实验二 实域填充算法

实验报告

一、综述

多边形在计算机中有顶点表示和点阵表示两种。顶点表示就是用多边形的顶点序列来表示多边形。点阵表示是用位于多边形内的象素集合来表示多边形。顶点表示占内存少，几何意义强，易于进行几何变换；而点阵表示丢失了许多几何信息（如边界、顶点）。但光栅显示图形需要点阵表示形式。多边形的扫描转换就是把多边形的顶点表示转换为点阵表示。

本次实验所涉及的算法有三种：逐点实域填充算法，有序边表实域填充算法和种子实域填充算法。在 MFC 环境中测试三种算法并分析三种算法的计算效率。

二、程序框架

实验程序为 MFC 框架，Ccg2019FillPolyView.h 为视图层的头文件，负责声明各种成员变量和成员函数。Ccg2019FillPolyView.cpp 为视图层的源文件，负责实现直线的三种绘制、误差分析并显示、圆与圆弧绘制的功能。CcgEditControl.h 为窗口面板中的按键及文本定义成员变量及成员函数，CcgEditControl.cpp 实现面板的功能，如鼠标交互绘制输入多边形，点击按键填充多边形等。

三、算法描述

1. 逐点实域填充算法

（1）将给定多边形输入；

（2）求出多边形的最小包含矩形；

（3）逐点扫描最小矩形的每一点，并判断是否位于多边形内部，从最小点到最大点一次判断，如果在该多边形内部，则将该点上色；

（4）判断位于多边形内部的方法是，过每一点水平向右作射线，与多边形边界求交点，如果交点个数为奇数，则说明该点在多边形内部，偶数则说明在多边形外部。

2. 有序边表实域填充算法

一.建立边表的方法：

（1）与 x 轴平行的边不计入；

（2）多边形的顶点分为两大类：一类是局部极值点，另外一类是非极值点。当扫面线与第一类顶点相交时，应看作两个点；而当扫描线与第二类顶点相交时，应视为一个点；
对于极值点则要记录两条边；

（3）扫描线按照 y 轴从低到高顺次记录；

（4）一条边按照 y 轴的高低记录；

（5）多条边以 x 轴递增顺序记录；

二．算法流程：

1、根据给出的多边形顶点坐标，建立 NET 表；

求出顶点坐标中最大 y 值 ymax 和最小 y 值 ymin。

2、初始化 AET 表指针，使它为空。

3、执行下列步骤直至 NET 和 AET 都为空.

（1）如 NET 中的第 y 类非空，则将其中的所有边取出并插入 AET 中；

（2）如果有新边插入 AET，则对 AET 中各边排序；

（3）对 AET 中的边两两配对，（1 和 2 为一对，3 和 4 为一对，…），

将每对边中 x 坐标按规则取整，获得有效的填充区段，再填充.

（4）将当前扫描线纵坐标 y 值递值 1；

（5）如果 AET 表中某记录的 ymax=yj，则删除该记录 （因为每条边被看作下闭上开的）；

（6）对 AET 中剩下的每一条边的 x 递增 dx，即 x’ = x+ dx .

三．图案填充：

只需要给出填充的图案，然后存放在二维数组 m_patternData 中即可，利用取余运算巧妙实现。

3. 扫描线种子填充算法

(1) 初始化一个空的栈用于存放种子点，将种子点(x, y)入栈；

(2) 判断栈是否为空，如果栈为空则结束算法，否则取出栈顶元素作为当前扫描线的种子点(x, y)，y 是当前的扫描线；

(3) 从种子点(x, y)出发，沿当前扫描线向左、右两个方向填充，直到边界。分别标记区段的左、右端点坐标为 xLeft 和 xRight；

(4) 分别检查与当前扫描线相邻的 y - 1 和 y + 1 两条扫描线在区间[xLeft, xRight]中的像素，从 xLeft 开始向 xRight 方向搜索，若存在非边界且未填充的像素点，则找出这些相邻的像素点中最右边的一个，并将其作为种子点压入栈中，然后返回第（2）步；

