计算机图形学案例的说明

1.五角星变换

五角星变换是通过计算圆上等间距的点，根据下标不同得出五角星和五边形，根据时间变换得出五角星的变换函数。

计算点的过程如下:

```
GLfloat a = m / sqrt((2 - 2 * cos(72 * Pi / 180)));
GLfloat bx = a * cos(18 * Pi / 180);
GLfloat by = a * sin(18 * Pi / 180);
GLfloat cx = a * cos(54 * Pi / 180);
GLfloat cy = -a * sin(54 * Pi / 180);
GLfloat Point[10] = { 0,a,bx,by,cx,cy,-cx,cy,-bx,by };
GLfloat Point1[10] = { 0,a,cx,cy,-bx,by,bx,by,-cx,cy };
GLfloat Point2[10];
```

变换过程如下:

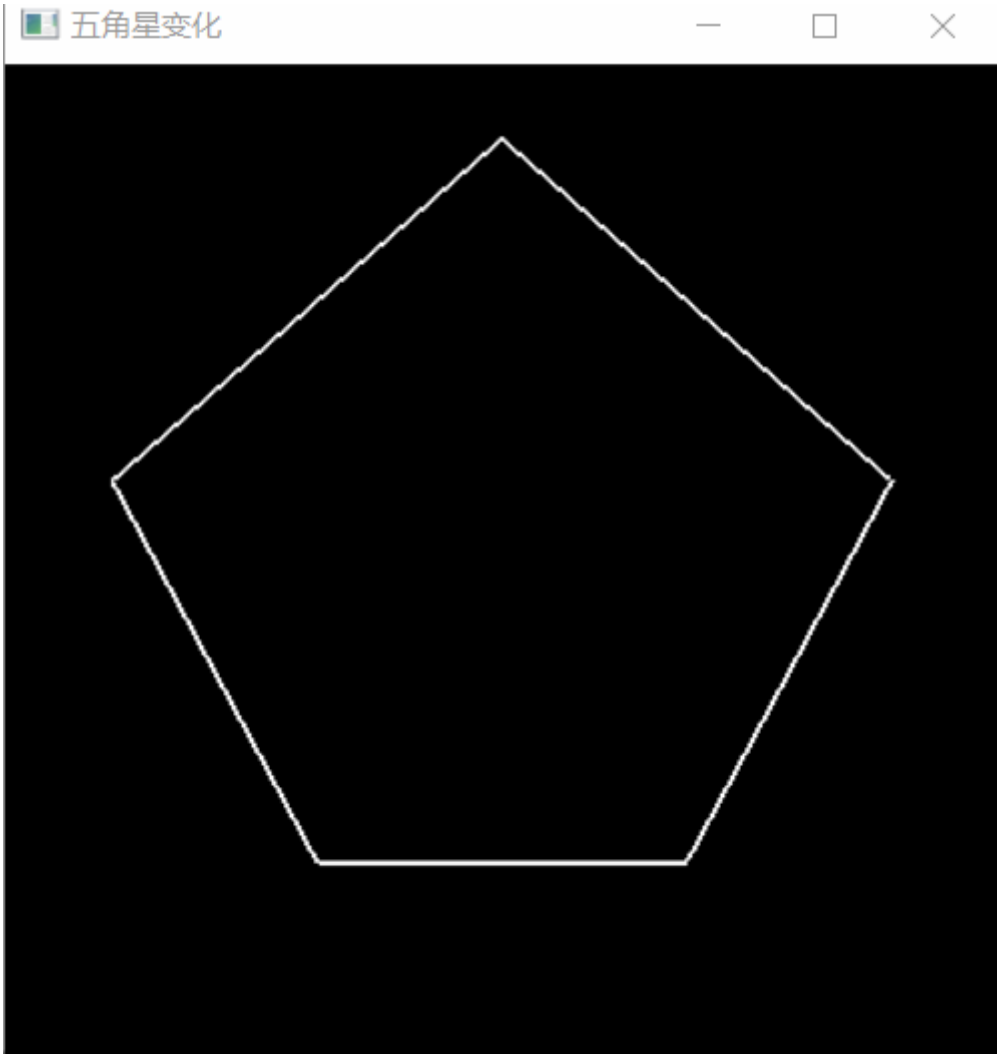
```
/*函数 定时器函数 用于修改点的坐标
* 输入 value 哪个定时器
* 调用函数： glutTimerFunc() gl的定时器函数
*           glutDisplayFunc 调用传递参数myDisplay 进行绘制
*/
void myTimerfunc(int value) {
    if (t > 1.0) t = 0;
    for (int i = 1; i < 10; i++) {
        Point2[i] = Point[i] * (1 - t) + Point1[i] * t;
    }
    t = t + 0.1;
    glutPostRedisplay();//重绘函数，调用mydisplay
    glutTimerFunc(500, myTimerfunc, 1);
}
```

绘制过程如下:

```
//绘制函数，采用GL_LINE_LOOP模式进行绘制
void myDisplay(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glPointSize(1.0f);//设置点的大小
    glLineWidth(1.5f);//设置直线的宽度
    glBegin(GL_LINE_LOOP);
    glVertex2f(Point2[0], Point2[1]);
    glVertex2f(Point2[2], Point2[3]);
    glVertex2f(Point2[4], Point2[5]);
    glVertex2f(Point2[6], Point2[7]);
    glVertex2f(Point2[8], Point2[9]);
    glEnd();
    glFlush();
```

```
}
```

实现效果如下所示：



2.旋转钻石的绘制

钻石的核心思想是将圆周进行n等分（一般为20），将这些点以此与其他进行相连，使其看上去像钻石图案。通过对计算过程中角度的调整，实现钻石的旋转。在这里需注意，因钻石绘制需要一定内存，若定时较快，可能会导致屏幕闪烁现象，采用双缓冲技术解决该问题。

钻石类的声明如下：

```
class Point
{
public:
    Point();
    ~Point();
    int x;
    int y;
private:

};
class diamond
{
```

```
public:
    diamond();
    diamond(int n, int r);//有参构造函数，等分数n，半径r
    ~diamond();
    int n;//点数
    void show();
    void revolve(int step);
private:
    void caculatePoint();
    int initial_r;
    int r;//半径
    Point* p;//顶点数组
    double theta;
    double alpha;
};
```

