

# 多边形三角剖分：画廊看守

October 17, 2021

## 1 看守与三角剖分

出自名家的绘画，心动的不止是艺术爱好者，罪犯亦如此。所以艺术画廊都对其拥有作品严加看管。白天由值班人员看守，晚上就由摄像机来看守。这样就引出了一个著名的 Art Gallery 问题：对于一个多边形的画廊，需要多少个摄像头（360 度无死角）才能完全覆盖？

为了确切艺术画廊问题的定义，需先将画廊的概念做形式化处理。画廊是三维空间，通过它的平面结构图，就可以确定摄像机的安放位置。因此，可以利用平面多边形的模型来表示画廊。还进一步做出限制，要求画廊的模型应是简单多边形—即由单个不自交的、封闭的多边形链所围出的区域。

为了看守一个简单多边形，需要多少台摄像机？这取决于具体的多边形：多边形越复杂，需要的摄像机就越多。因此，将根据多边形的顶点数  $n$ ，来界定摄像机的数量。即使顶点数相等的两个多边形，看守难度可能不一样。为了保险起见，我们将考虑最坏的情况—给出的只是一个上界，该上界适用于由  $n$  个顶点组成的所有简单多边形。

设  $P$  为包含  $n$  个顶点的简单多边形。在确定看守  $P$  所需摄像机的最小数目时，由于  $P$  的形状可能极为复杂，所以我们似乎无从下手。首先将  $P$  分解为很多块，每一块都很容易看守——“块”就是三角形。为完成这种分解，需要添加一些对角线，将某些顶点对联接起来。所谓对角线是一条开的线段，它联接于  $P$  的某两个顶点之间，而且完全落在  $P$  的内部。通过极大的一组互不相交的对角线，可将一个多边形分解为多个三角形——称作该多边形的三角剖分，参见图 1。通常，简单多边形的三角剖分不是唯一的。例如图 1 所示的这个多边形，就有多种不同的三角剖分方案。给定  $P$  的一个三角剖分  $T_P$ ，只要在每个三角形中放置一台摄像机，就可以实现对整个多边形的看守。然而，是否每个简单多边形总存在一个三角剖分呢？如果存在，其中三角形的数目又是多少呢？下面这则定理回答了这些问题。

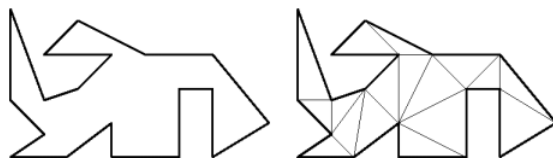


Figure 1: 一个简单多边形，及其可能的一个三角剖分

**定理 1** 任何简单多边形都存在（至少）一个三角剖分；若其顶点数目为  $n$ ，则它的每个三角剖分都恰好包含  $n - 2$  个三角形。

由此定理可得推论：包含  $n$  个顶点的任一简单多边形，都可用  $n-2$  台摄像机来看守。一个三角形配一台摄像机有些浪费。如果挑选出若干对角线，然后安装摄像机，就可能将摄像机的总数减少到大约  $n/2$ 。而更好的策略是，将摄像机安装在（多边形的）顶点上——毕竟，一个顶点可能同时与更多的三角形相关联，这样只需一台摄像机，就可以将与之相关联的所有三角形都看守住。这样，就导出了下面的方法。

令  $T_P$  为  $P$  的一个三角剖分。选出  $P$  的部分顶点组成一个子集，使得  $T_P$  中的每个三角形，都有至少一个顶点来自于该子集；然后，在被挑选出的每个顶点处，分别放置一台摄像机。为了找出这样一个子集，可以使用白、灰和黑三种颜色，给  $P$  的所有顶点染色（如图 2 所示）。染色方案必须满足：由任何边或者对角线联接的两个顶点，所染的颜色不能相同——称作“对经过三角剖分后的多边形的 3-染色”。三角剖分后的多边形经过如此染色，其中每个三角形都有（且仅有）一个白色、灰色和黑色的顶点。因此，只要在同色（比如灰色）的各顶点处分别放置一台摄像机，就必然可以看守整个多边形。进一步地，若选用点数最少的那一类同色顶点，并为它们配备摄像机，则只需不超过  $\lfloor n/3 \rfloor$  台摄像机，即可看守住  $P$ 。

