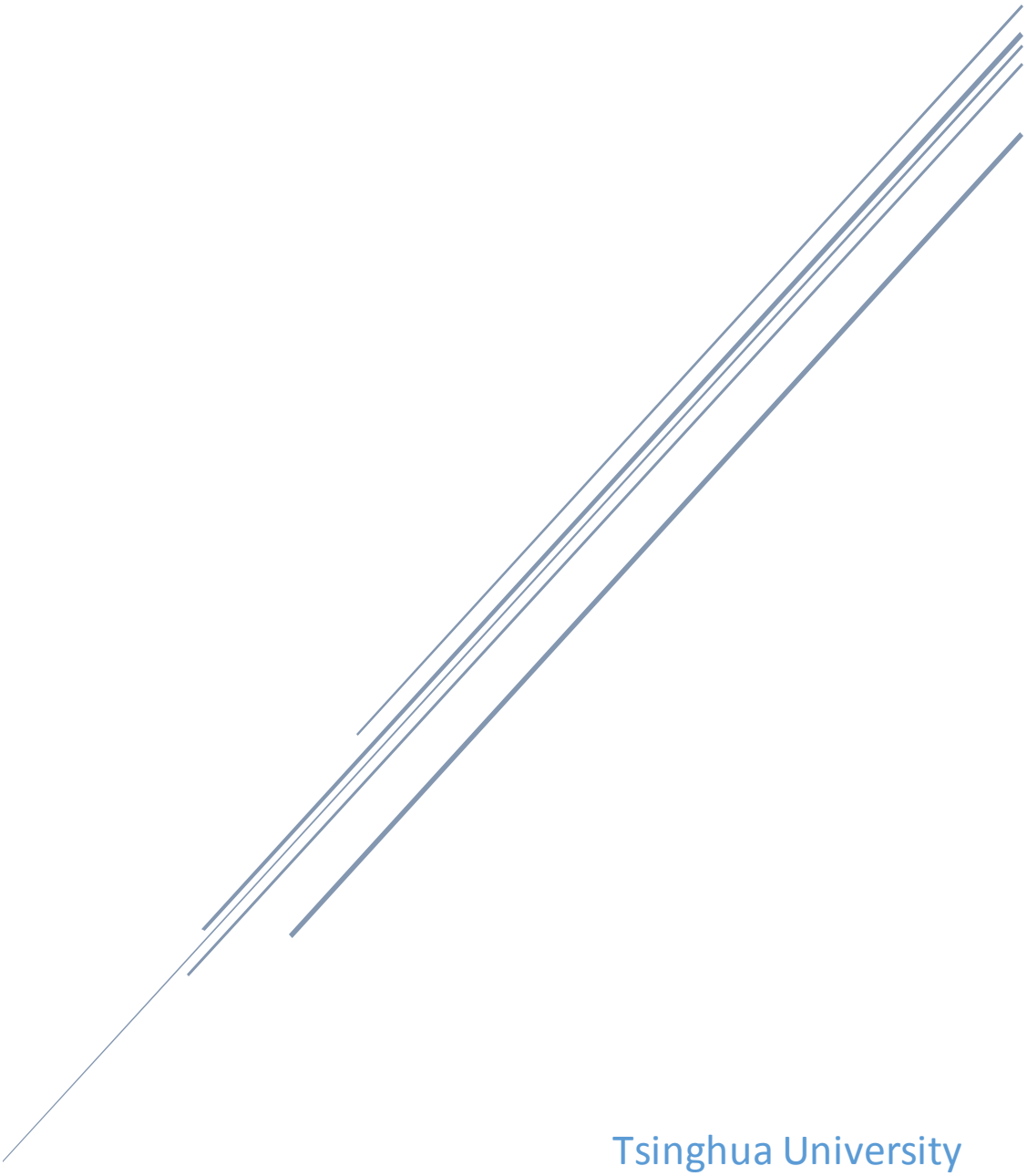
基于 VORONOI 和 DELAUNAY 的几何变换研究

易鑫 尚冲 肖楠



Tsinghua University

Computational Geometry 2014

目录

1 问题背景

人们往往将 Voronoi 图的发现归功与 Dirichlet 和 Voronoi。正因如此，有时其也称之为 Dirichlet 镶嵌。1644 年出版的《哲学原理》第三篇中，笛卡尔对宇宙爆炸做过分析，那里也出现了这类结构。就在那个世纪，Voronoi 图曾数次被重新发现。在生物学领域，这种再发现在极短的时间内疚发生过两次。1965 年，Browm 曾对树林中的数目密度做过研究。他定义了一个概念，称为“每棵树能够获得的面积”，实际上，如果把树当做基点，这一概念所指的也就是一棵树所对应的 Voronoi 但愿。一年之后，Mead 对一般的植物也采用了这一概念，他将 Voronoi 单元称为植物多边形。今天，与 Voronoi 图及其在各研究领域中的应用相关的文献已是浩若烟海。在 Okabe 等人的著作[1]中，对 Voronoi 图及其应用做了详实的介绍。

Lawson[2]曾经证明：通过一系列的边翻转操作，同一平面点集的不同三角剖分之间都可以相互转化。后来他还建议通过反复地进行边翻转，来找到一个好的三角剖分——每经过一次这样的边翻转，三角剖分的某个代价函数就会有一定的改善。在早期的一段时间内，人们已经注意到，所谓插值效果好的三角剖分，应该尽量避免狭长三角形的出现[3]。然而直到后来，才由 Sibson[4]将这一性质归纳为：若不考虑退化情况，则以角度向量为标准，局部最优的三角剖分必是唯一的——而这个三角剖分，也就是 Delaunay 三角剖分。

早期的凸包算法只能处理平面点集，在三维空间中构造凸包，难度要大很多。第一个此类算法，是 Chand 和 Kapur[5]提出的“礼品包扎”算法。该算法用一张平面不断地围绕已知的边“旋转”，从而逐一找出（构成凸包的）所有小平面；当重新回到起点时，算法终止。对于由 f 张小平面组成的一个凸包，该算法的运行时间为 O(nf)——最坏情况下为 $O(n^2)$。第一个运行时间不超过 O(nlogn)的算法，是由 Preparata 和 Hong 提出的，属于分治式算法。随后 Clarkson 和 Shor 提出了随机递增式算法[6]，可以利用 O(nlogn)的空间，在 O(nlogn)的时间内完成凸包的计算。

