9.1. 凸包简介

二维的多边形的英文表示是 Polygon，二维的凸包称为凸多边形，三维的多面体英文表示是 Polyhedron，三维的凸包称为凸多面体。二维的多边形和三维的多边形都可以称为多胞体，多胞体的英文表示是 Polytope，多胞体是任意维度上的几何对象的泛化表述。

凸多胞体有很多重要的应用，比如碰撞避免、计算最小包围盒[2]等问题，在不同的应用背景下也会有不同的定义方法，这里提供两种凸多胞体的定义：顶点定义法和半空间交定义法。此外，也简单介绍些与凸包问题相关的概念。

概念一. 顶点定义法

给定 R^n 空间上的有限点集 S，由点集中有限个极点（Extreme Point）构成的凸对象（Convex Object），就称为凸多胞体。

极点就可以看成是多胞体的顶点，以二维多边形为例，极点就相当于凸多边形的顶点。使用严格的数学定义来说，极点是指那些不能由凸多胞体内的其它点的凸组合（Convex Combination）表示出来的点。给定点 $x_1,\cdots,x_k$ 共 k 个点，它的凸组合可以用下面的等式表示：

$$a_1x_1+...+\mathrm{a}_k\,x_k, a_i\geq 0, a_1+...+a_k=1 \qquad (9.1)$$

一条线段就是它的端点的所有的凸组合，三角形就是它的三个顶点的凸组合，一个四面体是它的四个顶点的凸组合。

换句话，对于 R^n 空间中的对象 K，如果对象中的任意两个点构成的线段，也包含在对象 K 内，则称对象 K 是凸的。如图 9.1 所示的两个多边形 P_1 和 P_2，多边形 P_1 中的任意两点连成的线段也在多边形内，因此多边形 P_1 是凸的；但是多边形 P_2 中，存在两点的连成的线段不存多边形内的情况，如图中的线段 $\overline{A_2B_2}$，因此多边形 P_2 不是凸的。

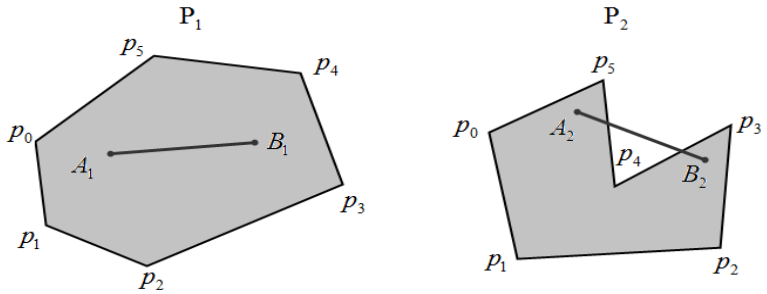

图 9.1 凸多边形和非凸多边形

概念二. 半空间交定义法

一个凸多胞体可以表示成有限个半空间（Half Space）相交的密闭空间。

先解释下什么叫半空间，例如在二维空间下，一条直线可以把平面分成两半，这两半就可以称为半空间，在三维空间下，一个平面可以把空间分成两半，这两半也就是两个半空间，在更高的维度下也可以有类似的表示。把 R^n 空间分成两半的平面称为**超平面**，例如二维的直线和三维的平面，都可以称为超平面。超平面 Γ 可以表示成如下形式：

$$\Gamma: a_1x_1+a_2x_2+....+a_nx_n=b \qquad (9.2)$$

由超平面 Γ 划分成的半空间可以用线性不等式来表示，等式（9.3）就是其中一个半空间的公式表示：

$$a_1x_1+a_2x_2+....+a_nx_n\leq b \qquad (9.3)$$

如果 m 个半空间形成一个封闭（Closed）的空间，该空间就是一个凸多胞体。m 个半空间，用 m 个线性不等式表示，形成一个线性不等式组如下所示：

$$\begin{cases} a_{11}x_1+a_{12}x_2+....+a_{1n}x_n\leq b_1 \\ a_{21}x_1+a_{22}x_2+....+a_{2n}x_n\leq b_2 \\ ..... \\ a_{m1}x_1+a_{m2}x_2+....+a_{mn}x_n\leq b_m \end{cases} \qquad (9.4)$$

二维空间下的超平面的几何表示就是一条直线，如图 9.2 所示，多边形 P 中的每一条边所在的直线可以把平面分成两个半空间，由 5 个超平面 $\Gamma_i, 1\leq i\leq 5$ 划分的半空间的交就形成

凸多边形P。

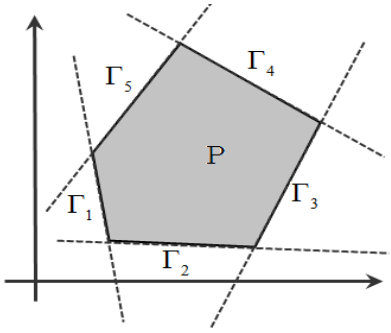

图 9.2 凸多边形P的空间交定义

概念三. 面（Facet）的定义

设P是在空间 R^n 上的一个凸多胞体，凸多胞体可以表示成多个半空间的交，分割半空间的超平面可以用等式（9.2）表示，则凸多胞体P的面用数学形式可以表示为：

$$F = P \bigcap \{x : \mathrm{a} \cdot \mathrm{x} = b\} \tag{9.5}$$

凸多胞体P的面的维度是 $n-1$，例如凸多面体的面是二维的。

概念四. 岭（Ridge）的定义

一个面的边界，就称为岭，岭表示的是两个面的邻接（Adjacency）部分，相当于是两个相邻面的交。对于空间 R^n 上的面的维度是 $n-1$，那么岭的维度就是 $n-2$。在三维空间上，多面体的面通常用三角形面片来表示，岭就是指三角形面片上的边。

概念五. 欧拉公式

在三维空间下，V, E, F 分别表示多面体的顶点数、边数、面数，则有下面的等式成立，O'Rourke[1](1998)对欧拉公式的证明做了详细的论述。

$$V - E + F = 2 \tag{9.6}$$

概念六. 正则多胞体

正则多边形，指每个边长度一样，每个内角相等的多边形，例如等边三角形，正方形等，显然有无数个正则多边形。

正则多面体，指各个面都一样且每个面有相同的内角的多面体，比如每个面都是正三角形、正五角形、五六角形等。正则多面体，也称为柏拉图立体（Platonic Solids），经证明只有 5 个这样的多面体，分别是正四面体、正六面体、正八面体、正十二面体、正二十面体，这些正则多面体的顶点、边、面信息如表 9.1 所示，它们的形状如图 9.3 所示，O'Rourke[1](1998)中详细证明了有且只有 5 个正则多面体。

正则多边形和正则多面体都可以统称为正则多胞体。

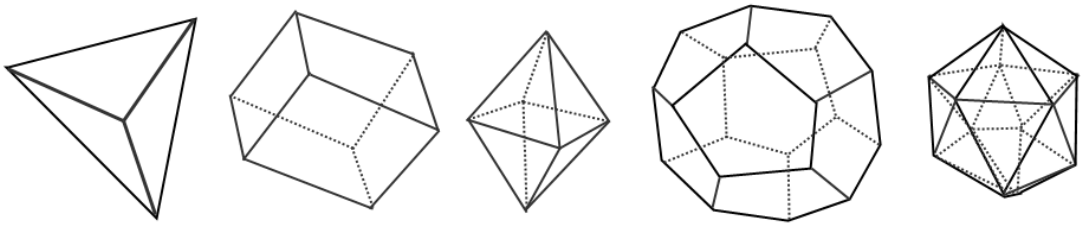

图 9.3 五种正多面体，从左到右分别是正四面体、正六面体、正八面体、正十二面体、正二十面体

表 9.1 五种正多面体的顶点、边和面信息

名字	V	E	F
正四面体	4	6	4
五六面体	8	12	6
正八面体	6	12	8
正十二面体	20	30	12
正二十面体	12	30	20

9.2. 凸包算法综述

9.2.1. 二维凸包

解决二维凸包问题，主要有 Jarvis 步进算法（Jarvis March），增量算法（Incremental Method），快速凸包算法（Quick Hull），分而治之算法（Divide and Conquer），Graham 扫描算法，单调链算法（Monotone Chain），Kirkpatrick–Seidel 算法，chan 算法。

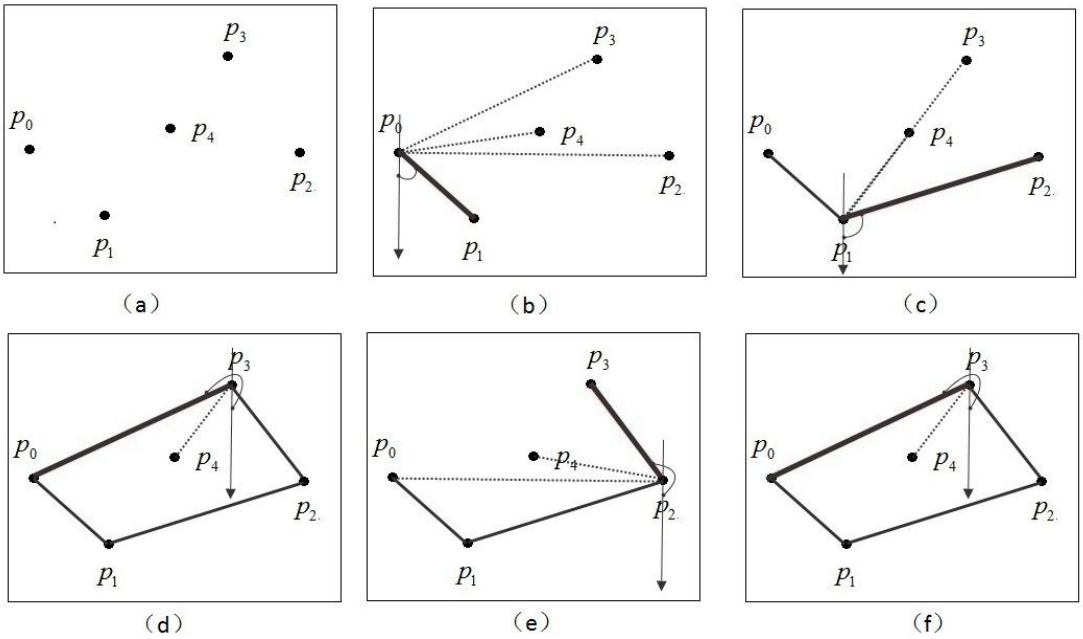

图 9.4 Jarvis 步进法

Jarvis 步进法是由 Jarvis[6](1973)提出的，但是它只是比它更早的 Chand&Kapur[3](1970)提出的礼物包裹（Gift Wrapping）算法在二维情况下的特例，Preparata&Shamos[4](1985)，O'Rourke[1](1998)和 Cormen el[5](2001)都有该算法更详细的描述。直观上，可以把 Jarvis 步进法看做是在集合 Q 的外面紧紧地包了一条线，线的一端固定在左下角的点上。把线向右拉使其绷紧，再拉高些，直到碰到一个点，这个点一定在凸包上。使线继续处于绷紧状态，查到下一个顶点，直至回来原点。更形式化的表示，是通过判断极角的大小，来确定下一个顶点，如图 9.4(a)所示，先找到最左下角的点 p_0（点的 X 分量最小，若多个点 X 分量相同，选择其中 Y 分量最小的点，它一定是凸包的一个顶点，搜索过程需要 $O(n)$ 的时间。接着相对于 p_0，如图(b)所示，找到极角最小的点 p_1,这一步搜索过程需要遍历点集中其它所有的点，算法复杂度是 $O(n)$。接着是分别搜索 p_2、p_3，直至回到起始点 p_0，算法复杂度与输出的凸包的顶点相关。

增量算法另外一种求解凸包问题的算法，由 Kallay[21](1984)提出，在 O'Rourke[1](1998)中有该算法详细的描述，增量算法的复杂度是 $O(n^2)$。它的基本思想是，随机选择若干个不共线的点，作为初始凸包，初始凸包可以是三角形也可以是四边形，为了使初始的凸包尽可能多的囊括非极点，可以选择最左、最右、最上、最下的四个点构成的四边形作为初始凸包。每次增加一个点，若该点在当前凸包内，则移除该点；否则，移除对该点可见的边，连接该点与当前凸包上和该点相切的点，产生新的凸包；重复上述步骤，直至点集中所有的点都处理为止。这里介绍两个重要的概念：可见与相切。若一个点 p 在凸多边形的边 e 指向外的法向量的一侧，称点 p 对边 e 可见或者称边 e 对点 p 可见。若点 p 在凸多边形外，那么对点 p 可见的边必然是连续的，它们可以表示为 $\overline{p_i p_{i+1}},\cdots,\overline{p_{j-1} p_j}$，称点 p_i, p_j 为凸多边形上与点 p 相切的两个点。注意，这里的可见和相切的概念只限于点在凸多边形的情况。设凸多边形上

