算法系列之十二：多边形区域填充算法 - - 扫描线填充算法（有序边表法） - oRbIt 的专栏 - CSDN博客

二、扫描线算法（Scan-Line Filling）

扫描线算法适合对矢量图形进行区域填充，只需要直到多边形区域的几何位置，不需要指定种子点，适合计算机自动进行图形处理的场合使用，比如电脑游戏和三维CAD软件的渲染等等。

对矢量多边形区域填充，算法核心还是求交。《计算几何与图形学有关的几种常用算法》一文给出了判断点与多边形关系的算法——扫描交点的奇偶数判断算法，利用此算法可以判断一个点是否在多边形内，也就是是否需要填充，但是实际工程中使用的填充算法都是只使用求交的思想，并不直接使用这种求交算法。究其原因，除了算法效率问题之外，还存在一个光栅图形设备和矢量之间的转换问题。比如某个点位于非常靠近边界的临界位置，用矢量算法判断这个点应该是在多边形内，但是光栅化后，这个点在光栅图形设备上看就有可能是在多边形外边（矢量点没有大小概念，光栅图形设备的点有大小概念），因此，适用于矢量图形的填充算法必须适应光栅图形设备。

2.1扫描线算法的基本思想

扫描线填充算法的基本思想是：用水平扫描线从上到下（或从下到上）扫描由多条首尾相连的线段构成的多边形，每根扫描线与多边形的某些边产生一系列交点。将这些交点按照x坐标排序，将排序后的点两两成对，作为线段的两个端点，以所填的颜色画水平直线。多边形被扫描完毕后，颜色填充也就完成了。扫描线填充算法也可以归纳为以下4个步骤：

（1） 求交，计算扫描线与多边形的交点

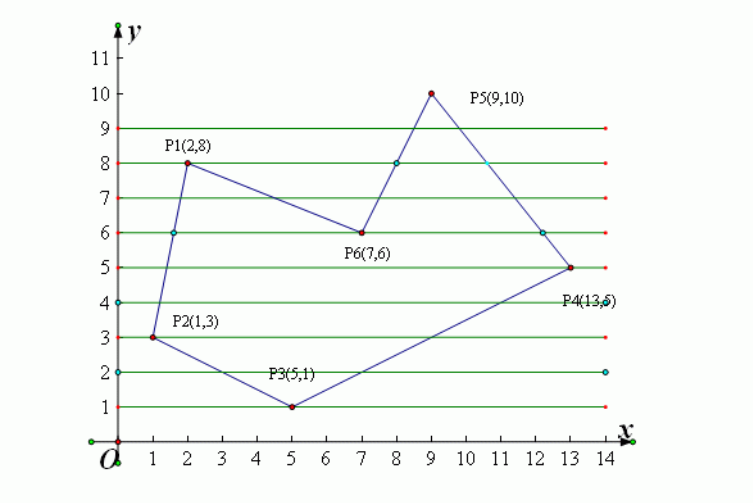
（2） 交点排序，对第2步得到的交点按照x值从小到大进行排序；

（3） 颜色填充，对排序后的交点两两组成一个水平线段，以画线段的方式进行颜色填充；

（4） 是否完成多边形扫描？如果是就结束算法，如果不是就改变扫描线，然后转第1步继续处理；

整个算法的关键是第1步，需要用尽量少的计算量求出交点，还要考虑交点是线段端点的特殊情况，最后，交点的步进计算最好是整数，便于光栅设备输出显示。

对于每一条扫描线，如果每次都按照正常的线段求交算法进行计算，则计算量大，而且效率底下，如图（6）所示：



图（6） 多边形与扫描线示意图

观察多边形与扫描线的交点情况，可以得到以下两个特点：

（1） 每次只有相关的几条边可能与扫描线有交点，不必对所有的边进行求交计算；

（2） 相邻的扫描线与同一直线段的交点存在步进关系，这个关系与直线段所在直线的斜率有关；

第一个特点是显而易见的，为了减少计算量，扫描线算法需要维护一张由“活动边”组成的表，称为“活动边表（AET）”。例如扫描线4的“活动边表”由P1P2和P3P4两条边组成，而扫描线7的“活动边表”由P1P2、P6P1、P5P6和P4P5四条边组成。

