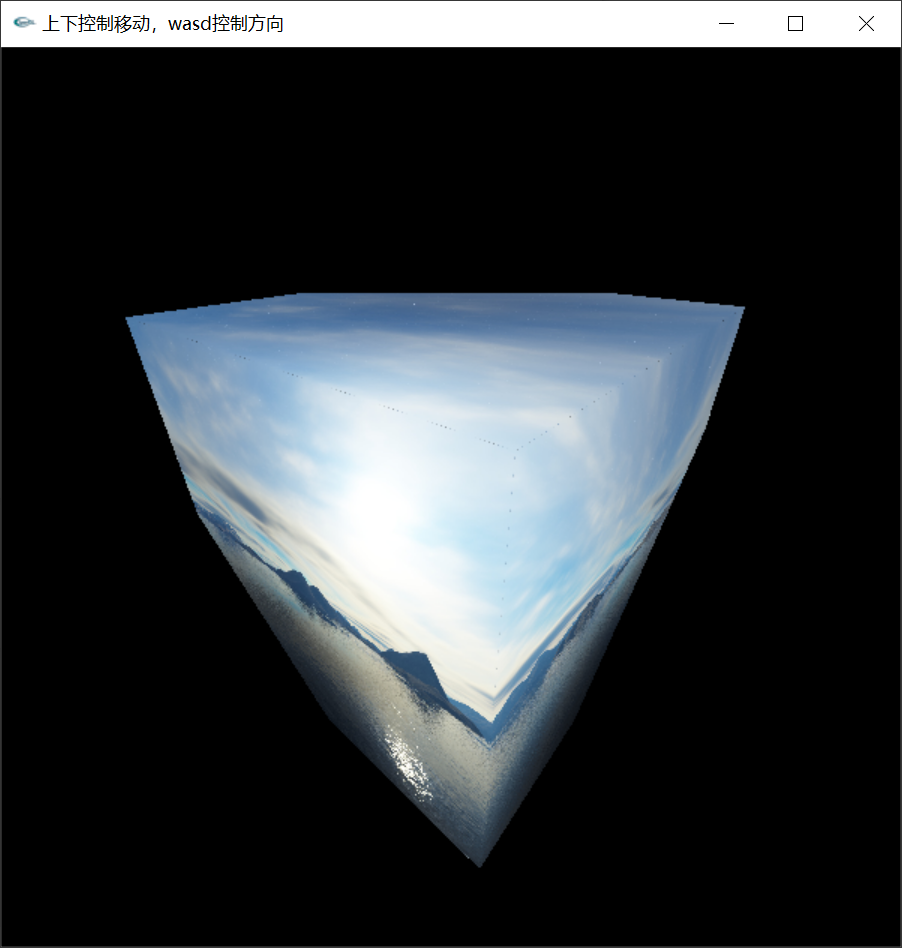
### 4.1 天空盒+几何模型+视点变换+坐标变换+纹理映射（立方体+球体）

**结果：**







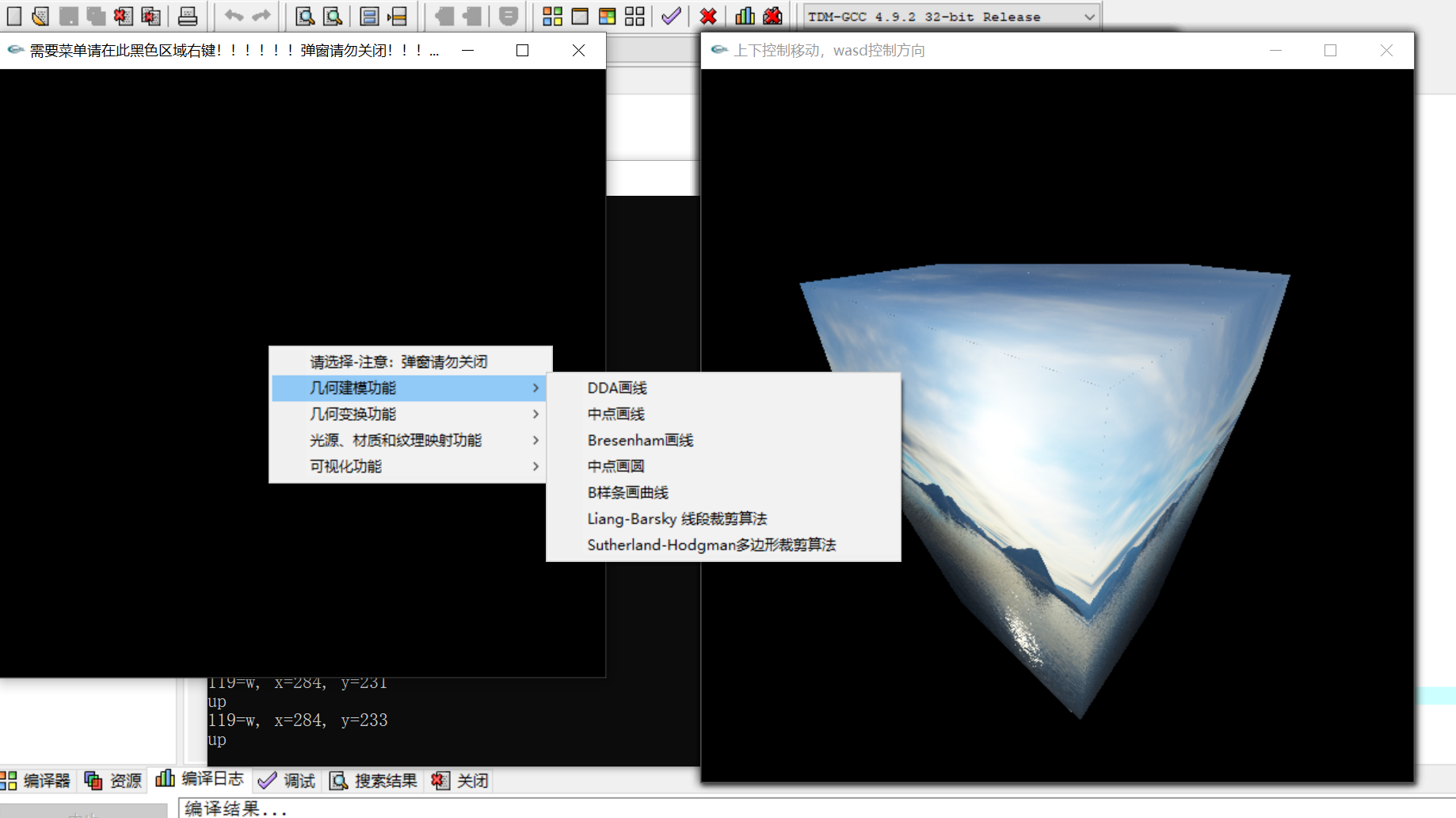


**分析：**

显示正常，天空盒效果还是不错的，还加入了3D球体纹理映射（可运动）、可通过wasd变换视角，可通过up和down来前进后退。

## 5、菜单功能

**结果：**



**分析：**

各个菜单显示正常，弹窗显示也正常，但是关闭一个弹窗就会关闭所有弹窗，这应该是这么多窗口都实际上在一个main里调用的原因。

# 四、结论与展望：

这次的大作业和以往不同，感觉更像是以往知识的一个整合，考验的是我们对框架的搭建能力、知识的掌握程度以及将这些知识转换成为代码的能力。

首先是框架，要想将这么多复杂的内容整合在一起，我们需要一个能够容纳这些内容的一个框架，也就是菜单。该怎么设计这个菜单呢？我选择的是多窗口的方式，也就是说，我先建立了一个操作界面用来选择需要查看的内容，选择后就会弹出一个新的窗口来展示这个内容。他的好处是，让代码的移植（也就是让我原先的代码移植到这个框架中去）变得非常方便，但是也有一些缺点。比如说，当你不小心关闭一个窗口的话，所有窗口都会随之关闭。

其次是知识掌握程度，这个其实没什么好说的，如果不掌握原理，那么写出这些代码的可能性几乎为零。当然，glut为我们提供给了丰富的库函数，很多地方其实都是可以通过直接调用库函数解决的，不过为了鉴于学习目的，我尽量没有使用这些库函数，也没有过多的去了解。未来如果有机会我想我会很乐意去了解下它们。

最后是代码能力，这其实是一个很重要的能力，在某种意义上和算法思想有着同等地位。继续拿框架举例子吧，知道了菜单这个东西之后我们该怎么插入我们的这些个内容呢？我一开始也没有什么想法，后来我发现我可以通过创建窗口来达到显示多个内容的目的。也就是说，在调用菜单之后我们实际要做的只不过是生成一个新的窗口，然后在这个窗口内展示我们选择的内容。

最后的最后，说下展望吧。我以前一直都感觉OpenGL是一个老古董了，如今有着很多和你还的图像软件，比如虚幻引擎啊之类的。知道我在某天打开GPU设置，在里头发现了一个熟悉的单词：OpenGL。底层的东西在很多情况下还是有其存在的价值的，未来我更要好好学习，努力为自己打下一个良好的基础。