一条边 $\overline{p_i p_{i+1}}$，该边指向外的法向量为 $\vec{n}$，若一个点 p 满足不等式 $(p-p_i)\cdot\vec{n}\succ 0$，则点 p 对边 $\overline{p_i p_{i+1}}$ 可见，否则，不可见。以如图 9.5(a)所示为例来解释下增量算法的基本过程，求点集 $\{p_0,p_1,p_2,p_3,p_4\}$ 构成的凸包。首先随机选择三个不共线的点 $\{p_0,p_1,p_4\}$，作为初始的凸包，如图(b)所示。接着，如图(c)所示，增加一个点 p_5， p_5 在当前凸包内，是非极点，直接排除。如图(d)所示，增加一个点 p_2，点 p_2 对边 $\overline{p_4p_0}$ 和边 $\overline{p_0p_1}$ 不可见，但点 p_2 对边 $\overline{p_1p_4}$ 可见，与点 p_2 相切的点就是 p_1 和 p_4，移除 p_1 和 p_4 之间的边，连接 p_1、 p_2 与 p_2、 p_4，生成新的凸包。按照相同的方法处理点 p_3，最后得到点集的凸包。搜索凸多边形上与点相切的两个顶点的暴力算法的复杂度是 $O(n)$，但是可以采用二分查找的方法进行优化，搜索的算法复杂度降低到 $O(\log n)$，因此增量算法的复杂度可以从 $O(n^2)$ 降低到 $O(n\log n)$。

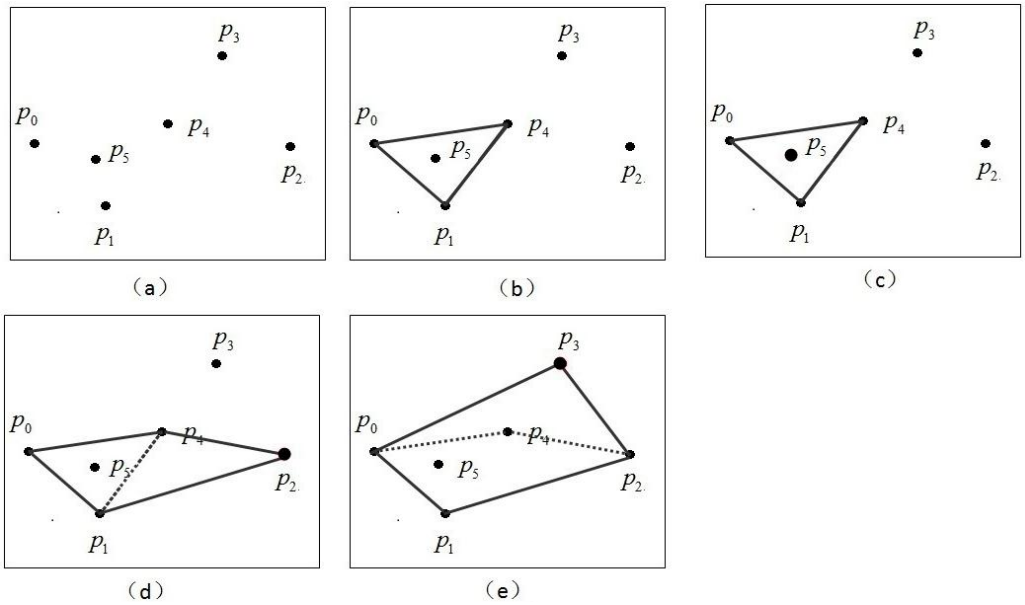

图 9.5 增量凸包算法

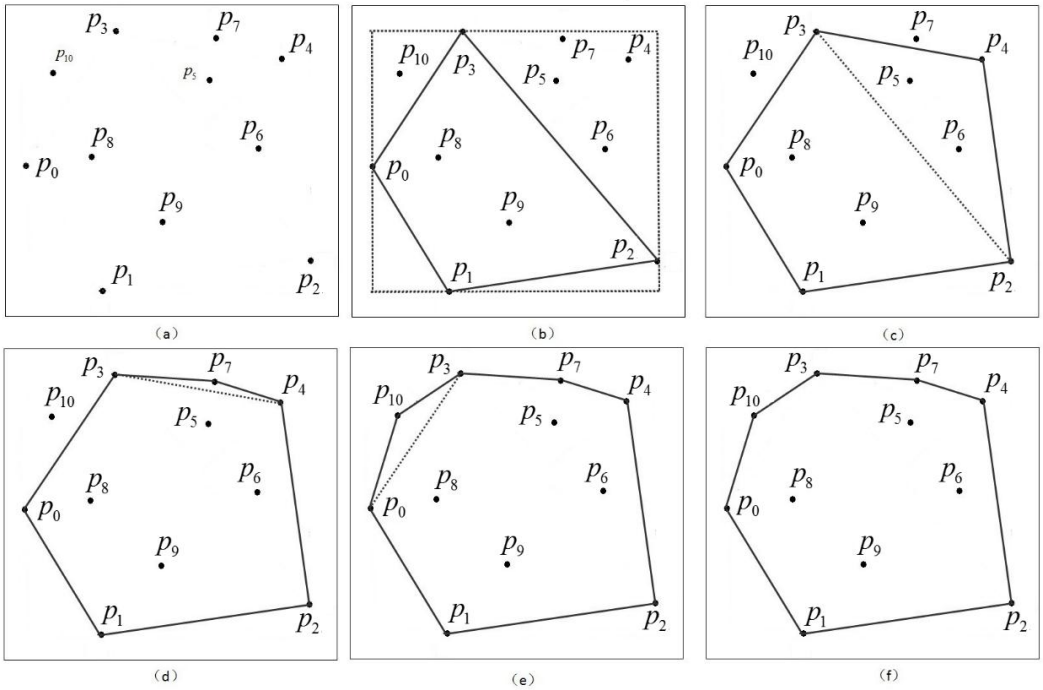

图 9.6 快速凸包算法

快速凸包算法的思想，很早是由多个研究者[15,16]独立提出的，由于该算法与快速排序的思想很相似，因此 Preparata&Shamos[4](1985)把它命名为快速凸包算法，并做了详细地描述，

在 O'Rourke[1](1998)中也有该算法的伪码表述。快速凸包算法在最好情况下，复杂度能达到 $O(n\log n)$，但是在最坏情况下复杂度是 $O(n^2)$，平均复杂度为 $O(n\log n)$。以图 9.6(a)所示为例，求点集 $\{p_0,...,p_{10}\}$ 的凸包。首先，找到点集中最左、最右、最上、最下的四个点，作为初始的凸包，尽可能多地舍弃初始凸包内的非极点，即舍弃点 p_8 和点 p_9。现在考虑当前凸包外侧还未处理的点，先考虑边 $\overline{p_2p_3}$,从边的外侧点集中，找到与边距离最远的点，如图(b)所示，边 $\overline{p_2p_3}$ 的外侧点集 $\{p_4,p_5,p_6,p_7\}$ 中，点 p_4 与边的距离最远，找到当前凸包上与点 p_4 相切的两个顶点，即顶点 p_2 和 p_3，删除当前凸包上 p_2 和 p_3 之间的边，连接 p_2、p_4 与 p_4、p_3，生成新的凸包。如图(c)所示，舍弃在新的凸包内的点，即舍弃点 p_5 和点 p_6。接着，只有边..和 p_3p_0 的外侧还存在未处理的点集，按照前面的操作进行处理，如图(d)和(e)所示，最终得到由 $p_0,p_1,p_2,p_4,p_7,p_3,p_{10}$ 共 7 个点构成的凸包，如图(f)所示。

分而治之算法最早是由 Preparata&Hong[9](1977)提出的，它的基本思想，就是把点集分割成左右或者上下两半，分别求出两半点集的凸包，再把它们合并，合并操作的算法复杂度是 $O(n)$，用 $T(n)$ 表示算法的复杂度，则有 $T(n)=2T(n/2)+\mathrm{O}(n)$，因此分而治之算法的复杂度是 $O(n\log n)$。以图 9.7(a)所示为例，求点集 $\{p_0,...,p_9\}$ 的凸包。可以把点集分成左右两半，如图(b)所示，左半点集产生一个子凸包 CH_1，右半点集产生另一个凸包 CH_2；接着，需要把这两个子凸包合并成一个大凸包，找到两个凸包的切线 $\overline{p_4p_8}$ 和 $\overline{p_2p_5}$，删除切线之间两个子凸包相对可见的边，连接两个子凸包的切线，完成子凸包的合并。相对于其它算法来说，分而治之算法的实现难度较大，Schneider& Eberly[11](2002)对二维分而治之凸包算法的实现有详细的表述。

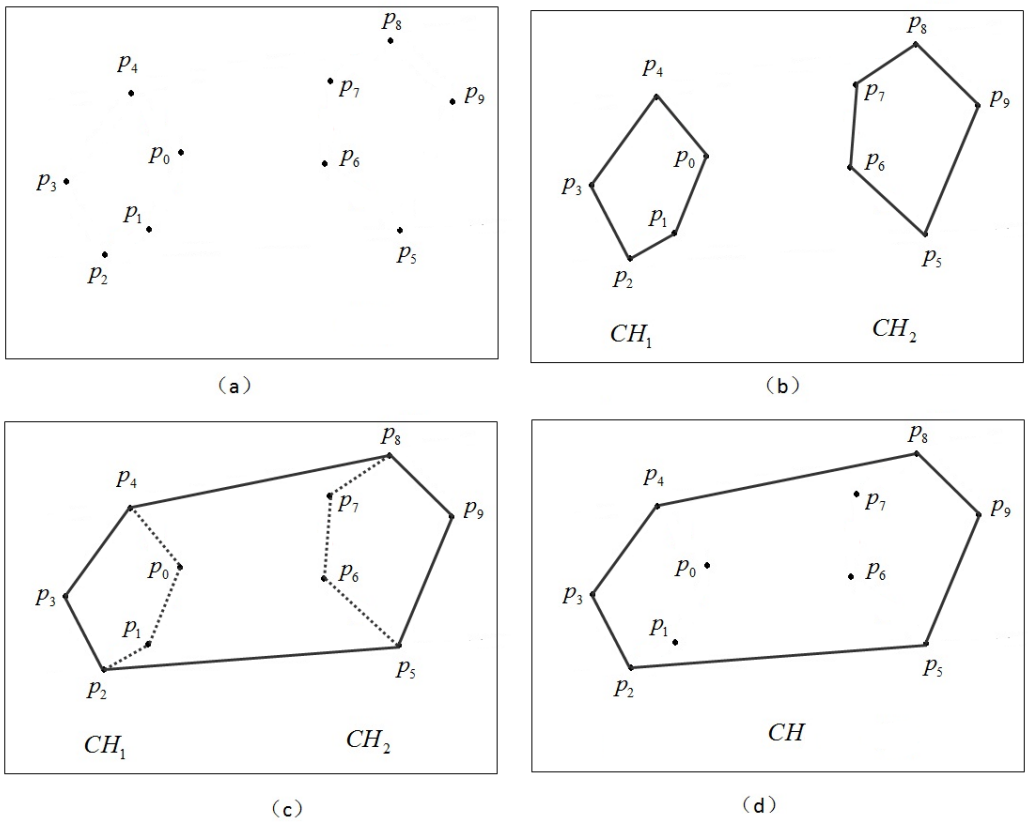

图 9.7 分而治之凸包算法

Graham 扫描算法最早是由 Graham[7]在 1972 年提出的，该算法最好和最坏的算法复杂度都是 $O(n\log n)$，实现也比较简单，能有效地处理各种退化情况，能有效的解决二维凸包问题，但是它无法扩展到三维，甚至于更高维的情况，因此对其它凸包算法的研究还是很有必要的，在后面的章节中将详细介绍该算法的思想和实现伪码。

单调链算法是由 Andrew[17]在 1979 年提出的，它与 Graham 扫描算法相似，首先对点集

进行排序，然后再利用有序的点集生成凸包，排序需要$O(n\log n)$时间，生成凸包需要$O(n)$，算法的复杂度也是$O(n\log n)$。不同的是：（1）单调链算法排序是根据点的 x,y 坐标的字典序进行；（2）根据有序点集生成凸包时，它会把点集分为上下两半，上点集生成凸包的上顶点链，下点集生成凸包的下顶点链，两个顶点就构成了一个完整的凸包，算法是以构造凸包的两个单调链为目的，这就是为什么称该算法是单调链算法的原因，**单调链的概念参考某节**。从效率上与 Graham 扫描算法相比较，单调链算法在排序中点的对比操作较简单，减少了计算量，但是增加了把点集划分为上下两半的操作。

