# 四、多边形的扫描转换算法原理和实践

## 1、原理

### 1）区域的“奇偶”性质

多边形扫描转换主要依据区域的一种“奇偶”性质，即一条直线与任意封闭的曲线相交时，总是从第一个交点进入内部，再从第二个交点退出，在交替的进入退出过程中对多边形进行填充。

### 2）活跃边表（Active Edge Table，AET）

用这个表存贮与当前扫描线相交的各边。每次离开一条扫描线进入下一条之前，将表中有但与下一条扫描线不相交的边清除出表，将与下一条扫描线相交而表中没有的边加入表中。**AET 中总是按 x 坐标递增排序**，因为在进行填充时，需要按该顺序判断是进入多边形内部还是从多边形内部出去。

### 3）边表（Edge Table，ET）

（1）在对多边形进行填充之前，首先应该建立多边形的边表来存储多边形的边的信息，边表是一种邻接表。**ET 中各登记项按 y 坐标递增排序**，每一登记项下的“吊桶”按所记 x 坐标递增排序，“吊桶”中各项的内容依次是：

① 边的另一端点的较大的 y 坐标 ymax。

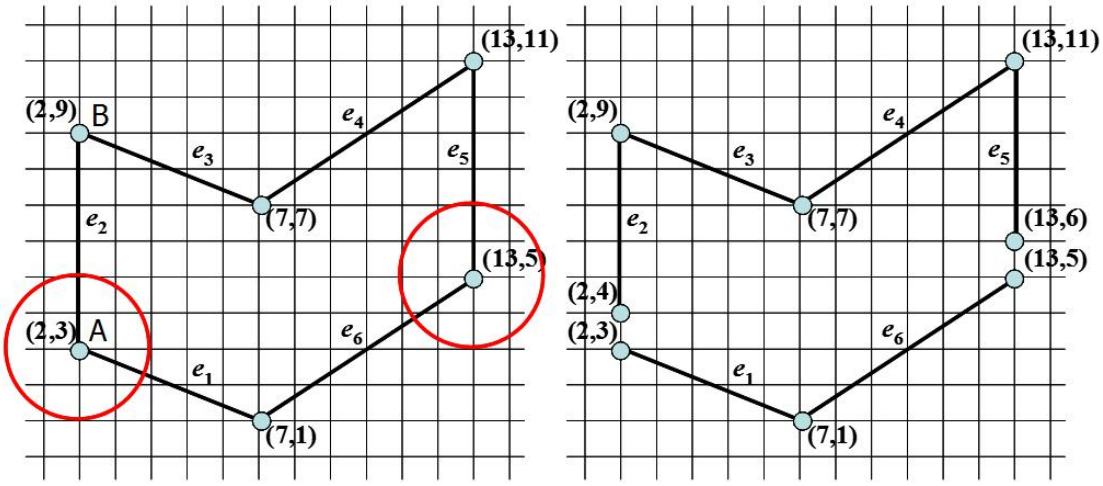
② 与较小的 y 坐标对应的边的端点的 x 坐标 xmin。

③ 斜率的倒数，即 1/m。

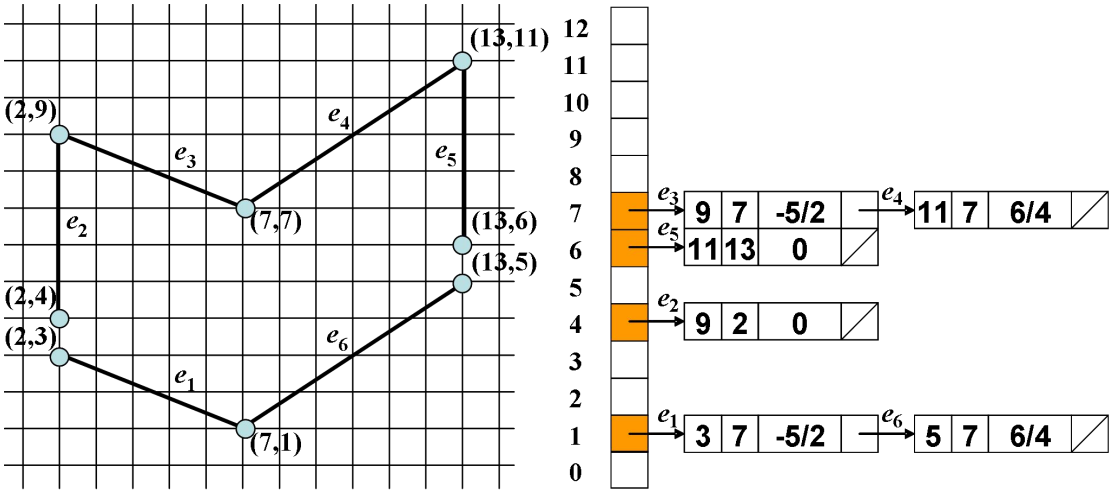
| $y_{max}$ | $x_{min}$ | 1/m | next |
|---|---|---|---|