计算钻石点表函数如下所示：

```
void diamond::caculatePoint() {
    double bata = 0;
    for (int i = 0; i < n; i++) {
        bata = ((i - 0.5) * theta + alpha) / 180 * M_PI;//其中，alpha为类的参数，表示旋转角度
        p[i].x = round(r * cos(bata)) + 100;
        p[i].y = round(r * sin(bata)) + 100;
    }
}
```

旋转角度的调整：

```
//使钻石点表根据旋转角度进行调整
//输入参数 step：调整步长，用于控制旋转角度的变化值
void diamond::revolve(int step=1) {
    if (alpha > 360) alpha = 0;
    caculatePoint();
    alpha += step;
}
```

绘制钻石函数如下：

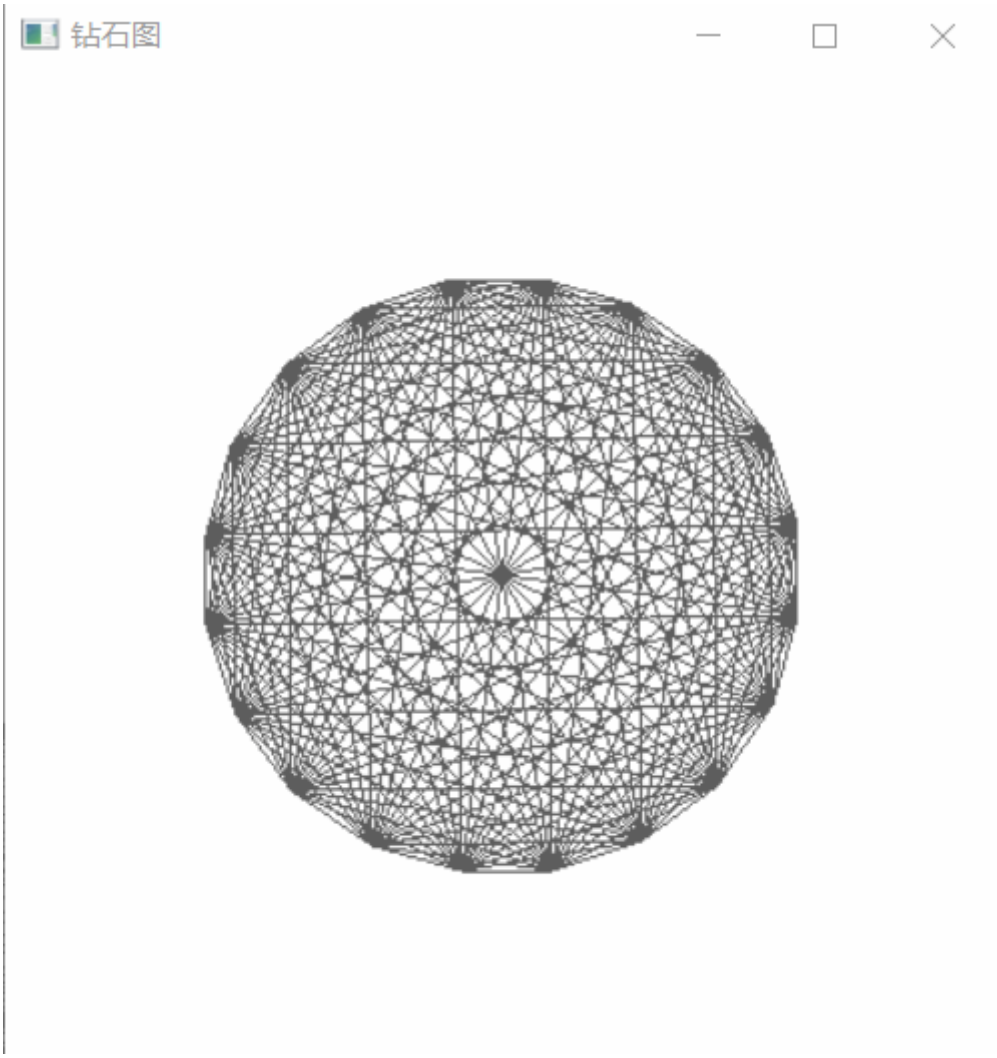
```
/*函数 show 展示钻石函数，有两种方式绘制
*/
void diamond::show() {

    int k = 0;
    double color = alpha / 400;
    glClear(GL_COLOR_BUFFER_BIT);         //用当前背景色填充窗口
    //glColor3f(0.0f, 0.0f, 0.0f);                    //设置当前的绘图颜色为黑色
    glColor3f(color, color, color);

    if (n % 2) {
        glBegin(GL_LINE_LOOP);
        for (int i = 1; i <= n; i++) {
            for (int j = 0; j < i * n; j = j + i) {
                k = j % n;
                glVertex2f(p[k].x, p[k].y);
```

```
            }
        }
    }//若n为奇数的一种绘制方法，对顶点进行遍历，每次循环确定起点或终点
    else {
        glBegin(GL_LINES);
        for (int i = 0; i < n - 1; i++) {
            for (int j = i; j < n; j++) {
                glVertex2f(p[i].x, p[i].y);
                glVertex2f(p[j].x, p[j].y);
            }
        }
    }//无论n为奇书还是偶数，都可采用
    glEnd();
}
```

双缓冲绘图实现：

```
diamond d(20,60);
double t = 0;
void show() {
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);//设置绘图模式为双缓冲
    d.show();//钻石对象的绘制函数
    glFlush();                          //清空OpenGL命令缓冲区，执行OpenGL命令
    glutSwapBuffers();//交换缓冲区

}
void myTimerfunc(int value) {
    d.revolve(1);//钻石旋转
    glutPostRedisplay();
    glutTimerFunc(10, myTimerfunc, 1);
}
```