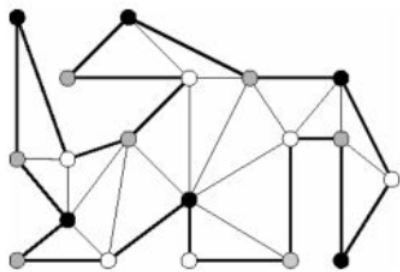


Figure 2: 根据三角剖分对顶点进行 3-染色

然而，3-染色方案是否总是存在？答案是肯定的。为了解这结论，来看看所谓“ $T_P$  的对偶图”——记为  $G(T_P)$ 。对应  $T_P$  中的每个三角形， $G(T_P)$  都有一个顶点。将对应顶点  $v$  的三角形记  $t(v)$ 。若  $t(v)$  与  $t(u)$  共用一条对角线，则在  $v$  和  $u$  之间就设置一条弧。这样， $G(T_P)$  中的各条弧就分别对应  $T_P$  中的各条对角线。任何一条对角线都会将  $P$  一分为二，故移去  $G(T_P)$  的任意一条弧， $G(T_P)$  都会分裂成两个（各自连通的）部分。因此， $G(T_P)$  必然是一棵树（如果允许多边形内有空洞，这个结论就不一定成立）。只要对该图进行一次（比如深搜）遍历，就可以得到一种 3-染色方案。具体做法：在深度优先遍历的过程中，始终都保证一点：已经访问过的三角形的所有顶点，都已被染上了白色、灰色或黑色；而且，任何一对（通过对角线或边）相互联接的顶点，颜色互异。由此保证：在访问完所有的三角形之后，可得到一个 3-染色方案。深度优先遍历可从  $G(T_P)$  的任一顶点开始；第一个被访问的三角形，其三个顶点将分别被染上白色、灰色或黑色（次序无所谓）。现在，假设从  $G$  的一个顶点  $u$  到达另一个顶点  $v$ 。那么  $t(v)$  和  $t(u)$  之间肯定存在一条公共对角线。由于  $t(u)$  的三个顶点都已经被染上了互异的颜色，所以  $t(v)$  的三个顶点中只有一个顶点需要染色。而且，只有一种颜色可供它使用——准确地，就是  $t(v)$  与  $t(u)$  之间公共对角线所没有用到的那种颜色。 $G(T_P)$  是一棵树，与  $v$  相邻（除  $u$  之外）的其它顶点都尚未访问到，因此的确可以将剩下的这一颜色赋给这个顶点。

总而言之，对于经过三角剖分的任意简单多边形，都能对其实施 3-染色。只需  $\lfloor n/3 \rfloor$  台摄像机，就可以看守住任何一个（包含  $n$  个顶点的）简单多边形。不过，我们的成本还能更低。毕竟，放置在顶点处的一台摄像机，其能够看守的范围，可能不止是与之相关联的那些三角形。然而不幸的是，对任何  $n \geq 3$ ，都存在一个（包含  $n$  个顶点的）简单多边形，它的确需要  $\lfloor n/3 \rfloor$  台摄像机。这样的一个例子就是所谓的“梳状多边形”：如图 3 所示，它有一条长长的水平基边，以及  $\lfloor n/3 \rfloor$  个分别由两条边形成的“梳齿”。任何两个相邻的梳齿之间，由一条水平边相联。只要适当地安排各顶点的位置，就总能够保证：单台摄像机无论放置在多边形内的什么位置，都不可能同时看到两个梳齿。因此，我们不能指望能够依靠某种策略，每次都找到少于  $\lfloor n/3 \rfloor$  台摄像机。换言之，就最坏情况来而言，上述 3-染色的方法已经是最优的了。

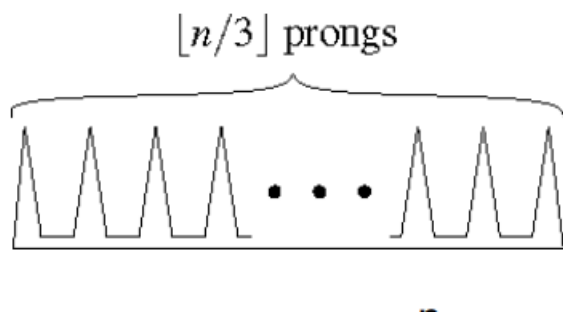


Figure 3: 梳状  $n$  边形需要于  $\lfloor n/3 \rfloor$

上述就证明了组合几何学的一个经典结果：

**定理 2** (艺术画廊定理) 包含  $n$  个顶点的任何简单多边形，只需（放置在适当位置的） $\lfloor n/3 \rfloor$  台摄像机就能保证：其中任何一点都可见于至少一台摄像机。有的时候，的确需要这样多台摄像机。

现在我们已经知道， $\lfloor n/3 \rfloor$  台摄像机总是够用的。然而，我们没有有效的算法，来计算出各台摄像机的具体位置。现需要一个快速的算法，以实现任何简单多边形的三角剖分。同时，通过该算法，

还应该能够导出一个合理的数据结构（比方说，双向链接边表），来表示三角剖分后的结果——这样，（在遍历时）只需常数时间，就可以从一个三角形转到它的一个邻居。一旦已经得到了这种形式的结构表示，就可以在线性时间内，按照上述方法——深度优先遍历对偶图，完成 3-染色，按照颜色将所有顶点分为三类，取出数量最少的一类顶点，并在这类顶点处放置摄像机——确定总数不超过  $\lfloor n/3 \rfloor$  台摄像机的具体位置。接下来一节，将介绍如何在  $O(n \log n)$  时间内构造一个三角剖分。提前借用这一结果，就可以得出下面有关多边形看守的最后结论：

**定理 3** 任给一个包含  $n$  个顶点的简单多边形  $P$ 。总可以在  $O(n \log n)$  时间内，在  $P$  中确定  $\lfloor n/3 \rfloor$  台摄像机的位置，使得  $P$  中的任何一点都可见于其中的至少一台摄像机

## 2 多边形的单调块划分

任给一个包含  $n$  个顶点的简单多边形  $P$ 。根据定理 1， $P$  的三角剖分总是存在。那个定理的证明本身就是构造式的，故马上就可以由此导出一个递归的三角剖分算法：找到一条对角线，将原多边形切分为两个子多边形，然后递归地对两个子多边形实施三角剖分。为了找到这样一条对角线，我们找出  $P$  中最靠左的顶点  $v$ ，然后试着将与  $v$  相邻的两个顶点  $u$  和  $w$  联接起来；如果不能直接联接这两个顶点，就在由  $u$ 、 $v$  和  $w$  确定的三角形内，找出距离  $uw$  最远的那个顶点，然后将它与  $v$  联接起来。按照这种方法，需要花费线性的时间才能找到一条对角线。而且，（在最坏情况下）这条对角线将  $P$  切分为一个三角形，以及一个含有  $n-1$  个顶点的多边形。我们的确可能一直都是联接  $u$  和  $w$ ——这就是最坏情况。在这种最坏情况下，上述三角剖分算法需要运行平方量级的时间。能否更快呢？对于某些类型的多边形，的确可以更快。

