计算几何

多边形三角剖分：画廊看守

- 0组：刘妍 郭鹏 林庆童 罗琼 陈沛沛 邢鹏
- 指导老师：齐全

目录

1

看守与三角剖分

画廊看守

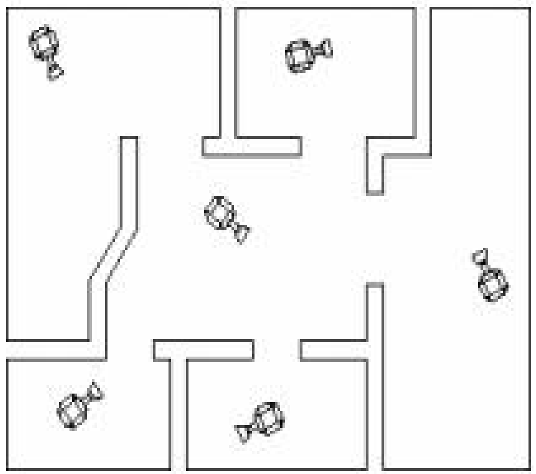


图3-2　监视画廊的一组摄像机

要求1

画廊内的每个角落，都必须被落在至少一台摄像机的视野之内

问题

需要多少台摄像机？分别安装在什么位置？

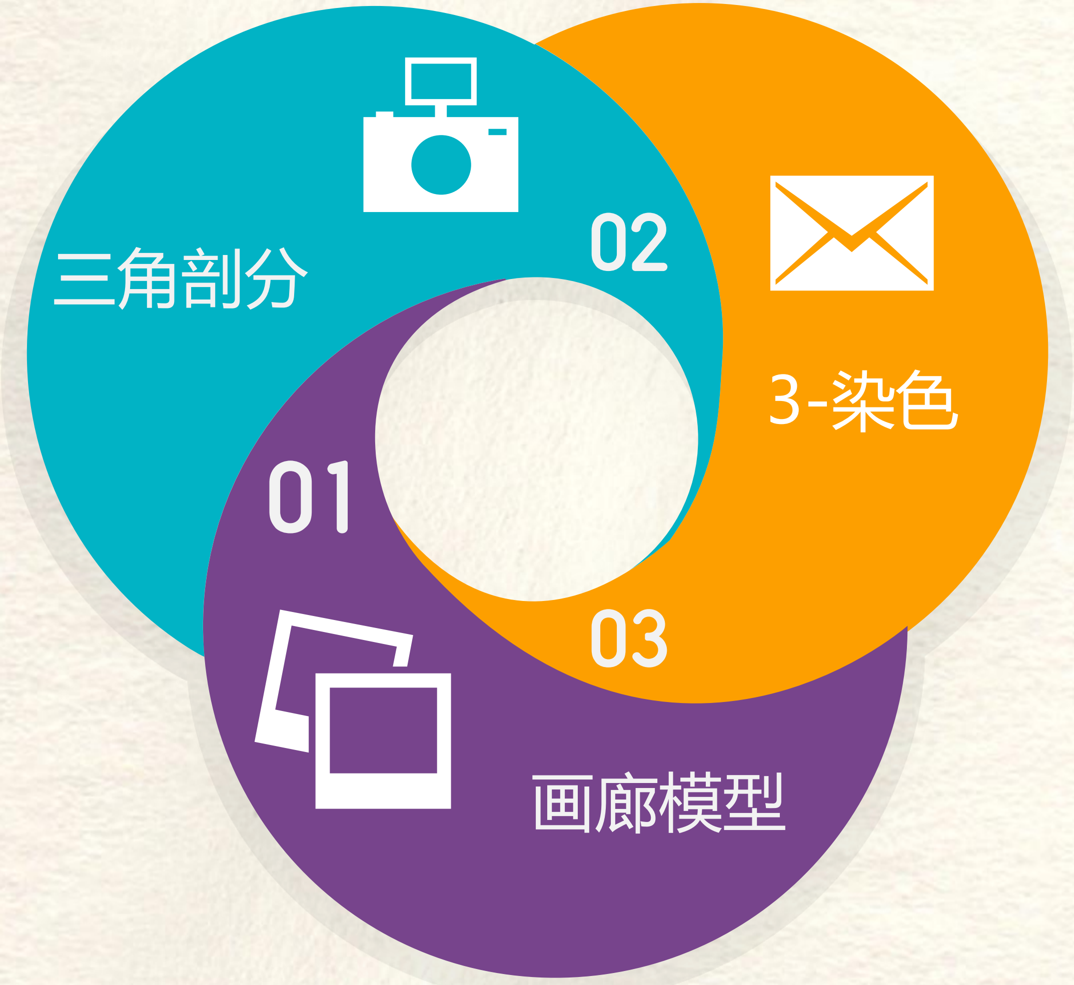
要求2

摄像机数目尽可能少

目标

使每台摄像机都能在画廊中照应到更大的范围

摄像机数量及安装位置



画廊模型

- 对三维空间的画廊做形式化处理，利用**简单多边形**的模型来表示一个画廊。
- 一台摄像机在画廊中的位置，对应于多边形中的一个点。
  如右图，对于多边形内部的任何一点，只要联接于它与某台摄像机之间的**开线段**完全落在多边形的内部，它就能被这台摄像机监视到。
- 根据多边形的顶点数目n，来界定所需摄像机的数量。
  对于由n个顶点组成的所有简单多边形，给出一个**上界**值。

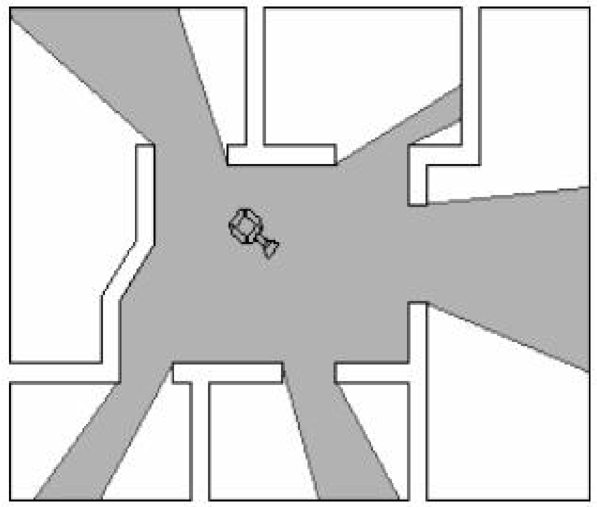


图3-3 单台摄像机所能看守的区域

三角剖分

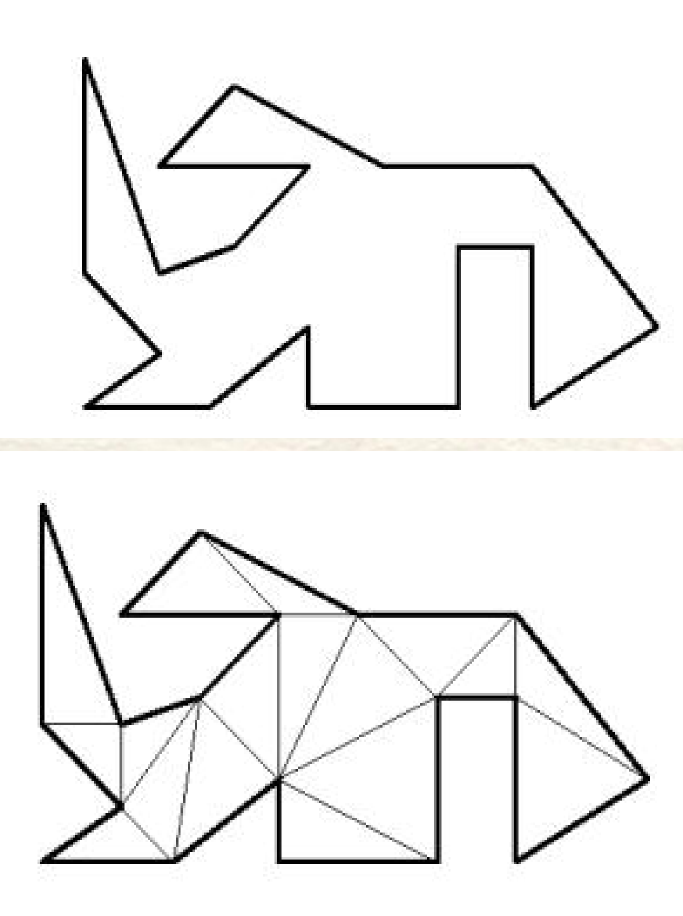


图3-4 简单多边形及可能的一个三角剖分

三角剖分

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

剖分方案

通常，简单多边形有多种不同的三角剖分方案，其三角剖分不是唯一的

多边形看守

只要在三角剖分的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

定理3.1

〖定理 3.1〗
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。

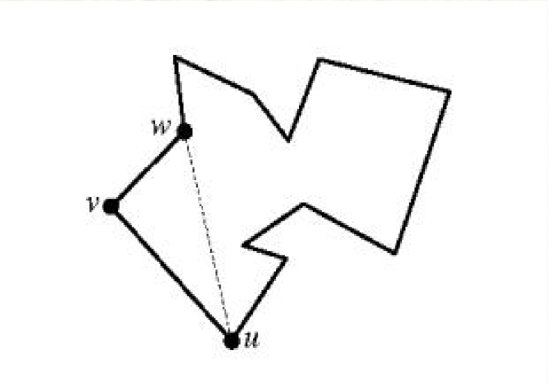


图3-5 线段uw完全落在简单多边形内部

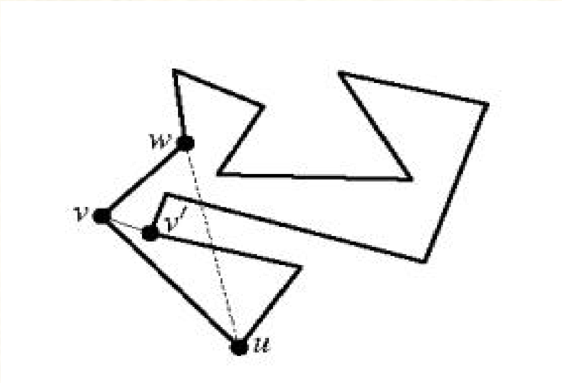


图3-6 线段uw不完全落在简单多边形内部

3-染色

对经过三角剖分后的多边形进行3-染色确定顶点

将摄像机安装在（多边形的）顶点上

由定理3.1：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

为每个三角形配备摄像机-浪费

选出简单多边形的部分顶点组成一个子集，使得三角剖分中的每个三角形，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。