首先对点集按照x, y递增的顺序排序，得到新的点集$S=\{p_0, p_1, \cdots, p_{n-1}\}$，设$x_{\min}, x_{\max}$分别是点集上$x$坐标最小的和最大的值，显然有$p_0.x=x_{\min}$。设$p_{\min,\min}$表示点的$x$坐标等于$x_{\min}$时，$y$坐标取最小值的点；同理$p_{\min,\max}$表示点的$x$坐标等于$x_{\min}$时，$y$坐标取最大值的点。若点集$S$上存在多个不同点的$x$坐标都是$x_{\min}$，那么$p_{\min,\min}$和$p_{\min,\max}$表示不同的点，否则它们表示相同的点。对于点$p_{\max,\min}, p_{\max,\max}$的定义也是类似的。如图 9.8 所示，连接$p_{\min,\min}, p_{\max,\min}$和$p_{\min,\max}, p_{\max,\max}$，分别用$L_{\min}$和$L_{\max}$表示。在直线$L_{\max}$上方的点集称为上点集，构成凸包的上顶点链；同理，在直线$L_{\min}$下方的点集称为下点集，由它构成凸包的下顶点链。上（下）点集构造单调链也是基于对栈的操作，跟 Graham 扫描算法相似。更详细的算法思想介绍和 C++实现可以参考 Dan Sunday[19]。

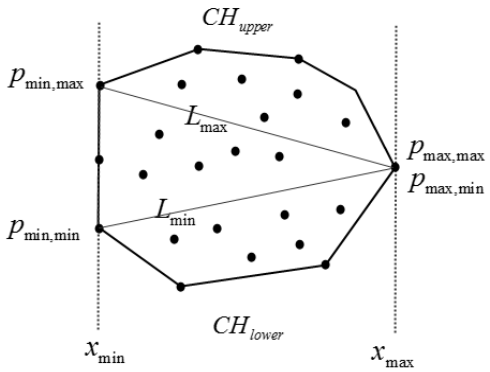

图 9.8 单调链算法

Kirkpatrick–Seidel 算法是由 Kirkpatrick&Seidel[18]在 1986 年提出的算法，以他们的名字来命名该算法，算法是分而治之（Divide and Conquer）算法的逆过程，分而治之算法是先把问题分解为一个个子问题（Divide），依次解决完每个问题后（Conquer），合并子问题的结果（Marry）。KS 算法则是先确定所有的子问题的结果是如何合并的，再考虑去解决每个子问题，Kirkpatrick&Seidel 称之为“Marriage-Before-Conquest”法则。算法复杂度为$O(n\log h)$，其中n表示点集的个数，h表示凸包的顶点个数，它的复杂度不仅与输入的点集相关，也与输出凸包的顶点个数相关。

Chan[20](1996)提出了另外一种实现更为简单、复杂度也是$O(n\log h)$的算法。

最后，对二维凸包算法进行总结，如表 9.2 所示：

表 9.2 二维凸包算法

算法	时间复杂度	来源
暴力算法		
Jarvis 步进法（礼物包裹算法）	$O(nh)$	Chand&Kapur(1970), Jarvis(1973)
Graham 扫描算法	$O(n\log n)$	Graham(1972)
快速凸包算法	$O(n\log n)$	Eddy(1977), Bykat(1978), O'Rourke(1998)
分而治之算法	$O(n\log n)$	Preparata&Hong(1977)
单调链算法	$O(n\log n)$	Andrew(1979)
增量算法	$O(n\log n)$	Kallay(1984)
Kirkpatrick–Seidel 算法	$O(n\log h)$	Kirkpatrick&Seidel(1986)
Chan 算法	$O(n\log h)$	Chan(1996)

9.2.2. 三维凸包算法

解决三维凸包问题，主要有礼物包裹算法、增量算法、快速凸包算法、分而治之算法。

礼物包裹算法最早由 Chand 和 Kapur[3] (1970)提出的，它不仅可以实现二维、三维凸包，还可以实现更高维的凸包，算法复杂度是 $O(nh)$， h 表示输出的面的数量， n 表示点集的个数，复杂度与输出凸包的面相关。直观上，三维的礼物包裹算法，可以看做是在点集的外面包围了一张纸，每次更新一个新的顶点，用纸覆盖住它，就像包裹礼物一样，直至包裹整个点集为止。

三维凸包的增量算法的复杂度是 $O(n^2)$，Clarkson&Peter[9](1989)提出了随机增量算法，它是增量算法的一种变种，通过把输入的点集进行随机化的操作，使算法的期望复杂度降为 $O(n\log n)$，O'Rourke[1](1998)提供了对增量算法 C 语言实现的详细描述，在开源的计算几何库 CGAL[14]提供了三维凸包的随机增量算法的实现。

Barber et al[10](1996)对三维的快速凸包算法进行了描述，并且通过实验证明快速凸包算法比随机增量算法的速度更快，需要的存储空间更小，但是没有理论上的依据，算法的平均复杂度为 $O(n\log n)$，最坏情况下的复杂度为 $O(n^2)$。其实，三维的快速凸包算法也可以看成是增量算法的一种变种，与随机增量算法相比，它们的不同就在于每次迭代从面的外部点集中选择的点不同。随机增量算法从外部点集中随机的选择一个点，但是快速凸包算法是选择距离最远的一个点。快速凸包算法已经有很广的应用，现在网上也有人提供有快速凸包算法健壮的算法实现，例如软件包 Qhull[12]就提供了健壮的快速凸包的算法实现，还比如开源的计算几何库 CGAL[13]也提供了快速凸包算法的实现，在后续的小节中将会详细介绍快速凸包算法。

凸包的分而治之算法最早由 Preparata&Hong[8](1977)提出，该方法理论上能实现二维、三维甚至于更高维的凸包，算法的复杂度达到 $O(n\log n)$，是三维凸包算法中理论上最快的。但是这种算法的应用并不广泛，最根本的原因就在于算法实现的难度比较大。Day[14](1990)提供了对三维凸包分而治之算法的 pascal 语言的实现，但是它实现的算法的复杂度是 $O(n^2)$，并没有达到 $O(n\log n)$。引用 O'Rourke[1](1998)的一句话来对该算做一个简单的表述：“This Algorithm is both theoretically important, and quite beautiful. It is, however, rather difficult to implement and seems not used as frequently in practice as other asymptotically slower algorithms”。

9.3. Graham 扫描算法

Graham 扫描算法会先将点按照极角的大小顺序排列，接着按顺序遍历每个点，通过夹角的大小判断哪些点在凸包上，算法的伪码如下所示:

求给定二维点集 $S=\{p_0,p_1,...,p_{n-1}\}$ 的凸包

1. 求出最左下角的点，即 X 分量的值最小点，若 X 分量值最小的点有多个，取 Y 分量最小的点，设为 p_0;
2. 剩下的点集，根据极角大小，逆时针排序，得到 $\{p_1,...,p_{m-1}\}$;
3. 令栈 *Stack* 为空，用 *Stack*[*i*] 表示栈中第 $i+1$ 个元素，用 k 表示栈中的元素个数；若栈中有 k 个元素，则 *Stack*[*k* - *1*] 即为栈顶元素；每一次 PUSH 操作，栈元素个数 k 加 1；每次 POP 操作，栈元素个数 k 减 1;
4. PUSH($Stack, p_0$);
5. PUSH($Stack, p_1$);
6. for($i=2$ ； $i \prec m$ ； $i=i+1$)
7. while ($k \geq 2$ && 由 $Stack[k-2], Stack[k-1]$ 和 p_i 的夹角 $\theta \geq 180$)
8. POP($Stack$);
9. end while;

10. end for;
11. if $k \prec 3$, then
12. 点集无法生成一个凸包;
13. else
14. 栈 *Stack* 中的 k 个元素，就是凸包的顶点;
15. end if;

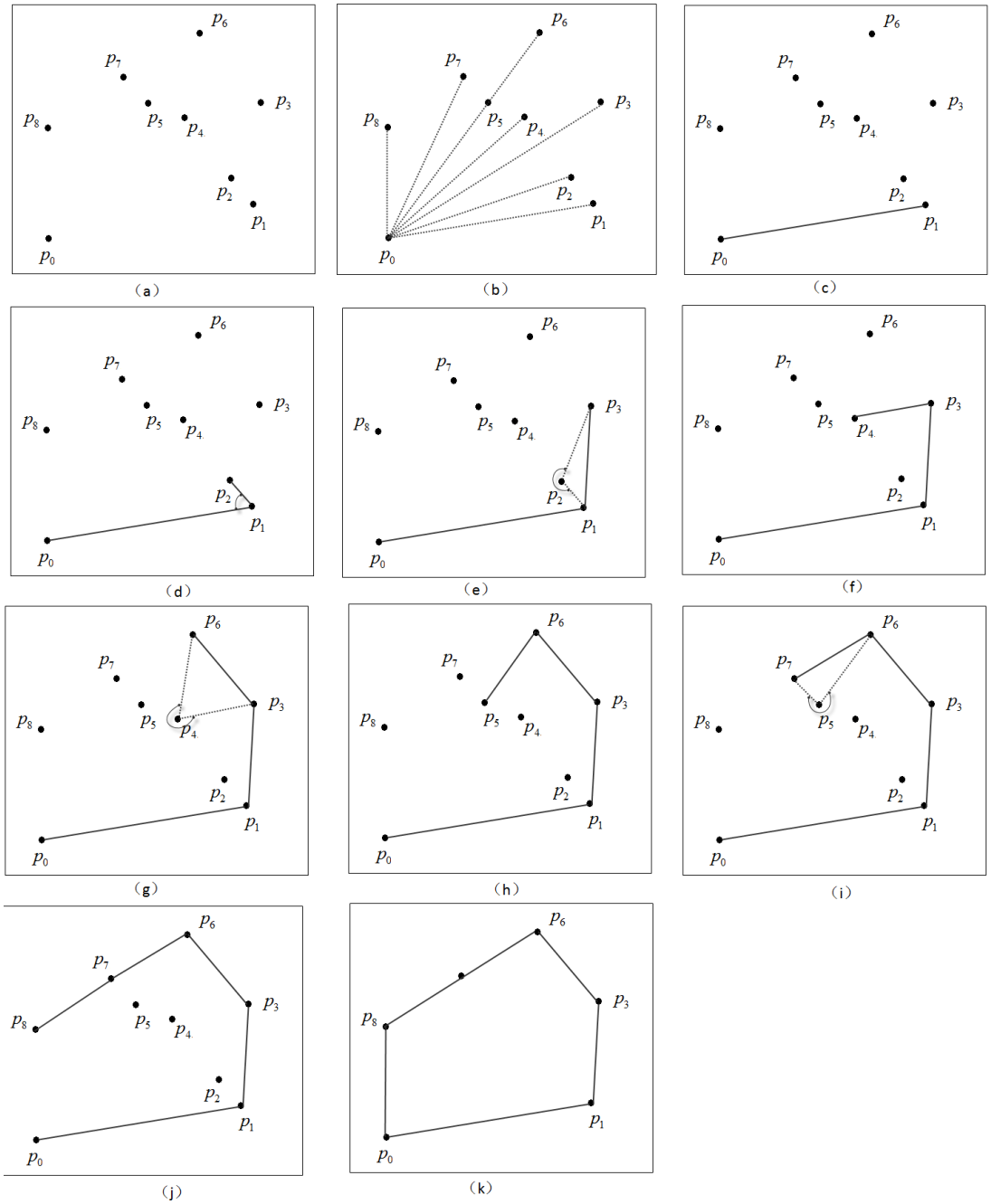

图 9.9 Graham 扫描算法过程

以如图 9.9(a)所示为例，给定一个点集，$\{p_0,\cdots,p_8\}$。第 1 步，选择最左下角的点 p_0，若 X 分量值最小的点有多个，取 Y 分量最小的点，**特别注意**，在实际实现中，可能存在多个相同的 p_0 点，必需把它们排除，保证点 p_0 是唯一的，否则在第 2 步中的排序会造成错误。第 2 步，如图(b)所示，对 $\{p_0,\cdots,p_8\}$ 按照极角从小到大进行排序，若极角相同，则根据与点 p_0 的距离从近到远排序。第 3~5 步，如图(c)所示，把 p_0 和 p_1 两个点压入栈中。逆时针方向处理每个顶点，第 6~10 步，如图(d)~(i)，若 $k \geq 2$ 且由 $Stack[k-2]$, $Stack[k-1]$ 和 p_i 的夹角大

于等式 180 度，则把栈顶元素出栈，否则把新点 p_i 压入栈。最终得到的凸包如图(k)所示，点 p_7 在线段 $\overline{p_6p_8}$ 上，是否把它作为凸包的顶点，可以根据具体的问题再作决定，上面提供的算法伪码，不允许把点 p_7 作为凸包的顶点。

对于对极点的排序，有多种算法实现，比如冒泡排序，快速排序，堆排序等，各种排序的性能对比，如表 9.3 的总结。若采用冒泡排序，则 Graham 扫描算法的复杂度将达到 $O(n^2)$，采用其它几种排序算法，Graham 算法的复杂度都能达到 $O(n\log n)$。采用快速排序，平均复杂度是 $O(n\log n)$，在 C++中，提供了 qsort()快速排序的算法，但在最坏情况下，Graham 算法的复杂度仍然是 $O(n^2)$。因此，可以选择归并排序或者堆排序。

表 9.3 有关排序算法的分析结果

算法	最坏情况	平均情况	时间上的最优性
冒泡排序	$O(n^2)$	$O(n^2)$	
快速排序	$O(n^2)$	$O(n\log n)$	平均时间最优
归并排序	$O(n\log n)$	$O(n\log n)$	最优
堆排序	$O(n\log n)$	$O(n\log n)$	最优