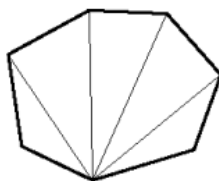


Figure 4: 凸多边形的三角剖分可以在线性时间内构造出来

比如凸多边形 (convex polygon) 就很容易：如图 4 所示，取出多边形的任何一个顶点，除了它的两个邻居之外，在这个顶点与其它的所有顶点之间分别联接一条对角线。整个过程只需要线性的时间。因此，对非凸多边形进行三角剖分的一种可能的方法就是：首先将  $P$  划分为多个凸块，然后分别对每块做三角剖分。然而不幸的是，将多边形划分为凸块的难度，与对它做三角剖分是一样的。因此，我们将把  $P$  划分为所谓的“单调块” (monotone piece)——这项工作要容易得多。

一个简单多边形称作“关于某条直线  $l$  单调” (monotone with respect to a line  $l$ )，如果对任何一条垂直于  $l$  的直线  $l'$ ， $l'$  与该多边形的交都是连通的。换言之，它们的交或者是一条线段，或者是一个点，也可能是空集

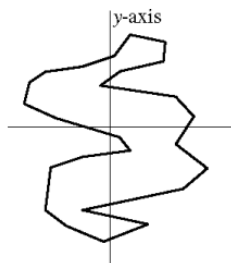


Figure 5: 单调多边形 (monotone polygon)

如果一个多边形关于  $y$  坐标轴单调 (图 5)，则称它是  $y$ -单调的 ( $y$ -monotone)。下面这个性质，是  $y$ -单调多边形 ( $y$  monotone polygon) 的一个特征：在沿着多边形的左 (右) 边界，从最高顶点走向最低顶点的过程中，我们始终都是朝下方 (或者水平) 运动，而绝不会向上。

我们对多边形  $P$  进行三角剖分的策略是：首先将  $P$  划分成若干个  $y$ -单调块，然后再对每块分别进行三角剖分。可以按照下面的方法，将一个多边形划分成单调块。设想我们沿着  $P$  的左或右边界，从其最高顶点走向最低顶点。在某些顶点处，我们的行进方向可能会从向下转成向上，或者从向上转成向下—这些位置称作拐点 (turn vertex)。为了将  $P$  划分成多个  $y$ -单调块，就必须消除这些拐点。为此可以引入对角线。如图 6 所示，若在某个拐点  $v$  处，与之关联的两条边都朝下，而且在此局部，多边形的内部位于  $v$  的上方，那么就构造一条从  $v$  出发、向上联接的对角线。



Figure 6: 通过引入对角线消除拐点

这条对角线将原多边形一分为二，而且在划分出来的两块中，顶点  $v$  都会出现。此外，在其中的任何一块中，与  $v$  相关联的两条边，必然有一条朝下（具体讲，就是从原多边形中继承下来的那条边），而另一条则朝上（也就是所引入的对角线）。也就是说，在两个子块中， $v$  都不再是一个拐点。如果与  $v$  相关联的两条边都朝上，而且在此局部，多边形的内部位于  $v$  的下方，那么就需要构造一条从  $v$  出发、向下联接的对角线。显然，拐点有多种不同类型，故需要更加准确地加以区分。

为了更加仔细地对不同类型的拐点做出定义，需要特别注意那些  $y$ -坐标相同的顶点。为此，要定义好“下方”和“上方”的概念：所谓“点  $p$  处于点  $q$  的下方”，是指  $p_y < q_y$ ，或者  $p_y = q_y$  而  $p_x > q_x$ ；而所谓“点  $p$  处于点  $q$  的上方”，是指  $p_y > q_y$ ，或者  $p_y = q_y$  而  $p_x < q_x$ 。（你可以想象着相对于原来的坐标系，沿顺时针方向，将整个平面旋转“一丁点”——这样，任何两个点都不会具有相同的  $y$ -坐标，而且上面所定义的上/下关系，在旋转后的平面上依然保持不变。）

$P$  的顶点可划分为五类（参见图 7）。其中四类都是拐点：起始顶点、分裂顶点、终止顶点以及汇合顶点。它们的定义如下。顶点  $v$  是一个起始顶点 (start vertex)，如果与它相邻的两个顶点的高度都比它低，而且在  $v$  处的内角小于  $\pi$ ；如果该内角大于  $\pi$ ， $v$  就是一个分裂顶点 (split vertex)。（注意，既然与  $v$  相邻的两个顶点都比  $v$  更低，此处的内角就不可能等于  $\pi$ 。）顶点  $v$  是一个终止顶点 (end vertex)，如果与它相邻的两个顶点的高度都比它高，而且在  $v$  处的内角小于  $\pi$ ；如果该内角大于  $\pi$ ， $v$  就是一个汇合顶点 (merge vertex)。这四类拐点以外的所有顶点，都是普通顶点 (regular vertex)。也就是说，在每个普通顶点的两个相邻顶点中，必然有一个比它高，而另一个则比它低。之所以要给不同类型的顶点取这样的名字，是因为我们的算法要进行一次自上而下的平面扫描，在此过程中，要维护扫描线与多边形的交集。当扫描线触及一个分裂顶点时，交集的某个（连通的）部分就要分裂；当扫描线触及一个汇合顶点时，则有两个（连通的）部分会汇合起来；诸如此类。