实现结果如下：

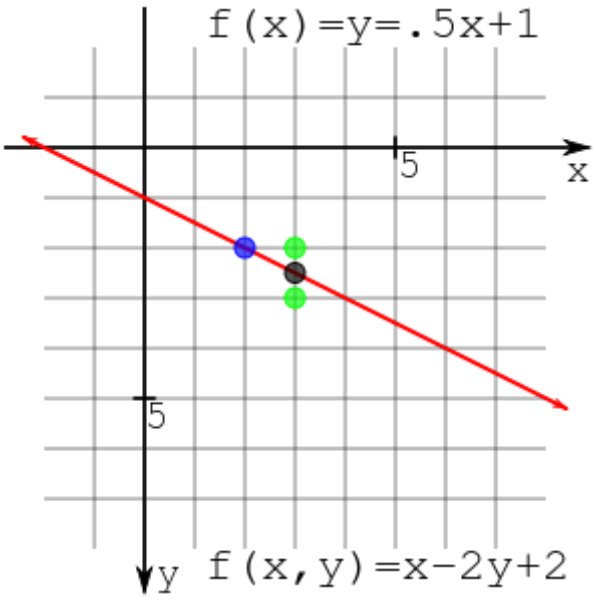


3.直线的扫描转换（Bresenham)

bresenham算法是计算机图形学中为了“显示器（屏幕或打印机）系由像素构成”的这个特性而设计出来的算法，使得在求直线各点的过程中**全部以整数来运算**，因而大幅度提升计算速度。

$$\text{直线方程采用} Ax + By + C = 0$$

利用直线斜率来计算每列像素中确定与理想直线最近的像素来进行转换。如下图所示，黑色的点代表实际中的点，而绿色的表示下一个点在计算机中可能的点，对该黑色点的y坐标四舍五入得到下一个点的坐标。而若直线斜率较大时，将x、y进行对调。当直线垂直y轴时做出特殊处理。



直线组类定义如下：

```
class linegroup
{
public:
    Point* p;
    double theta;
    double alpha;
    linegroup();
    linegroup(int n, int r);
    ~linegroup();
    void drawline(int x0, int y0, int x1, int y);//直线起始点、终点坐标
    void show();
    void revolve(int step);
private:

    void caculatePoint();
    int n;//线的条数
    int r;
};
```

通过等分圆计算直线顶点，并调用`drawline`函数绘制直线

顶点计算与钻石顶点计算相似，此外，使其旋转也与之相似，通过计算所得顶点绘制图案如下所示：

```
void linegroup::show()
{
    Point A{};
    A.x = A.y = 200;
    for (int i = 0; i < n; i++)
        drawline(A.x, A.y, p[i].x, p[i].y);
}
```

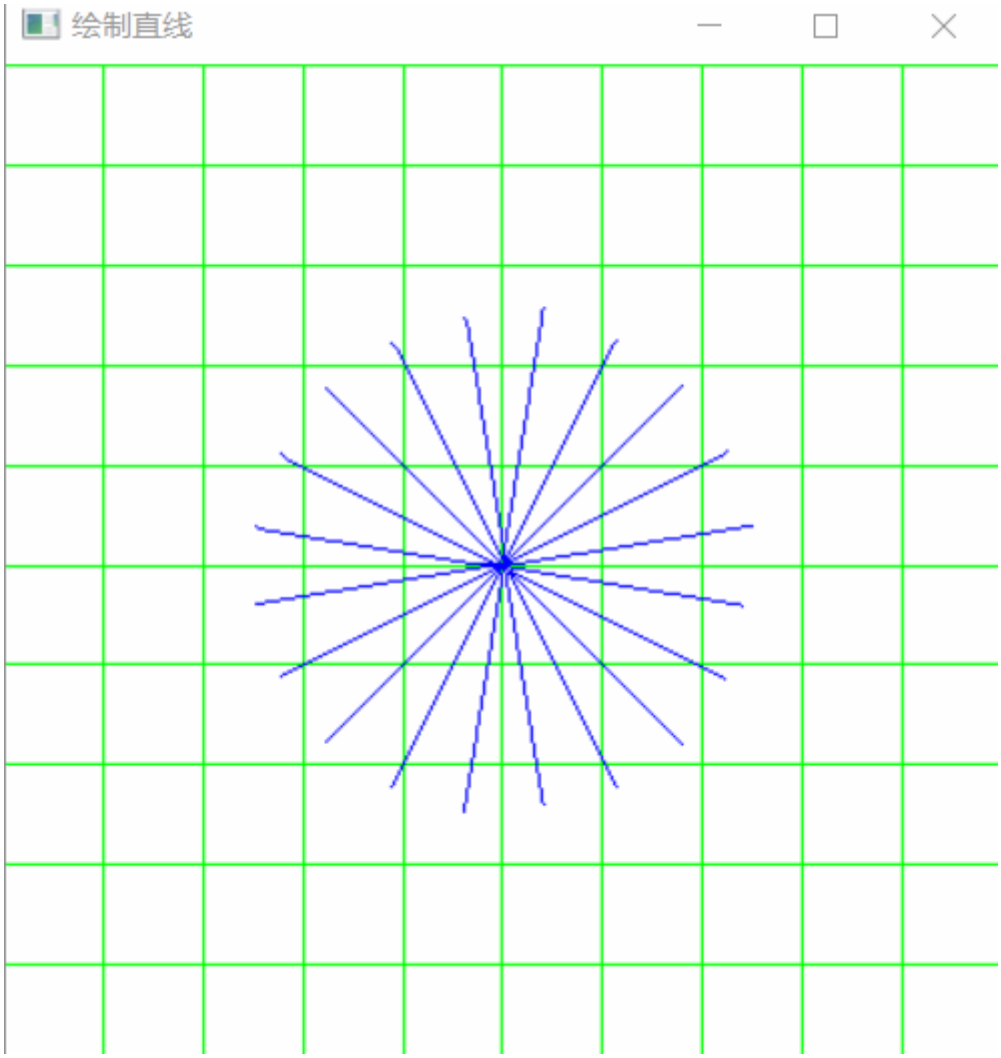
`drawline`函数绘制直线函数如下所示：

```
/*绘制直线函数
* 输入参数 直线起始点x、y、终点坐标y、z坐标
*/
void linegroup::drawline(int x0, int y0, int x1, int y1) {
    int x, y;
    if (x0 > x1) {
        x = x1;
        y = y1;
        x1 = x0;
        y1 = y0;
        x0 = x;
        y0 = y;
    }//起始点、终点交换
    else {
        if (x0 == x1) {
            x = x0;
            if (y0 > y1) {
                y = y1;
```

```
            y1 = y0;
        }
        else {
            y = y0;
        }
        while (y <= y1) {
            glColor3f(0.0f, 0.0f, 1.0f);
            glBegin(GL_POINTS);
            glVertex2d(x * 1.0, y * 1.0);
            glEnd();
            y++;
        }
        return;
    }//垂线绘制

}
x = x0;
y = y0;//初始化
double k = (y1 - y) * 1.0 / ((x1 - x) * 1.0);//直线斜率计算
double d = 0;
float e = d - 0.5;//用于判断下一点位置
GLfloat color = 0;
glColor3f(0.0f, 0.0f, 1.0f);
glBegin(GL_POINTS);
glVertex2d(x * 1.0, y * 1.0);
int dx = x1 - x0;
int dy = y1 - y0;
int flag = 0;//通过flag确定斜率正负影响的x、y、e值变化
if (k > 0) {
    flag = 1;
}
else {
    flag = -1;
}
int i = 0;
if (abs(k) <= 1) {
    while (i <= abs(dy) && x <= x1) {
        e = e + k;
        x++;
        if (e * flag > 0) {
            i++;
            y += flag;
            e -= flag;
        }
        glVertex2d(x * 1.0, y * 1.0);
    }
}//斜率在-1~1之间直线的绘制
else {
    k = 1 / k;

    while ((i <= abs(dy)) && x <= x1) {
        e = e + k;
        y += flag;
        if (e * flag >= 0) {
            i++;
            x++;
            e -= flag;
        }
```

```
            glVertex2d(x * 1.0, y * 1.0);
        }
    }//斜率绝对值大于一的直线绘制

    glEnd();
}
```