顶点子集与染色

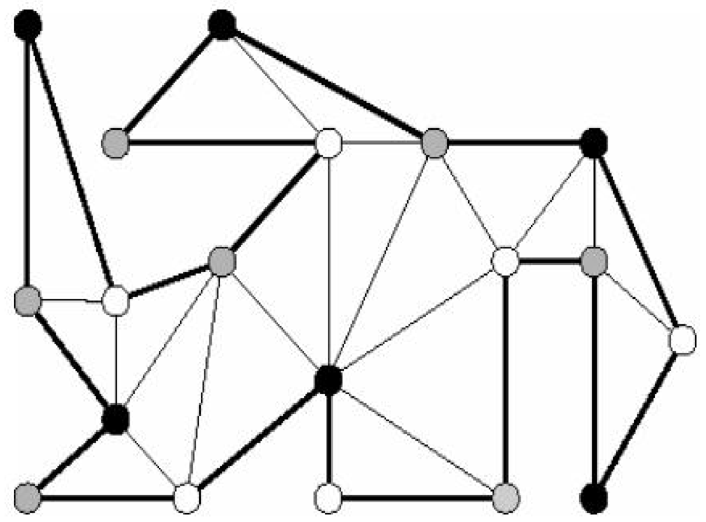


图3-7 根据三角剖分对顶点进行3-染色

染色方案

需满足：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

找子集

使用白、灰和黑三种颜色，给简单多边形的所有顶点染色，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

看守

只要在同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，若选用点数最少的那一类同色顶点配备摄像机，则只需不超过⌊ n/3⌋ 台摄像机，即可看守住P。

定理3.2与3.3

〖定理 3.2（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）⌊ n/3⌋ 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。

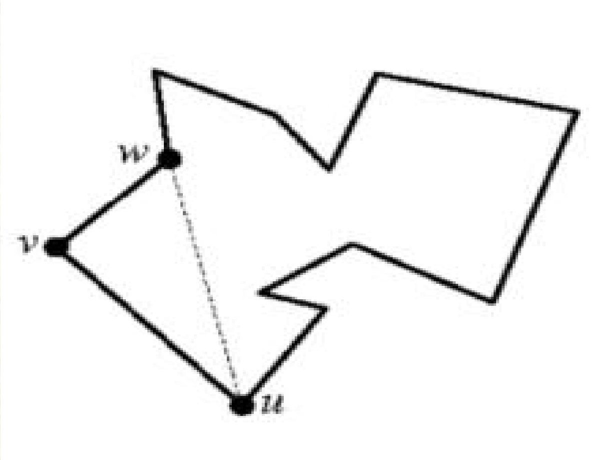
〖定理 3.3〗

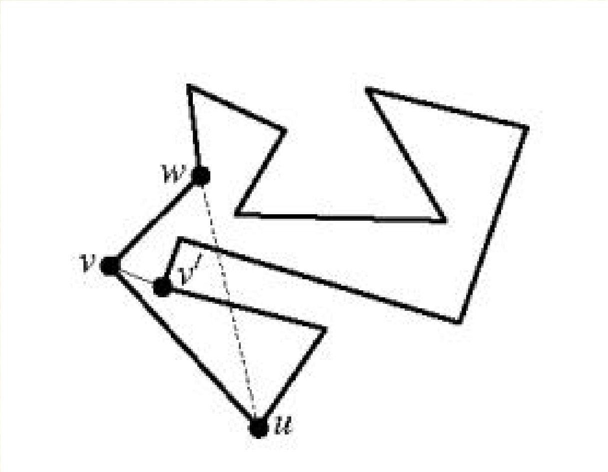
任给一个包含 n 个顶点的简单多边形 P。总可以在 O(nlogn)时间内，在 P 中确定⌊ n/3⌋ 台摄像机的位置，使得 P 中的任何一点都可见于其中的至少一台摄像机。

2

多边形单调块划分

三角剖分算法思路





- 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。

- 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?
- 某些类型的多边形，可以更快，比如凸多边形

单调多边形

问题描述：单调多边形指存在一个方向，垂直于此方向的所以扫描线与多边形只有两个交点。

从顶点T出发，从左边界(T-U-V-W-Z-A_1-B_1)或右边界(T-H_1-G_1-F_1-E_1-D_1-C_1-B_1)走向最低顶点B_1的路径上，高度一直都在下降。如图3-1所示：

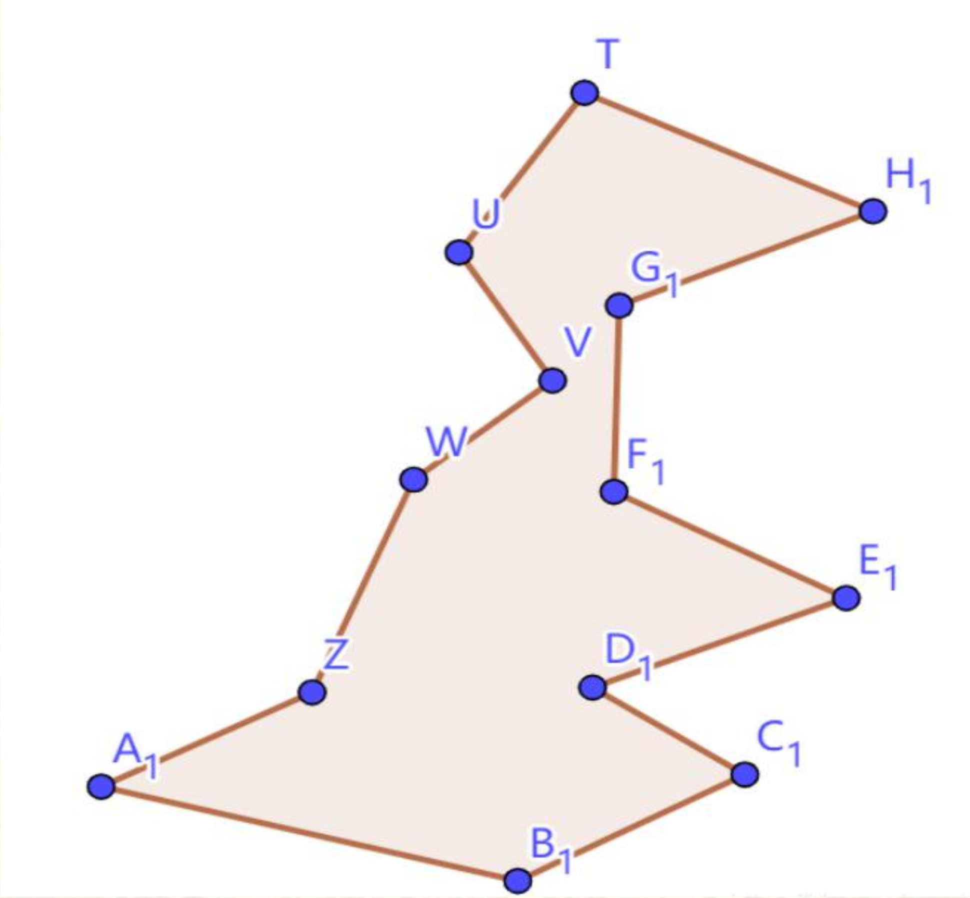
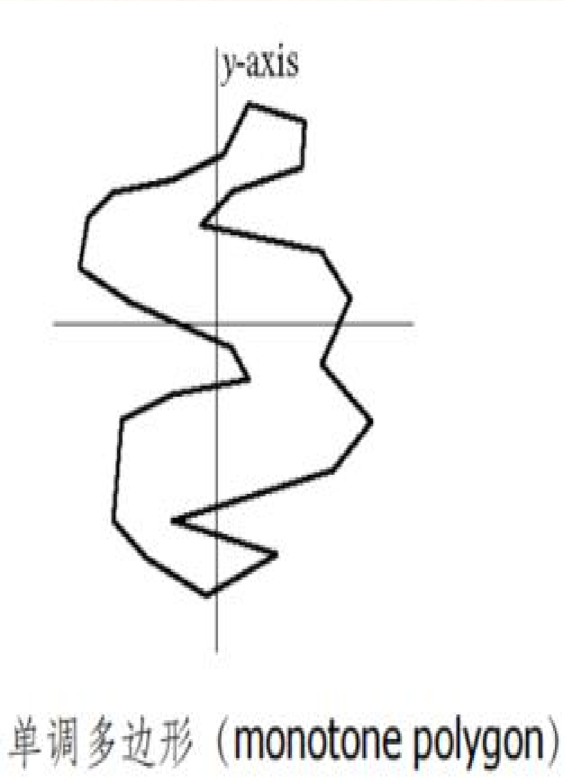
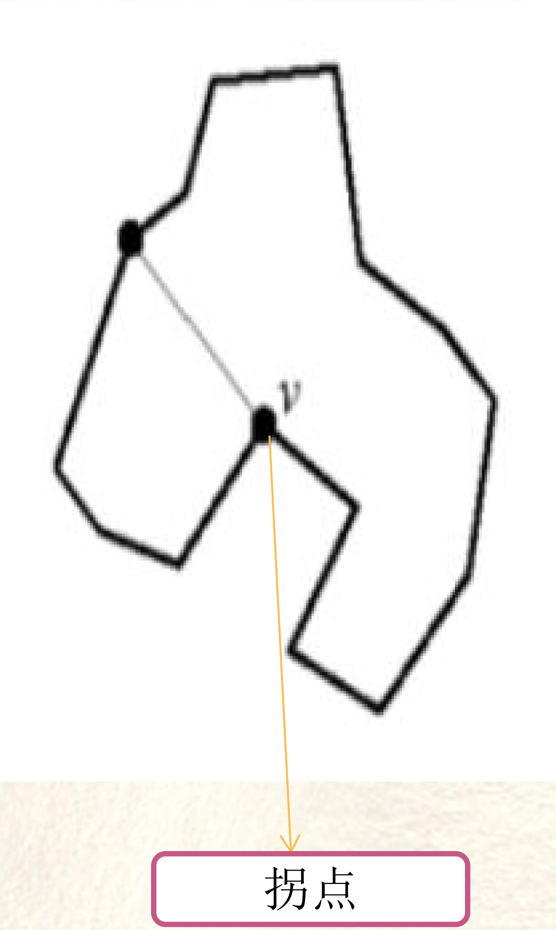