第二个特点可以进一步证明，假设当前扫描线与多边形的某一条边的交点已经通过直线段求交算法计算出来，得到交点的坐标为（x, y），则下一条扫描线与这条边的交点不需要再求交计算，通过步进关系可以直接得到新交点坐标为（x + △x, y + 1）。前面提到过，步进关系△x是个常量，与直线的斜率有关，下面就来推导这个△x。

假设多边形某条边所在的直线方程是：ax + by + c = 0，扫描线y_i和下一条扫描线y_{i+1}与该边的两个交点分别是（x_i，y_i）和（x_{i+1}，y_{i+1}），则可得到以下两个等式：

$ax_i + by_i + c = 0$ (等式 1)

$ax_{i+1} + by_{i+1} + c = 0$ (等式 2)

由等式1可以得到等式3：

$x_i = -(by_i + c) / a$ (等式 3)

同样，由等式2可以得到等式4：

$x_{i+1} = -(by_{i+1} + c) / a$ (等式 4)

由等式 4 – 等式3可得到

$x_{i+1} - x_i = -b (y_{i+1} - y_i) / a$

由于扫描线存在$y_{i+1} = y_i + 1$的关系，将代入上式即可得到：

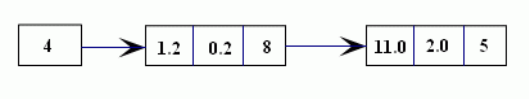
$x_{i+1} - x_i = -b / a$

即$\triangle x = -b / a$，是个常量（直线斜率的倒数）。

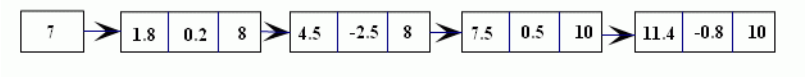
“活动边表”是扫描线填充算法的核心，整个算法都是围绕者这张表进行处理的。要完整的定义“活动边表”，需要先定义边的数据结构。每条边都和扫描线有个交点，扫描线填充算法只关注交点的x坐标。每当处理下一条扫描线时，根据△x直接计算出新扫描线与边的交点x坐标，可以避免复杂的求交计算。一条边不会一直待在“活动边表”中，当扫描线与之没有交点时，要将其从“活动边表”中删除，判断是否有交点的依据就是看扫描线y是否大于这条边两个端点的y坐标值，为此，需要记录边的y坐标的最大值。根据以上分析，边的数据结构可以定义如下：

```
65typedefstru
ct tagEDGE
66{
67double xi;
68double dx;
69int ymax;
74}EDGE;
```

根据EDGE的定义，扫描线4和扫描线7的“活动边表”就分别如图（7）和图(8)所示：

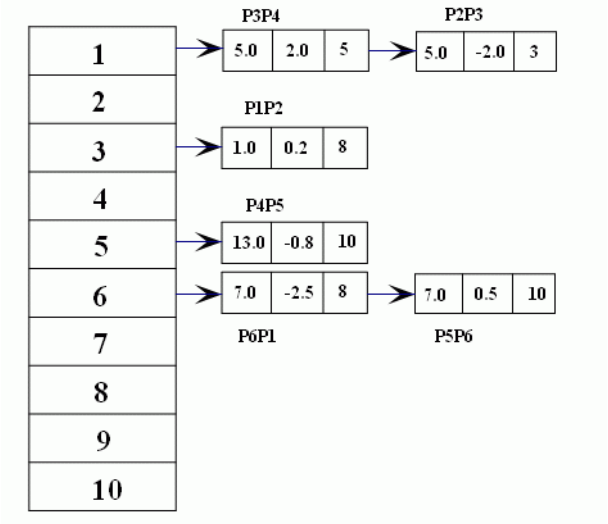


图（7） 扫描线4的活动边表



图（8） 扫描线7的活动边表

前面提到过，扫描线算法的核心就是围绕“活动边表（AET）”展开的，为了方便活性边表的建立与更新，我们为每一条扫描线建立一个“新边表（NET）”，存放该扫描线第一次出现的边。当算法处理到某条扫描线时，就将这条扫描线的“新边表”中的所有边逐一插入到“活动边表”中。“新边表”通常在算法开始时建立，建立“新边表”的规则就是：如果某条边的较低端点（y坐标较小的那个点）的y坐标与扫描线y相等，则该边就是扫描线y的新边，应该加入扫描线y的“新边表”。上例中各扫描线的“新边表”如下图所示：