在求解几何问题的时候，对偶变换可以使我们从另一个角度来进行思考；而对于计算几何学家们来说，对偶变换已经成为了一个标准的工具。有一种几何对偶称为反演。通过这种几何对偶，可以将平面上“点位于圆内”的关系，转化为三维空间中“点位于平面下方”的关系。更详细地说，任何一个点 p: (p_x, p_y)都被投影到三维空间中的一个单位抛物面 $z = x^2+y^2$ 上——也就是说：

$$p' = (p_x, p_y, p_x^2 + p_y^2)$$

平面上的任何一个圆 C: $(x-a)^2 + (y-b)^2 = r^2$，都可以按照下列方法，变换为三维空间中的某张平面：将该圆（上的各点）投影到单位抛物面上，然后取投影所在的那张平面。具体地讲，就是

$$C': z = a(x-a) + b(y-b) + r^2$$

这样，点 p 落在 C 的内部，当且仅当 p’位于 C’的下方。这个变换可以推广至高维空间——这种情况下，d 维空间中的一个超球，就相当于(d+1)维空间中的一张超平面。

2 算法及原理

2.1 二维 Delaunay 算法

计算二维点集的 Delaunay 三角剖分时，我们采用了随机增量（*RIC*）算法[7]。首先，用一个足够大的三角形将整个点集$\boldsymbol{P}$包围起来。为此，需要引入两个辅助点p_{-1}和p_{-2}，它们与$\boldsymbol{P}$中的最高点联合构成的三角形，将包含所有的点。也就是说，我们计算的是$\boldsymbol{P} \cup \{p_{-1}, p_{-2}\}$的（而不是$\boldsymbol{P}$的）Delaunay 三角剖分。这个三角剖分一旦构造出来，只需将p_{-1}、p_{-2}以及与他们关联的各边删去，即可得到$\boldsymbol{P}$的 Delaunay 三角剖分。为此，我们所选择的p_{-1}和p_{-2}必须相距足够远，才不致于对$\boldsymbol{P}$的 Delaunay 三角剖分中的任何三角形有所影响。如图所示：

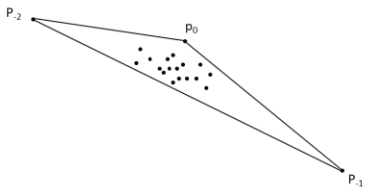


图 1

算法的伪代码如下所示：

算法 1 Delaunay Triangulation

```
算法 DELAUNAYTRIANGULATION(P)
输入：由平面上 n+1 个点组成的一个集合P
输出：P的一个 Delaunay 三角剖分
1.  令p0为 P 中依字典序最高的点，亦即，y 坐标最大的多个点中最靠右的那个
2.  在R2中选取相距足够远的点p-1和p-2，将 P 完全包含于三角形p0p-1p-2之中
3.  将T初始化为单独的一个三角形p0p-1p-2
4.  随机地选取 P\{p0}中各点的一个次序：p1,p2,…,pn
5.  for r ← 1 to n
6.    do (* 将pr插入到T中 *)
7.      找到pr所在的三角形pipjpk∈T
8.      if (pr落在三角形pipjpk的内部)
9.        then 分别将pr与三角形pipjpk的三个顶点联接起来
              (* 生成三条边，从而将三角形pipjpk一分为三 *)
10.           LEGALIZEEDGE(pr, pipj,T)
11.           LEGALIZEEDGE(pr, pjpk,T)
12.           LEGALIZEEDGE(pr, pkpv,T)
13.       else (*pr正好落在三角形pipjpk的某一条边（不妨设为pipj）上)
14.           将pr分别与pk以及与pipj关联的另一三角形的第三个顶点pl联接起来
            (* 从而将与pipj相关联的那两个三角形划分成四个三角形 *)
15.           LEGALIZEEDGE(pr, pipj,T)
16.           LEGALIZEEDGE(pr, pjpk,T)
17.           LEGALIZEEDGE(pr, pkpv,T)
18.           LEGALIZEEDGE(pr, pipv,T)
19.   将点p-1、p-2以及与之关联的所有边从T中剔除掉
20.   return (T)
```

其中，子函数 LEGALIZEEDGE 通过边翻转操作，将所有的非法边转换为合法边。其伪代码如下所示：

```
算法 LEGALIZEEDGE(p_r, p_i p_j, T)
1.  (*p_r为插入的点，p_i p_j为T中可能需要翻转的一条边 *)
2.  if (p_i p_j是非法的)
3.      then 令p_i p_j p_k为沿着边p_i p_j与p_r p_i p_j相邻的三角形
4.           将原来的边p_i p_j替换成边p_r p_k     (* 翻转p_i p_j *)
5.           LEGALIZEEDGE(p_r, p_i p_k, T)
6.           LEGALIZEEDGE(p_r, p_k p_j, T)
```

下面，还需要就以下两个重要步骤的实现方法做一详细描述：在算法 DELAUNAYTRIANGULATION 的第 7 行，如何才能找到p_r所处的那个三角形？在子函数 LEGALIZEEDGE 第二行的测试中，如何才能妥善地处理点p_{-1}和p_{-2}？

为了确定p_r落在哪个三角形之中，在构造 Delaunay 三角剖分的过程中，我们同时也维护一个有向无环图$\boldsymbol{D}$。$\boldsymbol{D}$中的各匹叶子，分别对应于当前三角剖分$\boldsymbol{T}$中的各个三角形，而且在这些叶子节点与对应的三角形之间，我们通过维护一些指针，使他们互相指向对方。$\boldsymbol{D}$中的每个内部节点，都对应于曾经在此前某个阶段的三角剖分中存在过的某个三角形。其在算法进行中的变化如图所示：