4. 多边形三角剖分算法：

（1）选择多边形的最左顶点 L+前后顶点，构成一个三角形。

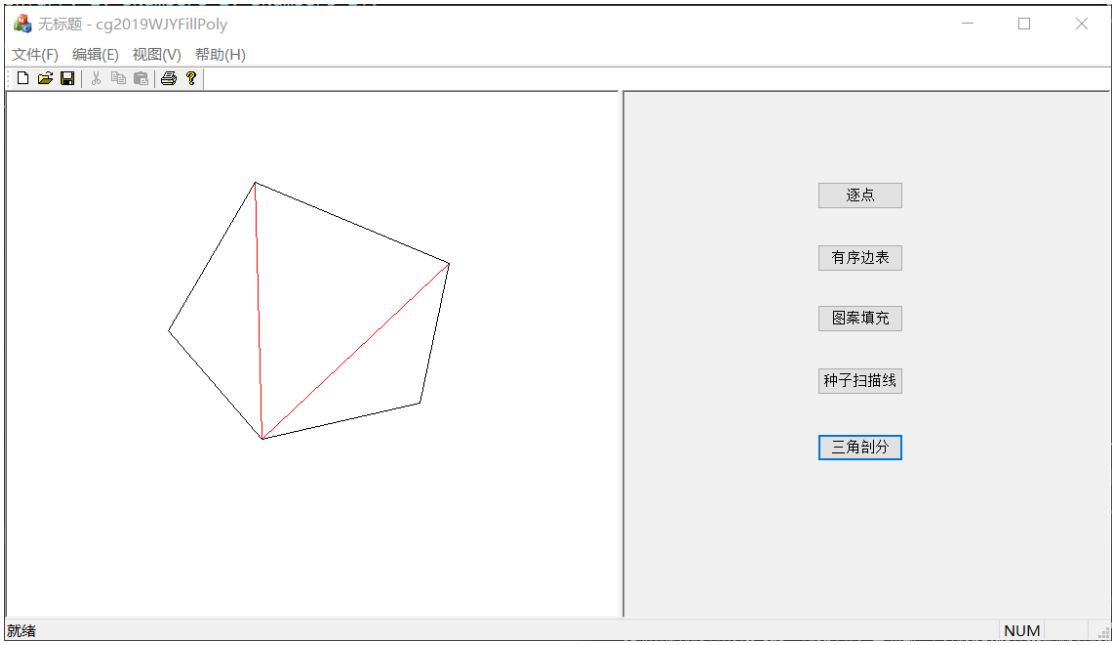
（2）检查该三角形内是否有其他顶点。

不包含其他顶点。则分割该三角形，递归调用第一步。

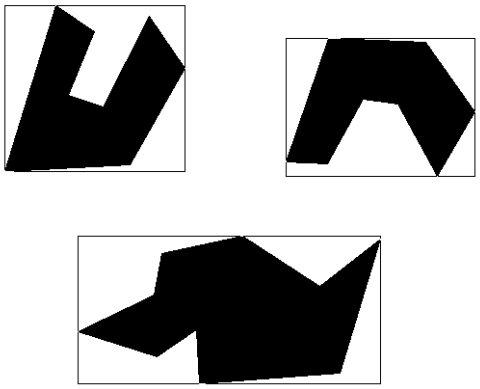
若包含其他顶点。则连接 L 与进入的顶点中最左侧的点，这样便会分割该多边形，然后对两个多边形再递归调用第一步。