9.4. 礼物包裹算法

9.4.1. 基本思想

礼物包裹算法最早由 Chand&Kapur[3] (1970)提出的，它不仅可以实现二维、三维凸包，还可以实现更高维的凸包，算法复杂度是 $O(hn)$，h 表示输出的面的数量，n 表示点集的个数，算法复杂度跟输出面相关。直观上，三维的礼物包裹算法，可以看做是在点集的外面包围了一张纸，每次更新一个新的顶点，用纸覆盖住它，就像包裹礼物一样，直至覆盖整个点集。本小节只描述三维的礼物包裹算法，对于更高维的情况，可以参考 Chand&Kapur[3] (1970)以及 Preparata&Shamos[4](1985)的描述。

礼物包裹算法基于一个简单的理论：任意一个三维凸包的每一条边有且只有两个相邻面。三维凸包上的每个面用三角形面片表示，若一个凸包有 h 个面，则凸包共有 $3h/2$ 条边，根据欧拉公式可知，顶点的个数是 $n=h/2+2$。

设三维凸包上的面用三角形面片表示，本节暂不考虑各种退化情况，三维凸包的礼物包裹算法的算法伪码如下所示：

```
三维空间下点集 S={p0, p1,..., pn-1} 的凸包
1.  面集合为 Q，存储边的集合 T;
2.  Q ← ∅;
3.  T ← ∅;
4.  找到一个初始的凸包面 F;
5.  Q ← F;
6.  T ← 面 F 的每条边信息和计数器;    //计数器初始值为 1
7.  while( Q 非空 )
8.      F ← Q;
9.      for 面 F 的每一条边 e, then
10.         根据边 e，从集合 T 中检索出相应的边结构 E;
11.         if E.count ≺ 2, then
12.             以边 e 为边界，找到新的面 F';
13.             Q ← F';
14              更新边集合 T;    //若面 F'的边存储在集合 T 中，对应边的计数器值加
                                 // 1;否则，创建新的边结构，存入集合 T 中，初始计
```

//数器的值为 1;
15. end if;
16. end for;
17. end while;

实现礼物包裹算法时，必需解决算法中要处理的三个关键的子问题：

i. 如何存储边结构的集合 T 。
ii. 给定一个凸包上的面，如何确定与该面片上任意一条边相邻的三角形面片；
iii. 如何确定在凸包上的初始三角形面片；

子问题 i：如何设计存储边的集合的 T 数据结构。

首先，确定两点大小关系的对比规则，设两点 $P_0(x_0,y_0,z_0)$ 和 $P_1(x_1,y_1,z_1)$，若 $P_0<P_1$，当且仅当 $(x_0<x_1)\vee(x_0=x_1\wedge y_0<y_1)\vee(x_0=x_1\wedge y_0=y_1\wedge z_0<z_1)$；若 $P_0=P_1$，当且仅当 $x_0=x_1\wedge y_0=y_1\wedge z_0=z_1$；否则，$P_0>P_1$。设两条边 $e_0(p_0,q_0)$和 $e_1(p_1,q_1)$，若 $e_0<e_1$，当且仅当 $(p_0<p_1)\vee(p_0=p_1\wedge q_0<q_1$；若 $e_0=e_1$，当且仅当 $p_0=p_1\wedge q_0=q_1$；否则，$e_0>e_1$。

接着，考虑存储边的数据结构的设计，每个边结构表示为 $\{p,q,count\}$，其中 p,q 表示边的两个端点，$count$ 记录与边相邻面的个数，通过计数器可以判定与该边是否存在相邻面，因为凸包上的每一条边当且仅当有 2 个相邻面。

把边结构存储在一个平衡二叉树的数据结构中，边结构的查询、插入操作只需要 $O(\log m)$，其中 m 表示边的个数。

子问题 ii：给定一个凸包上的三角形面片，如何确定与该面片上任意一条边相邻的三角形面片。

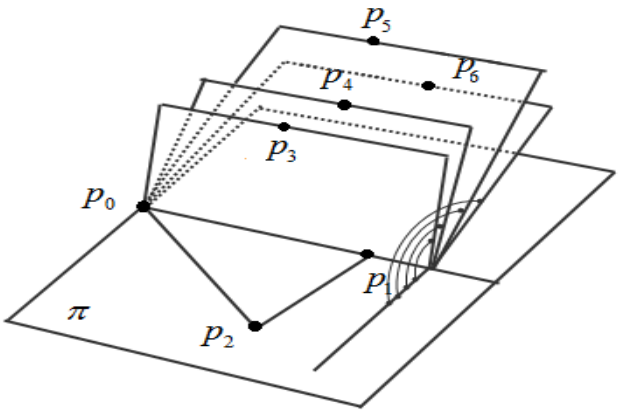

图 9.10 三角形是 $p_0p_1p_2$ 凸包上的面，在平面 π 上，不同的点 p_i 和边 p_0p_1 确定的平面跟平面 π 有不同的夹角大小

设三个点 $\{p_0,p_1,p_2\}$ 形成的三角形是凸包上的面片，三个点所在的平面是 π，从点集 $\{p_0,p_1,...,p_n\}$ 中到找一个点 p_i，使得由点 p_0，p_1 和 p_i 形成的平面 π_i 与平面 π 的夹角最大，那么由点 p_i 与边 $\overline{p_0p_1}$ 生成的三角形面片即是凸包的面。以图 9.10 所示为例，点 p_3,p_4,p_5,p_6 与边 $\overline{p_0p_1}$ 分别可以确定四个平面，这四个平面与平面 π 有一个夹角，选择夹角最大的平面，即由点 p_6 与边 p_0p_1 确定的平面，因此 $\{p_0,p_1,p_6\}$ 生成的三角形面片是凸包上的面。

在介绍了采用的策略后，接下来介绍如何实施策略，即如何判断哪个点与三角形面片确定的平面 π_i 与平面 π 之间的夹角最大。如图 9.11 所示，设平面 π 的法向量是 $\vec{n}$，若三角形点的顺序如该图所示，则法向量为 $\vec{n}=(-(p_1-p_0)\times(p_0-p_2))/\|(p_1-p_0)\times(p_0-p_2)\|$，即 $\|\vec{n}\|=1$，方向为垂直于平面 π 向上。因为 $\Delta p_0p_1p_2$ 在凸包上，所以对于任意一个点 p_i（$p_i\in\{p_0,\cdots,p_n\}$），都在平面上（On the plane）或者平面的上方空间（Above the plane）。指定一个方向向量 $\vec{a}$，令 $\vec{a}$ 同时垂直于法向量 $\vec{n}$ 和边 $\overline{p_0p_1}$，则向量 $\vec{a}=((p_1-p_0)\times\vec{n})/\|(p_1-p_0)\times\vec{n}\|$，即有 $\|\vec{a}\|=1$。点 p_2 在 $\vec{a}$ 所在的方向上。连接点 p_i 和点 p_0，得到向量 $v_i=p_i-p_0$，计算 v_i 在法向量 $\vec{n}$ 和向量 $\vec{a}$ 方向上的投影，分别是 $\vec{n}\cdot v_i$ 和 $\vec{a}\cdot v_i$，则对于不在平面 π 上的点 p_i 的夹角可以用等式（9.7）表示：

$$\rho_i = -\frac{\vec{a}\cdot v_i}{\vec{n}\cdot v_i} \tag{9.7}$$

遍历点集上的每个点，选择 ρ_i 值最大的点，就要所求的点，即：

$$\rho = \max \rho_i \tag{9.8}$$

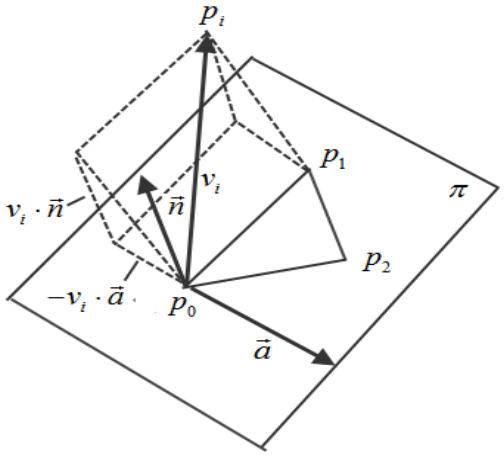

图 9.11 计算由点 p_i，p_0 和 p_1 确定的面，与平面 π 的夹角

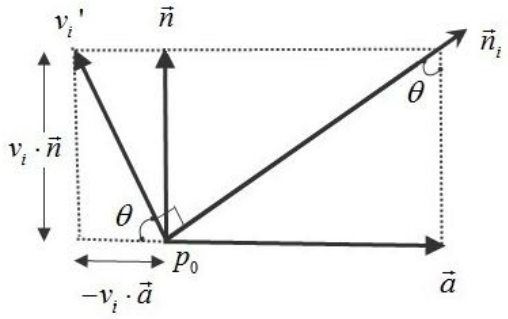

图 9.12 计算经过点 p_i 和边 p_0p_1 的面的法向量

由点 p_0 向点 p_1 方向观察图 9.11，可以得到横截面如图 9.12 所示，设 $\vec{n}_i$ 是由点 p_i，p_0，p_1 确定的平面的法向量，向量 v_i' 是向量 v_i 在如 9.12 所示的横截面上的投影，易知向量 v_i' 与法向量 $\vec{n}_i$ 垂直。设 $\vec{n}_i$ 在 $\vec{a}$ 方向上的投影为 λ_1，在 $\vec{n}$ 方向上的投影为 λ_2，由相似三角形的性质可得：

$$\frac{\lambda_1}{\lambda_2} = \frac{\vec{n}\cdot v_i}{\vec{a}\cdot v_i} \tag{9.9}$$

那么法向量 $\vec{n}_i$ 的方向可以表示为

$$\vec{n}_i = (\vec{n}\cdot v_i)\cdot\vec{a} - (\vec{a}\cdot v_i)\cdot\vec{n} \tag{9.10}$$

可以注意到，我们只需要找到 ρ_i 值最大的点，并不关心具体的值是多少，因此在计算向量 $\vec{n}$ 和 $\vec{a}$，可以不对它们进行归一化处理，即 $\vec{n} = -(p_1 - p_0)\times(p_0 - p_2), \vec{a} = (p_1 - p_0)\times\vec{n}$。未归一化的向量 $\vec{n}$ 和 $\vec{a}$ 也不会对等式（9.10）中法向量 $\vec{n}_i$ 的方向造成影响，读者可以自行证明。确定 ρ_i 值最大的点需要遍历点集，算法的复杂度为 $O(n)$。

子问题 iii：如何确定在凸包上的初始三角形面片。

算法的起始阶段，先排除点集中重复的点，找到初始的凸包面 $\triangle p_0p_1p_2$，如图 9.13 所示，从点集中找到最小的点 p_0，即点的 X 分量最小，若 X 分量最小的点有多个，则取其中 Y 分量最小的点，若 X,Y 分量最小的点有多个，则取其中 Z 分量最小的点。最小的点 p_0 是唯一的，经过点 p_0 作一个平面 π_0，平面的法向量表示为 $\vec{n}_0 = (a_1, 0, 0), a_1 \neq 0$，任取一个向量 $\vec{a}_0 = (0, d_1, 0), d_1 \neq 0$，遍历点集上的点，使用等式（9.7）计算得到 ρ_i 最大的点，设该点是 p_1。采用等式（9.10），计算出平面 π_1 的法向量 $\vec{n}_1 = (\vec{n}_0\cdot v_1)\cdot\vec{a}_0 - (\vec{a}_0\cdot v_1)\cdot\vec{n}_0$，其中 $v_1 = p_1 - p_0$，易知

法向量 $\vec{n}_1$ 是 $(b_1,b_2,0)$ 的形式，第 3 个分量为 0。接着，指定一个与法向量 $\vec{n}_1$ 和边 $\overline{p_0p_1}$ 垂直的向量 $\vec{a}_1=(e_1,e_2,e_3)$，可以采用等式 $\vec{a}_1=(p_1-p_0)\times\vec{n}_1$ 来计算。遍历点集上的点，使用等式（9.8）计算出 ρ_i 最大的点，设该点是 p_2。至此，初始凸包上的三角形面片的三个点 p_0,p_1,p_2 都计算出来了，算法需要三次遍历点集，算法规模是 $O(3n)$。