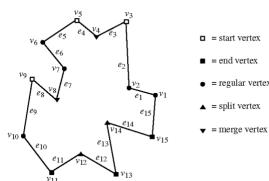


Figure 7: 五种类型的顶点

多边形中局部的非单调性，正来自于这些分裂顶点和汇合顶点。而且反过来，下面这个命题看似更强，却也竟然是成立的：

**定理 4** 一个多边形若既不含分裂顶点，也不含汇合顶点，则必然是  $y$ -单调的。

根据定理 3，只要将其中的分裂顶点和汇合顶点都消除掉，也就完成了将  $P$  划分为多个  $y$ -单调块的任务。为此，需要在每个分裂顶点处增加一条向上的对角线，也要在每个汇合顶点处增加一条向下的对角线。当然，这些对角线必须互不相交。一旦这些工作完成， $P$  也就已经被划分为多个  $y$ -单调块了。

首先来看看，在一个分裂顶点处应该如何引入一条对角线。这里采用平面扫描的方法。按照顺时针的方向，令  $P$  的所有顶点排列为  $v_1, v_2, \dots, v_n$ 。再令  $P$  的各边为  $e_1, e_2, \dots, e_n$ ，其中对任何的  $1 \leq i < n$ ，都有  $e_i = \overline{v_i v_{i+1}}$ ；另外， $e_n = \overline{v_n v_1}$ 。按照平面扫描算法，一条假想的水平扫描线  $l$  自上而

下地扫过整个平面。在一些被称为事件点 (event point) 的位置, 扫描线会稍做停留。就目前这一问题而言, 这些事件点包括  $P$  的所有顶点; 不过, 在整个扫描的过程中, 不会产生任何新的事件点。所有的事件点被组织成一个事件队列 (event queue)  $Q$ 。该事件队列实际上是一个优先队列, 各顶点的优先级就是其各自的  $y$ -坐标。如果两个顶点的  $y$ -坐标相同, 则居于左边 ( $x$ -坐标更小) 的那个顶点具有更高的优先级。这样, 每次只需  $O(\log n)$  时间, 就可以找出下一待处理的顶点。(既然在扫描过程中不会出现新的事件, 不妨在扫描之前将所有顶点按照  $y$ -坐标排一次序—经过这一预处理, 每次只需  $O(1)$  时间就可以确定下一事件点。)

扫描的目的, 是为了将每个分裂顶点, 与位于其上方的某个顶点联接起来, 从而引入一条对角线。如图 8 所示, 试考虑扫描线触及某个分裂顶点  $v_i$  的时刻。此时, 应该将  $v_i$  与哪个顶点相联呢? 与  $v_i$  相距较近的顶点, 是一个不错的选择—这样, 在将它与  $v_i$  联接起来之后, 连线不与  $P$  的任何边相交的可能性更大。让我们更准确地做一解释。沿着当前的扫描线, 令居于  $v_i$  的左侧、与之相邻的那条边为  $e_j$ ; 令居于  $v_i$  的右侧、与之相邻的那条边为  $e_k$ 。

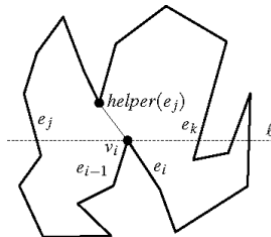


Figure 8: 分裂顶点的处理

现在考虑介于  $e_j$  和  $e_k$  之间、位于  $v_i$  上方的那些顶点, 若这些顶点至少存在一个, 则总可以将其中最低的那个与  $v_i$  联接起来 (构成一条合法的对角线)。若这类顶点根本不存在, 则可将  $v_i$  与  $e_j$  或  $e_k$  的上端点联接起来。无论如何, 我们都将这个顶点称作 “ $e_j$  的助手” (helper of  $e_j$ ), 记作  $\text{helper}(e_j)$ 。按照正式的定义,  $\text{helper}(e_j)$  应该是 “在位于扫描线上方、通过一条完全落在  $P$  内部的水平线段与  $e_j$  相联的那些顶点中, 高度最低的那个顶点”。请注意,  $\text{helper}(e_j)$  可能就是  $e_j$  自己的上端点。

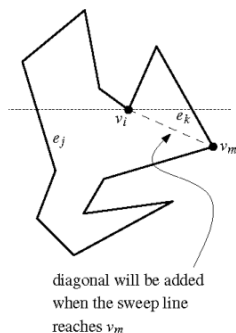


Figure 9: 汇合顶点的消除: 当扫描线扫过  $v_m$  时, 将输出一条对角线

这样, 我们就知道了消除分裂顶点的方法—分别将它们与各自左侧那条相邻边的助手相联。那么, 汇合顶点呢? 从表面上看, 它们似乎更难以消除—因为, 对称地, 它们各自需要借助一个位置更低的顶点, 才能引入一条对角线。然而, 位于扫描线下面的那些部分尚未访问到, 所以在遇到一个汇合顶点时, 并不能参照上面的方法构造出一条对角线。幸运的是, 该问题并不像乍看起来那样困难。试考虑扫描线刚刚触及某一汇合顶点  $v_i$  的时刻。沿着扫描线的方向, 令  $e_j$  ( $e_k$ ) 为居于  $v_i$  左 (右) 侧、与之相邻的边。请注意以下事实: 在到达  $v_i$  的时候, 它也就成为了  $e_j$  的新助手。这样, 就可以从介于  $e_j$  和  $e_k$  之间、位于当前扫描线下方的所有顶点中, 选出其中的最高者, 然后将  $v_i$  与之相联。这个过程, 与处理分裂顶点的情况正好相反—在那里, 我们是在从介于  $e_j$  和  $e_k$  之间、位于当前扫描线上方的所有顶点中, 选出其中的最低者, 然后将  $v_i$  与之相联。这也不值得奇怪—实际上, 只要将上和下颠倒过来, 汇合顶点也就相当于分裂顶点。当然, 在扫描线触及  $v_i$  那一时刻, 我们还不知道哪个才是位于扫描线下方的最高顶点。然而我们马上就会看到, 这并不难判断出来。如图 9 所示, 此后将遇到某个顶点  $v_m$ , 它将取代  $v_i$  的地位, 成为  $e_j$  的新助手—这时,  $v_m$  就是我们所寻找的顶点。因此, 在每次更换某条边的助手时, 都要通过检查以确认 (被替换的) 先前的助手是否为一个汇合顶点。如果是, 就在新、老助手之间引入一条对角线。若新助手是一个分裂顶点, 则这条对角线本来就需要被加入进来,