绘制结果如下：



可看出直线在斜率偏离1最远处走样最严重

4.多边形有效边表填充

区域填充是利用颜色或者图案来填充一个多边形内部区域。

多边形的表示方法主要有两种：

- 顶点表示法：使用顶点进行描述，但此时多边形仅仅是一些封闭线段，内部是空的，且不能直接进行填充上色
- 点阵表示法：使用大量的点进行描述，描述完成之后，得到的就是完整形态的多边形，内部已被填充，可直接针对点来进行上色

多边形的扫描转换就是从顶点表示法转换到点阵表示法的过程。

有效边表填充算法按照扫描线从小到大的移动顺序，计算当前扫描线与有效边的交点，然后把这些交点按x的值递增顺序进行排序，配对，以确定填充去间，最后用指定颜色填充区间内的所有像素，即完成填充工作。通过维护边表和有效边表，避开了扫描线与多边形所有边求交的复杂运算，性能提升巨大。
边界像素处理原则：对于边界像素，采用“左闭右开”和“下闭上开”的原则

有效边：多边形与当前扫描线相交的边

有效边表：把有效边按照与扫描线交点x坐标递增的顺序存放在一个链表中，称为有效边表

桶表：按照扫描线顺序管理边的数据结构

创建MFC应用，用CDC类型为画布进行绘制。

将多边形顶点传入Fill类，即可调用Fill类的函数进行多边形填充

Fill类的定义如下所示：

```
class CFill
{
public:
    CFill(void);
    virtual ~CFill(void);
    void SetPoint(CPi2* p, int);//类的初始化
    void CreateBucket();//创建桶
    void CreateEdge();//边表
    void AddEt(CAET*);//合并ET表
    void EtOrder();//ET表排序
    void Gouraud(CDC*);//填充多边形
    CRGB Interpolation(double, double, double, CRGB, CRGB);//线性插值
    void ClearMemory();//清理内存
    void DeleteAETChain(CAET* pAET);//删除边表
public:
    int     PNum;//顶点个数
    CPi2* P;//顶点坐标动态数组
    CAET* pHeadE, * pCurrentE, * pEdge; //有效边表结点指针
    CBucket* pHeadB, * pCurrentB;         //桶表结点指针
};
```

填充多边形函数如下所示：

```
void CFill::Gouraud(CDC* pDC)//填充多边形
{
    CAET* pT1 = NULL, * pT2 = NULL;
    pHeadE = NULL;
    for (pCurrentB = pHeadB; pCurrentB != NULL; pCurrentB = pCurrentB->pNext)
    {
        for (pCurrentE = pCurrentB->pET; pCurrentE != NULL; pCurrentE =
pCurrentE->pNext)
        {
            pEdge = new CAET;
            pEdge->x = pCurrentE->x;
            pEdge->yMax = pCurrentE->yMax;
            pEdge->k = pCurrentE->k;
            pEdge->ps = pCurrentE->ps;
            pEdge->pe = pCurrentE->pe;
            pEdge->pNext = NULL;
            AddEt(pEdge);//合并边表
```

```
        }
        EtOrder();//对边表按照起始点、斜率大小进行排序
        pT1 = pHeadE;
        if (pT1 == NULL)
        {
            return;
        }
        while (pCurrentB->ScanLine >= pT1->yMax)//下闭上开
        {
            CAET* pAETTEmp = pT1;
            pT1 = pT1->pNext;
            delete pAETTEmp;
            pHeadE = pT1;
            if (pHeadE == NULL)
                return;
        }
        if (pT1->pNext != NULL)
        {
            pT2 = pT1;
            pT1 = pT2->pNext;
        }
        while (pT1 != NULL)
        {
            if (pCurrentB->ScanLine >= pT1->yMax)//下闭上开
            {
                CAET* pAETTemp = pT1;
                pT2->pNext = pT1->pNext;
                pT1 = pT2->pNext;
                delete pAETTemp;
            }
            else
            {
                pT2 = pT1;
                pT1 = pT2->pNext;
            }
        }
        CRGB Ca, Cb, Cf;//Ca、Cb代表边上任意点的颜色，Cf代表面上任意点的颜色
        Ca = Interpolation(pCurrentB->ScanLine, pHeadE->ps.y, pHeadE->pe.y,
pHeadE->ps.c, pHeadE->pe.c);//对颜色进行插值
        Cb = Interpolation(pCurrentB->ScanLine, pHeadE->pNext->ps.y, pHeadE-
>pNext->pe.y, pHeadE->pNext->ps.c, pHeadE->pNext->pe.c);
        BOOL Flag = FALSE;
        double xb, xe;//扫描线和有效边相交区间的起点和终点坐标
        for (pT1 = pHeadE; pT1 != NULL; pT1 = pT1->pNext)
        {
            if (Flag == FALSE)
            {
                xb = pT1->x;
                Flag = TRUE;
            }
            else
            {
                xe = pT1->x;
                for (double x = xb; x < xe; x++)//左闭右开
                {
                    Cf = Interpolation(x, xb, xe, Ca, Cb);
                    pDC->SetPixel(ROUND(x), pCurrentB->ScanLine, RGB(Cf.red *
255, Cf.green * 255, Cf.blue * 255));
```

```
                }
                Flag = FALSE;
            }
        }
        for (pT1 = pHeadE; pT1 != NULL; pT1 = pT1->pNext)//边的连续性
        {
            pT1->x = pT1->x + pT1->k;
        }
    }
}
```