（2）**注意：**在建立边表时，当顶点表现为是局部极大或局部极小时（下图中的 B 点），就看做是二个，否则看做一个。实际处理这个问题时，对局部极大或局部极小的顶点无需处理，而其它的顶点应该沿着边的方向缩进一个单位。

理由如下：例如在下图中，对于 B 点来来说，无需多做处理，因为当扫描线 y = 9 时，e2 和 e3 都会包含到 AET 中，也就是说 B 点在当前 AET 中会出现两次，当进行填充时，在该点会完成一次进入和退出，也就不会出现错误填充的情况。而对于 A 点来说，如果不进行缩进，当扫描线 y = 3 时，e1 和 e2 都会包含到 AET 中，所以 A 点同样在当前 AET 中出现两次，此时，AET 中有三个点，y = 3 这一行上 A 点之后的多边形内部区域并不会进行填充，发生错误，而且 AET 中应该永远包含偶数个点才对；进行缩进后，A 点在当前 AET 中只出现一次，并且此时 AET 中有偶数个点，填充正确。




（3）建立好的 ET 如下所示：




## 4）扫描过程

AET指针

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 初始化 | / | | | | | | | | | | | |
| 扫描线1 | $e_1$ | 3 | 7 | -5/2 | $e_6$ | 5 | 7 | 6/4 | / | | | |
| 扫描线2 | $e_1$ | 3 | $4^1/_2$ | -5/2 | $e_6$ | 5 | $8^1/_2$ | 6/4 | / | | | |
| 扫描线3 | $e_1$ | 3 | 2 | -5/2 | $e_6$ | 5 | 10 | 6/4 | / | | | |
| 扫描线4 | $e_2$ | 9 | 2 | 0 | $e_6$ | 5 | $11^1/_2$ | 6/4 | / | | | |
| 扫描线5 | $e_2$ | 9 | 2 | 0 | $e_6$ | 5 | 13 | 6/4 | / | | | |
| 扫描线6 | $e_2$ | 9 | 2 | 0 | $e_5$ | 11 | 13 | 0 | / | | | |
| 扫描线7 | $e_2$ | 9 | 2 | 0 | $e_3$ | 9 | 7 | -5/2 | $e_4$ | 11 | 7 | 6/4 | $e_5$ | 11 | 13 | 0 | / |
| 扫描线8 | $e_2$ | 9 | 2 | 0 | $e_3$ | 9 | $4^1/_2$ | -5/2 | $e_4$ | 11 | $8^1/_2$ | 6/4 | $e_5$ | 11 | 13 | 0 | / |
| 扫描线9 | $e_2$ | 9 | 2 | 0 | $e_3$ | 9 | 2 | -5/2 | $e_4$ | 11 | 10 | 6/4 | $e_5$ | 11 | 13 | 0 | / |
| 扫描线10 | $e_4$ | 11 | $11^1/_2$ | 6/4 | $e_5$ | 11 | 13 | 0 | / | | | | | | | | |
| 扫描线11 | $e_4$ | 11 | 13 | 6/4 | $e_5$ | 11 | 13 | 0 | / | | | | | | | | |
| 扫描线12 | / | | | | | | | | | | | | | | | | |

## 5）填充算法的伪代码

```
void Polygonfill(EdgeTable ET,   COLORREF color)
{
    1.y=边表 ET 中各登记项对应的 y 坐标中最小的值;
    2.活跃边表 AET 初始化为空表;
    3. while(ET 表中仍有扫描线未被处理) //处理 ET 表中的每一条扫描线
      {
        3.1 将 ET 中登记项 y 对应的各“吊桶”合并到表 AET 中，
            将 AET 中各吊桶按 x 坐标递增排序；
        3.2 在扫描线 y 上,按照 AET 表提供的 x 坐标对，用 color 实施填充；
        3.3 将 AET 表中有 y=ymax 的各项清除出表；
        3.4 对 AET 中留下的各项,分别将 x 换为 x+1/m.
        3.5 由于前一步可能破坏了 AET 表中各项 x 坐标的递增次序，
            故按 x 坐标重新排序；//非简单多边形
        3.6 y++，去处理下一条扫描线。
      }
}
```

## 2、实践

## 1）题目要求

通过鼠标输入顶点的方法绘制多边形，并实现多边形的扫描转换算法完成对该多边形的填充，要求使用“0908”数字图案对多边形内部进行填充，并且多边形边界为蓝色，填充颜色为红色。

## 2）分析

（1）定义一个 30*60 的 bool 型数组 m_pattern，用来存储需要填充的“0908”图案，在画像素点(x, y) 时进行判断，如果 m_pattern[y%30][x%60] 为 true 则进行填充，否则不进行填充。

（2）通过鼠标点击和移动绘制多边形的边，当多边形新绘制的点和第一个点的距离相差 5 个像素时，认为该多边形绘制完成。并且在绘制过程中完成边表的建立。对顶点进行处理时，取出当前点的前两个点，判断前一个点是否局部极大或局部极小，来决定是否要对前一个点进行缩进；并且在画最后一个顶点（也就是第一个顶点）时，既要判断前一个点是否要进行缩进，也要判断当前点是否要缩进。