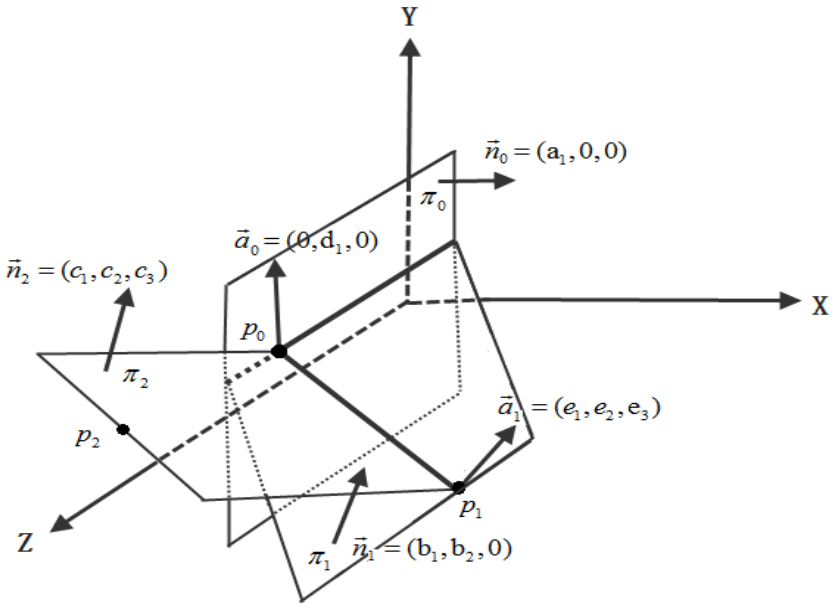

图 9.13 在凸包上的初始三角形面片 $\Delta p_0p_1p_2$

综上，在礼物包裹算法中，确定初始的三角形面片，需要时间 $O(3n)$；每确定一个新的凸包面，需要时间 $O(\log m)+O(n)$，其中 m 表示边，又 $m\le 3(n-2)$，即时间复杂度为 $O(n)$。因此，三维礼物包裹算法的时间复杂度为 $O(nh)$。

9.4.2. 特殊情况

1. 多点共面

如果算法实现中，没有考虑多点共面的情况，会引起错误。如图9.14所示，求与平面 π_0 上边 p_0p_1 相邻在面，得到平面 π_1 的夹角最大，但是在平面 π_1 上，存在4个或者大于4个点，图中有7个点 $\{p_0,p_1,p_3,p_4,p_5,p_6,p_7\}$ 在平面 π_1。若随机地选择一个点来更新平面F'，可能会选择点 p_5，也可能是点 p_7，但是显然点 p_7 不是凸包的极点，就引发了错误。若选择与边距离最远的点来更新平面F'，就会选择点 p_4，它一定是凸包上的极点，但可能会引发生成的面的边可能会发生交叉的情况。如图9.15所示，考虑边 $\overline{p_0p_1}$ 的相邻面，最远点是 p_4，连接3点得到凸包上的面 $\overline{p_0p_1p_4}$；接着考虑边 $\overline{p_1p_6}$ 的相邻面，最远点是 p_3，连接3点得到凸包上的另外一个三角形面片 $\overline{p_1p_6p_3}$，这就造成了边发生相交的情况。

这里介绍一种方法，设三维空间上的点集 $R=\{q_0,q_1,...,q_m\},m\ge 3$ 在同一个平面上，把点集 R 投影到二维空间上，得到新的点集 $R'=\{q_0',q_1',...,q_m'\}$。三维点投影到二维空间上，可以采用的方法是：设点集所在的平面的法向量为 $\vec{m}$，移除法向量 $\vec{m}$ 的绝对值最大的分量，即对于任意一个点 $q_i=(x_i,y_i,z_i)$，投影后的点 q_i' 如等式（9.11）所示。

$$q_i'=\begin{cases}(y_i,z_i),\|m_x\|=\max\{\|m_x\|,\|m_y\|,\|m_z\|\}\\(x_i,z_i),\|m_y\|=\max\{\|m_x\|,\|m_y\|,\|m_z\|\}\\(x_i,y_i),\|m_z\|=\max\{\|m_x\|,\|m_y\|,\|m_z\|\}\end{cases}\qquad(9.11)$$

使用二维凸包算法，求出点集 R' 的凸包，设求出的二维凸包为 $CH'=\{p_0',p_1',\cdots,p_k'\}$，

那么三维点构成的对应的凸多边形为$CH=\{p_0,p_1,\cdots,p_k\}$，其中，点$p_i'$是点$p_i$在二维空间对应的投影点，$0\le i\le k$。可以把$CH$划分成$\Delta p_0p_1p_2,\Delta p_0p_2p_3,...,\Delta p_0p_{m-1}p_m$共$m-1$个三角形面片，这种方法生成的三角形面片不会发生相交的情况。以图9.14为例，凸包为$p_0p_1p_6p_5p_4p_3$，就可以划分为$\Delta p_0p_1p_6,\Delta p_0p_6p_5,\Delta p_0p_5p_4,\Delta p_0p_4p_3$共4个三角形面，如图9.16所示。

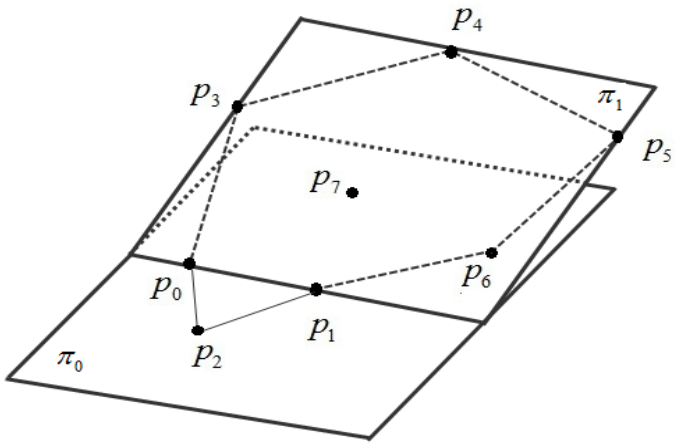

图9.14 多个点共面的情况

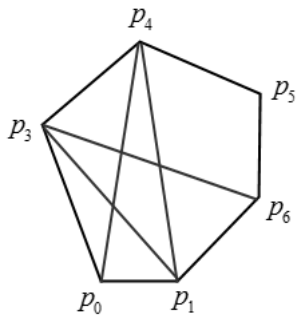

图9.15 选择与边距离最远点更新新面，可能产生的问题

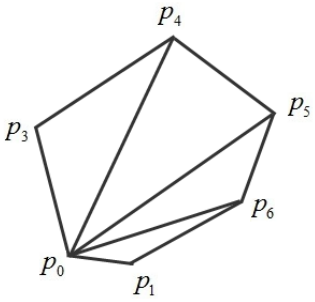

图9.16 把凸包上的点，划分为不相交的三角形面

在确定凸包上的初始三角形面片时，也可能有多个点在同一个面上，此时不需将共面点集投影到二维平面上求出子凸包，只需选择与指定边距离最远的点即可，该点一定是凸包的极点，避免了不必要的算法操作。

2. 分母为零的情况

若所求的点p_i在平面π上，则v_i在法向量方向$\vec{n}$上的投影$\vec{n}\cdot v_i=0$，显然，等式（9.7）的分母为零。此时，需要根据分子$\vec{a}\cdot v_i$的符号判断该点是否有效。若$\vec{a}\cdot v_i\ge 0$，说明该点是无效点，直接忽略；若$\vec{a}\cdot v_i\prec 0$，说明该点是有效的，则夹角最大的点在平面π上。以图 9.17所示为例，点$p_i,1\le i\le 9$在平面π上，在求与三角形面$\Delta p_0p_1p_2$的边$\overline{p_0p_1}$相邻的凸包面时，点p_6,p_7,p_8,p_9是有效点，但是点p_0,p_1,p_2,p_3,p_4,p_5都是无效点。

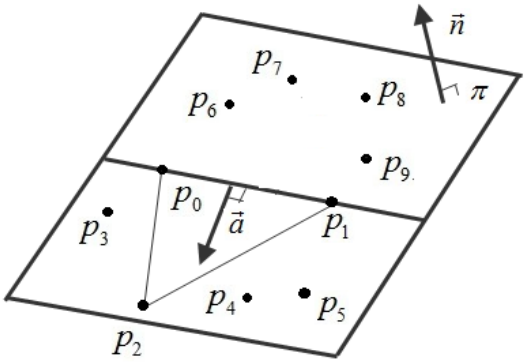

图9.17 分母为零的情况

9.4.3. 算法实现

存储点的数据结构可以表示为$\{x, y, z, IsOn\}$，其中，x, y, z 分别表示点的 X, Y, Z 方向上的坐标，$IsOn$ 是标志位，是一个布尔数，表示指定点是否是凸包上的极点，这个数据在输出结果时会起到作用，对于标志位为 $FALSE$ 的点可以删除，保留下标志位为 $TRUE$ 的点。起始时，对于任意一个点 $p \in S$，$p.IsOn$ 的值都为 $FALSE$。

前面介绍了边结构可以用 $\{p, q, count\}$ 来表示，包括边的两个端点 p, q 和相邻面个数的记数器 $count$。可以采用点的索引来记录边，减少存储空间，因此边结构中的 p, q 表示点的索引，用 V 表示全局点集，那么边结构 e 的端点信息就是 $V[e.p]$ 和 $V[e.q]$。使用平衡二叉树来作为存储边结构的容器，用 T 表示，边结构的查询、插入操作只需要 $O(\log m)$ 的时间，其中，m 表示边的个数。对存储边结构的平衡二叉树 T 的查询和插入的表示方法如下所示：

1. T.find($\{p,q\}$)，从 T 中检索出以 $V[p], V[q]$ 为端点的边结构，并将它返回；
2. T.insert($\{p,q\}$)，若 T 中存在以 $V[p], V[q]$ 为端点的边，则将相应的边结构的 $count$ 值加 1；否则，创建一个边结构 $\{p, q, 1\}$，并把它插入 T 中。

三维凸包上的面用三角形面片表示，存储面的数据结构可以表示为 $\{v_0, v_1, v_2\}$，与边结构类似，为了减少存储内存，v_0, v_1, v_2 表示点的索引，如果给定一个三角形面片 F，则它的 3 个顶点信息就是 $V[F.v_0], V[F.v_1], V[F.v_2]$。三角形面片的朝向是固定的，我们规定，通过等式 $\vec{n} = (V[F.v_1] - V[F.v_0]) \times (V[F.v_2] - V[F.v_0])$，计算出的面片的法向量指向凸包外。

算法中还需要设计一种存储三角形面片的容器，由于我们只需要将三角形面片插入容器首部（尾部）元素、删除容器首部（尾部）元素、遍历操作，所以可以采用链表这种数据结构作为存储三角形面片的容器。它一方向可以用于存储候选三角形面片；另一方面用于存储求出的凸包面片，最终作为结果输出。对链表数据结构 $List$ 主要的操作的表示方法如下所示：

1. $List$.push(O)，把元素 O 插入链表 $List$ 内；
2. $List$.pop()，从链表 $List$ 中弹出一个元素，并将它返回；

下面给出算法的伪码，考虑到了输入的点集存在重合或在同一个平面上，以及上一节描述的多点共面、分母为零的情况。

采用礼物包裹算法求三维空间下点集 $S = \{p_0, p_1, ..., p_{n-1}\}$ 的凸包

GIFT_WRAPPING(S)

1. 对点集 S 中的点从小到大排序;
2. 移除点集 S 中重复的点，得到新的点集 $V = \{q_0, q_1, \cdots, q_{m-1}\}$;
3. if $\|V\| \le 3$ ‖ 点集 V 共面, then
4. return $\varnothing$;
5. end if;
6. $Z = \varnothing$; // Z 存储用于输出的面片，链表

7. $Q=\varnothing$; //Q 存储候选面片，链表
8. $T=\varnothing$; //T 存储边结构，平衡二叉树
9. $F=\text{INIT\_FIRST\_FACET}(V)$;
10. $T.\text{insert}(\{p:F.v_0, q:F.v_1, count:1\})$;
11. $T.\text{insert}(\{p:F.v_1, q:F.v_2, count:1\})$;
12. $T.\text{insert}(\{p:F.v_1, q:F.v_0, count:1\})$;
13. $Q.\text{push}(F)$;
14. $Z.\text{push}(F)$;
15. while(Q 非空)
16. $F=Q.\text{pop}()$;
17. $\vec{n}=-(V[F.v_1]-V[F.v_0])\times(V[F.v_2]-V[F.v_0])$;
18. for (i=0 ; i<3 ; i+=1)
19. $p=F.v_i, q=F.v_{(i+1)\%3}$;
20. $E=T.\text{find}(\{p,q\})$;
21. if $E.count\geq 2$, then
22. continue;
23. end if;
24. $\vec{a}=\vec{n}\times(V[p]-V[q])$;
25. $\{PS,k_{max}\}=\text{GET\_EXTREME}(V,\vec{n},\vec{a},V[p])$;
26. $\vec{d}=(V[p]-V[q])\times(V[k_{\max}]-V[q])$;
27. if $\|PS\|\succ 1$, then
28. $CH=SUB\_\text{CH}(\vec{d},E,PS)$;
29. else
30. $CH=PS$
31. $CH.\text{push\_front}(q)$;
32. $CH.\text{push\_front}(p)$;
33. end if;
34. for each $k\in CH$
35. $V[k].IsOn=TRUE$;
36. end for;
37. $m=\|CH\|$ //凸包 CH 的顶点个数
38. $h_0=CH[0], h_1=CH[1], h_2=CH[2], h=2$;
39. while($h\prec m$)
40. $T.\text{insert}(\{h_2,h_1\})$;
41. $F'=\{h_0,h_2,h_1\}$;
42. $Z.\text{push}(F')$;
43. $Q.\text{push}(F')$;
44. $h+=1$;
45. if $h==m-1$, then
46. $T.\text{insert}(\{h_0,h_2\})$;
47. else
48. $h_1=h_2$;
49. $h_2=CH[h]$;
50. end if;
51. end while;
52. end for;
53. end while;
54. return Z;