该函数主要使用了边表和桶表，两者建立过程如下图所示：

```
void CFill::CreateBucket()//创建桶表
{
    int yMin, yMax;
    yMin = yMax = P[0].y;
    for (int i = 0; i < PNum; i++)//查找多边形所覆盖的最小和最大扫描线
    {
        if (P[i].y < yMin)
        {
            yMin = P[i].y;//扫描线的最小值
        }
        if (P[i].y > yMax)
        {
            yMax = P[i].y;//扫描线的最大值
        }
    }
    for (int y = yMin; y <= yMax; y++)
    {
        if (yMin == y)//如果是扫描线的最小值
        {
            pHeadB = new CBucket;//建立桶的头结点
            pCurrentB = pHeadB;//pCurrentB为CBucket当前结点指针
            pCurrentB->ScanLine = yMin;
            pCurrentB->pET = NULL;//没有链接边表
            pCurrentB->pNext = NULL;
        }
        else//其他扫描线
        {
            pCurrentB->pNext = new CBucket;//建立桶的其他结点
            pCurrentB = pCurrentB->pNext;
            pCurrentB->ScanLine = y;
            pCurrentB->pET = NULL;
            pCurrentB->pNext = NULL;
        }
    }
}

void CFill::CreateEdge()//创建边表
{
    for (int i = 0; i < PNum; i++)
    {
        pCurrentB = pHeadB;
        int j = (i + 1) % PNum;//边的第2个顶点，P[i]和P[j]点对构成边
        if (P[i].y < P[j].y)//边的终点比起点高
        {
```

```
            pEdge = new CAET;
            pEdge->x = P[i].x;//计算ET表的值
            pEdge->yMax = P[j].y;
            pEdge->k = (P[j].x - P[i].x) / (P[j].y - P[i].y);//代表1/k
            pEdge->ps = P[i];//绑定顶点和颜色
            pEdge->pe = P[j];
            pEdge->pNext = NULL;
            while (pCurrentB->ScanLine != P[i].y)//在桶内寻找当前边的yMin
            {
                pCurrentB = pCurrentB->pNext;//移到yMin所在的桶结点
            }
        }
        if (P[j].y < P[i].y)//边的终点比起点低
        {
            pEdge = new CAET;
            pEdge->x = P[j].x;
            pEdge->yMax = P[i].y;
            pEdge->k = (P[i].x - P[j].x) / (P[i].y - P[j].y);
            pEdge->ps = P[i];
            pEdge->pe = P[j];
            pEdge->pNext = NULL;
            while (pCurrentB->ScanLine != P[j].y)
            {
                pCurrentB = pCurrentB->pNext;
            }
        }
        if (P[i].y != P[j].y)
        {
            pCurrentE = pCurrentB->pET;
            if (pCurrentE == NULL)
            {
                pCurrentE = pEdge;
                pCurrentB->pET = pCurrentE;
            }
            else
            {
                while (pCurrentE->pNext != NULL)
                {
                    pCurrentE = pCurrentE->pNext;
                }
                pCurrentE->pNext = pEdge;
            }
        }
    }
}
```

其中桶表是按照扫描线顺序管理边的数据结构，每个桶节点主要包括扫描线值、桶上的边表指针、下一个桶表节点指针；边表是多边形的边的存储，每个节点包括扫描线与边相交点的x值（起始时使y最小）、y的最大值、边的起点、终点。

创建一多边形，其填充过程如下：

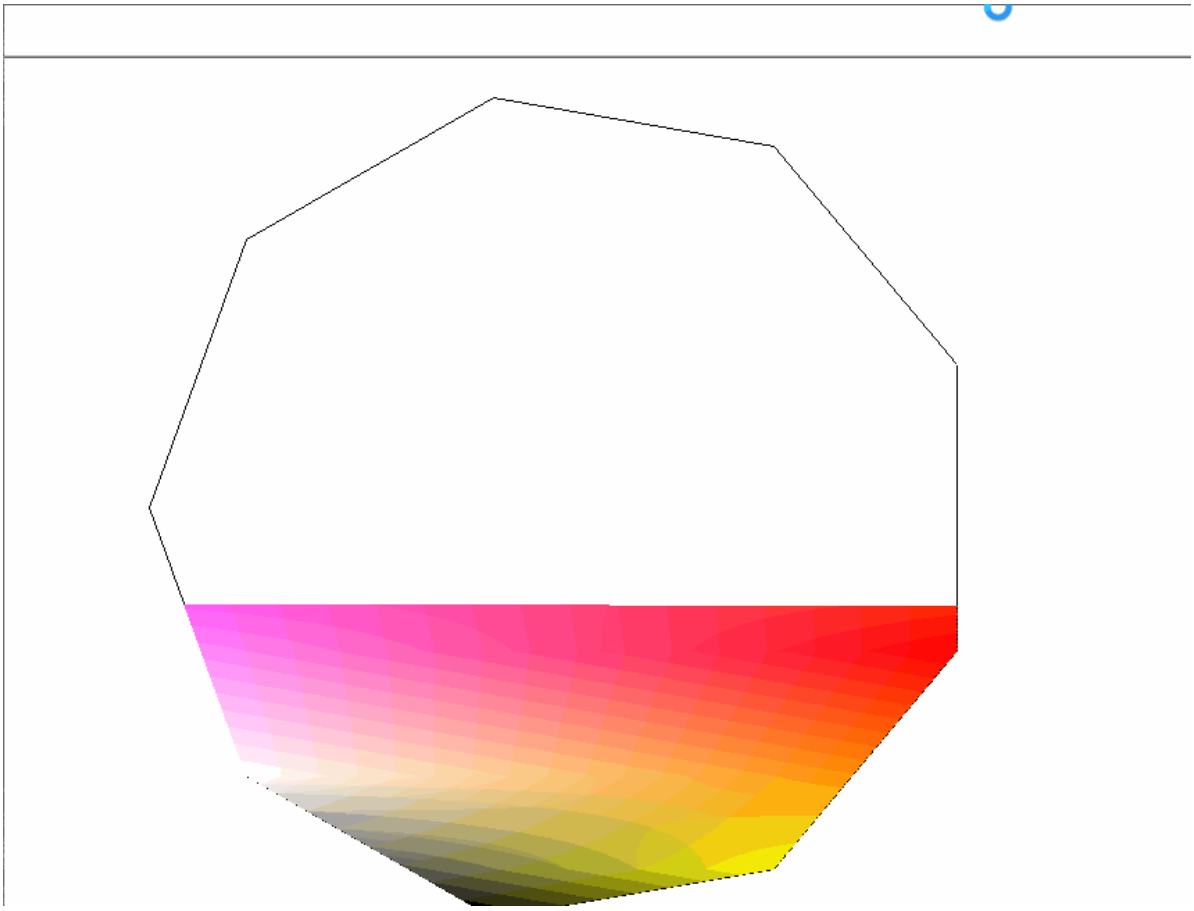
```
void CHexagon::FillPolygon(CDC* pDC)//填充多边形
{
    for (int i = 0; i < n; i++)
    {
        PO[i].x = P[i].x;
```

```
        PO[i].y = Round(P[i].y);
        PO[i].c = P[i].c;
    }
    CFill* fill = new CFill;       //动态分配内存
    fill->SetPoint(PO, n);        //初始化Fill对象
    fill->CreateBucket();        //建立桶表
    fill->CreateEdge();          //建立边表
    fill->Gouraud(pDC);          //填充多边形
    delete fill;                 //撤销内存
}
```