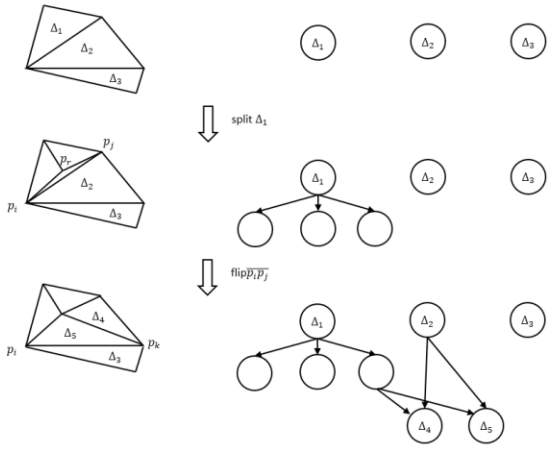


图 2

针对无穷远点的引入，作以下说明：

首先，对于任意一对点(x_p,y_p)和(x_q,y_q)，若$y_p > y_q$，或者$y_p = y_q$且$x_p > x_q$，则称“p高于q”。依此定义，即可在$\boldsymbol{P}$中各点之间确定一个字典序。

分别取位于整个点集$\boldsymbol{P}$之下、之上的一对水平线l_{-1}和l_{-2}。假想地在l_{-1}上取p_{-1}，沿l_{-1}向右移动p_{-1}，直到它不再落在$\boldsymbol{P}$中任何三个点的外接圆内，并使$\boldsymbol{P}$中各点相对于p_{-1}的极角次序，与他们的字典序完全一致。然后，假想地在l_{-2}上取p_{-2}，沿l_{-2}向左移动p_{-2}，直到它不再落在$\boldsymbol{P} \cup \{p_{-1}\}$中任何三个点的外接圆内，并使$\boldsymbol{P} \cup \{p_{-1}\}$中各点相对于$p_{-2}$的极角次序，与他们的字典序完全一致。

在点定位阶段，需要判断点p_j相对于有向直线$\overrightarrow{p_i p_k}$的位置。以下条件是等价的：

- p_j位于有向直线$\overrightarrow{p_i p_{-1}}$的左侧
- p_j位于有向直线$\overrightarrow{p_{-2} p_i}$的左侧
- 按字典序，p_j大于p_i

在判断某条边的合法性时，设待测试的边为$\overrightarrow{p_i p_j}$，与该边关联的两个三角形（如果都存在的话）的第三个顶点分别为p_k和p_l。若四个顶点不构成凸四边形，则$\overrightarrow{p_i p_j}$必定为合法边，否则，按照如下判据进行判断：

- $\overrightarrow{p_i p_j}$为三角形$p_0 p_{-1} p_{-2}$的一条边。这类边必定合法

- 下标i、j、k和l均非负。这是最常见的情况。此时，$\overrightarrow{p_ip_j}$非法，当且仅当p_l落在p_i、p_j和p_k的外接圆内
- 所有其余的情况。这些情况下，$\overrightarrow{p_ip_j}$合法当且仅当$min(k,l) < min(i,j)$。

2.2 三维 Convex Hull 算法

在计算三维 Convex Hull 时，我们实现了两种算法，随机增量算法（Random Incremental Construction）和快速凸包算法（Quick Hull）。RIC 在计算凸包时，是通过不断向当前的凸包中**随机**加入新点直到遍历完所有的点，其中我们需要维护一个点和面的可见图（Conflict Graph）。而 Quick Hull 算法思想是在 RIC 基础上，每次选择的新点是距离当前面的最远的点。将最远的点加入到凸包中，意味着最远点到当前面中间的点部分会落于新构建的凸包之内，这样就可以减少循环次数，提高原有的 RIC 算法的速度，不过我们要付出的代价是对每个面都要维护一个按照距离远近排序的可见点集合。

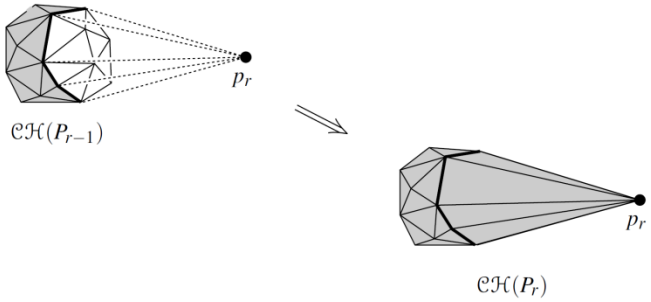


图 3 增量式构造凸包

2.2.1RIC 算法

RIC 算法将会维护一个不断增长的凸包，每次迭代都会在凸包外随机选择一个点P，将P加入到凸包，每次增加一个点只是在原来凸包上做一下改动：将点P的可见面删除，然后连接P和地平线形成新的面，将新面加入凸包。在这个过程中，涉及两个重要概念：1）一个记录着所有点和所有面的可见性的冲突图；2）点P 到凸包的地平线。关于这些概念，在 Deberg 的书中有详细解释。