以消除这一分裂顶点。若同时老助手是一个汇合顶点，则这条对角线将把一个分裂顶点和一个汇合顶点同时消除掉。还有一种可能：在扫描线越过  $v_i$  之后， $e_j$  的助手不再会被更换—在这种情况下，可以将  $v_i$  与  $e_j$  的下端点联接起来。

按照上述方法，还需要找出居于每个顶点左侧、与之紧邻的那条边。为此，可使用一棵动态二分查找树  $T$ ，将  $P$  中与当前扫描线相交的所有边存放在该树的叶子中。 $T$  中所有叶子从左到右的次序，对应于这些边从左到右的次序。既然我们只关心在左侧与各分裂顶点或汇合顶点紧邻的边，故在  $T$  中，只需存放  $P$  的内部（在局部）位于其右侧的那些边。对  $T$  中的每一条边，我们都记录其对应的助手。树  $T$  以及所存储的各边的助手，构成了扫描线算法的状态（Status）。随着扫描线的推进，状态会相应地变化：有些边可能开始与扫描线相交，原来与扫描线相交的一些边可能不再相交，同时某条边原先的助手可能会被新助手替换掉。

采用上述算法对  $P$  进行划分之后，得到的各个子多边形还必须经过后续的处理。为了能够方便地访问到这些子多边形，需要将  $P$  导出的子区域划分（subdivision）存储起来，并且将对角线加入到双向链接边表  $D$  之中。我们假定， $P$  原本就是以双向链接边表（doubly-connected edge list）形式给出的；否则—比如，仅表示为所有顶点的一个逆时针列表—就需要首先为  $P$  构造出一个双向链接边表。随后，为每个分裂顶点和汇合顶点引入的对角线，都必须加入到这个双向链接边表之中。为了访问该双向链接边表，需要将状态结构与双向链接边表中对应的各边通过指针链接起来。借助于指针的操作，可以在常数时间内引入一条对角线。这样，就得到了如下的主算法：

---

**Algorithm 1:** MAKEMONOTONE( $P$ )

---

**输入：**表示为双向链接边表  $D$  的一个简单多边形  $P$

- 1  $T \leftarrow P$  的单调子多边形划分，同样地存储在  $D$  中
  - 2 以  $y$ -坐标为优先级，将  $P$  的所有顶点组成一个优先队列  $Q$  若有多个顶点的  $y$ -坐标相同，则  $x$ -坐标小者优先级更高
  - 3 初始化一棵空的二分查找树  $T$
  - 4 while ( $Q$  非空)
  - 5   do 从  $Q$  中取出优先级最高的顶点  $v$
  - 6   根据该顶点的类型，选用适当的子程序加以处理
- 

接下来，详细介绍不同事件点的处理方法。刚开始阅读这些算法时，你可以暂不考虑任何退化情况；以后可以反过来验证，它们也能够正确处理各种退化情况。（当然，对在 HANDLESPLITVERTEX 第一行和 HANDLEMERGEVERTEX 第二行中出现的“在左侧紧邻”的概念，你必须给出恰当的定义。）在处理任何一个顶点的时候，我们都需要完成两项任务。首先，必须通过检查确定，是否需要引入一条对角线。若是分裂顶点，或者某条边的助手被替换了，而前任助手本身是一个汇合顶点，则需要引入对角线。其次，还要对状态结构  $T$  所存储的信息进行更新。处理各类事件的详细算法将在下面给出。你可以参照如图 10 所示的例子来体会一下，在不同情况下将发生什么变化。

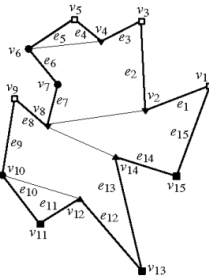


Figure 10: 单调剖分实例

---

**Algorithm 2:** HANDLESTARTVERTEX( $v_i$ )

---

- 1 将  $e_i$  插入  $T$  中，将  $\text{helper}(e_i)$  设为  $v_i$
- 

例如，在如图实例中  $v_5$  处，要将  $e_5$  插入到树  $T$  之中

---

**Algorithm 3:** HANDLEENDVERTEX( $v_i$ )

---

- 1 if (helper( $e_{i-1}$ ) 为一个汇合顶点)
- 2     then 在  $v_i$  和 helper( $e_{i-1}$ ) 之间生成一条对角线, 并将该对角线插入到 D 中
- 3 在 T 中删除  $e_{i-1}$

在上述运行实例中, 当到达终止顶点  $v_{15}$  时, 虽然  $e_{14}$  的助手为  $v_{14}$ , 但因为  $v_{14}$  不是一个汇合顶点, 所以并不需要在此引入一条对角线。

---

**Algorithm 4:** HANDLESPLITVERTEX( $v_i$ )

---

- 1 对 T 进行搜索, 查找在左侧与  $v_i$  紧邻的那条边  $e_j$
- 2 在  $v_i$  和 helper( $e_j$ ) 之间生成一条对角线, 并将该对角线插入到 D 中
- 3 helper( $e_j$ )  $\leftarrow v_i$
- 4 将  $e_i$  插入到 T 中, 将 helper( $e_i$ ) 设置为  $v_i$