（3）当最后一个点按下后，开始进行填充，开始时扫描线 y 值初始化为边表的第一个值（该值为多边形的最小的 y 值），之后扫描线逐步向上移动，然后按照边表更新活跃边表，并按区域的“奇偶”性质进行填充。

## 3）代码实现

（1）构建边表的部分代码：

```
//如果条横边，不用加入边表
if (m_lastPoint.y != point.y)
{
//当顶点表现为是局部极大或局部极小时，就看做是二个，否则看做一个
if (((m_beforLastPoint.y <= m_lastPoint.y) && (point.y <= m_lastPoint.y)) ||
    ((m_beforLastPoint.y >= m_lastPoint.y) && (point.y >= m_lastPoint.y)))
{
    //局部极大或局部极小时不作处理
    CPoint maxPoint = m_lastPoint.y > point.y ? m_lastPoint : point;
    CPoint minPoint = m_lastPoint.y > point.y ? point : m_lastPoint;

    pEdge edge = new Edge();
    edge->ymax = maxPoint.y;
    edge->xmin = minPoint.x;
    edge->next = NULL;
    edge->dx = (double)(maxPoint.x - minPoint.x) / (maxPoint.y - minPoint.y);

    //将边插入到正在绘制的多边形的边表中
    m_ETs.back()->insertEdge(minPoint.y, edge);
}
else
{
    //将 m_lastPoint 点缩进一个像素
    pEdge edge = new Edge();
    edge->dx = (double)(m_lastPoint.x - point.x) / (m_lastPoint.y - point.y);
    edge->next = NULL;

    if (m_lastPoint.y > point.y)
    {
        edge->ymax = m_lastPoint.y - 1;
        edge->xmin = point.x;

        //将边插入到正在绘制的多边形的边表中
        m_ETs.back()->insertEdge(point.y, edge);
    }
    else
    {
        edge->ymax = point.y;
        edge->xmin = m_lastPoint.x + edge->dx;
```

```
            //将边插入到正在绘制的多边形的边表中
            m_ETs.back()->insertEdge(m_lastPoint.y + 1, edge);
        }
}
```

（2）多边形的扫描转换算法实现：

```
/*
    et 是要绘制多边形的边表
    color 是要填充的颜色
*/
void CTestView::Polygonfill(CDC * pDC, EdgeTable * et, COLORREF color)
{
    pDC->SetROP2(R2_COPYPEN);

    double y = et->vertexes.front()->ymin;       //扫描线
    double ymax = et->vertexes.back()->ymin;    //边表中的最大 y 坐标值

    if (m_AET)
    {
        delete m_AET;
        m_AET = NULL;
    }

    m_AET = new Vertex();
    m_AET->edge = NULL;

    pVertex tmp1 = NULL, tmp2 = NULL;
    pEdge tmp3 = NULL, tmp4 = NULL;
    while (m_AET->edge || y <=ymax)
    {
        //更新活跃边表
        tmp1 = m_AET;
        tmp2 = et->getVertex(y);
        m_AET = new Vertex();
        m_AET->edge = NULL;
        if (tmp1)
        {
            tmp3 = tmp1->edge;
            while (tmp3)
            {
                tmp4 = new Edge();
                tmp4->next = NULL;
                tmp4->dx = tmp3->dx;
                tmp4->xmin = tmp3->xmin;
                tmp4->ymax = tmp3->ymax;
```

```
                m_AET->insertEdge(tmp4);
                tmp3 = tmp3->next;
            }
        }
        if (tmp2)
        {
            tmp3 = tmp2->edge;
            while (tmp3)
            {
                tmp4 = new Edge();
                tmp4->next = NULL;
                tmp4->dx = tmp3->dx;
                tmp4->xmin = tmp3->xmin;
                tmp4->ymax = tmp3->ymax;

                m_AET->insertEdge(tmp4);
                tmp3 = tmp3->next;
            }
        }

        //tmp2 是从边表中取出的所以不能 delete
        if (tmp1)
        {
            delete tmp1;
            tmp1 = NULL;
        }

        //在扫描线 y 上,按照 AET 表提供的 x 坐标对，用 color 实施填充
        int yvalue = (int)y;
        int x1 = 0, x2 = 0;
        for (int i = 0; m_AET->getEdge(i); i += 2)
        {
            x1 = m_AET->getEdge(i)->xmin + 0.5 + 1;
            x2 = m_AET->getEdge(i + 1)->xmin + 0.5;
            for (; x1 < x2; x1++)
            {
                if (m_pattern[yvalue % 30][x1 % 60])
                {
                    pDC->SetPixel(x1, yvalue, color);
                }
            }
        }
```

```
            //将 AET 表中有 y=ymax 的各项清除出表
            m_AET->removeEdge(y);
            //对 AET 中留下的各项,分别将 xmin 加上 dx
            m_AET->addDx();
            //去处理下一条扫描线
            y++;
        }
}
```

## 4）实现效果