在mfc中绘制图像如下所示：

```
void CPolygonFillView::DrawGraph()
{
    CRect rect;                                          //定义客户区
    GetClientRect(&rect);                                //获得客户区的大小
    CDC* pDC = GetDC();                                    //定义设备上下文指针

    pDC->SetMapMode(MM_ANISOTROPIC);                     //自定义坐标系
    pDC->SetWindowExt(rect.Width(), rect.Height());       //设置窗口比例
    pDC->SetViewportExt(rect.Width(), -rect.Height());    //设置视区比例，且x轴水平向
右，y轴垂直向上
    pDC->SetViewportOrg(rect.Width() / 2, rect.Height() / 2);//设置客户区中心为坐标
系原点
    rect.OffsetRect(-rect.Width() / 2, -rect.Height() / 2);  //矩形与客户区重合
    hex.DrawPolygon(pDC);//绘制多边形
    hex.FillPolygon(pDC);//填充多边形
    ReleaseDC(pDC);//释放DC
}
```

实现效果如下所示：



5.种子填充

种子填充算法是从多边形内部一个已知像素出发，找到区域内的所有像素，并把这些像素置成多边形色。它要求填充的区域必须是连通区域（所谓连通区域是指从区域内任何一点出发，可以通过不同的方向组合在不越出区域的前提下，到达区域内的任何一点。有八连通和四连通之分。

最简单的邻域填充原理：种子像素入栈，当栈非空时执行如下三步操作①栈顶元素出栈；②将出栈元素置为多边形色；③按四邻域顺序检查出栈元素的四个邻域点，将不在边界，且未置色的点填充；缺点，邻域的邻域仍然为邻域，运算的复杂度太高。

栈节点定义如下所示：

```
#pragma once
class CStackNode//栈结点
{
public:
    CStackNode();
    virtual ~CStackNode();
public:
    CPoint PixelPoint;
    CStackNode* pNext;
};
```

图片导入如下（这里要注意，若要使用资源符号访问图片，要通过添加资源将图片加入资源中）：

```
CBitmap NewBitmap, * pOldBitmap;
    NewBitmap.LoadBitmap(IDB_BITMAP1);
    pOldBitmap = memDC.SelectObject(&NewBitmap);
    BITMAP bmp;
    NewBitmap.GetBitmap(&bmp);
    int nX = rect.left + (nClientWidth - bmp.bmWidth) / 2;//计算位图在客户区的中心点
    int nY = rect.top + (nClientHeight - bmp.bmHeight) / 2;
    pDC->BitBlt(nX, nY, nClientWidth, nClientHeight, &memDC, 0, 0, SRCCOPY);
    memDC.SelectObject(pOldBitmap);
    NewBitmap.DeleteObject();
```

通过鼠标交互确定种子点位置：

```
void CSeedFillView::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: 在此添加消息处理程序代码和/或调用默认值
    Seed = point;//选择种子位置
    BoundaryFill();//填充图形
    CView::OnLButtonDown(nFlags, point);
}
```

通过四邻域进行填充

```
void CSeedFillView::BoundaryFill()
{
    CDC* pDC = GetDC();
    pHead = new CStackNode; //建立栈头结点
    pHead->pNext = NULL;    //栈头结点的指针域为空
    Push(Seed);             //种子像素入栈
    if (BkClr != pDC->GetPixel(Seed.x, Seed.y) && BoundaryClr != pDC-
>GetPixel(Seed.x, Seed.y))
    {
        while (NULL != pHead->pNext)//如果栈不为空
        {
            CPoint PopPoint;
            Pop(PopPoint);
            if (SeedClr == pDC->GetPixel(PopPoint.x, PopPoint.y))
                continue;//加速处理
            pDC->SetPixelV(PopPoint.x, PopPoint.y, SeedClr);
            Left.x = PopPoint.x - 1;//搜索出栈结点的左方像素
            Left.y = PopPoint.y;
            COLORREF CurPixClr;     //当前像素的颜色
            CurPixClr = pDC->GetPixel(Left.x, Left.y);
            if (BoundaryClr != CurPixClr && SeedClr != CurPixClr)   //不是边界色并
且未置成填充色
                Push(Left);//左方像素入栈
            Top.x = PopPoint.x;//搜索出栈结点的上方像素
            Top.y = PopPoint.y + 1;
            CurPixClr = pDC->GetPixel(Top.x, Top.y);
            if (BoundaryClr != CurPixClr && SeedClr != CurPixClr)
                Push(Top);//上方像素入栈
            Right.x = PopPoint.x + 1;//搜索出栈结点的右方像素
            Right.y = PopPoint.y;
            CurPixClr = pDC->GetPixel(Right.x, Right.y);
            if (BoundaryClr != CurPixClr && SeedClr != CurPixClr)
                Push(Right);//右方像素入栈
```

```
            Bottom.x = PopPoint.x;//搜索出栈结点的下方像素
            Bottom.y = PopPoint.y - 1;
            CurPixClr = pDC->GetPixel(Bottom.x, Bottom.y);
            if (BoundaryClr != CurPixClr && SeedClr != CurPixClr)
                Push(Bottom);//下方像素如栈
        }
    }
    else
        MessageBox(_T("请在空心字体内部单击鼠标左键!"), _T("提示"));
    delete pHead;
    pHead = NULL;
    ReleaseDC(pDC);
}
```