四、处理流程说明

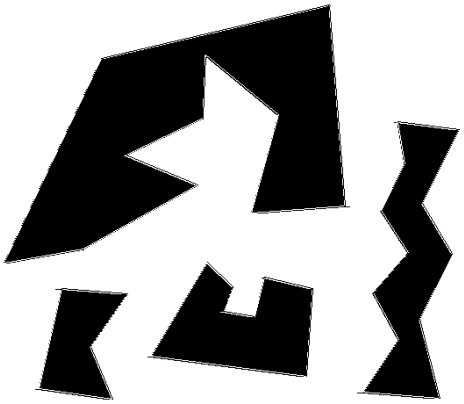
操作面板：


无标题 - cg2019WJYFillPoly
文件(F) 编辑(E) 视图(V) 帮助(H)
逐点
有序边表
图案填充
种子扫描线
三角剖分
就绪
NUM

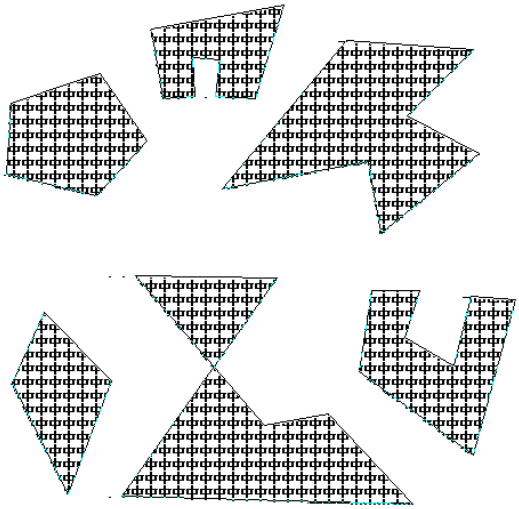
1. 逐点法：



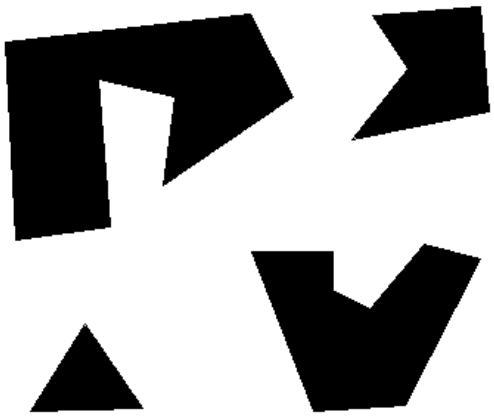
2. 有序边表



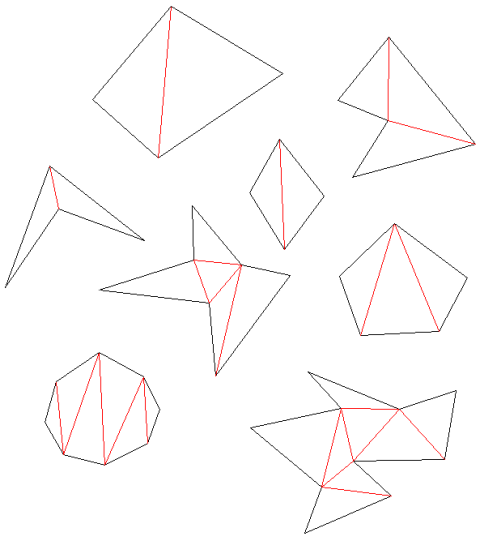
3. 图案填充：



4. 扫描线种子填充：



5. 多边形三角剖分：



五、算法对比

从计算效率来看，逐点扫描填充算法效率最低，由于需要遍历最小包含矩形的每一点并进行逻辑判断，速度慢；有序边表法由于采用了活性边表，大大减少了计算量，提高了运行效率。种子扫描算法利用了栈结构，算法减少了每个像素的访问次数，所需堆栈深度较浅，每次递归填充一行像素，速度较快，效率较高。