图3-1

三角剖分策略





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。

为了将P划分成多个y-单调块，就必须消除这些拐点。

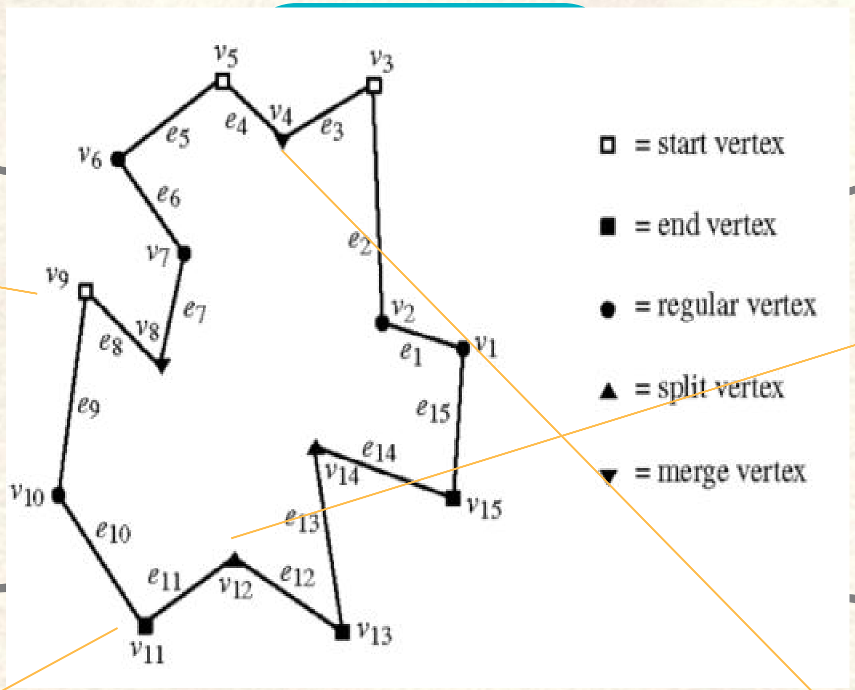
什么是拐点?

起始顶点

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π

分裂顶点

与它相邻的两个顶点
的高度都比它低，而且在v处的
内角大于π



终止顶点

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π

除普通顶点外的四种类顶点都是拐点

汇合顶点

与它相邻的两个顶点的高度都比
它高，而且在v处的内角大于π

引理3.4

〖引理 3.4〗
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。

假设 P 不是 y-单调的。我们来证明，P 中必然含有一个分裂顶点，或者一个汇合顶点。

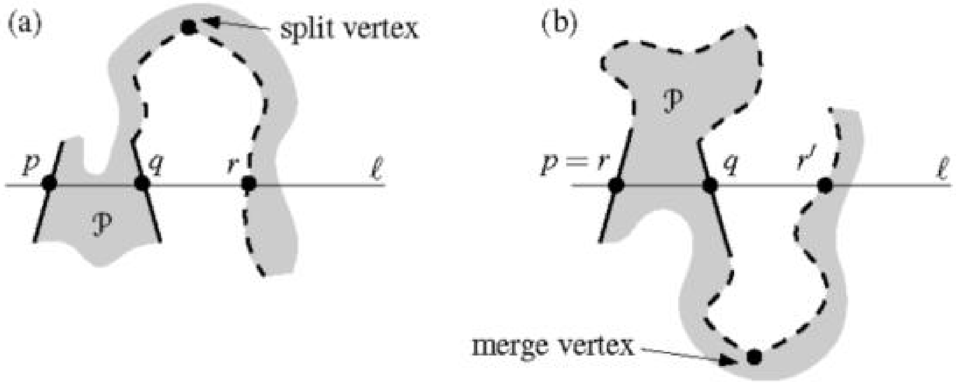


图3-14 〖引理3.4〗的证明中所涉及到的两种情况

消除分裂顶点

采用平面扫描的方法在一个分裂顶点处引入一条对角线

现在考虑介于ej和ek之间、位于vi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vi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vi与ej或ek的上端点联接起来。无论如何，我们都将这个顶点称作“ej的助手”（helper of ej），记作helper(ej)。

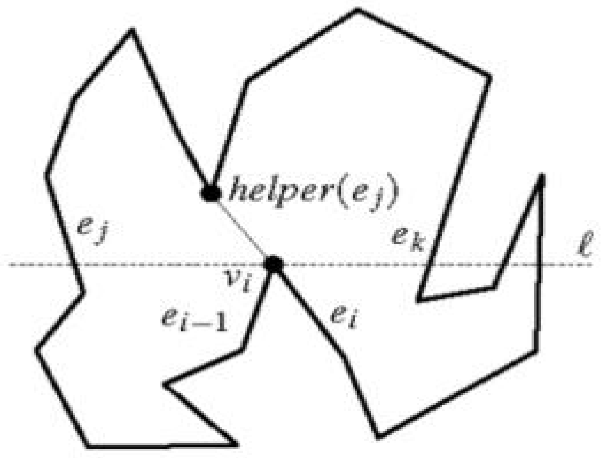


图3-15 分裂顶点的处理

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

消除汇合顶点

从介于ej和ek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vi与之相联

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vi之后，ej的助手不再会被更换——在这种情况下，可以将vi与ej的下端点联接起来

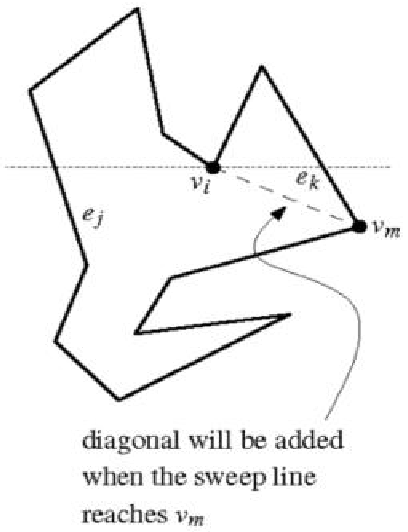


图3-16 汇合顶点的消除：当扫描线扫过vm时，将输出一条对角线

要将上和下颠倒过来，汇合顶点也就相当于分裂顶点

主算法

算法 MAKEMONOTONE(P)
输入：表示为双向链接边表 D 的一个简单多边形 P
输出：P 的单调子多边形划分，同样地存储在 D 中
1. 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
2. 初始化一棵空的二分查找树 T
3. while (Q 非空)
4. do从 Q 中取出优先级最高的顶点 vi
5. 根据该顶点的类型，选用适当的子程序加以处理

不同事件点的处理方法

起始顶点

算法 HANDLESTARTVERTEX(vi)
1. 将 ei 插入 T 中，将 helper(ei)设为 vi

终止顶点

算法 HANDLEENDVERTEX(vi)
1. if (helper(ei-1)为一个汇合顶点)
2. then 在 vi 和 helper(ei-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 ei-1