实现结果如下：

对八邻域测试图片进行填充测试，发现结果并不满意，并且当点击边界后，程序直接陷入不断压栈过程：

6.反走样WU算法

在显示器上显示的图形是由一系列具有一定亮度的离散像素构成的，像素有大小，不是一个点，因而图像会失真，这种用离散量表示连续量的失真叫做走样，wu算法原理与扫面转换类似，在进行实际像素点（就是在屏幕上显示的那个点）的计算的时候，根据实际像素点与理想直线的差值来调整亮度，从而骗过人眼，所以可以在Bresenham算法的基础上进行改进。

反走样WU算法基于Bresenham算法进行改进，以下为实现该算法的过程

为降低代码量，将按颜色和位置绘制点封装为一个函数

```
void linegroup::drawPoint(int x, int y, float e)
{
    glColor3f(e, 0.0f, 0.0f);
    glBegin(GL_POINTS);
    glVertex2d(x * 1.0, y * 1.0);
    glEnd();
}
```

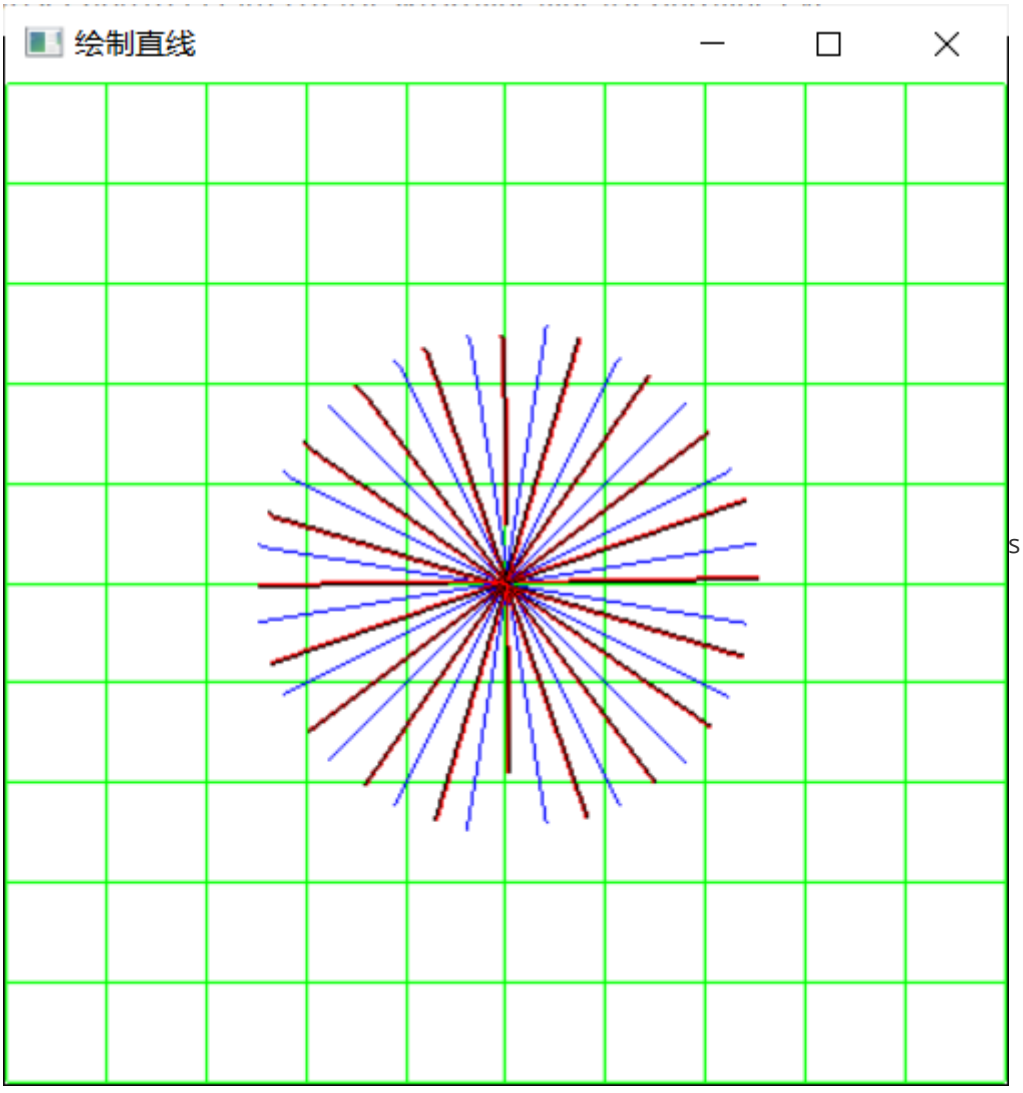
反走样算法实现：

```
void linegroup::drawLineWU(int x0, int y0, int x1, int y1)
{
    int x, y;
    if (x0 > x1) {
        x = x1;
        y = y1;
        x1 = x0;
        y1 = y0;
        x0 = x;
        y0 = y;
    }//起始点、终点交换
    else {
        if (x0 == x1) {
```

```
            x = x0;
            if (y0 > y1) {
                y = y1;
                y1 = y0;
            }
            else {
                y = y0;
            }
            while (y <= y1) {
                glColor3f(0.0f, 0.0f, 1.0f);
                glBegin(GL_POINTS);
                glVertex2d(x * 1.0, y * 1.0);
                glEnd();
                y++;
            }
            return;
        }//垂线绘制

    }
    x = x0;
    y = y0;//初始化
    double k = (y1 - y) * 1.0 / ((x1 - x) * 1.0);//直线斜率计算
    double d = 0;
    float e = d - 0.5;//用于判断下一点位置
    GLfloat color = 0;
    //glColor3f(0.0f, 0.0f, 1.0f);
    //glBegin(GL_POINTS);
    //glVertex2d(x * 1.0, y * 1.0);
    int dx = x1 - x0;
    int dy = y1 - y0;
    int flag = 0;
    if (k > 0) {
        flag = 1;
    }
    else {
        flag = -1;
    }
    int i = 0;
    if (abs(k) <= 1) {
        while (i <= abs(dy) && x <= x1) {
            e = e + k;
            x++;
            drawPoint(x, y, e);
            drawPoint(x, y + flag, 1 - e);
            if (e * flag > 0) {
                i++;
                y += flag;
                e -= flag;
            }


        }
    }//斜率在-1~1之间直线的绘制
    else {
        k = 1 / k;

        while ((i <= abs(dy)) && x <= x1) {
            e = e + k;
```

```
            y += flag;
            drawPoint(x, y, e);
            drawPoint(x + flag, y, 1 - e);
            if (e * flag >= 0) {
                i++;
                x++;
                e -= flag;
            }
            //glVertex2d(x * 1.0, y * 1.0);
        }
    }//斜率绝对值大于一的直线绘制
}
```