六．源代码

```
//图案
int m_patternData[12][12] = {
    { 0,0,0,0,0,1,1,0,0,0,0,0 },
    { 0,0,0,0,0,1,1,0,0,0,0,0 },
    { 0,0,0,0,0,1,1,0,0,0,0,0 },
    { 0,1,1,1,1,1,1,1,1,1,1,1 },
    { 0,1,1,0,0,1,1,0,0,1,1,0 },
    { 0,1,1,0,0,1,1,0,0,1,1,0 },
    { 0,1,1,0,0,1,1,0,0,1,1,0 },
    { 0,1,1,1,1,1,1,1,1,1,1,0 },
    { 0,0,0,0,0,1,1,0,0,0,0,0 },
    { 0,0,0,0,0,1,1,0,0,0,0,0 },
    { 0,0,0,0,0,1,1,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0 }

};
```

```
//有序边表法填充
void Ccg2019FillPolyView::Fillpolygon(int pNumbers, CPoint* points, CDC* pDC)
{

    m_edgeNumbers = 0;
    pLoadPolygon(pNumbers, points);   // Polygon Loading, calculates every edge's
m_yMax[],m_yMin[],m_Xa[],m_Dx[]

    //求交边集范围，因为数组已经根据y值大小进行边的排序，所以end向后移动即代表有边进入，
start向后移动，代表有边退出
    m_Begin = m_End = 0;
    m_Scan = (int)m_yMax[0];          //从顶向下扫描
    Include();                        //检查是否有边进入扫描线
    UpdateXvalue();                   //检查是否有边退出扫描线
    while (m_Begin != m_End) {
        pFillScan(pDC);
        m_Scan--;
        Include();
        UpdateXvalue();
    }
}

void Ccg2019FillPolyView::pLoadPolygon(int pNumbers, CPoint* points)
{
    float x1, y1, x2, y2;

    x1 = points[0].x;    y1 = points[0].y + 0.5;
    for (int i = 1; i < pNumbers; i++) {
        x2 = points[i].x;    y2 = points[i].y + 0.5;
        if (abs(int(y2 - y1)) >= 0)    //水平线不做处理
        {
            pInsertLine(x1, y1, x2, y2);
            x1 = x2;      y1 = y2;
        }
        else
            x2 = x1;
    }
}

void Ccg2019FillPolyView::pInsertLine(float x1, float y1, float x2, float y2)
{
    int i;
    float Ymax, Ymin;
```

```
    Ymax = (y2 > y1) ? y2 : y1;
    Ymin = (y2 < y1) ? y2 : y1;
    i = m_edgeNumbers;
    //根据y值的大小，进行排序插入，大的在前面
    while (i != 0 && m_yMax[i - 1] < Ymax) {
        m_yMax[i] = m_yMax[i - 1];
        m_yMin[i] = m_yMin[i - 1];
        m_Xa[i] = m_Xa[i - 1];
        m_Dx[i] = m_Dx[i - 1];
        i--;
    }
    m_yMax[i] = Ymax;
    m_yMin[i] = Ymin;
    if (y2 > y1) m_Xa[i] = x2;//根据y大小确定Xa的值，y大的会先于扫描线相交
    else         m_Xa[i] = x1;

    m_Dx[i] = (x2 - x1) / (y2 - y1);    //斜率的倒数
    m_edgeNumbers++;
}

void Ccg2019FillPolyView::Include()
{
    //end向后移动，找出所有边最高点y值大于当前扫描线的边，看是否有新的边进入交集
    while (m_End < m_edgeNumbers && m_yMax[m_End] >= m_Scan) {
        //有边进入，调整起始点位置，然后将Dx调整为位移量
        m_Xa[m_End] = m_Xa[m_End] + (-0.5) * m_Dx[m_End];
        m_Dx[m_End] = -m_Dx[m_End];
        m_End++;
    }
}

void Ccg2019FillPolyView::UpdateXvalue()
{
    int i, start = m_Begin;

    for (i = start; i < m_End; i++) {
        if (m_Scan > m_yMin[i]) {
            //当前边没有退出，则移动x，然后在进行排序
            m_Xa[i] += m_Dx[i];
            pXsort(m_Begin, i);
        }
        else {
            //有边退出，更新数组，然后begin++
            for (int j = i; j > m_Begin; j--) {
```

```
                m_yMin[j] = m_yMin[j - 1];
                m_Xa[j] = m_Xa[j - 1];
                m_Dx[j] = m_Dx[j - 1];
            }
            m_Begin++;
        }
    }
}

void Ccg2019FillPolyView::pXsort(int Begin, int i)
{
    float temp;

    while (i > Begin&& m_Xa[i] < m_Xa[i - 1]) {
        temp = m_Xa[i];   m_Xa[i] = m_Xa[i - 1];   m_Xa[i - 1] = temp;
        temp = m_Dx[i];   m_Dx[i] = m_Dx[i - 1];   m_Dx[i - 1] = temp;
        temp = m_yMin[i]; m_yMin[i] = m_yMin[i - 1]; m_yMin[i - 1] = temp;
        i--;
    }
}


void Ccg2019FillPolyView::pFillScan(CDC* pDC)
{
    int x, y;
    Ccg2019FillPolyDoc* pDoc = GetDocument();


    for (int i = m_Begin; i < m_End; i += 2) {

        if (pDoc->m_displayMode == 1) {
            pDC->MoveTo(m_Xa[i], m_Scan);
            pDC->LineTo(m_Xa[i + 1], m_Scan);