算法第 1~5 步，排除点集 S 中重复的点、以及点集 S 在同一个平面上的情况。第 6~17

步，初始化必要的存储结构，确定在凸包上的初始三角形面片。接着，进入 while 循环，从 Q 中取出一个候选面片 F，分别处理面 F 的 3 条边，从 T 中查找到相应存储有相应边的边结构 E。如果 $E.count \geq 2$，说明该边已存在两个相邻面，不作任何处理，否则，计算出与其相邻面的 F'，将面 F' 的另外两条边插入到平衡二叉树 T 中，可以采用 $T.\text{insert}(\{p,q\})$ 方法。

其中，函数 GET_EXTREME($V,\vec{n},\vec{a},p$) 实现的功能是：给定一个凸包上的面 F，确定与面片 F 上的边 E 相邻的三角形面片，PS 表示一个点的索引集合，因为有可能与 E 相邻的三角形面片所在的平面上有多个点，即前面叙述的多点共面的情况，$k_{\max}$ 表示 PS 中与边 E 距离最远的点的索引。设面 F 所在的平面为 π，算法的伪码如下所示：

GET_EXTREME($V,\vec{n},\vec{a},p$)
return ($\{PS,k_{max}\}$)

1. $i=0$;
2. $PS=\varnothing$
3. while($i \prec \|V\|$) //确定 $vk,uk,d,k_{\max},PS$ 的初值
4. $v=V[i]-p$;
5. $vk=\vec{a}\cdot v, uk=\vec{n}\cdot v$;
6. $i+=1$;
7. if ($uk==0$)&&($vk\geq 0$), then
8. continue;
9. else if $uk\neq 0$, then
10. $d=-vk/uk$;
11. end if;
12. $k_{\max}=k$;
13. PS.push(i);
14. break;
15. end while;
16. while($i \prec \|V\|$)
17. $v=V[i]-p$;
18. $tpVk=\vec{a}\cdot v, tpUk=\vec{n}\cdot v$;
19. $i+=1$;
20. if ($tpVk==0$)&&($tpUk==0$), then //待测点 $V[i]$ 在边 E 上
21. continue;
22. else if($uk==0$), then //遇到 2 个或 2 个以上分母为零的点
23. if ($tpUk==0$)&&($tpVk \prec vk$), then
24. $vk=tpVk, k_{\max}=i$;
25. end if;
26. if $tpVk \prec 0$, then //分母为零时，有效点的判定条件
27. PS.push(i);
28. end if;
29. else if ($tpUk==0$)&&($tpVk \prec 0$), then //第 1 次遇到分母为零的点
30. $vk=tpVk, uk=0$;
31. $k_{\max}=i$;
32. $PS=\varnothing$;
33. PS.push(i);
34. else //分母不为零
35. $tpD=-tpVk/tpUk$; //参见等式（9.7）
36. if $tpD \succ d$, then
37. $vk=tpVk, uk=tpUk$;
38. $d=tpD$;
39. $k_{\max}=i$;

40. $PS=\varnothing$;
41. $PS.\text{push}(i)$;
42. else if $tpD==d$, then //存在多点共面的情况
43. if $tpUk\succ uk$, then
44. $vk=tpVk, uk=tpUk$;
45. $k_{\max}=i$;
46. end if;
47. end if;
48. end if;
49. end while;
50. return ($\{PS,k_{max}\}$)

其中，函数 INIT_FIRST_FACET(V)实现的功能是：确定在凸包上的初始三角形面片，算法的伪码如下所示：

INIT_FIRST_FACET(V)

1. 设 $0\le p\prec\|V\|$，且 $V[p]$ 是点集 V 中的最小点的索引;
2. $\vec{n}=(1,0,0), \vec{a}=(0,1,0)$;
3. $\{PS,q\}=\text{GET\_EXTREME}(V,\vec{n},\vec{a},V[p])$;
4. $v=V[q]-V[p]$;
5. $vk=\vec{a}\cdot v, uk=\vec{n}\cdot v$;
6. $\vec{n}=-vk\cdot\vec{n}+uk\cdot\vec{a}$;
7. $\vec{a}=v\times\vec{n}$;
8. $\{PS,r\}=\text{GET\_EXTREME}(V,\vec{n},\vec{a},V[q])$;
9. return $\{p,r,q\}$;

其中，函数 $SUB\_\text{CH}(\vec{d},E,PS)$ 实现的功能是：解决多点共面这种退化情况，把三维的点 $\{E.p,E.q\}\cup PS$ 投影到二维空间上，再根据二维凸包算法计算出凸多边形，最后把结果返回，即 $CH=SUB\_\text{CH}(\vec{d},E,PS)$。可以确定 $\{E.p,E.q\}\subseteq CH$，函数 $SUB\_\text{CH}(\vec{d},E,PS)$ 需要确保凸多边形 CH 是以 $E.p,E.q,c_0,c_1,\cdots$ 这样一个方向。三维点投影到二维，以及求二维凸包算法，在前面都已详细介绍，这里不再赘述算法。

最后，以图 9.18 所示为例来演示礼物包裹算法 GIFT_WRAPPING(S) 的流程，可以结合伪码，模拟下算法的过程。图中的点集为 $S=\{p_0,\cdots,p_8\}$，分布在一个 $2\times2\times2$ 的矩形方块上，点集的坐标如表 9.4 所示。礼物包裹算法会先确定一个初始的三角形面片，设为 $\Delta p_0p_4p_3$，把面片存入 Q 和 Z。接着，进入 while 循环，从 Q 中取出第 1 个面片 $\Delta p_0p_4p_3$，先处理面片上的边 $\overline{p_0p_4}$，产生新的面片 $\Delta p_0p_2p_4$；再处理边 $\overline{p_4p_3}$，此时出现多点共面的情况，点 p_6,p_7 与边 $\overline{p_4p_3}$ 共面，且产生的夹角最大，随机启动“多点共面情况”的子算法，最终产生两个面片 $\Delta p_3p_4p_6$ 和 $\Delta p_4p_7p_6$；最后处理边 $\overline{p_3p_0}$，产生新的面片 $\Delta p_3p_0p_1$。在处理完第 1 个三角形面片后，候选面片集 $Q=\{\Delta p_0p_2p_4,\Delta p_3p_4p_6,\Delta p_4p_7p_6,\Delta p_3p_0p_1\}$。由于 Q 非空，第 2 次进行 while 循环内的迭代操作，处理面片 $\Delta p_0p_2p_4$，先处理边 $\overline{p_0p_2}$，生成新的面片 $\Delta p_0p_1p_2$；再处理边 $\overline{p_2p_4}$，生成 $\Delta p_2p_7p_4$；最后处理边 $\overline{p_4p_0}$，由于该边的两邻面个数为 2，已不存在其它相邻面，不作处理，此时候选面片集 $Q=\{\Delta p_3p_4p_6,\Delta p_4p_7p_6,\Delta p_3p_0p_1,\Delta p_0p_1p_2,\Delta p_2p_7p_4\}$。由于 Q 非空，从中取出一个平面，重复上述操作，直至 Q 为空，输出三维凸多边体的面片，算法结束。如表 1.4 所示，由于篇幅限制，只记录了前 4 个面片的处理结果。

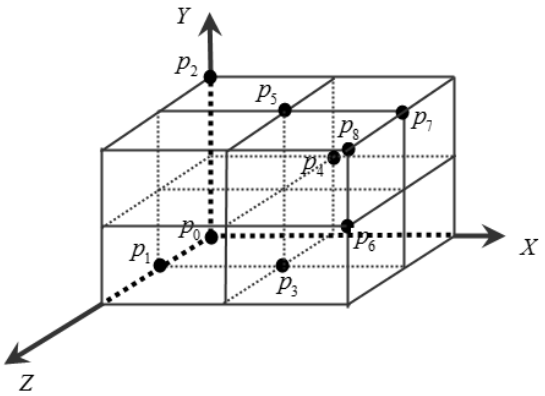

图1.18 礼物包裹算法过程

表9.4 点集坐标和算法循环体内的处理结果

点	坐标
p_0	(0,0,0)
p_1	(0,0,1)
p_2	(0,2,0)
p_3	(1,0,1)
p_4	(1,1,0)
p_5	(1,2,1)
p_6	(2,1,2)
p_7	(2,2,1)
p_8	(2,2,2)

处理面片	处理面片的边	输出面片
$\triangle p_0p_4p_3$	$\overline{p_0p_4}$	$\triangle p_0p_2p_4$
	$\overline{p_4p_3}$	$\triangle p_3p_4p_6$, $\triangle p_4p_7p_6$
	$\overline{p_3p_0}$	$\triangle p_3p_0p_1$
$\triangle p_0p_2p_4$	$\overline{p_0p_2}$	$\triangle p_0p_1p_2$
	$\overline{p_2p_4}$	$\triangle p_2p_7p_4$
	$\overline{p_4p_0}$	相邻边个数为 2
$\triangle p_3p_4p_6$	$\overline{p_3p_4}$	相邻边个数为 2
	$\overline{p_4p_6}$	子凸多边形内边
	$\overline{p_6p_3}$	$\triangle p_1p_3p_6$
$\triangle p_4p_7p_6$	$\overline{p_4p_7}$	相邻边个数为 2
	$\overline{p_7p_6}$	$\triangle p_6p_7p_8$
	$\overline{p_6p_4}$	子凸多边形内边

9.5. 快速凸包算法

9.5.1. 基本思想

快速凸包算法也可以看成是增量算法的一种变种，与随机增量算法相比，它们的不同就在于每次迭代从面的外部点集中选择的点不同。随机增量算法从外部点集中随机的选择一个点，但是快速凸包算法是选择距离最远的一个点，著名的开源代码 Qhull[12]、 CGAL[13]都有快速凸包算法的 C++实现。

在介绍快速三维凸包算法前，先介绍算法中会频繁使用到的两个基本的几何操作：

i. 给定 3 个三维空间上的点，确定一个平面。

平面可以用方程 $\vec{n}\cdot P+d=0$ 表示，设 3 个点分别是 $\{v_0,v_1,v_2\}$，则有 $\vec{n}=(v_1-v_0)\times(v_2-v_0)$，$d=-\vec{n}\cdot v_0$，叉积的顺序影响法向量的方向，若 $\vec{n}$ 是零向量，说明 v_0,v_1,v_2 三点共线。

ii. 计算三维空间上的点到平面的有符号距离。

设三维点为 P，平面为 $\Gamma:\vec{n}\cdot P+d=0$，那么点 P 到平面 Γ 的有符号距离表示为 $dist(P)=(\vec{n}\cdot P+d)/\|\vec{n}\|$。若 $dist(\mathrm{P})\succ 0$，称点 P 在平面上方（Above the Plane），；若 $dist(\mathrm{P})\prec 0$，称点 P 在平面下方（Below the Plane）；若 $dist(\mathrm{P})=0$，称点 P 在平面上（On the Plane）。

快速凸包算法是基于 Beneath Beyond 理论，增量算法也同样基于该理论，该理论如下所示，这里只考虑三维空间下的凸包：

设 H 是一个 R^3 空间上的凸包， p 是 R^3-H 上的一个点。 F 是凸包 $conv(p\cup H)$ 上的面，当且仅当

（1） F 是凸包 H 的一个面且点 p 在面 F 的下方；

（2） F 不是凸包 H 的一个面， F 是由凸包 H 的边和点 p 构成，点 p 在该边相邻的一个面的上方，在该边相邻的另一个面的下方。

若点 p 在 H 内，则 $conv(p\cup H)$ 与 H 重合，显然，结论成立；若点 p 在 H 外，分为两种情况，以图 9.19 所示为例， H 是由极点 $\{p_0,p_1,p_2,p_3,p_4,p_5\}$ 构成的凸包， $conv(p\cup H)$ 是由极点 $\{p,p_0,p_1,p_2,p_4,p_5\}$ 构成的凸包。对于凸包 H 来说，点 p 在三角形面片 $\Delta p_0p_4p_3$，$\Delta p_0p_3p_5$，$\Delta p_2p_3p_4$，$\Delta p_2p_5p_3$ 的上方，点 p 在三角形面片 $\Delta p_0p_1p_4$，$\Delta p_0p_5p_1$，$\Delta p_2p_4p_1$，$\Delta p_2p_1p_5$ 的下方，因此在边 $\overline{p_2p_4}$，$\overline{p_4p_0}$，$\overline{p_0p_5}$，$\overline{p_5p_2}$ 一侧的面片是在点 p 的上方，另一侧的面片在点 p 的下方，我们称这样的边为临界边。当一个面 F 在 $conv(p\cup H)$ 上时，若点 p 在面 F 的下方，则 F 是 H 的一个面，否则，它是由点 p 与临界边构成的面片。