实现结果如下所示，红色为反走样线条，蓝色为走样线条，在视觉效果上，红色线条明显优于蓝色线条



7、窗口裁剪

7.1、编码

采用编码裁剪，将窗口进行延申，屏幕上得到九个区域，按照一定的编码规则（上下左右分别代表四位二进制编码的一位（由高到底），在哪个区域，哪位为一），依照上述规则对直线的端点进行编码；设线段的两个端点为`P_1（x_1，y_1）`和`P_2（x_2，y_2）`可以求出`P_1`和`P_2`所在区域的分区代码`C_2`和`C_2`。根据`C_1`和`C_2`的具体值，可以有三种情况：

（1）`C_1=C_2=0`，表明两端点全在窗口内，因而整个线段也在窗内，应予保留。

（2）`C_1&C_2≠0`（两端点代码按位作逻辑乘不为0），即C_1和C_2至少有某一位同时为1，表明两端点必定处于某一边界的同一外侧，因而整个线段全在窗外，应予舍弃。

（3）不属于上面两种情况，均需要求交点。

7.2、求交点

假设算法按照：左、右、下、上边界的顺序进行求交处理，对每一个边界求完交点，并相关处理后，算法转向第2步，重新判断，如果需要接着进入下一边界的处理。为了规范算法，令线段的端点`P_1`为外端点，如果不是这样，就需要`P_1`和`P_2`交换端点。当条件(C_1&0001≠0)成立时，表示端点`P_1`位于窗口左边界外侧，按照前面介绍的求交公式，进行对左边界的求交运算。依次类推，对位于右、下、上边界外侧的判别，应将条件式中的0001分别改为0010、0100、1000即可。求出交点P后，用`P_1`=P来舍去线段的窗外部分，并对`P_1`重新编码得到C_1，接下来算法转回第2步继续对其它边界进行判别。

在代码中以提前绘制好的矩形做为一个窗口，绘制一条与窗口相交的直线进行裁剪，添加了键盘响应事件，便于看出效果；

代码如下：

```
//准备工作
void LineGL(int x0, int  y0, int x1, int y1)
{
    glBegin(GL_LINES);
    glColor3f(0.0f, 1.0f, 0.0f);
    glVertex2f(x0, y0);
    glVertex2f(x1, y1);
    glEnd();
}

struct MyRect
{
    float xmin, xmax, ymin, ymax;
};
//编码，根据窗口进行编码，这样与运算很方便，就是不太直观
int CompCode(int x, int y, MyRect rect)
{
    int code = 0x00;
    if (y < rect.ymin)
        code = code | 4;
    if (y > rect.ymax)
        code = code | 8;
    if (x > rect.xmax)
        code = code | 2;
    if (x < rect.xmin)
        code = code | 1;
    return code;
}

//裁剪算法主角
int cohenSutherlandLineClip(MyRect  rect, int& x0, int& y0, int& x1, int& y1)
{
    int accept, done;
    float x, y;
    //标识位
```

```
    accept = 0;
    done = 0;

    int code1, code2, codeout;
    code1 = CompCode(x0, y0, rect);
    code2 = CompCode(x1, y1, rect);
    do {
        //理想情况全在窗口内，不用裁剪
        if (!(code1 | code2))
        {
            accept = 1;
            done = 1;
        }
        //理想情况全在窗口外
        else if (code1 & code2)
            done = 1;
        //裁剪开始
        else
        {
            //确定p1(强制以p1为内点)
            if (code1 != 0)
                codeout = code1;
            else
                codeout = code2;
            /*求交点，其中 constexpr auto LEFT_EDGE = 1;;
                          constexpr auto RIGHT_EDGE = 2;
                          constexpr auto BOTTOM_EDGE = 4;
                          constexpr auto TOP_EDGE = 8;*/
            if (codeout & LEFT_EDGE) {
                y = y0 + (y1 - y0) * (rect.xmin - x0) / (x1 - x0);
                x = (float)rect.xmin;
            }
            else if (codeout & RIGHT_EDGE) {
                y = y0 + (y1 - y0) * (rect.xmax - x0) / (x1 - x0);
                x = (float)rect.xmax;
            }
            else if (codeout & BOTTOM_EDGE) {
                x = x0 + (x1 - x0) * (rect.ymin - y0) / (y1 - y0);
                y = (float)rect.ymin;
            }
            else if (codeout & TOP_EDGE) {
                x = x0 + (x1 - x0) * (rect.ymax - y0) / (y1 - y0);
                y = (float)rect.ymax;
            }
             //继续进行裁剪判断
            if (codeout == code1)
            {
                x0 = x; y0 = y;
                code1 = CompCode(x0, y0, rect);
            }
            else
            {
                x1 = x; y1 = y;
                code2 = CompCode(x1, y1, rect);
            }
        }
        //判断是否裁剪完成
    } while (!done);
```

```
    if (accept)//进行绘制
        LineGL(x0, y0, x1, y1);
    return accept;
}
```

实现结果如下图所示：