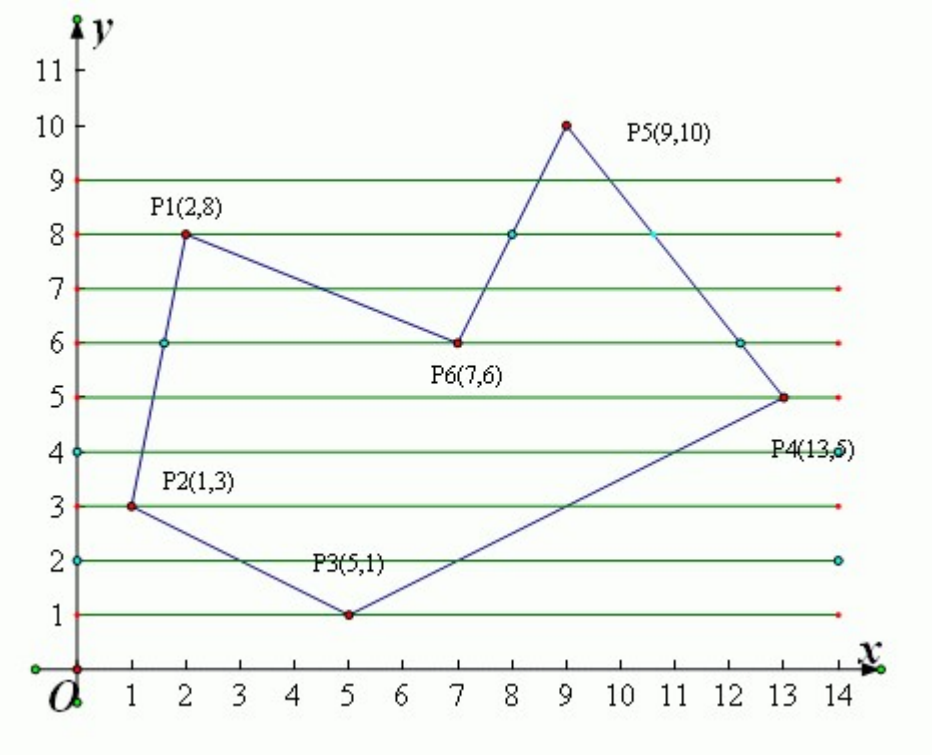
二：扫描线算法(有序边表法)：

以前有人提出过这样一种算法，如果一个点在多边形内部，则绘制该点，否则不绘制，也就是说需要对屏幕上的每一个点进行判断，这样的话无用计算量太大了，不在多边形内部的像素也需要判断。所以有人发明了一种使用扫描线填充的算法，这种算法只需要处理多边形内部的像素即可。

2.1 扫描线算法的基本思想：

用水平扫描线从上到下(或者从下到上) 对多边形进行扫描，每条扫描线会和多边形的某些边相交，产生一系列的交点，然后将这些交点先按照 x 坐标排序，两两成对作为线段的的两个端点，用所选颜色绘制这些水平线段，当多边形被扫描完毕时，颜色也填充完成了。如下图所示：



当扫描线多边形时，只需要求出每条扫描线和多边形的交点，然后绘制这些交点中间的线段即可绘制出该多边形。如当扫描线行号为 2 时，该扫描线和多边形的交点为(3,2)和(9,2)，对于该条扫描线来说，只要绘制出(3,2)到(9,2)这条线段即可。