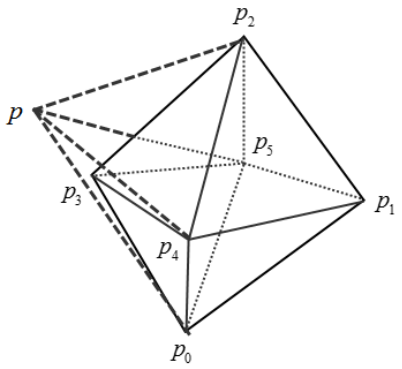

图 9.19 凸包 $conv(p\cup H)$

快速凸包算法的伪码如下所示

用快速凸包算法求三维空间上的点集 $S=\{p_0,p_1,...,p_{n-1}\}$ 的凸包

```
1.  初始化一个由四个点构成的四面体;
2.  for 四面体的每个面 F
3.      for 点集中的每个未被分配的点 p
4.          if p 在平面 F 的上方, then
5.              把 p 分配到面 F 的外部点集中;
6.          end if;
7.      end for;
8.  end for;
9.  把外部点集非空的面保存进待定面集 Q 中;
10. for 待定面集 Q 中的每个面 F
11.     从 F 的外部点集中找到距离面 F 最远的点 p;
12.     初始化可见面集 V，把面 F 保存进 V 中;
13.     for 可见面集 V 中每个面的未被访问过的邻居面 N
14.         if p 在面 N 的上方, then
15.             把 N 加到集合 V 中;
16.         end if;
17.     end for;
18.     把集合 V 中每个面的外部点集汇总到一个点集 L 中;
19.     集合 V 中所有面的临界边，构成一个集合 H;
20.     for 集合 H 中的每个边界边 R
21.         连接点 p 和临界边 R，创建一个新的面，加入到集合新面集 NS;
```

22. 更新新的面的相邻面;
23. end for;
24. for 新面集 *NS* 中的每一个新的面 *F*'
25. for 点集 *L* 中每个未分配的点 *q*
26. if *q* 在平面 *F*' 的上方，then
27. 把 *q* 分配到平面 *F*' 的外部点集中;
28. end if;
29. end for;
30. end for
31. 若待定面集 *Q* 和可见面集 *V* 的交不为空，则从待定面集 *Q* 中移除它们的交集，并删除可见面集 *V*;
32. 把新面集 *NS* 中外部点集非空的新面 *F*' 添加到待定面集 *Q* 中;
33. end for;

第 1 步，初始化一个由四个点构成的四面体。从点集中选择 4 个点生成初始的四面体，要保证这 4 个点生成的四面体是非退化的，尽可能选择跨度大的点。比如选择 *X* 分量最小的点和最大的点，*Y* 最小的点和最大的点，*Y* 最小的点和最大的点，这样可以使尽可能多的点包含在初始四面体内，达到排除尽可能多的非极点的目的。

第 2~8 步，存储每个面的数据结构包含面的 3 个顶点信息、它的外部点集和它的 3 个相邻面，面的外部点集指在面上方的、未分配的点构成的集合。“未分配”有两层意思：（1）该点不作为当前凸包的顶点，（2）其它面的外部点集不包含该点。若一个点在多个面的上方，则把它随机地分配到任意一个面的外部点集中。因此，若一个面的外部点集不为空，说明它一定不是的凸包 *conv*(*S*) 的面；相反，若一个面的外部点集为空，也不能保证它一定是凸包 *conv*(*S*) 的面。对于四面体的每个面 *F*，遍历点集，找到所有在面 *F* 上方的点，保存在面 *F* 的外部点集中，每个面结构都记录有一个外部点集，把外部点集非空的面保存在一个集合中，称这个集合为待定面集。

第 9 步，把外部点集非空的面保存进待定面集 Q 中。

第 11 步，从面 *F* 的外部点集中找到与面 *F* 距离最远的点 *p*，并且把点 *p* 从面 *F* 的外部点集中移除。

第 12 步，初始化可见面集 *V*，这里的可见面，是相对于点来说的，若点 *p* 在面的上方，则称该面是相对于点 *p* 的可见面。初始情况，可见面集 *V* 中只有一个面，就是面 *F*。

第 13~17 步，可见面集 *V* 中每个面中的未被访问过的邻居面 *N*，如果点 *p* 在面 *N* 的上方，把 *N* 加到集合 *V* 中。可见面集 *V* 的初始情况只有一个面 *F*，从面 *F* 向周围的面遍历，将所有对点 *p* 可见的面保存在可见集 *V* 中。以图 9.19 所示为例，设点 *p* 是面 $\Delta p_0 p_4 p_3$ 的外部点集中距离最远的点，经过算法第 13~17 步，得到可见面集为 $V=\{\Delta p_0 p_4 p_3, \Delta p_0 p_3 p_5, \Delta p_2 p_3 p_4, \Delta p_2 p_5 p_3\}$。

第 18 步，把可见面集合 *V* 中每个面的外部点集汇总到一个点集 *L* 中，因为可见面集中的面需要从当前凸包中删除。

第 19 步，可见面集合 *V* 中所有可见面的临界边，构成一个集合 *V*，若一条边的两个相邻面中，一个面是可见的，另外一个面是不可见的，那么该边就是临界边。

第 20~23 步，连接点 *p* 和集合 *H* 中的边界边，创建出新的面，更新新的面的相邻面。

第 24~30 步，对于每个新的面 *F*'，遍历点集 *L*，如果对于点集 *L* 中未分配的点 *q*，它在在 *F*' 的上方，则把它添加到面 *F*' 的外部点集中。

第 31 步，若待定面集 *Q* 和可见面集 *V* 的交不为空，则从待定面集 *Q* 中移除它们的交集，用一个直观数学表示就是 $Q=Q-Q\cap V$，删除可见面集中保存的面。移除交集的目的是为了保证已经删除的可见面，不会在待定面集 *Q* 中，导致后面的重复处理。

第 32~33 步，对于每个新的面 *F*'，若它的外部点集非空，则把它添加到待定面集 Q 中，进行下次的迭代。

这里考虑一个问题，在快速凸包算法第 2~8 步和第 24~30 步中，若一个点在多个面的上方，则把它随机地分配到任意一个面的外部点集中，是否会造成错误，即点集 *S* 中存在一

个极点最终不会是凸包 $conv(S)$ 上的顶点。我们可以证明下这条结论：

若一个极点在多个面的上方，把它随机的分配给任意一个可见面，该点最终都会作为凸包的顶点。

我们用反证法来证明，若一个极点 p 未分配给任意一个面的外部点集，则它一定不会出现在凸包的顶点上，这就与它是极点相矛盾，因此它一定是初始化的四面体的某个面的外部点集。令 p 和 q 在相同面的外部点集中，设 p 在面的上方但是不在由 q 与边生成的新的面的外部点集中，因此 p 在可见面上，但是在由 q 与临界边生成的新的面的下方，这表明 p 在凸包的内部，则它不是一个极点，发生矛盾。综上，我们可以得到结论。

快速凸包算法首先会初始化一个 4 个点的凸包，然后把点集分割成各个面的外部点集，挑选距离面最远的点进行处理。任意一个极点都不会因为分割为外部点集导致被丢弃。根据理论 Beneath Beyond 理论（1），一个面要么保持不变，要么根据理论 Beneath Beyond 理论（2）创建新的面，算法能产生已处理点的凸包，采用这种方法不断的加入新的点进行处理，直至点集上的点都处理结束为止。

算法的平均复杂度为 $O(n\log n)$，最坏情况下的复杂度为 $O(n^2)$。

9.5.2. 算法实现

为了方便后面的叙述，规定待定面集用 Q 表示，可见面集用 V 表示，临界边集用 H 表示。

存储点的数据结构可以表示为 $\{x, y, z, IsOn\}$，其中，x, y, z 分别表示点的 X, Y, Z 方向上的坐标，$IsOn$ 是标志位，是一个布尔数，表示指定点是否是凸包上的极点，这个数据在输出结果时会起到作用，对于标志位为 $FALSE$ 的点可以删除，保留下标志位为 $TRUE$ 的点。起始时，对于任意一个点 $p \in S$，$p.IsOn$ 的值都为 $FALSE$。

存储临界边的数据结构可以表示为 $\{v_0, v_1, f_0, f_1\}$，其中，v_0, v_1 表示边的两个端点，我们规定临界边的方向为由 v_0 指向 v_1；f_0, f_1 表示边的两个相邻面，因为它是临界边，所以两个相邻面中，必然有一个面是可见面，另一个面是不可见面，我们规定 f_0 表示可见面，f_1 表示不可见面。算法中，最远点与它的临界边，构造新的面。

存储面的数据结构可以表示为 $\{v_0, v_1, v_2, f_0, f_1, f_2, OS, flag\}$，因为三维空间上凸包的每个面用三角形表示。其中，$v_0, v_1, v_2$ 表示面片的三个顶点，顶点的顺序影响面的法向量的方向，我们规定采用等式 $\vec{n} = (v_1 - v_0) \times (v_2 - v_0)$ 计算出来的法向量指向凸包的外侧；f_0, f_1, f_2 表示与面片的三条边相邻的三个面，我们规定 f_0 是边 $\overline{v_0v_1}$ 的相邻面，f_1 是边 $\overline{v_1v_2}$ 的相邻面，f_2 是边 $\overline{v_2v_0}$ 的相邻面；OS 表示面的外部点集；$flag$ 是标志位，通过标志位来判定面是否被访问过，以及若面被访问过，是否为可见面，通过标志位来确定哪些边属于临界边。采用这种面数据结构，给定一个面，就可以通过深度优先搜索或宽度优先搜索遍历凸包上的每一个面，如图 9.20 所示，采用宽度优先搜索遍历凸包，三角形上的数字就表示搜索的层数，这种结构对搜索某个点在凸包上的可见面的效率特别高。

存储点集和面集的数据结构，可以是链表，因为元素的插入、删除操作为 $O(1)$，算法中只需要对点集、面集进行元素的插入、删除、遍历操作。用 $List$ 表示链表，对链表的操作可以表示为：

1. $List$.empty()，若 $List$ 为空，则返回 $TRUE$；否则，返回 $FALSE$；
2. $List$.erase(p)，从 $List$ 中删除 p；
3. $List$.front()（$List$.back()），返回 $List$ 首部（尾部）的元素；
4. $List$.pop_front()（$List$.pop_back()），删除 $List$ 首部（尾部）的元素；
5. $List$.push_front(p)（$List$.push_back(p)），把元素 p 插入到链表的首部（尾部）；
6. $List$.clear()，将链表中的元素清空；
7. SPLICE($dest$,src)，把链表 src 中的元素合并到链表 $dest$ 的尾部，并将链表 src 清空；

8. 给定链表的一个元素 it，它的上一个元素表示为 $it = List.\text{last}(it)$，它的下一个元素表示为 $it = List.\text{next}(it)$，$List.\text{begin}()$ 是链表起始的元素，但是 $List.\text{end}()$ 是紧随着链表最后一个元素的标志位。链表的遍历可以表示为：

```
for ( it = List.begin() ; it != List.end() ; it = List.next(it) )        //顺序遍历
                ..................
    end for;
或
    for ( it = List.rbegin() ; it != List.rend() ; it = List.rnext(it) )    //逆序遍历
                ..................
    end for;
```

每次迭代，所有的临界边会围成一圈，形成一个环，例如若只有三条临界边，它们一定是由三个点构成的环，设三个点是 $\{v_0, v_1, v_2\}$，则三条边可以表示为 $\overline{v_0v_1}$，$\overline{v_1v_2}$ 和 $\overline{v_2v_0}$。连接最远点 p 和临界边，构造新的平面后，需要更新新的平面的相邻面，所以临界边集合 H 采用的存储结构必需满足快速查询的要求。显然，链表不适合快速查找，可以采用平衡二叉树来存储临界边，使得元素的插入、删除和查询操作在 $O(\log m)$ 时间内完成，其中，m 表示临界边的个数。插入平衡二叉树的元素由两部分组成——{关键字，数据}，元素的关键字域作为二叉树中元素间对比用，数据域作为元素的数据部分。对于给定两个三维点 p 和 q，若有 $p \prec q$，当且仅当 $(p_x \prec q_x) \vee (p_x = q_x \wedge p_y \prec q_y) \vee (p_x = q_x \wedge p_y = q_y \wedge p_z \prec q_z)$；显然，当满足 $(not\ p \prec q) \wedge (not\ q \succ p)$ 时，$p = q$。