对图例中的顶点  $v_{14}$  而言, 在其左侧与之紧邻的边为  $e_9$ 。该边的助手为  $v_8$ , 故要在  $v_{14}$  与  $v_8$  之间引入一条对角线。

---

**Algorithm 5:** HANDLEMERGEVERTEX( $v_i$ )

---

- 1 if (helper( $e_{i-1}$ ) 为一个汇合顶点)
- 2     then 在  $v_i$  和 helper( $e_{i-1}$ ) 之间生成一条对角线, 并将该对角线插入到 D 中
- 3 在 T 中删除  $e_{i-1}$
- 4 对 T 进行搜索, 查找在左侧与  $v_i$  紧邻的那条边  $e_j$
- 5 if (helper( $e_j$ ) 为一个汇合顶点)
- 6     then 在  $v_i$  和 helper( $e_j$ ) 之间生成一条对角线, 并将该对角线插入到 D 中
- 7 helper( $e_j$ )  $\leftarrow v_i$

在图例中的顶点  $v_8$  处, 边  $v_7$  的助手为  $v_2$ , 它是一个汇合顶点, 故需要在  $v_8$  与  $v_2$  之间引入一条对角线

最后需要介绍的, 只剩下处理普通顶点的子程序。对一个普通顶点的处理方法, 取决于在其邻域 P 到底是处于它的左侧还是右侧。

---

**Algorithm 6:** HANDLEREGULARVERTEX( $v_i$ )

---

- 1 if (P 的内部处于  $v_i$  的右侧)
- 2     then if (helper( $e_{i-1}$ ) 是一个汇合顶点)
- 3         then 生成一条对角线, 联接  $v_i$  和 helper( $e_{i-1}$ ), 并将该对角线插入到 D 中
- 4         在 T 中删除  $e_{i-1}$
- 5         将  $e_i$  插入到 T 中, 将 helper( $e_i$ ) 设置为  $v_i$
- 6     else 对 T 进行搜索, 查找在左侧与  $v_i$  紧邻的那条边  $e_j$
- 7         if (helper( $e_j$ ) 是一个汇合顶点)
- 8             then 在  $v_i$  和 helper( $e_j$ ) 之间生成一条对角线, 并将该对角线插入到 D 中
- 9             helper( $e_j$ )  $\leftarrow v_i$

比如在图例中的普通顶点  $v_6$  处, 需要在  $v_6$  与  $v_4$  之间引入一条对角线。

现在只需证明: 算法 MAKEMONOTONE 的确能够正确地将 P 划分为多个单调块。

**定理 5** 通过引入一系列互不相交的对角线, 算法 MAKEMONOTONE 能够将 P 划分为多个单调子多边形。

试考察在到达  $v_i$  的高度时, 由 HANDLESPLITVERTEX 所引入的对角线  $\overline{v_m v_i}$ 。令在  $v_i$  左侧与其紧邻的那条边为  $e_j$ , 而在  $v_i$  右侧与其紧邻的那条边为  $e_k$ 。于是, 在触及  $v_i$  时, helper( $e_j$ ) =  $v_m$ 。

首先说明:  $\overline{v_m v_i}$  不会与 P 的任何一条边相交。为此, 可考察图 11 中由  $e_j$ 、 $e_k$  以及分别通过  $v_m$  和  $v_i$  的两条水平线所确定的那个四边形 Q。我们断言: Q 的内部不含 P 的任何顶点。否则,  $v_m$  就不可能成为  $e_j$  的助手。现在假设:  $\overline{v_m v_i}$  与 P 的某条边相交。既然这条边的端点都不可能落在 Q 中, 而且多边形的边互不相交, 故这条边要么跨越联接于  $v_m$  与  $e_j$  之间的水平线段, 要么跨越联接于  $v_i$  与

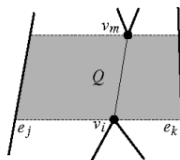


Figure 11: 介于  $v_m$  和  $v_i$  之间水平梯形  $Q$  内部必空

$e_j$  之间的水平线段。然而，这两种情况都不可能出现—因为，无论是对  $v_m$  还是  $v_i$  而言，在其左侧与之紧邻的边都是  $e_j$ 。因此， $\overline{v_m v_i}$  不会与  $P$  的任何边相交。

最后，再来考虑此前所引入的那些对角线。既然  $Q$  的内部不含  $P$  的任何顶点，而且此前所引入的每一条对角线的两个端点都要高于  $v_i$ ，故它们都不可能与—— $\overline{v_m v_i}$  相交。

下面对该算法的运行时间做一分析。构造优先队列  $Q$  需要线性的时间，而树  $T$  的初始化只需常数时间。在扫描过程中，每次处理一个事件点，都只需要对  $Q$  执行一次操作；对于树  $T$ ，最多只分别做一次查找、一次插入和一次删除；对于  $D$ ，最多插入两条对角线。无论是优先队列，还是平衡查找树，都可以在  $O(\log n)$  时间内完成一次查找或一次更新；而将一条对角线插入到  $D$  中，只需  $O(1)$  时间。因此，只需  $O(\log n)$  时间，就可以处理完一次事件，于是整个算法所需的时间就是  $O(n \log n)$ 。显然，该算法只需线性的空间—在  $Q$  中，每个顶点至多被存储一次；在  $T$  中，每条边至多存储一次。这样，结合定理 4，就可以得出如下定理

**定理 6** 使用  $O(n)$  的存储空间，可以在  $O(n \log n)$  时间内将包含  $n$  个顶点的任何简单多边形分解为多个  $y$ -单调的子块。

### 3 单调多边形的三角剖分

在本节中，将说明：可以在线性的时间内，完成对单调多边形的三角剖分。只有将这一结果与前一节的结果联系起来，才能得出结论：对任何简单多边形的三角剖分，都可以在  $O(n \log n)$  时间内完成。