        }
        else if (pDoc->m_displayMode == 4) {    //图案填充
            y = m_Scan;
            for (int x = m_Xa[i]; x < m_Xa[i + 1]; x++)
                if (m_patternData[y % 12][x % 12])
                    pDC->SetPixel(x, y, RGB(255, 0, 0));

        }
    }
}
```

```
//逐点法填充

void Ccg2019FillPolyView::PointFillpoly(int pNumbers, CPoint *points, CDC *pDC)
{
    BoxRect_t polyRect;

    polyRect = getPolygonRect(pNumbers, points);

    m_pDC->MoveTo(polyRect.minX, polyRect.minY);

    m_pDC->LineTo(polyRect.minX, polyRect.maxY);
    m_pDC->LineTo(polyRect.maxX, polyRect.maxY);
    m_pDC->LineTo(polyRect.maxX, polyRect.minY);
    m_pDC->LineTo(polyRect.minX, polyRect.minY);

    CPoint testPoint;
    //从最小点到最大点一次判断，如果在该多边形内部，则进行填充
    for (testPoint.x = polyRect.minX; testPoint.x < polyRect.maxX; testPoint.x++)
        for (testPoint.y = polyRect.minY; testPoint.y < polyRect.maxY; testPoint.y++) {
            if (PointInPolygon(m_pNumbers, m_pAccord, testPoint))
                pDC->SetPixel(testPoint.x, testPoint.y, RGB(255, 255, 255));
        }
}

//得到该多边形的最大、最小的y、x值
BoxRect_t Ccg2019FillPolyView::getPolygonRect(int pointNumOfPolygon, CPoint tarPolygon[])
{
    BoxRect_t boxRect;

    boxRect.minX = 50000;
    boxRect.minY = 50000;
    boxRect.maxX = -50000;
    boxRect.maxY = -50000;

    for (int i = 0; i < pointNumOfPolygon; i++) {
        if (tarPolygon[i].x < boxRect.minX) boxRect.minX = tarPolygon[i].x;
        if (tarPolygon[i].y < boxRect.minY) boxRect.minY = tarPolygon[i].y;
        if (tarPolygon[i].x > boxRect.maxX) boxRect.maxX = tarPolygon[i].x;
        if (tarPolygon[i].y > boxRect.maxY) boxRect.maxY = tarPolygon[i].y;
    }
    return boxRect;
}

//判断点是否位于区域内
```

```
BOOL Ccg2019FillPolyView::PointInPolygon(int pointNumOfPolygon, CPoint tarPolygon[],
CPoint testPoint)
{

    if (pointNumOfPolygon < 3)
        return false;

    bool  inSide = FALSE;
    float lineSlope, interSectX;
    int   i = 0, j = pointNumOfPolygon - 1;

    for (i = 0; i < pointNumOfPolygon; i++) {
        if ((tarPolygon[i].y < testPoint.y && tarPolygon[j].y >= testPoint.y ||
            tarPolygon[j].y < testPoint.y && tarPolygon[i].y >= testPoint.y) &&
            (tarPolygon[i].x <= testPoint.x || tarPolygon[j].x <= testPoint.x)) {
            if (tarPolygon[j].x != tarPolygon[i].x) {
                lineSlope = (float)(tarPolygon[j].y - tarPolygon[i].y) /
(tarPolygon[j].x - tarPolygon[i].x);
                interSectX = (int)(tarPolygon[i].x + (testPoint.y - tarPolygon[i].y) /
lineSlope);
            }
            else
                interSectX = tarPolygon[i].x;
            if (interSectX < testPoint.x)
                inSide = !inSide;
        }
        j = i;
    }

    return inSide;
}


//扫描线种子填充
int SetRP(int x, int y, COLORREF color, COLORREF mColor, CDC* pDC) {
    while (pDC->GetPixel(CPoint(x, y)) == mColor) {
        pDC->SetPixel(x, y, color);
        x++;
    }
    return x - 1;
}
int SetLP(int x, int y, COLORREF color, COLORREF mColor, CDC* pDC) {
    while (pDC->GetPixel(CPoint(x - 1, y)) == mColor) {
        pDC->SetPixel(--x, y, color);
```

```
    }
    return x + 1;
}
void NewLineSeed(std::stack<CPoint>* stk, int lx, int rx, int y, COLORREF color, COLORREF
mColor, CDC* pDC) {
    int x, e;
    for (x = lx + 1, e = rx + 1; x < e; x++) {
        //找出每一个区间的最右像素，入栈
        if (pDC->GetPixel(CPoint(x, y)) != mColor) {
            if (pDC->GetPixel(CPoint(x - 1, y)) == mColor)
                stk->push(CPoint(x - 1, y));
        }
    }
    //把rx所在点入栈
    if (pDC->GetPixel(CPoint(x - 1, y)) == mColor)