分裂顶点

算法 HANDLESPLITVERTEX(vi)
1. 对 T 进行搜索，查找在左侧与 vi 紧邻的那条边 ej
2. 在 vi 和 helper(ej)之间生成一条对角线，并将该对角线插入到 D 中
3. helper(ej) ← vi
4. 将 ei 插入到 T 中，将 helper(ei)设置为 v

不同事件点的处理方法

算法 HANDLEMERGEVERTEX(vi)
1. if (helper(ei-1)为一个汇合顶点)
2. then 在 vi 和 helper(ei-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 ei-1
4. 对 T 进行搜索，查找在左侧与 vi 紧邻的那条边 ej
5. if (helper(ej)为一个汇合顶点)
6. then 在 vi 和 helper(ej)之间生成一条对角线，并将该对角线插入到 D 中
7. helper(ej) ← vi

算法 HANDLEREGULARVERTEX(vi)
1. if (P 的内部处于 vi 的右侧)
2. then if (helper(ei-1)是一个汇合顶点)
3. then 生成一条对角线，联接 vi 和 helper(ei-1)，并 将该对角线插入到 D 中
4. 在 T 中删除 ei-1
5. 将 ei 插入到 T 中，将 helper(ei)设置为 vi
6. else 对 T 进行搜索，查找在左侧与 vi 紧邻的那条边 ej
7. if (helper(ej)是一个汇合顶点)
8. then 在 vi 和 helper(ej)之间生成一条对角线，并将该对角线插入到 D 中
9. helper(ej) ← v

引理3.5 3.6

〖引理 3.5〗
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。

〖定理 3.6〗
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。

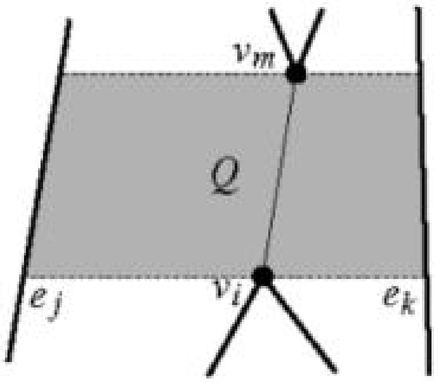


图3-18 介于v_m和v_i之间水平梯形Q内部必空

3

单调多边形的三角剖分

单调多边形

问题描述：单调多边形指存在一个方向，垂直于此方向的所以扫描线与多边形只有两个交点。

从顶点T出发，从左边界(T-U-V-W-Z-A_1-B_1)或右边界(T-H_1-G_1-F_1-E_1-D_1-C_1-B_1)走向最低顶点B_1的路径上，高度一直都在下降。在此过程中，只要有可能，就引入对角线。如图3-1所示：

由此，引出三角剖分的贪婪算法。

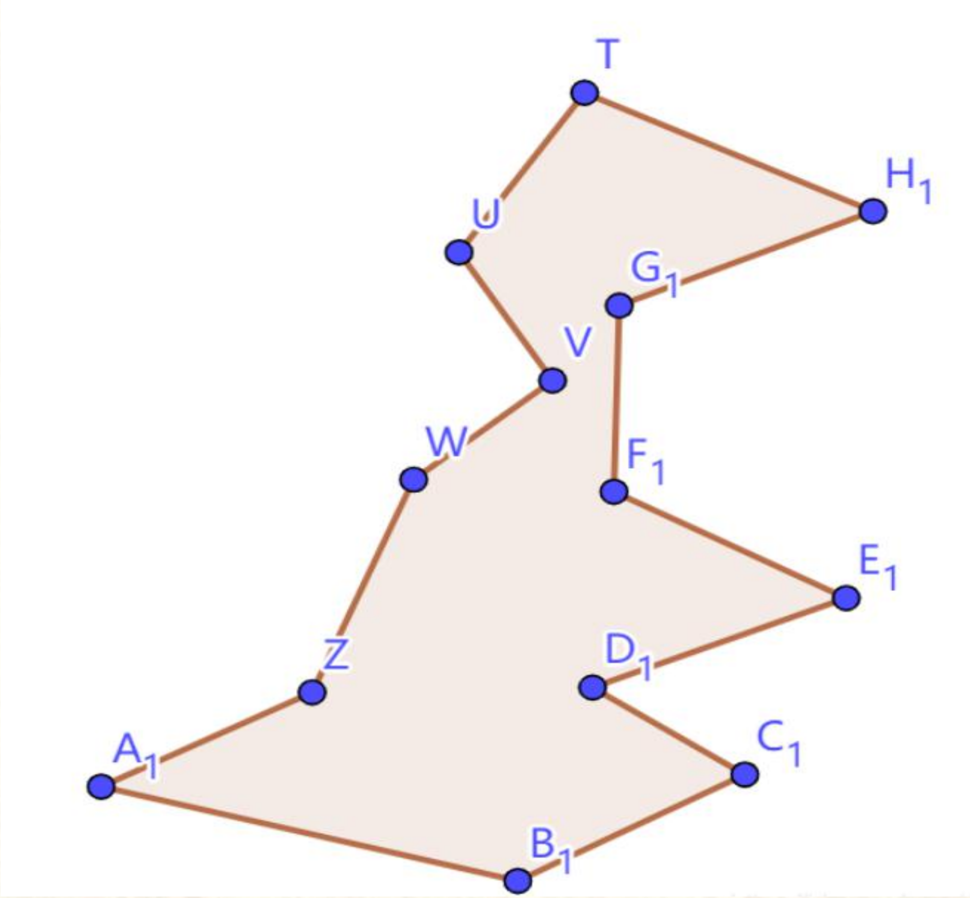


图3-1

单调多边形的三角剖分算法：

该算法原理：首先对多边形顶点按照Y坐标大小降序排列（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），由此可以得到一个有序的序列，从最高点到最低点遍历单调多边形的每一个顶点，然后根据其左右序列属性执行相应的操作。

同时，还需要利用一个空的栈S作为辅助的数据结构。在算法过程中，栈S存放了在P中已经被发现、却仍然可以生出更多对角线的顶点。并且在处理每个顶点的时候，需尽可能地在这个顶点与栈中的各顶点之间引入对角线。通过这些对角线从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。

单调多边形的三角剖分算法：

倒置漏斗：

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——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**）。如图3-2所示。

并且这些顶点各自对应的内角都不小于 **180°** 。当处理完接下来的一个顶点之后，这个性质依然保持。

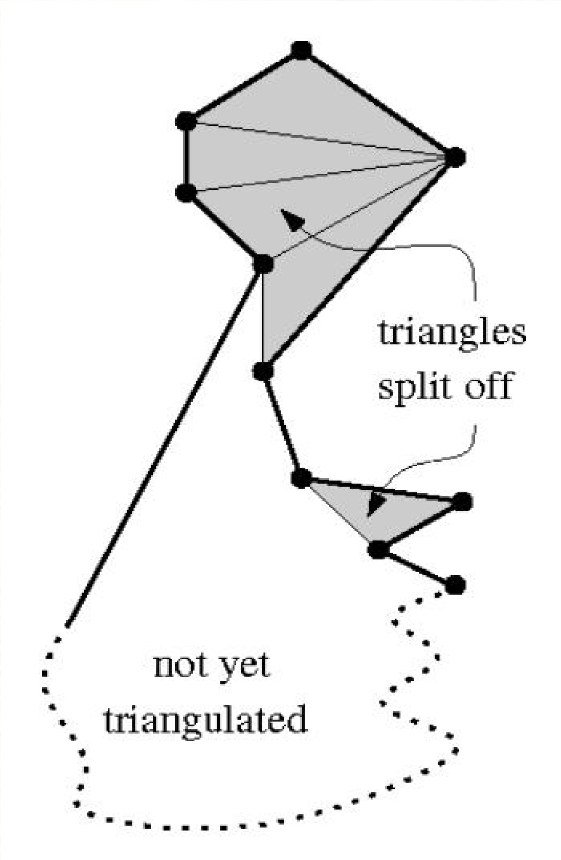


图3-2

单调多边形的三角剖分算法：

根据倒置漏斗的形状，在处理下一个顶点v_j时，可以引入的对角线分两种情况。

第一种情况：接受处理的下一顶点v_j，与栈中的凹顶点处于漏斗的同一侧

第二种情况：接受处理的下一顶点v_j，与栈中的凹顶点处于漏斗的另一侧

单调多边形的三角剖分算法：

第一种情况：v_j与栈中的凹顶点属于漏斗的同一侧，但可能出现以下两种情况。

(1)：从v_j出发，不能够与栈中的每一个顶点都联接一条合法的对角线，剖分出三角形。如图3-3所示，过程是先pop出与v_j已经相连的v_k，再判断此时的栈顶顶点，能否与v_j连接，结果是不能，然后把顶点v_k再push进栈，再把顶点v_j，push进栈。