2.2 扫描线和多边形求交的问题

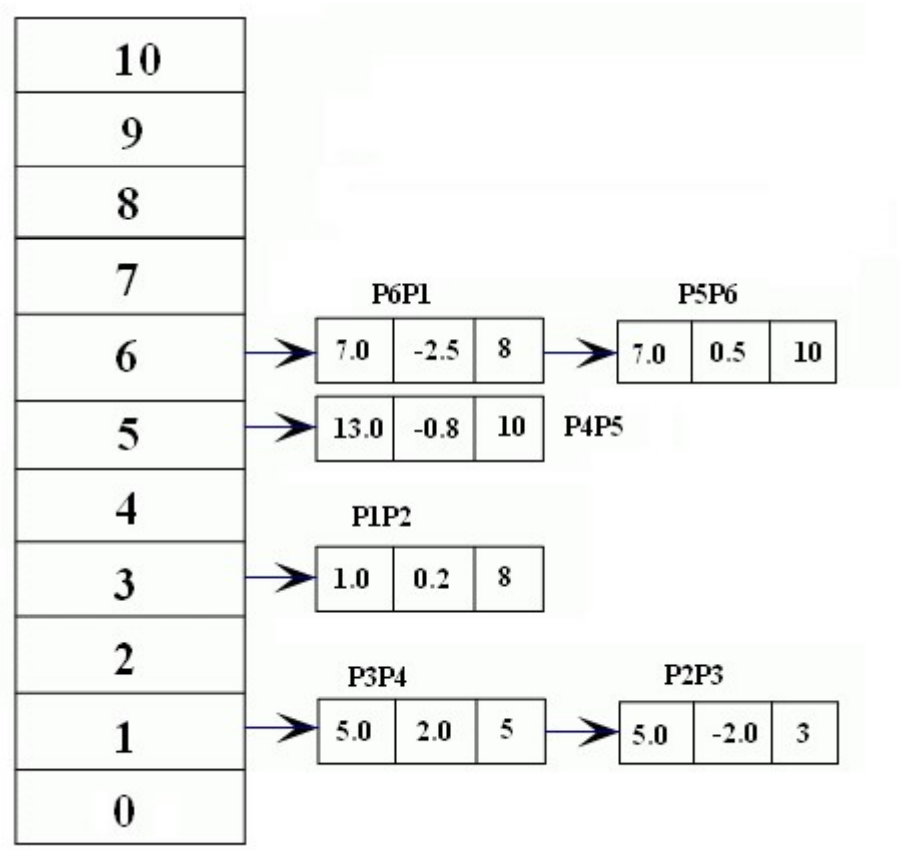
如果每条扫描线都要和多边形求一次交点，那么计算量将会非常的大。通过对上图的观察我们可以发现以下两点：

1. 每条扫描线只会和多边形有限的几条边相交，所以不必对所有的边都求一次交，这和边的起终点相关。
2. 相邻的扫描线对同一条边的交点存在一个步进关系，这个步进和边的斜率相关。

我们只需要记录每条边的起点，终点，斜率就能确定哪些扫描线和边相交以及这些扫描线和边的交点。

2.3 使用了以上特性的多边形扫描线算法

创建一个新边表 NET(new edge table)如下：



（NET）

左边的每一行表示边起点的 Y 值，有的行没有指向任何元素，表示没有任何边以该行作为起点。而每个元素中的第一个数字表示边起点的 X 值，第二个数字表示当扫描线每增加一行时，该条边的 X 值的增量，第三个数字表示该条边终点的 Y 值，对于一条边来说，起点是 Y 值较小的那个顶点，如果边和扫描线平行，认为这条边和任何扫描线都没有交点。

接下来构建一个链表 AEL(active edge list)用于表示当前扫描线和多边形边的相交情况，这些交点必定是成对出现在 AEL 中的，将这些交点按照 X 值的大小从小到大排序(如果 X 一样则按照 dx 排序，因为下一条扫描线会导致 dx 大的边 X 更大)，并且两两取出，绘制一条横着的线段<(Xa,P),(Xb,P)>即可。AEL 从下往上处理扫描线(扫描区间为多边形的最小 Y 值到多边形的最大 Y 值)，遇到 NET 中有新的边信息则将这些边添加到 AEL 中，并且在处理完当前扫描线之后会对 AEL 中的每个数据做出如下处理：

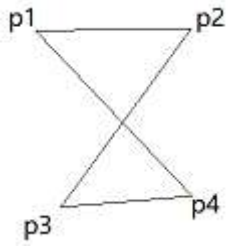
1. 如果当前扫描线已经是某条边的最大 Y 值(终点的 Y 值)，则将该边从 AEL 中移除；
2. 剩下的所有边的 X 值增加该边 X 增量大小。

在对 AEL 排序时可以选择不同的时机对其排序：

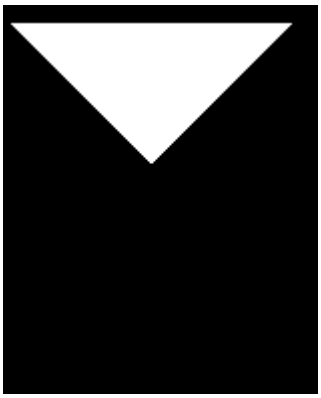
1. 在每次将 AEL 移动到下一行时进行排序，代码如下(FillPolygon1/FillPolygon/main.cpp 61 行)

```
if (!NET[y].empty())//如果当前扫描线对应的NET不为空
{
    AEL.splice(AEL.end(), NET[y]);//将其添加到AEL中
}
AEL.sort(SortEdge);//将边排序
```

2. 在每次往 AEL 中添加新元素的时候排序，可以减少计算量，但是如果多边形的边有自交，则会丢失交点往上的内容,如下图所示：



这里边<p2,p3>和边<p1,p4>出现了自相交，绘制出来的图案如下图:



少了下面的一部分。

见代码(FillPolygon2/FillPolygon /main.cpp 61 行):

如果能够确定该多边形不会自交，应该用方案 2，这样可减少计算量。