        stk->push(CPoint(x - 1, y));
}
void Ccg2019FillPolyView::ScanLineFill4(int x, int y, COLORREF color, CDC* pDC)
{
    int pRight, pLeft;
    std::stack<CPoint> stk;
    int mColor = pDC->GetPixel(x, y);  //给定种子

    stk.push(CPoint(x, y));
    while (!stk.empty()) {
        CPoint p = stk.top(); //栈顶像素出栈
        stk.pop();

        pRight = SetRP(p.x, p.y, color, mColor, pDC);    //向左向右进行填充
        pLeft = SetLP(p.x, p.y, color, mColor, pDC);

        //上下两条扫描线处理
        NewLineSeed(&stk, pLeft, pRight, p.y + 1, color, mColor, pDC);
        NewLineSeed(&stk, pLeft, pRight, p.y - 1, color, mColor, pDC);
    }
}


void Ccg2019FillPolyView::FloodFill4(int x, int y, int fillColor, int oldColor, CDC* pDC)
{
    int current;
    current = pDC->GetPixel(x, y);
    if (current == oldColor) {
        pDC->SetPixel(x, y, fillColor);
```

```
        FloodFill4(x + 1, y, fillColor, oldColor, pDC);
        FloodFill4(x - 1, y, fillColor, oldColor, pDC);
        FloodFill4(x, y + 1, fillColor, oldColor, pDC);
        FloodFill4(x, y - 1, fillColor, oldColor, pDC);
    }
}

//多边形三角剖分
void Ccg2019FillPolyView::Triangulation(CPoint* points, int pNumbers, int number)
{

    if (pNumbers == 3) {  //出口
        return;
    }

    int k, xMin = 100000;
    for (int j = 0; j < pNumbers; j++) //找出当前多边形的最左侧顶点
    {
        if (points[j].x < xMin) {
            k = j;
            xMin = points[j].x;
        }
    }

    CPoint arry[3];
    arry[0] = points[k];  //最左侧顶点
    int next = (k + 1) % pNumbers;
    arry[1] = points[next];   //后一个顶点
    int pre = k - 1;
    if (pre < 0)
        pre += pNumbers;
    arry[2] = points[pre];//前一个顶点

    //围成的该三角形内是否有其他顶点
    CPoint in_point[N];
    int in_number = 0, i = 0;
    xMin = 1000;
    for (int j = 0; j < pNumbers; j++)
    {
        if (j == k || j == next || j == pre)    //三角形的三个顶点不算
            continue;
        if (PointInPolygon(3, arry, points[j])) {   //找出在该三角形内的顶点，并找到其
中的最左侧顶点
            in_point[in_number] = points[j];
```

```
            if (in_point[in_number].x < xMin) {
                i = j;
                xMin = in_point[in_number].x;
            }
            in_number++;
        }
    }
    if (in_number >= 1) { //若存在，则链接三角形内的最左侧点，分割多边形，递归调用

        CPen pen(PS_SOLID, 1, RGB(255, 0, 0));//创建画笔对象
        CClientDC dc(this);
        CPen* pOldPen = dc.SelectObject(&pen);

        dc.MoveTo(arry[0].x, arry[0].y);    //连接线到该最左侧顶点
        dc.LineTo(points[i].x, points[i].y);
        dc.SelectObject(&pOldPen);


        CPoint arry_1[N], arry_2[N];   //分割成两个多边形
        int pNumbers_1 = 0, pNumbers_2 = 0;

        if (k > i) {
            int temp = i; i = k; k = temp;
        }

        for (int j = 0; j < pNumbers; j++)
        {
            if (j >= k && j <= i) {
                arry_1[pNumbers_1++] = points[j];
            }
            if (j <= k || j >= i) {
                arry_2[pNumbers_2++] = points[j];
            }
        }
        Triangulation(arry_1, pNumbers_1, pNumbers_1);
        Triangulation(arry_2, pNumbers_2, pNumbers_2);
    }
    else {   //若不存在，则分割该三角形，递归调用

        CPen pen(PS_SOLID, 1, RGB(255, 0, 0));//创建画笔对象
        CClientDC dc(this);
        CPen* pOldPen = dc.SelectObject(&pen);

        dc.MoveTo(arry[1].x, arry[1].y);    //剖分线
```

```
            dc.LineTo(arry[2].x, arry[2].y);
            dc.SelectObject(&pOldPen);

            for (int j = k; j < pNumbers - 1; j++)  //删除k顶点
            {
                points[j] = points[j + 1];
            }
            pNumbers--;

            Triangulation(points, pNumbers, number);
        }
    }
```