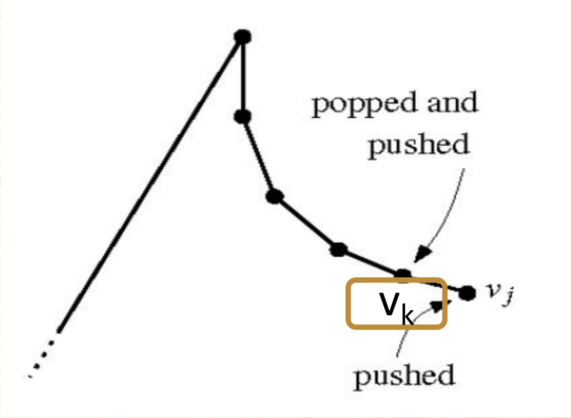


图3-3

(2)：从v_j出发，v_j能够与部分顶点联接(这些点一定连续)，剖分出一些新的三角形出来。如图3-4所示，过程是，循环判断栈顶元素能否连接对角线，可以的pop出来，连线，直到最高的顶点v_h，判断出不能再与v_j连接了，结束循环，然后把最后与v_j顶点连接的v_k再push进栈，再把顶点v_j，push进栈。

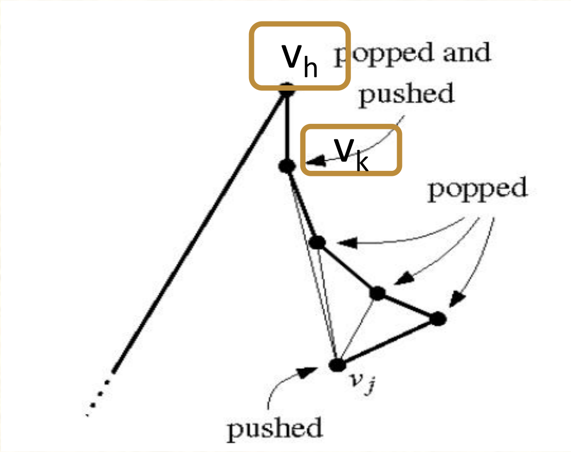


图3-4

单调多边形的三角剖分算法：

需做以下处理：

Step1：检查栈顶处的顶点能否与v_j相连进行三角剖分，如果可以执行**step2**，否则执行**Step3.**

Step2： 从栈中pop出栈顶顶点，与v_j相连（已经与v_j相连的点就不必再连了），跳回Step1.

Step3，将最后一个从栈内pop出的、能与vj相连的节点再push进栈s，再将节点v_j，push进栈。

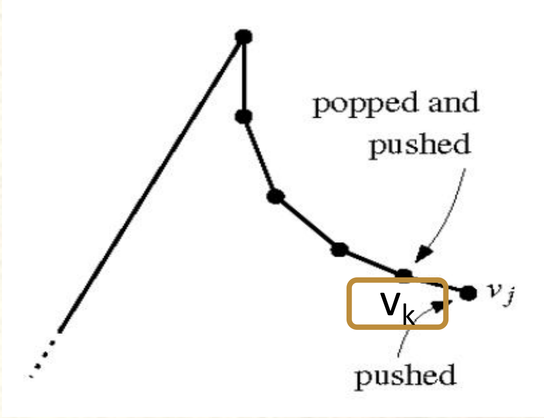


图3-3

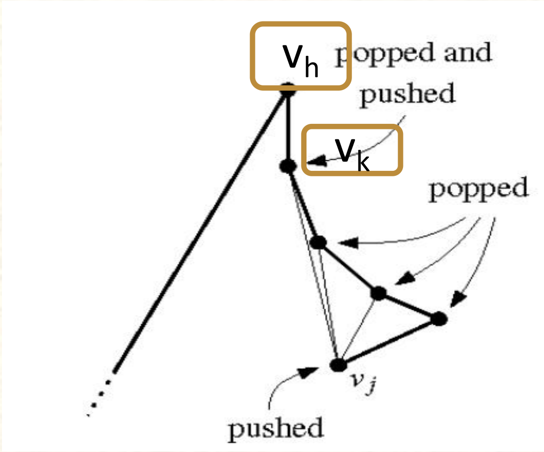


图3-4

单调多边形的三角剖分算法：

第二种情况：v_j与栈中的凹顶点处于漏斗的另一侧，此时v_j必然是独自界定该漏斗一侧边界的那条边e的下端点。如图3-5所示。这时v_j一定可以与栈内的所有点连接对角线，且除v_j和v_k外，其他点都已完成三角剖分。所以最后只许再将v_k，v_j依次入栈。

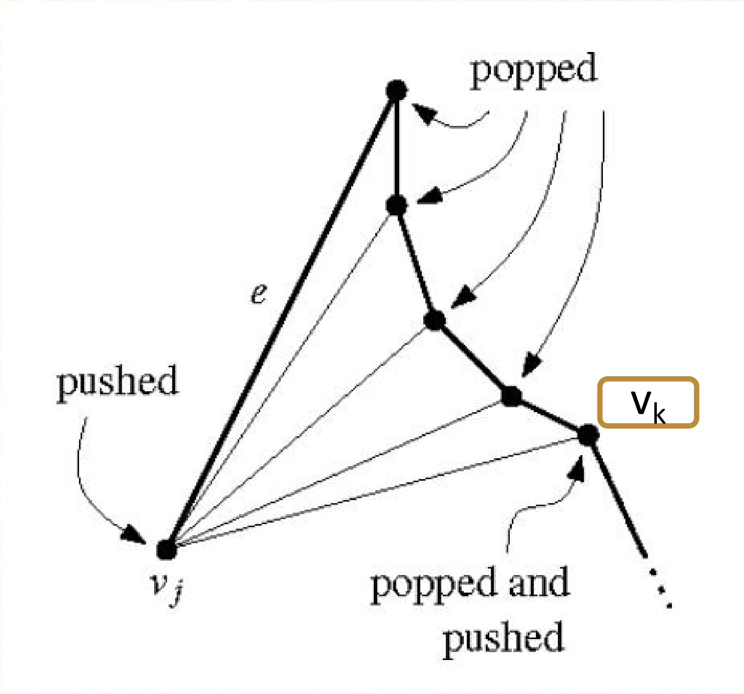


图3-5

算法步骤

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**
输出:P的三角剖分和对角线D

步骤如下:

S1 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，得到排序后的序列为U=[u_1, …, u_n] （需要线性时间）

S2 初始化一个空栈 S，然后将 u1和 u2压入其中 （需要常数时间）

S3 依次遍历U中剩余的元素 （运行时间为o(n)）

S4 完成对单调多边形P的三角剖分

算法步骤

对于第三步(S3)，做如下详细描述：

for (j ← 3 to n-1)

do if (uj处于与 S 栈顶顶点对面的一侧)

then 弹出 S 中的所有顶点
对于弹出的（除最后一个外的）每个顶点在 uj与该顶点之间生成一条对角线
将 uj-1和 uj压入 S

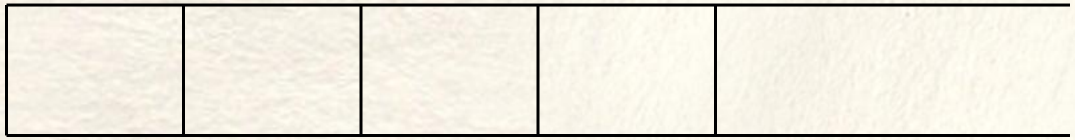
else 弹出 S 的栈顶
不断检查当前栈顶处的顶点：
只要它与 u_j的联线完全落在 P 的内部，就弹出该顶点把这些联线当作对角线，插入到 D 中
将最后弹出的那个顶点，重新压入 S 中

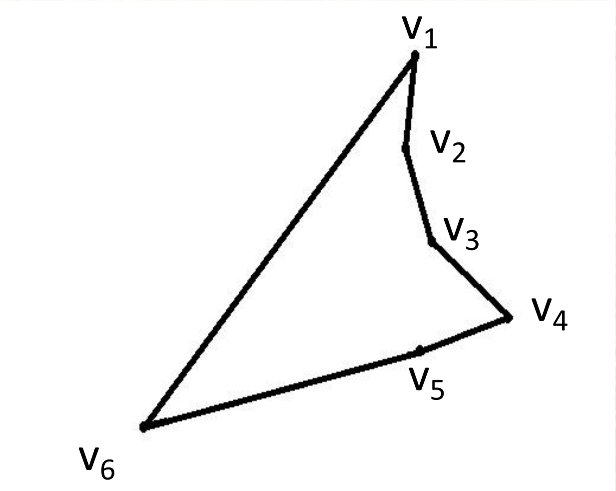
将 uj压入 S 中
将 un与栈中（除第一个和最后一个外的）每个顶点相联构成对角线

栈内流程

S.push(v_1);
S.push(v_2);

栈 S

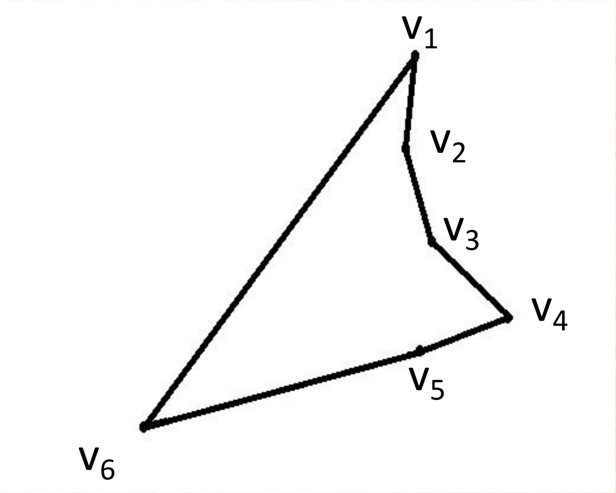




栈内流程

S.push(v_1);
S.push(v_2);