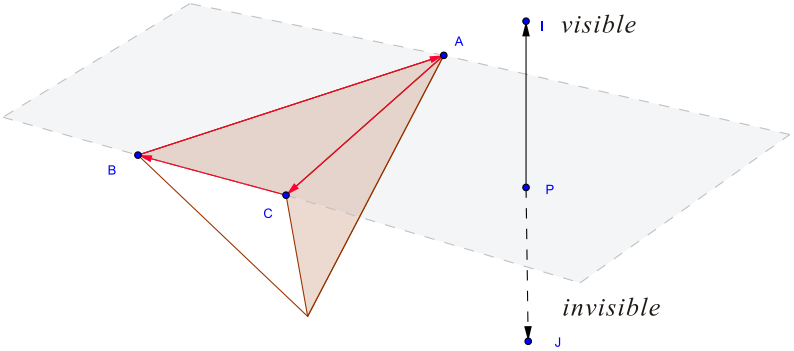


图 4 三维空间中点和面可见性关系

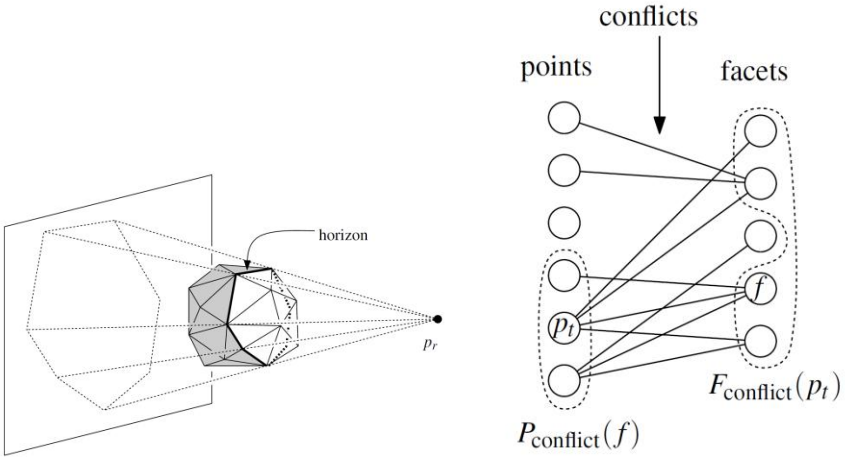


图 5 地平线（左）和冲突图（右）

算法 2 Convex Hull (RIC)

算法 RIC_CONVEXHULL(P)
输入：三维空间中的 n 个点组成的一个集合$\boldsymbol{P}=\{\boldsymbol{p}_1,\boldsymbol{p}_2,\ldots,\boldsymbol{p}_n\}$
输出：集合$\boldsymbol{P}$构成的三维凸包 $\mathbb{C}\mathcal{H}(\boldsymbol{P})$
初始化：随机选择 4 个不共面的点，构成四面体；对当前的 4 个面和剩余的点计算其可见图$\mathcal{g}$；任取其余各点的一个随机排列；
迭代：

```
1.  for r ← 5 to n
2.  do (*将p_r 插入到 C 中*)
3.      if (g(p_r)非空) (* 即，p_r落在C 外部 *)
4.      then 将g(p_r) 中所有的小平面从C 中删除
5.           沿着（恰好由g(p_r) 中各小平面组成的）p_r可见区域的边界行进将沿地平线的各边组织成一个有序表 L
6.           for (所有 e∈L)
7.               do 通过生成一张三角形小平面 f，将 e 与p_r联接起来
8.                if (和 f 相邻于 e 的小平面 f'与f 共面)
9.                then 将 f 与 f'合二为一，合并后小平面的冲突列表与 f'的相同
10.               else (* 确定 f 引起的冲突 *)
11.                  在 G 中生成一个对应于 f 的节点
12.                  考虑原先凸包上与 e 关联的那两张小平面，令为f_1和f_2
13.                  P(e) ←g(f_1) ∪g(f_2)
14.                  for (所有点p∈P(e))
15.                     do 若 f 与 p 可见，则将(p, f)加入到 g中
16.           在g 中删除以下点和弧：
                 a. 对应于p_r的节点、
                 b. 与g(p_r)中各小平面对应的节点、以及
                 c. 与之关联的所有弧
17.           return(C)
```

2.2.2Quick Hull 算法

Quick Hull 算法是对 RIC 算法的改进。RIC 维护一个点到面的二部图，其中的边表示两者相互可见。Quick Hull 改进了 RIC 随机选择可见点的做法，它每次都会选择距离当前面最

远的点。

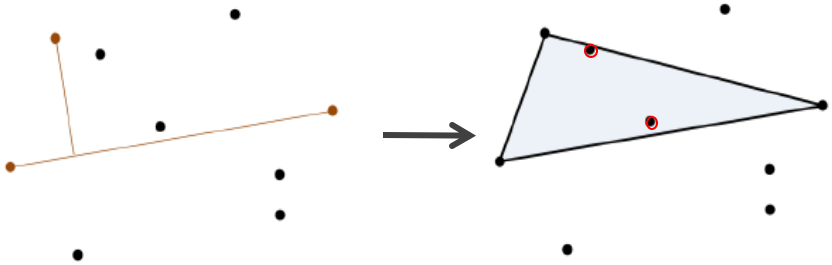


图 6 Quick Hull 在二维数据上选择最远点示意图(因为选择的最远点，红色点直接被圈入凸包中)

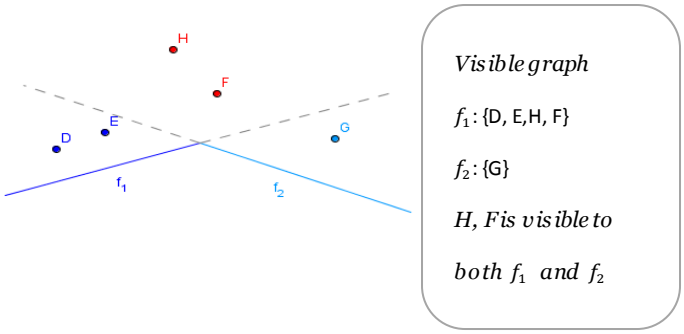


图 7 对于每个面维护的单向可见点链表

由于迭代过程需要查询最远点，我们将 RIC 算法中的冲突图改成所有面的可见点链表。这是一个单向的结构，每个点都被加入它的第一个可见面的链表中。所以这样做的最终结果是每个面只是记录它的部分可见点，每个点并没有指向它的可见面的链表。所以在我们的算法中，我们维护一个栈，栈中记录着所有可见点链表非空的面，每次迭代都从栈中弹出一个面F，从F的可见点链表中找到距离F最远的可见点P，然后再对P寻找P的所有可见面集合L。L的边界就是点P的地平线，根据P和地平线就能得到新的面，随后需要对新的面构建它们的可见点链表。具体实现中，我们使用了按距离优先的队列来表示每个面的可见点链表。