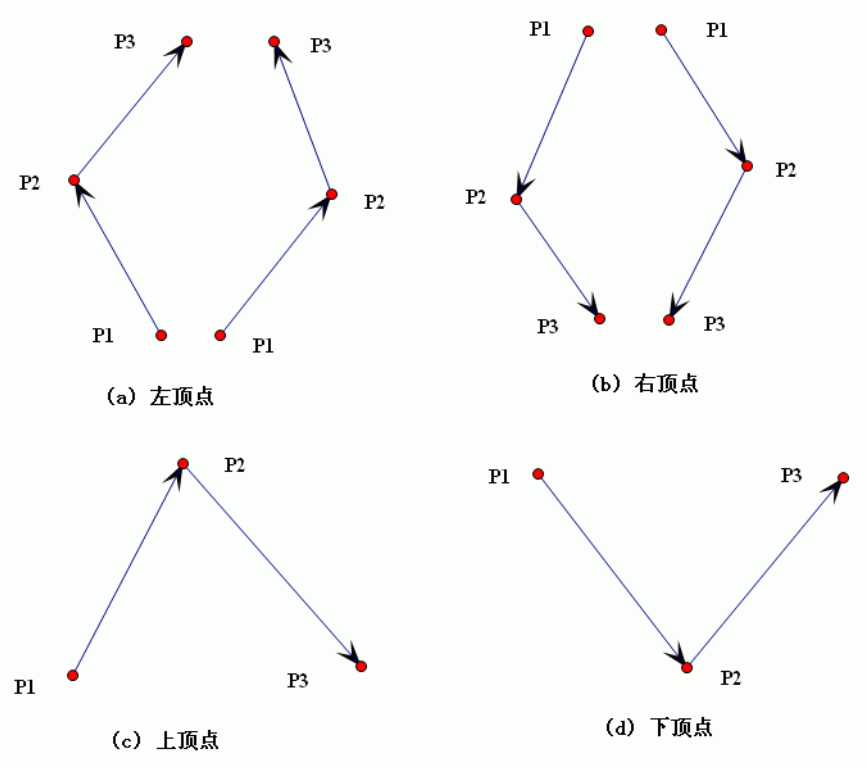
图（9） 各扫描线的新边表

讨论完“活动边表（AET）”和“新边表（NET）”，就可以开始算法的具体实现了，但是在进一步详细介绍实现算法之前，还有以下几个关键的细节问题需要明确：

（1） 多边形顶点处理

在对多边形的边进行求交的过程中，在两条边相连的顶点处会出现一些特殊情况，因为此时两条边会和扫描线各求的一个交点，也就是说，在顶点位置会出现两个交点。当出现这种情况的时候，会对填充产生影响，因为填充的过程是成对选择交点的过程，错误的计算交点个数，会造成填充异常。

假设多边形按照顶点P1、P2和P3的顺序产生两条相邻的边，P2就是所说的顶点。多边形的顶点一般有四种情况，如图（10）所展示的那样，分别被称为左顶点、右顶点、上顶点和下顶点：



图（10） 多边形顶点的四种类型

左顶点——P1、P2和P3的y坐标满足条件：y1 < y2 < y3；

右顶点——P1、P2和P3的y坐标满足条件：y1 > y2 > y3；

上顶点——P1、P2和P3的y坐标满足条件：y2 > y1 && y2 > y3；

下顶点——P1、P2和P3的y坐标满足条件：y2 < y1 && y2 < y3；

对于左顶点和右顶点的情况，如果不做特殊处理会导致奇偶奇数错误，常采用的修正方法是修改以顶点为终点的那条边的区间，将顶点排除在区间之外，也就是删除这条边的终点，这样在计算交点时，就可以少计算一个交点，平衡和交点奇偶个数。结合前文定义的“边”数据结构：EDGE，只要将该边的ymax修改为ymax – 1就可以了。