栈 S					
	v_1	**v_2**			

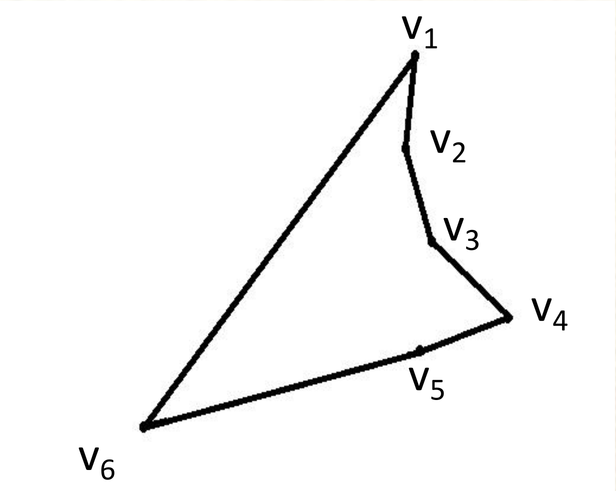


栈内流程

枚举v_3

S.pop()

栈 S	v_1	v_2			

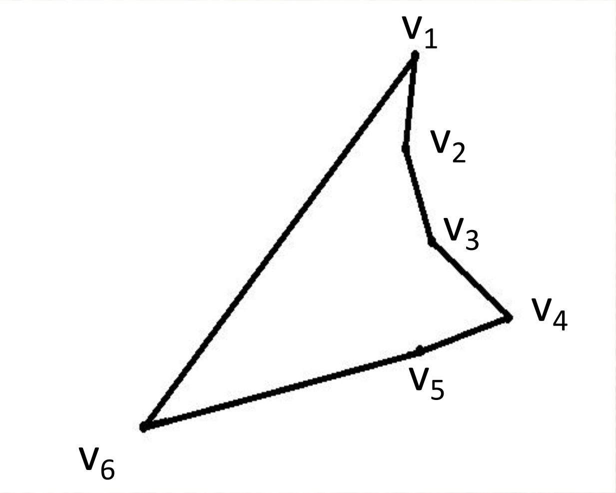


栈内流程

枚举v_3

S.pop()

栈 S	v_1				

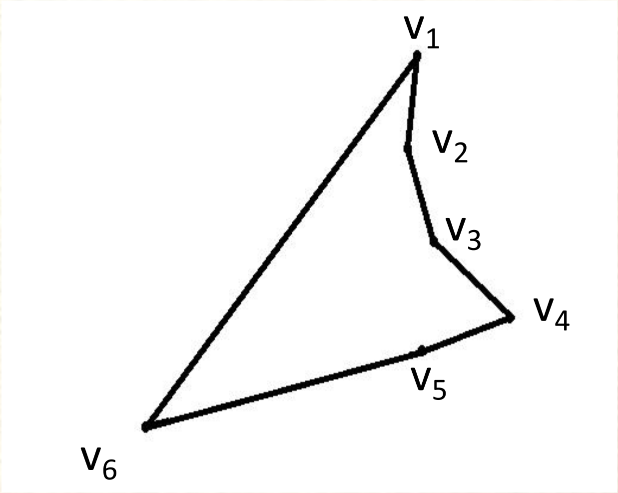


栈内流程

枚举v_3

S.pop()

栈 S	v_1				



判断出S.top()不能与v_3连接对角线

S.push(v_2);
S.push(v_3);

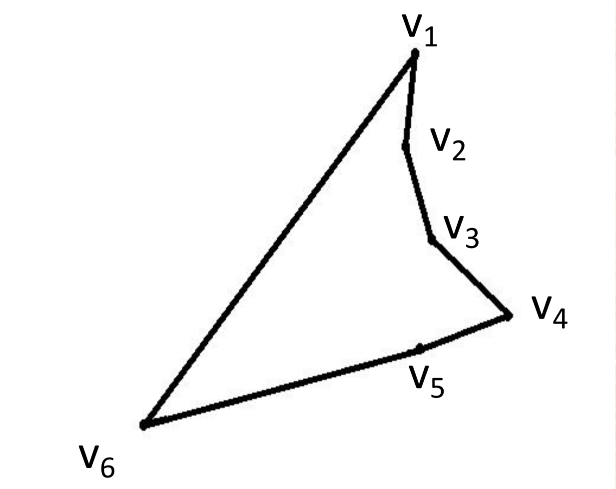
栈内流程

枚举v_3

S.pop()

栈 S

v_1	v_2	v_3		



判断出S.top()不能与v_3连接对角线

S.push(v_2);

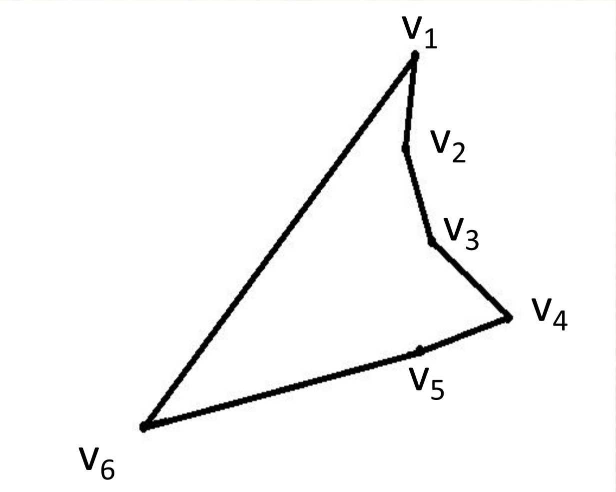
S.push(v_3);

栈内流程

枚举v_4

S.pop()

栈 S	v_1	v_2	v_3		

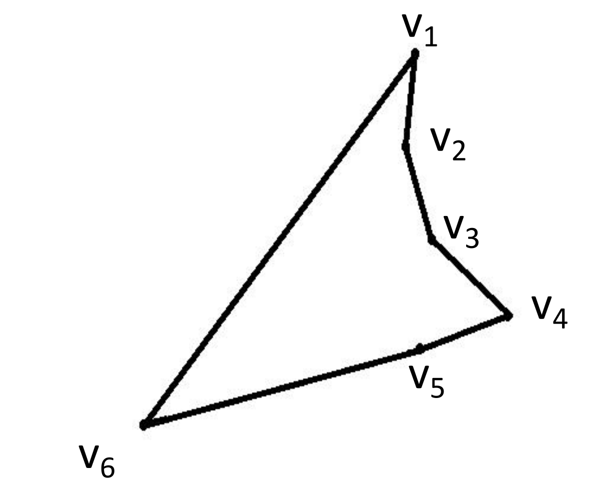


栈内流程

枚举v_4

S.pop()

栈 S	v_1	v_2			



判断出S.top()不能与v_4连接对角线

S.push(v_3);

S.push(v_4);

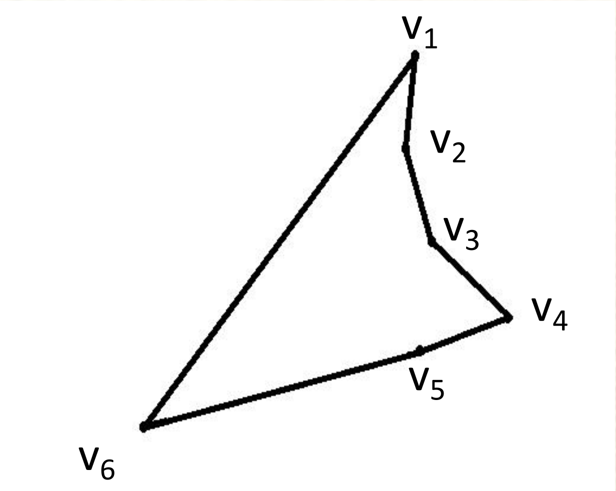
栈内流程

枚举v_4

S.pop()

栈 S

v_1	v_2	v_3	v_4	



判断出S.top()不能与v_4连接对角线

S.push(v_3);

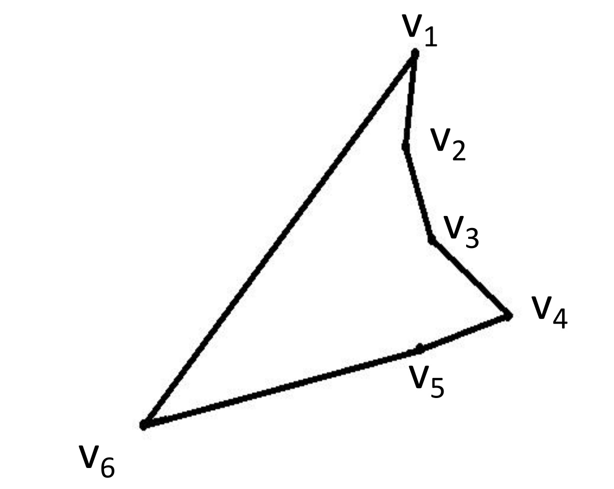
S.push(v_4);

栈内流程

枚举v_5

S.pop()