规定，存储在临界边集 H 中的元素的关键值域是一个三维坐标点，数据域是临界边结构。对平衡二叉树 $Tree$ 的操作可以表示为：

1. $Tree.\text{insert}(\{key,value\})$，将元素 $\{key,value\}$ 插入到 $Tree$ 中，若 $Tree$ 中已经存在关键域为 key 的元素，则插入失败；
2. $Tree.\text{empty}()$，若 $Tree$ 为空，则返回 $TRUE$；否则，返回 $FALSE$；
3. $Tree.\text{find}(key)$，从 $Tree$ 中检索出关键值域为 key 的元素，并返回它的数据域；
4. $Tree.\text{erase}(key)$，将关键值域为 key 的元素从 $Tree$ 中删除。

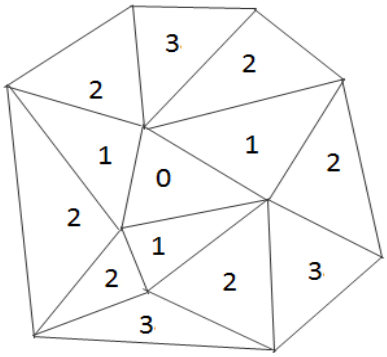

图 9.20 采用宽度优先搜索法遍历凸包

快速凸包算法的伪码如下所示：

用快速凸包算法求三维空间上的点集 $S = \{p_0, p_1, ..., p_{n-1}\}$ 的凸包

```
1.  NS = ∅;                                          //链表结构，存储创建的新面
2.  Q = ∅;                                           //链表结构，存储待定面
3.  if  ! INIT_TETRAHEDRON(S,NS) , then     //初始化四面体
4.      return  FALSE ;
5.  end if;
6.  PARTITION(Q,NS,S,R) ;                            // R 是一个面结构
7.  V = ∅;                                           //链表结构，存储可见面
8.  H = ∅;                                           //平衡二叉树结构，存储临界边
9.  L = ∅;                                           //链表结构，存储点集
```

10. while(Q 非空)
11. $F = Q.\text{front}()$; //此时，Q,V,H,L 均为空
12. $Q.\text{pop\_front}()$;
13. $p = \text{FIND\_FURTHEST}(F)$; //从面 F 的外部点集中找到与面 F 距离最远的点
14. $(F.OS).\text{erase}(p)$;
    //从凸包上找到点 p 的可见面，存入 V 中，并将临界边插入到 H 中
15. $F.flag = VISITED$;
16. $V.\text{push\_back}(F)$;
17. for ($it = V.\text{begin}()$; $it != V.\text{end}()$; $it = V.next(it)$)
18. for ($i = 0$; $i < 3$; $i = i + 1$)
19. $N = F.f_i$;
20. if $N.flag == NOT\_VISITED$, then
21. $N.flag == VISITED$;
22. $\vec{n} = (N.v_1 - N.v_0) \times (N.v_2 - N.v_0), d = -\vec{n} \cdot N.v_0$; //确定面 N 所在的平面
23. if $\vec{n} \cdot p + d > 0$, then
24. $V.\text{push\_back}(N)$;
25. else
26. $N.flag = BOUNDARY$;
27. 设 E 是一个临界边结构;
28. $E.f_0 = it, E.f_1 = N$;
29. $E.v_0 = it.v_i, E.v_1 = it.v_{(i+1)\%3}$;
30. $H.insert(\{E.v_0, E\})$;
31. end if;
32. else if $N.flag == BOUNDARY$, then
33. 设 E 是一个临界边结构;
34. $E.f_0 = it, E.f_1 = N$;
35. $E.v_0 = it.v_i, E.v_1 = it.v_{(i+1)\%3}$;
36. $H.insert(\{E.v_0, E\})$;
37. end if;
38. end for;
39. end for;
    //把集合 V 中每个面的外部点集汇总到一个点集 L 中，若待定面集 Q 和可见面集 V 的交不为空，则从待定面集 Q 中移除它们的交集
41. for ($it = V.\text{begin}()$; $it != V.\text{end}()$; $it = V.next(it)$)
42. if $!(it.OS).\text{empty}()$, then
43. $\text{SPLICE}(L, it.OS)$;
44. end if;
45. if $it \in Q$, then
46. $Q.\text{erase}(it)$;
47. end if;
48. end for;
    //连接临界边和点 p，构造新的面，并将新的面存入 NS，更新受影响面的相邻面
49. 从 H 中获取一个元素 E;
50. while($!H.\text{empty}()$)
51. $(E.f_1).flag = NOT\_VISITED$;
52. $F'.v_0 = p, F'.v_1 = E.v_0, F'.v_2 = E.v_1$;
53. $F'.f_1 = E.f_1$;
54. $NS.\text{push\_back}(F')$;

```
55.             for( i = 0 ; i < 3 ; i = i + 1 )
56.                 if (E.f1).fi == E.f0 , then
57.                     (E.f1).fi == F';
58.                     break;
59.                 end if;
60.             end for;
61.             H.erase(E.v0);
62.             E = H.find(E.v1)
63.         end while;
64.     lit = NS.last(NS.end());
65.         it = V.begin();
66.         while( it != NS.end() )
67.             it.f0 = lit;
68.             lit.f2 = it;
69.             lit = it;
70.             it = NS.next(it);
71.         end while;
72.         PARTITION(Q,NS,L,R);
73.         V.clear(), K.clear(), L.clear();
74.     end while;
75. 利用面结构 R，通过宽度优先搜索算法，得到三维凸包;
76.     return TRUE;
```

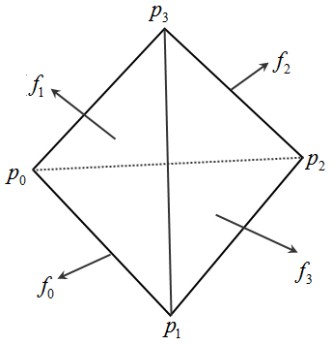

图 9.21 初始四面体

上述伪码中，函数 INIT_TETRAHEDRON(*S*,*NS*) 用于初始化四面体，初始的四面体如图 9.21 所示，对于任意一个面 f，保证得到的法向量 $\vec{n}=(f.v_1-f.v_0)\times(f.v_2-f.v_0)$ 的方向为垂直于面指向凸包外侧，算法伪码如下所示。

```
    INIT_TETRAHEDRON(S,NS)          // S 表示点集， NS 表示面集
1.  if 点集 S 共线，then
2.      return FALSE;
3.  end if;
4.  找到点集上 X 分量最小的和最大的点 p0, p1， Y 分量最小的点 p2;
5.  if p0, p1, p2 共线， then
6.      随机找到 3 个不共线的点 p0, p1, p2;
7.  end if;
8.  n = (p1 - p0) × (p2 - p0), d = -n · p0;
9.  itmin = itmax = it = S.begin();
10. distmin = distmax = n · it + d;
11. it = S.next(S);
12. while( it != S.end() )
13.     dist = n · it + d;
```

14. if $dist \prec dist_{\min}$, then
15. $dist_{\min} = dist$;
16. $it_{\min} = it$;
17. end if;
18. if $dist \succ dist_{\max}$, then
19. $dist_{\max} = dist$;
20. $it_{\max} = it$;
21. end if;
22. end while;
23. $p_3 = it_{\max}$;
24. if p_0, p_1, p_2, p_3 共面, then
25. SWAP(p_0, p_2); //交换点 p_0 与点 p_2 的值
26. $p_3 = it_{\min}$;
27. if p_0, p_1, p_2, p_3 共面, then
28. return $FALSE$;
29. end if;
30. end if;
31. $f_0.v_0 = p_0, f_0.v_1 = p_2, f_0.v_2 = p_1$; //4 个面的顶点
32. $f_1.v_0 = p_0, f_1.v_1 = p_1, f_1.v_2 = p_3$;
33. $f_2.v_0 = p_0, f_2.v_1 = p_3, f_2.v_2 = p_2$;
34. $f_3.v_0 = p_1, f_2.v_1 = p_2, f_2.v_2 = p_3$;
35. $f_0.f_0 = f_2, f_0.f_1 = f_3, f_0.f_2 = f_1$; //4 个面的相邻面
36. $f_1.f_0 = f_0, f_1.f_1 = f_3, f_1.f_2 = f_2$;
37. $f_2.f_0 = f_1, f_2.f_1 = f_3, f_2.f_2 = f_0$;
38. $f_3.f_0 = f_0, f_3.f_1 = f_2, f_3.f_2 = f_1$;
39. $S.erase(p_0)$; $S.erase(p_1)$;
40. $S.erase(p_2)$; $S.erase(p_3)$;
41. NS.push_back(f_0); NS.push_back(f_1);
42. NS.push_back(f_2); NS.push_back(f_3);

在函数 PARTITION(Q,NS,L,R) 中，Q 表示待定面集，NS 表示新创建的面集，L 表示集点，R 是一个面结构，通过它可以获取凸包上的所有面片，函数实现的功能是：若对于每个点 $p \in L$，存在一个面 $F \in NS$，则把点 p 加入到面 F 的外部点集中，若不存在这样的面，说明点 p 在当前凸包内部。

PARTITION(Q,NS,L,R)

1. for ($fit = NS$.begin() ; $fit != NS$.end() ; $fit = NS.next(fit)$)
2. $\vec{n} = (fit.p_1 - fit.p_0) \times (fit.p_2 - fit.p_0), d = -\vec{n} \cdot fit.p_0$;
3. for ($vit = L$.begin() ; $vit != L$.end() ; $vit = L.next(fit)$)
4. if $\vec{n} \cdot vit + d \succ 0$, then
5. $fit.OS$.push_back(vit);
6. L.erase(vit);
7. end if;
8. end for;
9. end for;
10. for ($fit = NS$.begin() ; $fit != NS$.end() ; $fit = NS.next(fit)$)
11. if ! $fit.OS$.empty(), then
12. Q.push_back(fit);
13. else
14. $R = fit$;

15. end if;
16. end for;

参考

[1] Joseph O'Rourke. *Computational geometry in C*. Cambridge university press, 1998.
[2] Stefan Gottschalk. "Collision queries using oriented bounding boxes." PhD diss., The University of North Carolina, 2000.
[3] Donald R. Chand, and Sham S. Kapur. "An algorithm for convex polytopes."*Journal of the ACM (JACM)*, vol.17, no.1, pp.78-86, 1970.
[4] Franco P. Preparata, and Michael Ian Shamos. *Computational geometry. an introduction*. Texts and Monographs in Computer Science, New York: Springer, 1985.
[5] Thomas H. Cormen, et al. *Introduction to algorithms*. Vol. 2. Cambridge: MIT press, 2001.
[6] Ray A Jarvis. "On the identification of the convex hull of a finite set of points in the plane." *Information Processing Letters*, vol.2, no.1, pp.18-21, 1973.
[7] Ronald L Graham. "An efficient algorithm for determining the convex hull of a finite planar set." *Information processing letters*, vol.1, no.4, pp.132-133, 1972.
[8] Franco P. Preparata, and Se June Hong. "Convex hulls of finite sets of points in two and three dimensions." *Communications of the ACM*, vol.20, no.2, pp.87-93, 1977.
[9] Kenneth L. Clarkson, and Peter W. Shor. "Applications of random sampling in computational geometry, II." *Discrete & Computational Geometry*, vol.4, no.1, pp.387-421, 1989.
[10] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. "The quickhull algorithm for convex hulls." *ACM Transactions on Mathematical Software (TOMS)*, vol.22, no.4, pp.469-483, 1996.
[11] Philip Schneider, and David H. Eberly. *Geometric tools for computer graphics*. Morgan Kaufmann, 2002.
[12] Qhull. "The Geometry Center Home Page." website, <http://www.qhull.org/>, last access in October,2014.
[13] CGAL Open Source Project. "Computational Geometry Algorithms Library", website, <https://www.cgal.org/>, last access in October,2014.
[14] A. M Day. "The implementation of an algorithm to find the convex hull of a set of three-dimensional points." *ACM Transactions on Graphics (TOG)*, vol.9, no.1, pp.105-132, 1990.
[15] William F Eddy. "A new convex hull algorithm for planar sets." *ACM Transactions on Mathematical Software (TOMS)*, vol.3, no.4, pp.398-403, 1977.
[16] Alex Bykat."Convex hull of a finite set of points in two dimensions." *Information Processing Letters*, vol.7, no.6, pp.296-298, 1978.
[17] Alex M Andrew. "Another efficient algorithm for convex hulls in two dimensions." *Information Processing Letters*, vol.9, no.5, pp.216-219, 1979.
[18] David G. Kirkpatrick, and Raimund Seidel. "The ultimate planar convex hull algorithm?." *SIAM journal on computing*, vol.15, no.1, pp.287-299, 1986.
[19] Dan Sunday. "The Convex Hull of a Planar Point Set." website, <http://geomalgorithms.com/a10-_hull-1.html>, last acess in October, 2014.
[20] Timothy M Chan. "Optimal output-sensitive convex hull algorithms in two and three dimensions." *Discrete & Computational Geometry*, vol.16, no.4, pp.361-368, 1996.
[21] Michael Kallay. "The complexity of incremental convex hull algorithms in R^d." *Information Processing Letters*, vol.19, no.4, pp.197, 1984.