给定一个包含  $n$  个顶点的  $y$ -单调多边形  $P$ 。 $P$  是严格  $y$ -单调的，即它不仅是  $y$ -单调，而且还不含任何水平边。可以从最高顶点开始，沿着  $P$  的（左、右）两条边界链，走向最低的顶点，再次过程中，只要有可能，就引入对角线。下面详细介绍一下三角剖分的贪婪算法。

该算法按照  $y$ -坐标递减的次序，依次处理各个顶点。若有两个顶点的  $y$ -坐标相等，则其中靠左的顶点将被优先处理。该算法需要利用一个栈  $S$  做为辅助的数据结构。一开始，该栈为空；在算法过程中，它存放了在  $P$  中已经被发现、却仍然可以生出更多对角线的那些顶点。在处理每个顶点的时候，我们将尽可能地在这个顶点与栈中的各顶点之间引入对角线。这些对角线会从  $P$  中分离出若干三角形。已经做过一些处理，但尚未从原多边形中分离出来的那些顶点（亦即仍滞留在栈中的顶点）都散落在  $P$  中尚未被三角剖分的部分（与已处理过的部分之间）的边界上。这些顶点中位置最低的那个（亦即最后开始接受处理的那个顶点），就位于栈顶的位置；高度次低的那个顶点，则位于次栈顶的位置；依此类推。如图 3 所示，在已经被发现的那些顶点之上， $P$  中还有一些部分尚待剖分，这些部分具有特定的形状—犹如一个倒置的漏斗。这个漏斗（左或右）一侧的边界，由  $P$  的某条边独立地界定；而沿着它在另一侧的边界，所有的顶点都是凹顶点（reflex vertex）—亦即，这些顶点各自对应的内角都不小于  $180^\circ$ （其中最高的那个顶点除外，它是凸的）。在我们处理完接下来的一个顶点之后，这个性质依然保持—也就是说，这是该算法所具有的一个不变性。

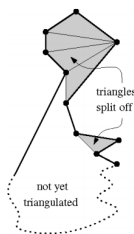


Figure 12: 已经三角剖分的部分与尚未三角剖分的部分

现在来看看，在处理下一个顶点的时候，可以引入哪些对角线。分两种情况处理：接受处理的下一顶点  $v_j$ ，与栈中的那些凹顶点处于（漏斗的）同一侧；或者，处于对面的另一侧。若是后一种情况



(图 12), 则  $v_j$  必然就是独自界定该漏斗一侧边界的那条边  $e$  的下端点。鉴于其漏斗的形状, 我们可以从  $v_j$  出发, 与当前栈中除最后一个 (即居于栈底的那个) 顶点之外的每个顶点, 分别联接一条对角线。而实际上, 此时栈中的最后一个顶点, 就是  $e$  的上端点—也就是说, 该顶点实际上已经与  $v_j$  联接了。所有这些顶点都将从栈中弹出。此后, 在  $v_j$  之上, 原多边形中尚待三角剖分的部分, 将由此前生成的、联接  $v_j$  与栈顶顶点的那条对角线界定; 而该部分的范围, 将在该顶点处向下延伸—因此, 这部分仍然是 (倒立的) 漏斗状, 故上述不变性依然保持。该顶点以及  $v_j$  仍然属于多边形中尚待三角剖分的那部分, 因此, 需要将它们 (再次) 压入栈中。

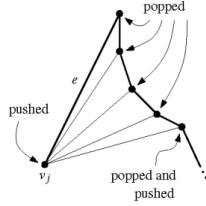


Figure 13: 接受处理的下一顶点  $v_j$  处于对面的另一侧

在另一种情况中,  $v_j$  与栈中的那些凹顶点同属于漏斗的一侧。此时, 从  $v_j$  出发, 就不见得能够与栈中的每一个顶点都联接一条 (合法的) 对角线。尽管如此,  $v_j$  还是能够与其中的某些顶点联接—这些顶点必然是依次相邻的, 而且在栈中都位于顶部。因此可按如下方法处理: 首先, 从栈中弹出一个顶点 (该顶点已经通过  $P$  的一条边, 与  $v_j$  联接); 然后, 依次从栈中弹出各顶点, 并将其与  $v_j$  联接。不断重复这一过程, 直到不能如此联接的某个顶点。为确定能否在  $v_j$  与栈中的某个顶点  $v_k$  之间联接一条对角线, 只需检查  $v_j$ 、 $v_k$  以及此前刚刚被弹出的那个顶点。一旦遇到一个不能与  $v_j$  相联的顶点, 就将被弹出的前一顶点重新压入栈中。若此前确实联接出过至少一条对角线, 则该顶点就是最后那条对角线的端点; 若根本就没有生成过任何对角线, 则沿着  $P$  的边界该顶点必然与  $v_j$  相邻 (参见图 13)。完成上述操作之后, 将  $v_j$  再次压入栈中。此时, 无论是哪种情况, 不变性又重新恢复了—漏斗的一侧边界由多边形的一条边独立界定, 而另一侧边界则由一串 (依次相邻的) 凹顶点确定。由此可以得出如下算法:

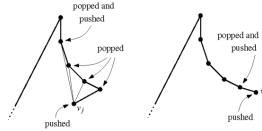


Figure 14: 接受处理的下一顶点  $v_j$  处于对面的另一侧

这个算法需要运行多长的时间呢? 第 1 步需要线性的时间, 第 2 步需要常数时间。for-循环共有  $n-3$  轮, 每一轮最多可能需要线性的时间。然而, 在每一轮 for-循环中, 需要压入  $S$  的顶点不会超过两个。因此, 加上第 2 步中的两次压栈操作, 压栈操作的总数不会超过  $2n-4$ 。自然地, 退栈操作的次数不可能超过压栈, 故 for-循环总共的运行时间为  $O(n)$ 。算法最后一步所需的时间, 也不会超过线性的量级。总而言之, 该算法的运行时间为  $O(n)$ 。