栈 S	v_1	v_2	v_3	v_4	

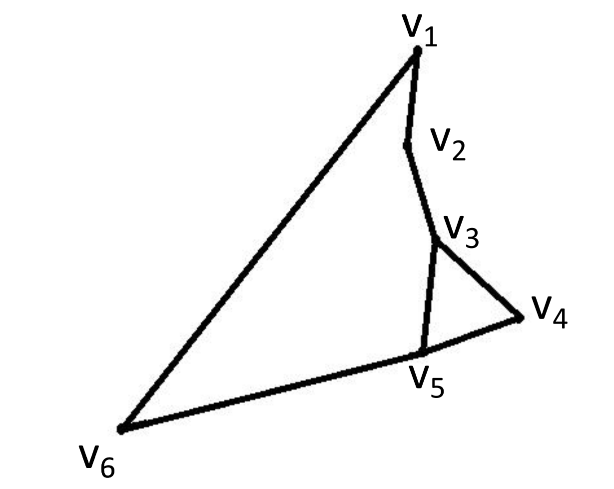


栈内流程

枚举v_5

S.pop()

栈 S	v_1	v_2	v_3		



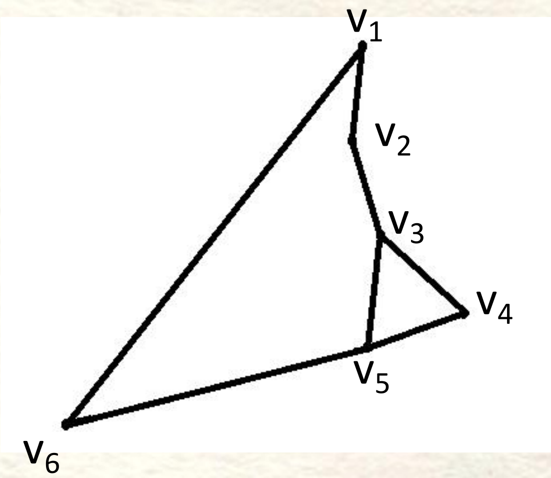
判断出S.top()可以与v_5连接对角线，
连接v_3，v_5的对角线;

栈内流程

枚举v_5

S.pop()

栈 S	v_1	v_2	v_3		

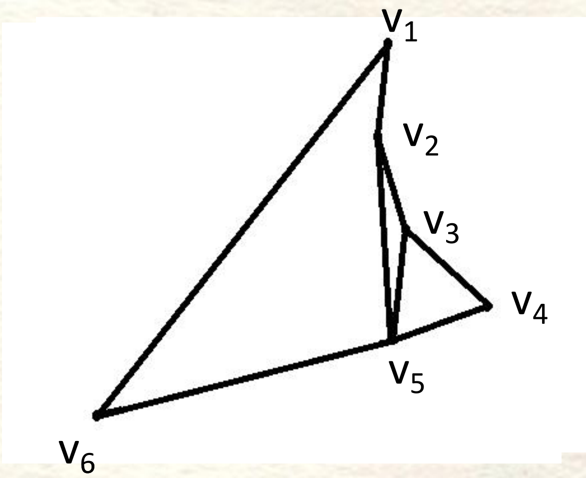


栈内流程

枚举v_5

S.pop()

栈 S	v_1	v_2			



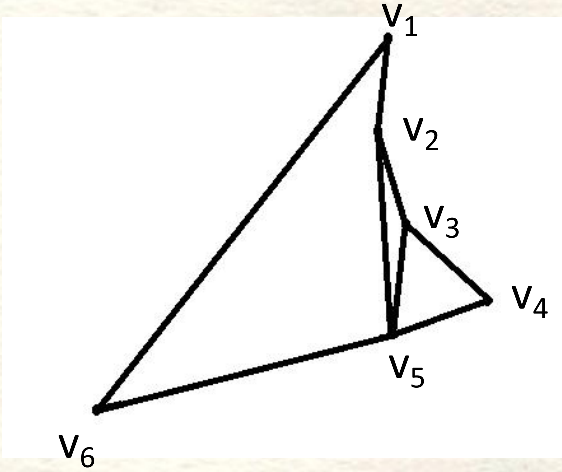
判断出S.top()可以与v_5连接对角线，
连接v_2, v_5的对角线;

栈内流程

枚举v_5

S.pop()

栈 S	v_1	v_2			



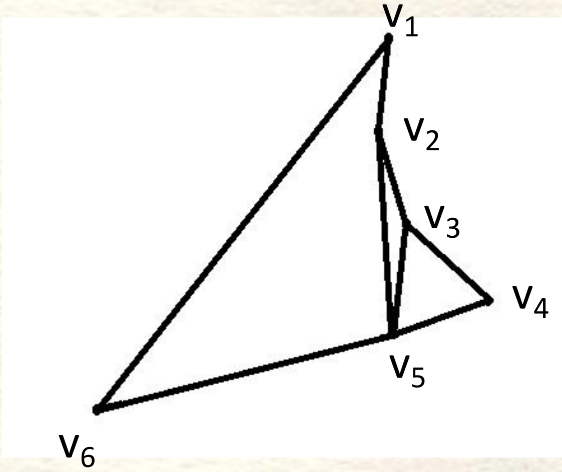
栈内流程

枚举v_5

S.pop()

栈 S

v_1				



判断出S.top()不能与v_5连接对角线

S.push(v_2);

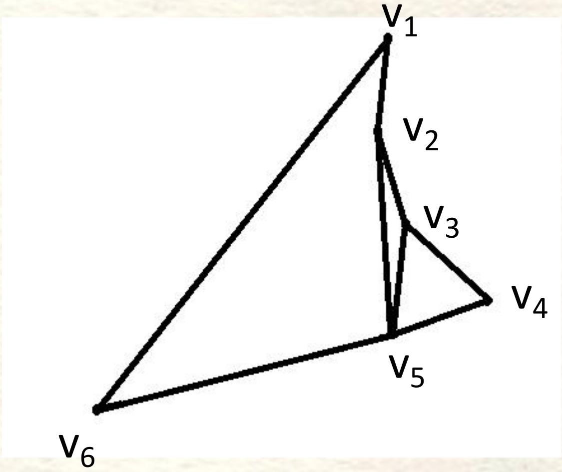
S.push(v_5);

栈内流程

枚举v_6,在直边一侧

栈 S

v_1	v_2	v_5		



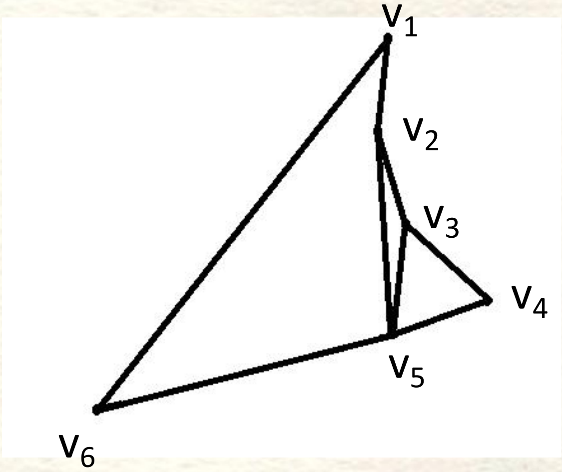
S.top()与v_6连接对角线；

栈内流程

枚举v_6,在直边一侧

S.pop()

栈 S	v_1	v_2	v_5		



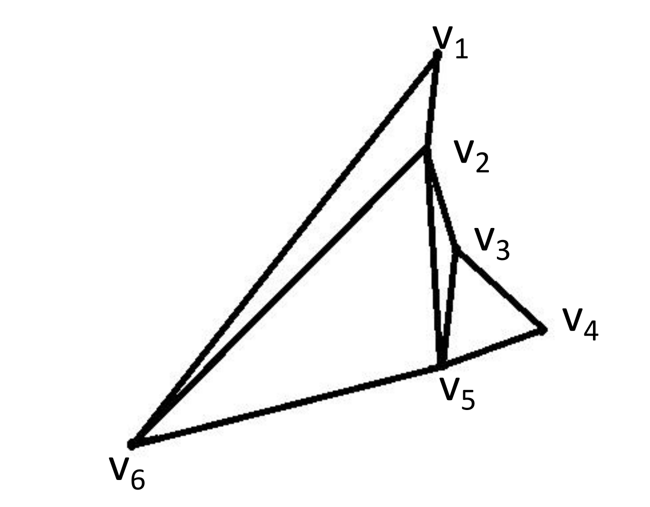
栈内流程

枚举v_6,在直边一侧

S.pop()

栈 S	v_1	v_2			

S.top()与v_6连接对角线;