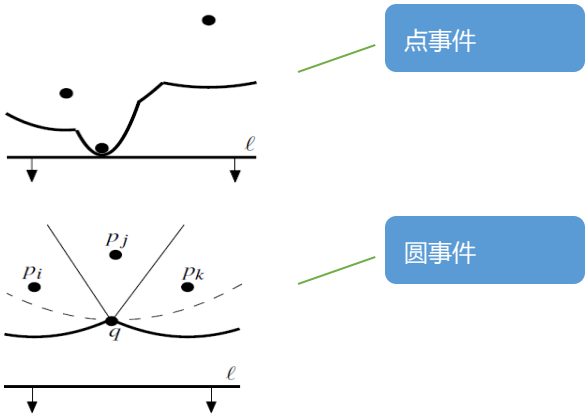
算法 3 Quick Hull

算法 QUICK_CONVEXHULL(P)
输入：三维空间中的 n 个点组成的一个集合$P = \{p_1, p_2, \ldots, p_n\}$
输出：集合P构成的三维凸包 $\mathbb{C}\mathcal{H}(P)$
初始化：计算初始的四面体：从 6 个 {最大|最小的 X|Y|Z} 的点中，选择 4 个不共面的点构成四面体；
建立单向可见图：剩余的每个点都被加到它的第一个可见面的链表中，并使该链表按距离排序。
将初始的四个面加入到栈S中。
迭代：
1. while S 非空
2. do (*将p_r 插入到 C 中*)
3. f ← S 栈顶
4. if g(f) 非空
5. 找到对 f 最远的点 p
6. 对于 p，找到所有 p 的可见的面集$F = \{f_1, f_2 \ldots\}$(*这些面肯定和 f 相邻，实现中只要以 f 为起点广度优先搜索*)
7. 沿着 F 边界构成一个地平线边集合，有序表 L
8. 由 L 构建新的面集合$F' = \{f_1, f_2 \ldots\}$，并将这些面加入到当前的凸包中
9. 更新F' 的可见点的链表：F' 的可见点的集合肯定是F 的子集，在实现中只需要遍历F中每个面的可见点集合，将每个点 p 插入到第一个可见面f'的可见点链表中（使链表有序）

```
10.          将F′之中那些可见点链表非空的面压入栈S 中
11.          在当前的凸包C 中删除F
17.   return(C)
```

2.3 二维 Voronoi 算法

根据 Voronoi 图和 Delaunay 三角剖分的关系，在已有 Delaunay 三角剖分时，将每两个相邻三角形的外接圆的圆心连接起来就能形成部分 Voronoi 图（得到完整 Voronoi 图还需要加入无穷远边）。不过我们所实现的是一个完整的 Voronoi图的 BeachLine 算法。在 Sweep Line 过程中，前锋面被当作几段相连的抛物面，两个抛物面相连的点称作断点，断点的轨迹就是 voronoi 图的边。在经过一个点事件会增加一个抛物面，经过一个圆事件会减少一个或多个抛物面，该算法主要是通过处理这两种事件，来得到最后的 Voronoi 图。



3

图 8 BeachLine 算法两类事件

算法 4 Voronoi Diagram 的 Beach line 算法

```
算法 VORONOIDIAGRAM(P)
输入：平面点集P = {p1,p2,…,pn}
输出：以双向链接边表 D 表示的（限制在一个足够大的包围框之内的）Voronoi 图 Vor(P)
初始化：事件队列 Q：将所有的基点事件插入其中
        状态结构 T：将其置空
        双向链接边表 D：将其置空
1.  while (Q 非空)
2.     do 将 y-坐标最大的事件从 Q 中取出
3.        if (这是一个发生于基点pi处的基点事件)
4.        then HANDLESITEEVENT(pi)
             (* 新加入一个抛物面，在 T 找到添加的位置 *)
             (* 检查当前点与其左右相邻的三个点能否引起圆事件，若能则将圆心作为圆事件加入 Q 中 *)
5.        else HANDLECIRCLEEVENT(γ)
             (* 这里的γ是 T 的一匹叶子，它对应于那段即将消失的弧 *)
             (* 从 T 中删除γ，并使得树平衡 *)
             (* 更新 voronoi 图的结构，增加一个顶点 *)
             (* 检查删除当前抛物面后是否会引起新的圆事件，若有则插入到 Q 中 *)
6.  (* 仍然存在于 T 中的那些内部节点，对应于 Voronoi 图的单向无穷边 *)
    计算出一个包围框，其尺寸之大，应足以容下 Voronoi 图中的所有顶点
```

通过对双向链接边表的适当调整，将这些单向无穷边都联接到这个包围框上

7. 遍历双向链接边表中的所有半边

增加相应的单元记录

设置好指向这些单元的指针，以及由这些单元发出的（指向对应各边的）指针

3.1 几何变换

Delaunay 三角剖分、Voronoi 图与凸包之间的联系通过提升变换得以实现。其将d维空间中的一系列点映射为$d+1$维空间中的对象（点或超平面）。

3.1.1提升变换

我们针对二维情况进行进一步的阐述。这种变换的基础是一个单位抛物面$\mathrm{z}=x^2+y^2$。其任意垂直截面都是一条抛物线，而任意水平截面都是一个圆。二维平面上的点提升到该抛物面上的像点为$(\mathrm{x},\mathrm{y},x^2+y^2)$。

给定二维平面上的点集$\boldsymbol{S}$，用$\boldsymbol{S_0}$表示$\boldsymbol{S}$提升到抛物面上的点集。考虑$\boldsymbol{S_0}$的下凸包，若将它投影回平面上，那么得到的将是$\boldsymbol{S}$的 Delaunay 三角剖分。特别地，对于$\boldsymbol{S}$中的元素(p,q,r)，用p_0,q_0,r_0表示他们在抛物面上的像点，那么，$p_0q_0r_0$构成$\boldsymbol{S_0}$的下凸包的一个面当且仅当pqr是$\boldsymbol{S}$的 Delaunay 剖分中的一个三角形。这个过程如下图所示。