**定理 7** 由  $n$  个顶点组成的任一严格  $y$ -单调多边形, 都可以在线性时间内被三角剖分。

我们希望把单调多边形的三角剖分算法, 做为对任意简单多边形进行三角剖分的一个子程序。按照这一构思, 首先要将多边形划分为若干单调子块, 然后分别对各单调子块进行三角剖分。看起来, 似乎所有必需的条件都已具备。然而, 还有一个问题—本节一直假定: 输入都是严格  $y$ -单调的多边形; 然而按照前一节所介绍的算法, 生成的单调子块中有可能含有水平边。你应该记得, 在前一节中, 对于  $y$ -坐标相等的顶点, 我们是按照自左向右的次序进行处理的。其效果等同于沿顺时针方向, 将整个平面做一足够小角度的旋转, 从而使得任何两个顶点都不会处于同一水平高度上。于是, 在这个经过小角度旋转之后的平面上, 由上节的算法划分出来的单调子多边形, 必然都是严格单调的。这样, 只要我们依然按照自左向右的次序处理那些  $y$ -坐标相同的顶点 (这等同于在旋转后的平面上进行计算), 本节的三角剖分算法就能正常地工作。因此, 我们可以将这两个算法结合起来, 得出一个适用于任何简单多边形的三角剖分算法。

这个三角剖分算法的运行时间有多长? 由定理 5, 可在  $O(n \log n)$  时间内将一个多边形分解为多个单调子块。第二个阶段可采用本节的算法, 在线性时间内对各单调子块分别进行三角剖分。由于所有子块包含的顶点总数为  $O(n)$ , 故第二个阶段总共需要  $O(n)$  时间。由此可以归纳出如下结论:

---

**Algorithm 7: TRIANGULATEMONOTONEPOLYGON(P)**

---

**输入:** 表示为双向链接边表  $D$  的一个严格  $y$ -单调的多边形  $P$

- 1  $T_{\text{put}}$   $P$  的三角剖分, 同样存储在  $D$  中
- 2 将  $P$  左、右侧边界上的所有顶点合并起来, 按照  $y$ -坐标排成一个递减的序列若有多个顶点的  $y$ -坐标相等, 则  $x$ -坐标小者在前 (令排序后的序列为  $u_1, \dots, u_n$ )
- 3 初始化一个空栈  $S$ , 然后将  $u_1$  和  $u_2$  压入其中
- 4 for ( $j \leftarrow 3$  to  $n-1$ )
- 5   do if ( $u_j$  处于与  $S$  栈顶顶点对面的一侧)
- 6     then 弹出  $S$  中的所有顶点
- 7       对于弹出的 (除最后一个外的) 每个顶点
- 8       在  $u_j$  与该顶点之间生成一条对角线
- 9       将  $u_{j-1}$  和  $u_j$  压入  $S$
- 10   else 弹出  $S$  的栈顶
- 11     不断检查当前栈顶处的顶点:
- 12     只要它与  $u_j$  的连线完全落在  $P$  的内部, 就弹出该顶点
- 13     把这些连线当作对角线, 插入到  $D$  中
- 14     将最后弹出的那个顶点, 重新压入  $S$  中
- 15 将  $u_n$  与栈中 (除第一个和最后一个外的) 每个顶点相联构成对角线

---

**定理 8** 使用  $O(n)$  的存储空间, 可以在  $O(n \log n)$  时间内对由  $n$  个顶点组成的任一简单多边形进行三角剖分。

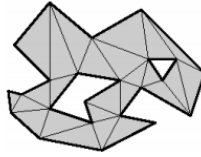


Figure 15: 带洞多边形的三角剖分

我们已经知道了应该如何对简单多边形进行三角剖分。但是, 对那些内部含有空洞的多边形 (图 15) 呢? 它们也能够如此轻易地被三角剖分吗? 答案是肯定的。实际上, 我们所介绍的算法同样适用于内部存在空洞的多边形—在将一个多边形分解为多个单调子块的过程中, 我们本来就没有要求多边形是简单的。该算法甚至还适用于另外一种更具一般性的情况—给定一个平面子区域划分  $S$ , 要求对  $S$  进行三角剖分。对这一问题更准确的描述是: 如果  $S$  的所有边都落在某一包围框 (bounding box)  $B$  的内部, 我们希望构造出由互不相交的对角线—也就是联接于  $S$  和  $B$  的顶点之间、与  $S$  的边不相交的线段—组成的一个极大集合, 这些对角线将  $B$  划分为多个三角形

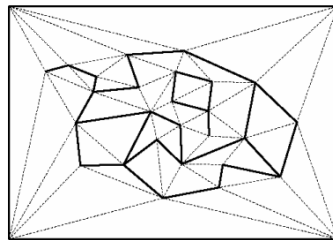


Figure 16: 经三角剖分后的一个子区域划分

图 16 所显示的, 就是一个子区域划分的三角剖分。图中, 用粗线条来表示原子区域划分的边以及包围框的边。可以采用本章所介绍的算法, 来构造这样一个三角剖分—首先, 将该子区域划分分解为多个单调子块; 然后, 分别对各子块做三角剖分。由此可以得出如下定理:

**定理 9** 使用  $O(n)$  存储空间, 可以在  $O(n \log n)$  时间内对包含  $n$  个顶点的任一平面子区域划分进行三角剖分。