对于上顶点和下顶点，一种处理方法是将交点计算做0个，也就是修正两条边的区间，将交点从两条边中排除；另一种处理方法是不做特殊处理，就计算2个交点，这样也能保证交点奇偶个数平衡。

（2） 水平边的处理

水平边与扫描线重合，会产生很多交点，通常的做法是将水平边直接画出（填充），然后在后面的处理中就忽略水平边，不对其进行求交计算。

（3） 如何避免填充越过边界线

边界像素的取舍问题也需要特别注意。多边形的边界与扫描线会产生两个交点，填充时如果对两个交点以及之间的区域都填充，容易造成填充范围扩大，影响最终光栅图形化显示的填充效果。为此，人们提出了“左闭右开”的原则，简单解释就是，如果扫描线交点是1和9，则实际填充的区间是[1,9），即不包括x坐标是9的那个点。

2.2扫描线算法实现

扫描线算法的整个过程都是围绕“活动边表（AET）”展开的，为了正确初始化“活动边表”，需要初始化每条扫描线的“新边表（NET）”，首先定义“新边表”的数据结构。定义“新边表”为一个数组，数组的每个元素存放对应扫描线的所有“新边”。因此定义“新边表”如下：

```
510 std::vecto
r< std::list<E
DGE>> slNet
(ymax - ymin
+1);
```

ymax和ymin是多边形所有顶点中y坐标的最大值和最小值，用于界定扫描线的范围。slNet 中的第一个元素对应的是ymin所在的扫描线，以此类推，最后一个元素是ymax所在的扫描线。在开始对每条扫描线处理之前，需要先计算出多边形的ymax和ymin并初始化“新边表”：

```
503void ScanLinePolygonFill(const Polygon& py,int color)
504{
505 assert(py.IsValid());
506
507int ymin = 0;
508int ymax =0;
509 GetPolygonMinMax(py, ymin, ymax);
510 std::vector< std::list<EDGE>> slNet(ymax - ymin +1);
511 InitScanLineNewEdgeTable(slNet, py, ymin, ymax);
512//PrintNewEdgeTable(slNet);
513 HorizonEdgeFill(py, color);//水平边直接画线填充
514 ProcessScanLineFill(slNet, ymin, ymax, color);
515}
```

InitScanLineNewEdgeTable()函数根据多边形的顶点和边的情况初始化“新边表”，实现过程中体现了对左顶点和右顶点的区间修正原则：

```
#include<stdio.h>
#include<math.h>
#include<algorithm>
#include<iostream>
using namespace std;
const int MAXN=1000;

struct point
{
    int x,y;
};
point list[MAXN];
int stack[MAXN],top;

int cross(point p0,point p1,point p2) //计算叉积  p0p1 X p0p2
{
    return (p1.x-p0.x)*(p2.y-p0.y)-(p1.y-p0.y)*(p2.x-p0.x);
}
double dis(point p1,point p2)  //计算 p1p2的 距离
{
    return sqrt((double)(p2.x-p1.x)*(p2.x-p1.x)+(p2.y-p1.y)*(p2
```

```
.y-p1.y));
}
bool cmp(point p1,point p2) //极角排序函数， 角度相同则距离小的在前面
{
    int tmp=cross(list[0],p1,p2);
    if(tmp>0) return true;
    else if(tmp==0&&dis(list[0],p1)<dis(list[0],p2)) return true;
    else return false;
}
void init(int n) //输入，并把 最左下方的点放在 list[0] 。并且进行极角排序
{
    int i,k;
    point p0;
    scanf("%d%d",&list[0].x,&list[0].y);
    p0.x=list[0].x;
    p0.y=list[0].y;
    k=0;
    for(i=1;i<n;i++)
    {
        scanf("%d%d",&list[i].x,&list[i].y);
        if( (p0.y>list[i].y) || ((p0.y==list[i].y)&&(p0.x>list[i].x)) )
        {
            p0.x=list[i].x;
            p0.y=list[i].y;
            k=i;
        }
    }
    list[k]=list[0];
    list[0]=p0;

    sort(list+1,list+n,cmp);
}

void graham(int n)
{
    int i;
    if(n==1) {top=0;stack[0]=0;}
    if(n==2)
    {
        top=1;
```

```
        stack[0]=0;
        stack[1]=1;
    }
    if(n>2)
    {
        for(i=0;i<=1;i++) stack[i]=i;
        top=1;

        for(i=2;i<n;i++)
        {
            while(top>0&&cross(list[stack[top-1]],list[stack[top]],list[i])<=0) top--;
            top++;
            stack[top]=i;
        }
    }
}
```