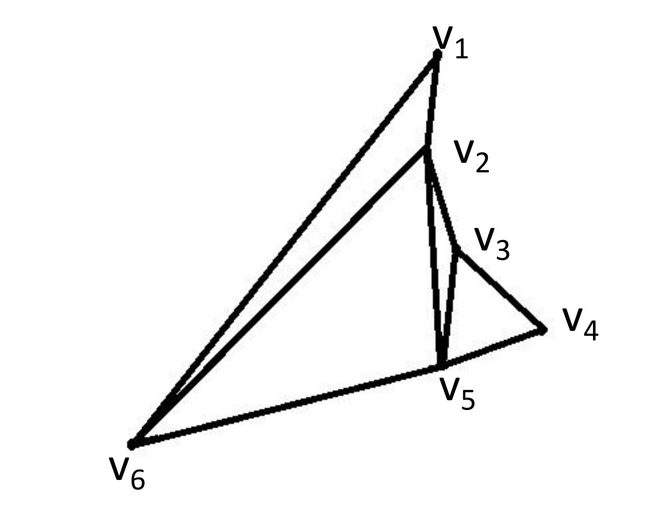


栈内流程

枚举v_6,在直边一侧

S.pop()

栈 S	v_1	v_2			



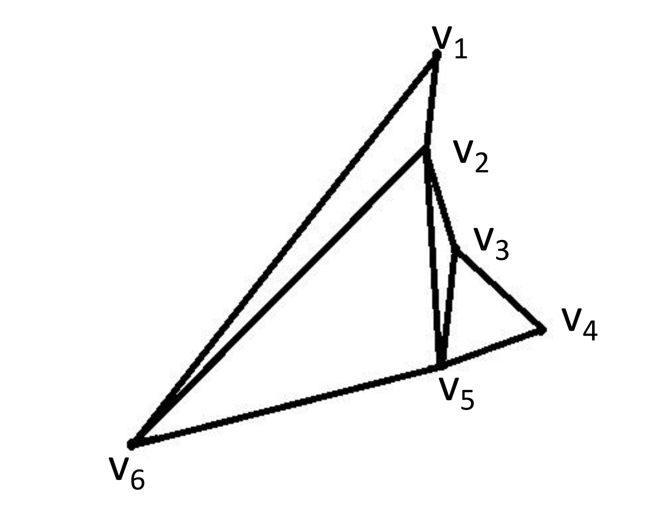
栈内流程

枚举v_6,在直边一侧

S.pop()

栈 S

v_1				



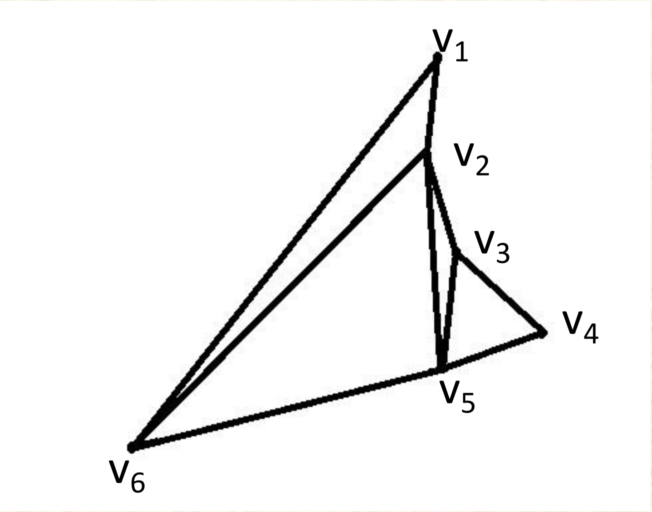
S内只有一个节点，并且已于v_6连接了

栈内流程

枚举v_6,在直边一侧

S.pop()

栈 S	v_1				

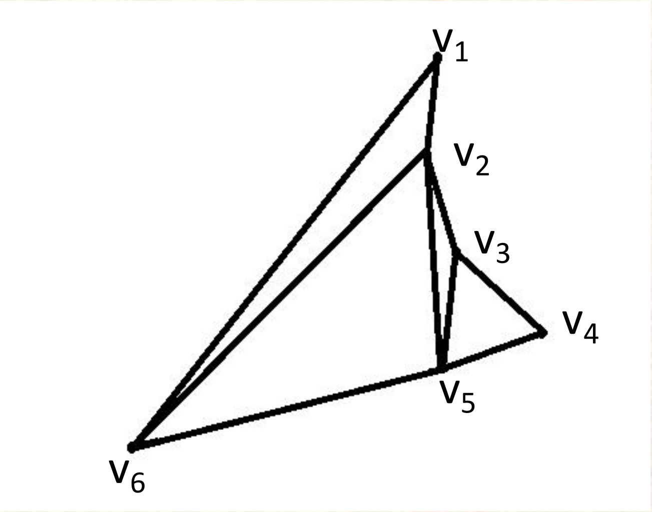


栈内流程

枚举v_6,在直边一侧

S.pop()

栈 S



栈空，除v_5，v_6，当前其他点已确定完成三角剖分。

S.push(v_5);

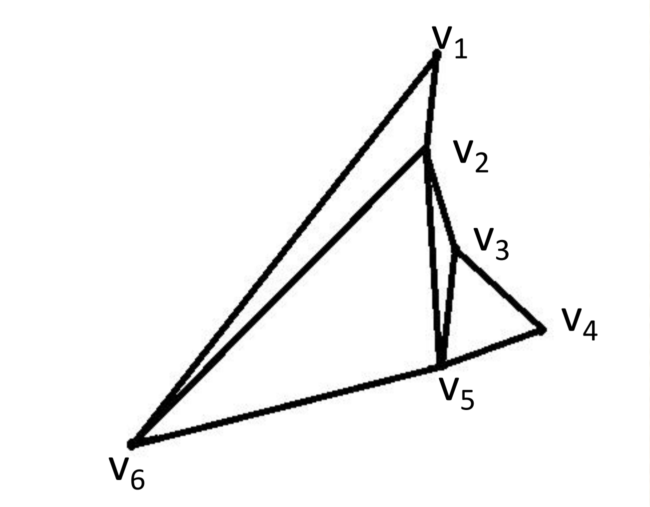
S.push(v_6);

栈内流程

枚举结束

栈 S

v_5	v_6			



定理3.2与3.3

〖定理 3.2（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）⌊n/3⌋ 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。

〖定理 3.3〗

任给一个包含 n 个顶点的简单多边形 P。总可以在 O(nlogn)时间内，在 P 中确定⌊n/3⌋ 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。

4

注释及评论

艺术画廊问题

第一次提出 | 首次证明n/3 | 平面扫描算法 | 线性时间算法

第一次提出

艺术画廊问题是由Klee在 1973 年与Vasek Chvatal的一次交谈中提出的

首次证明

1975 年，Chvatal第一次证明：n/3台摄像机总是足够的，而且有时是必需的。这一结论被称为艺术画廊定理，或者看守者定理

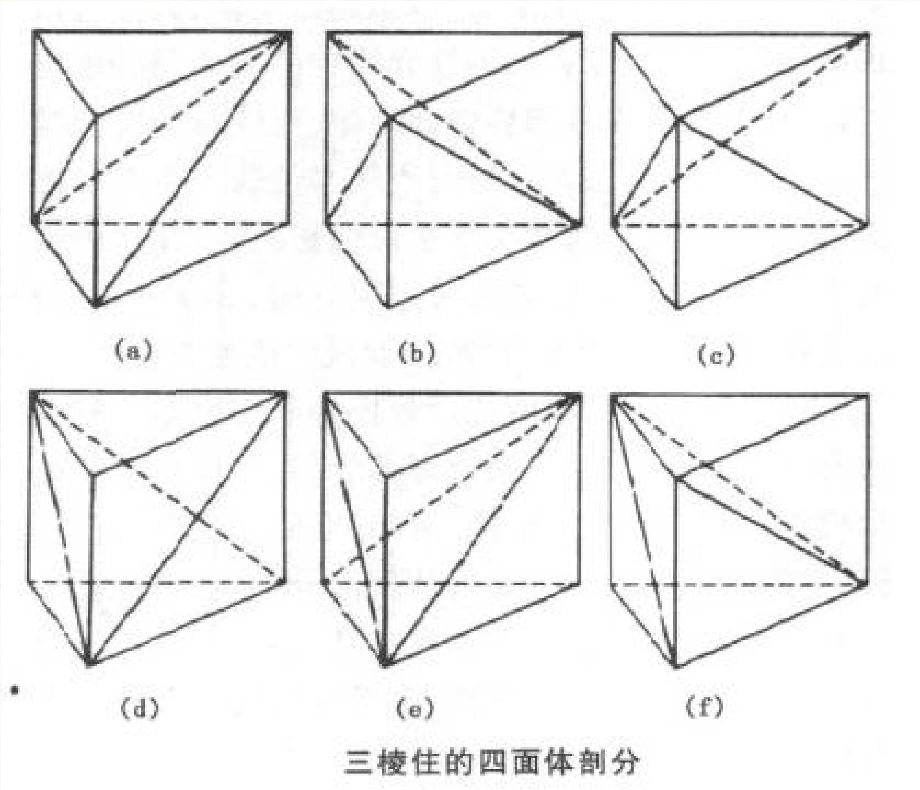
平面扫描算法

将多边形分解为单调子块的平面扫描算法，则是由Lee和Preparata提出的

线性时间算法

本章所介绍的单调多边形三角剖分的线性时间算法，是由Garey等人提出的

三维空间中的对应问题



给定一个多胞体（*polytope*），要求将它分解为互不相交的四面体（*tetrahedron*），其中各四面体的所有顶点，都必须是原多胞体的顶点。多胞体的这种分解被称为四面体剖分（*tetrahedralization*）。

0组全体同学

谢谢您的观看指导!