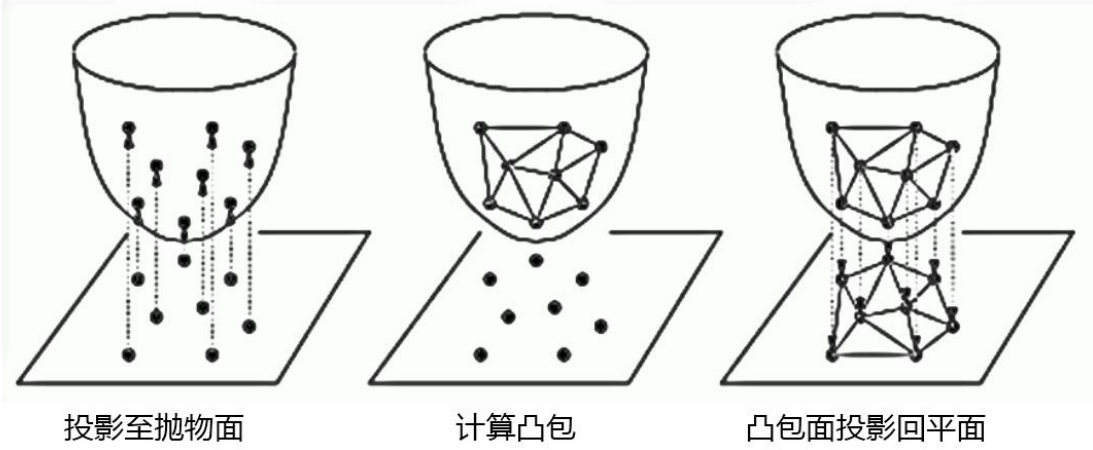


图 9

3.1.2Delaunay 三角剖分

为了证明这种对应的正确性，我们需要在 Delaunay 剖分的三角形与凸包的面之间建立联系。首先，回顾如下性质：

- 【Delaunay 剖分】$\boldsymbol{S}$中的三个点p,q,r构成 Delauany 剖分中的一个三角形当且仅当

这三个点的外接圆中不包含$\boldsymbol{S}$中其他的点。

- 【凸包】$\boldsymbol{S_0}$中的三个点p_0,q_0,r_0组成其凸包的一个面当且仅当经过$p_0q_0r_0$的平面使得$\boldsymbol{S_0}$中所有的点都在其的同一侧。

显然，我们需要在二维的空外接圆性质与三维中的空半空间性质之间建立联系。我们将证明如下结论：

引理 1：*考虑平面中的四个点$pqrs$，用p_0,q_0,r_0,s_0表示他们在单位抛物面上的像点。s落在pqr的外接圆内当且仅当s_0落在经过p_0,q_0,r_0的平面下方。*

为了证明这个引理，我们注意到，对于点(a,b)，在该点上方与抛物面相切的平面方程为

$$\mathrm{z} = 2\mathrm{ax} + 2\mathrm{by} - (a^2 + b^2)$$

现在，我们将该平面向上移动距离ρ^2，那么平面方程变为

$$\mathrm{z} = 2\mathrm{ax} + 2\mathrm{by} - (a^2 + b^2) + \rho^2$$

通过将其与抛物面方程$\mathrm{z} = x^2 + y^2$联立，解得交点满足的方程为

$$(\mathrm{x} - \mathrm{a})^2 + (y - b)^2 = \rho^2$$

即超平面与抛物面的交线（其实是一个椭圆）在二维平面上的投影为一个圆。进一步地，这个圆半径的平方就等于抛物面与该平面在点(a,b)处的竖直距离。

所以，任意平面下方的半空间与抛物面的交点，若投影到(x,y)平面上，都落在对应圆的内部。回到引理，当把点pqr映射到抛物面上，其对应点p_0,q_0,r_0构成一个平面。由于p_0,q_0,r_0落在该平面与抛物面的交线上，所以对应的圆就是经过pqr的外接圆。点s落在该外接圆内，当且仅当其在抛物面上的像点s_0落在平面下方的半空间内。

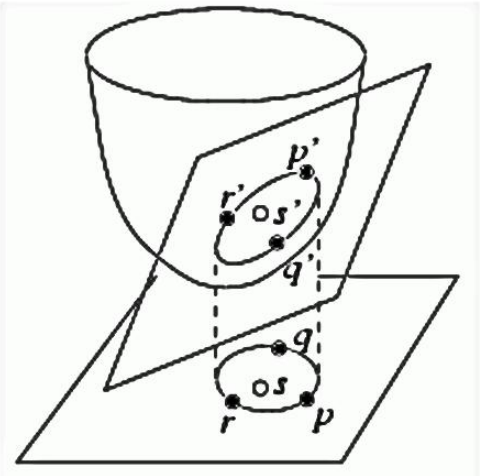


图 10

3.1.3Voronoi 图

给定点$p = (a,b)$，超平面$H(p)$表示在p的像点处与抛物面相切的平面，即

$$\mathrm{z} = 2\mathrm{ax} + 2\mathrm{by} - (a^2 + b^2)$$

考虑平面上任意点$q=(\alpha,\beta)$，其到抛物面的竖直距离为$\alpha^2+\beta^2$。而q到超平面$H(p)$的竖直距离为$2a\alpha+2b\beta-(a^2+b^2)$ 。

用$\Delta(p,q)$表示这两个距离之间的差，那么有

$$\Delta(p,q)=\alpha^2+\beta^2-2a\alpha-2b\beta+(a^2+b^2)=(a-\alpha)^2+(b-\beta)^2$$

即，$\Delta(p,q)$表示了p和q在二维平面上的距离的平方。

现在，考虑平面$z=0$上任意两点p_1和p_2，我们声明q距离p_1更近当且仅当在$q=(\alpha,\beta)$，平面$H(p_1)$在平面$H(p_2)$上方（距离抛物面更近）。由上述竖直距离的计算，该结论是显然的。因此，我们得到了如下引理：

引理 2： *设$p_1,p_2,\dots,p_n$是平面$z=0$上的一系列点。点q在 Voronoi 图中属于点p_i的区域当且仅当$H(p_i)$在q处是最高的平面。*