多边形的定义Polygon和本系列第一篇《计算几何与图形学有关的几种常用算法》一文中的定义一致，此处就不再重复说明。算法通过遍历所有的顶点获得边的信息，然后根据与此边有关的前后两个顶点的情况确定此边的ymax是否需要-1修正。ps和pe分别是当前处理边的起点和终点，pss是起点的前一个相邻点，pee是终点的后一个相邻点，pss和pee用于辅助判断ps和pe两个点是否是左顶点或右顶点，然后根据判断结果对此边的ymax进行-1修正，算法实现非常简单，注意与扫描线平行的边是不处理的，因为水平边直接在HorizonEdgeFill()函数中填充了。

ProcessScanLineFill()函数开始对每条扫描线进行处理，对每条扫描线的处理有四个操作，如下代码所示，四个操作分别被封装到四个函数中：

```
467void ProcessScanLineFill(std::vector< std::list<EDGE>>& slNet,
468int ymin,int ymax,int color)
469{
470 std::list<EDGE> aet;
471
472for(int y = ymin; y <= ymax; y++)
473{
474 InsertNetListToAet(slNet[y - ymin], aet);
475 FillAetScanLine(aet, y, color);
476//删除非活动边
477 RemoveNonActiveEdgeFromAet(aet, y);
478//更新活动边表中每项的xi值，并根据xi重新排序
479 UpdateAndResortAet(aet);
480}
481}
```

InsertNetListToAet()函数负责将扫描线对应的所有新边插入到aet中，插入操作到保证aet还是有序表，应用了插入排序的思想，实现简单，此处不多解释。FillAetScanLine()函数执行具体的填充动作，它将aet中的边交点成对取出组成填充区间，然后根据“左闭右开”的原则对每个区间填充，实现也很简单，此处不多解释。RemoveNonActiveEdgeFromAet()函数负责将对下一条扫描线来说已经不是“活动边”的边从aet中删除，删除的条件就是当前扫描线y与边的ymax相等，如果有多条边满足这个条件，则一并全部删除：

```
439bool IsEdgeOutOfActive(EDGE e,int y)
440{
441return(e.ymax == y);
442}
443
444void RemoveNonActiveEdgeFromAet(std::list<EDGE>& aet,int y)
445{
446 aet.remove_if(std::bind2nd(std::ptr_fun(IsEdgeOutOfActive),y));
447}
```

UpdateAndResortAet()函数更新边表中每项的xi值，就是根据扫描线的连贯性用dx对其进行修正，并且根据xi从小到大的原则对更新后的aet表重新排序：

```
449void UpdateAetEdgeInfo(EDGE& e)
450{
451 e.xi += e.dx;
452}
453
454bool EdgeXiComparator(EDGE& e1,EDGE& e2)
455{
456return(e1.xi <= e2.xi);
457}
458
459void UpdateAndResortAet(std::list<EDGE>& aet)
460{
461//更新xi
462 for_each(aet.begin(), aet.end(), UpdateAetEdgeInfo);
463//根据xi从小到大重新排序
464 aet.sort(EdgeXiComparator);
465}
```

其实更新完xi后对aet表的重新排序是可以避免的，只要在维护aet时，除了保证xi从小到大的排序外，在xi相同的情况下如果能保证修正量dx也是从小到大有序，就可以避免每次对aet进行重新排序。算法实现也很简单，只需要对InsertNetListToAet()函数稍作修改即可，有兴趣的朋友可以自行修改。

至此，扫描线算法就介绍完了，算法的思想看似复杂，实际上并不难，从具体算法的实现就可以看出来，整个算法实现不足百行代码。

<下一篇：一种改进的扫描线算法>