因此，$p_1,p_2,\dots,p_n$的 Voronoi 图即为$H(p_i)$下的半空间的逐点最大值，或说这些平面的上包络，在平面$z=0$上的投影。

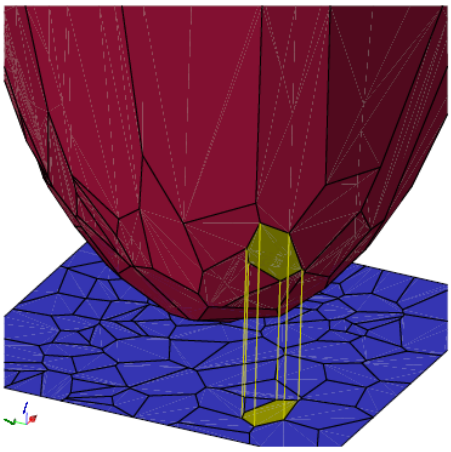


图 11

4 系统设计

程序基于 QT 实现，采用了标准的 MVC（Model-View-Controller）模型。下表针对该结构进行了具体的说明：

Model
根据不同的几何对象定义了不同的类，包括 Point2D、Edge2D、Face2D、Point3D、Edge3D、Face3D 等，分别负责自身的绘制和交互。 **对于算法计算，也根据需要提供了对应的类，如 DT、VD 类等，分别支持 Delaunay 算法和 Voronoi 算法。**
View
Geometrymagician 类实现了最高层控制，其下包含了 Q2DCanvas 和 Q3DCanvas 类，分别实现 2D 面板和 3D 面板的显示和交互。

Controller
根据不同的算法，分别实现了对应的类，包括：ConvexHullAlgo、QuickHullAlgo、DelaunayAlgo 和 VoronoiAlgo。每个类负责根据输入完成结果的计算，以及完成演示步骤的步进功能。 **每个算法根据 setPoint()接受输入，利用 calc()实现结果的计算，通过 nextStep()完成算法步进。通过 GetPoint(), GetDT(), GetCH()等函数，实现对结果的获取。**

其中，**Delaunay 算法**类的演示状态机如下所示：

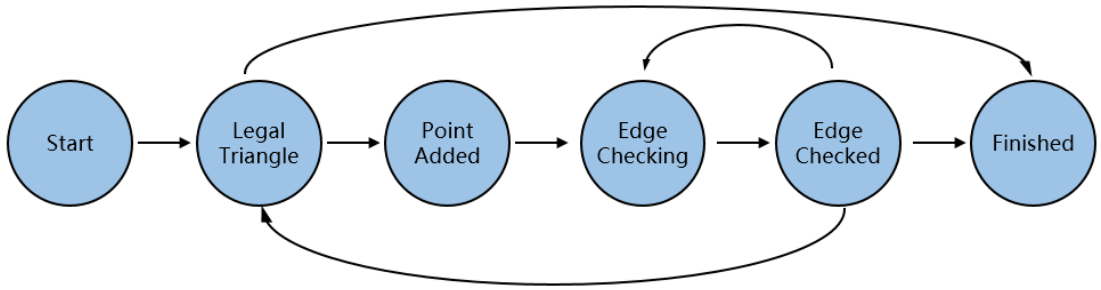


图 12

1、初始阶段，算法处于 Start 状态。

2、在 Start 状态，首先加入 3 个点，构成初始三角形，进入 Legal Triangle 状态。

3、在 Legal Triangle 状态，若已加入所有的点，则进入 Finished 状态，否则加入下一个点，进入 Point Added 状态。

4、在 Point Added 状态，找到新加入的点所在的三角形，将其与对应顶点连接，进入 Edge Checking 状态。

5、在 Edge Checking 状态，找到一条当前需要检查的边，进入 Edge Checked 状态。

6、在 Edge Checked 状态，根据当前边的状态进行操作，若合法，则继续，否则翻转之。若还有待检查的边，则返回 Edge Checking 状态，否则若所有点都已经加入，进入 Finished 状态，若还有点未加入，则进入 Legal Triangle 状态。

7、在 Finished 状态，算法已经结束，状态不更新。

其中，**Convex Hull (RIC)算法**的演示状态机如下所示：

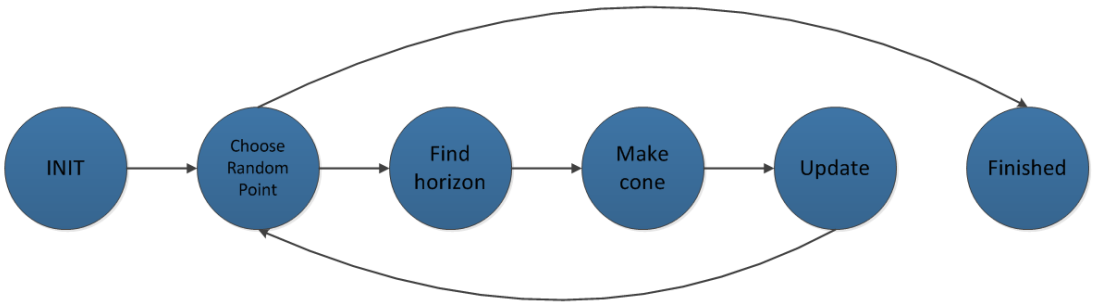


图 13

1、初始阶段，算法由 Start 状态进入 INIT。

2、在 INIT 状态，首先选取不共面的 4 个点，构成初始一个四面体，进入 Choose Random Point 状态。

3、在 Choose Random Point 状态，若遍历完所有的点，则进入 Finished 状态，否则选取

一个点，进入 Find Horizon 状态。

4、在 Find Horizon 状态，找到当前的点的可见面以及其边界构成的地平线。进入下一状态 Make Cone。

5、在 Make Cone 状态，由当前点和地平线构建新面，并进入 Update 状态。

6、在 Update 状态，更新算法维护的可见图（冲突图），并删除旧面，以及与其对应的可见图中的边。然后重新返回 Choose Random Point。

7、在 Finished 状态，算法已经结束，状态不更新。

其中，**Convex Hull (Quick Hull)算法**的演示状态机如下所示：

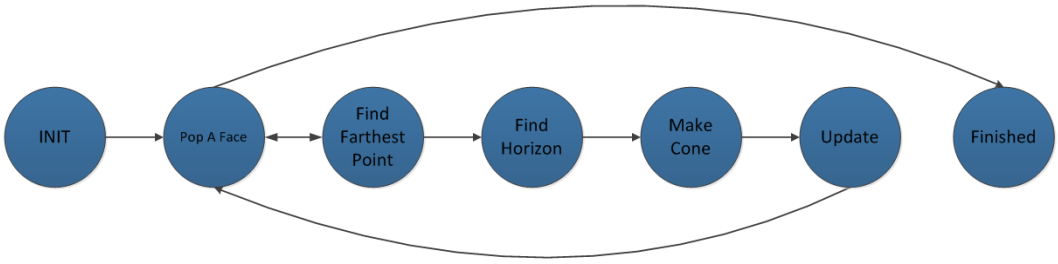


图 14

我们在上面已经介绍了 Quick Hull 的算法，与 RIC 算法不同在于 Quick Hull 每次从一个栈中取出一个面，寻找改面的最远点。其余部分就和 RIC 算法类似，故不做赘述。

5 实验结果

5.1 功能演示

程序的界面如下所示：

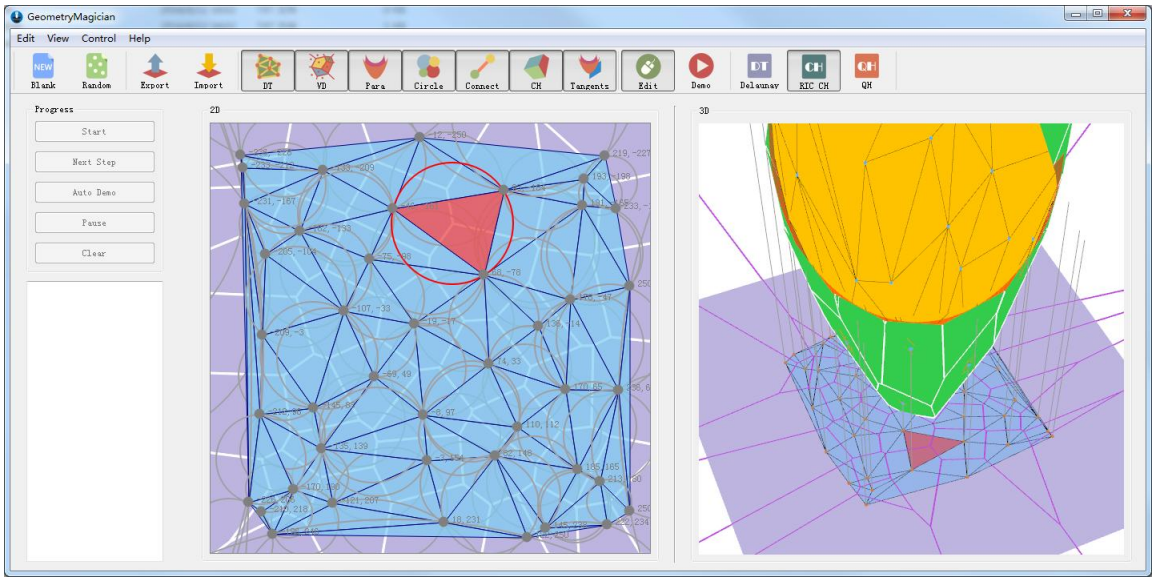


图 15

其中，顶部包含菜单栏、工具栏。左侧为进度控制与信息栏，两个主要面板分别显示 2D 与 3D 的信息，并支持交互。

在 2D 面板中，用户可以通过鼠标右键增加一个点，也可以通过鼠标左键选择点、Delaunay 三角形或 Voronoi 面，3D 面板会实时同步显示。在 3D 面板中，用户可以通过鼠标左键拖动调整视角，鼠标右键拖动旋转视角。

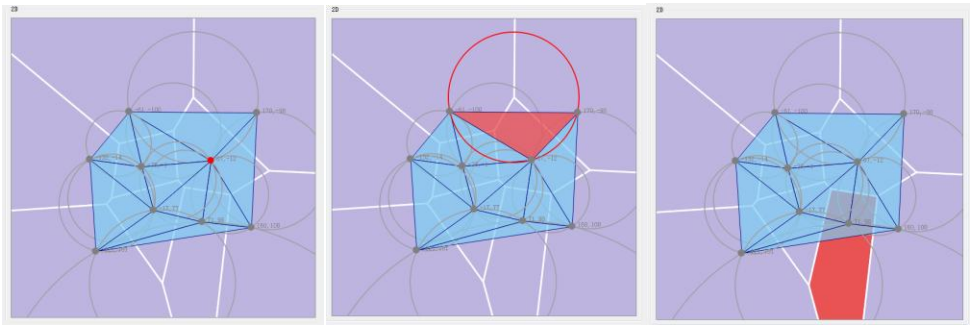


图 16

程序支持两种模式：**编辑模式**与**演示模式**，可以通过按钮进行切换。

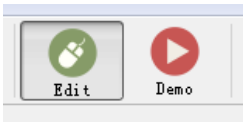


图 17

其中，编辑模式中支持点集的输入和即时修改，以及算法选择，程序会实时计算出当前所有结果（Delaunay 三角剖分，Voronoi 图、三维凸包等），并动态更新界面。编辑模式下还可以自由选取目标对象（点、Delaunay 三角形、Voronoi 面等）以高亮，以便观察三维映射关系。

演示模式支持点集的输入和算法选择，之后可以通过进度控制面板控制演示的流程。演示模式下不支持点集的动态编辑。操作流程为：增加点，点击 Start 确认输入，之后可以通过 Next Step 或者 Auto Demo 实现单步执行或自动执行。在自动执行时，可以点击 pause 暂停。也可以随时通过 Clear 重置，以重新输入点。在算法流程中，会高亮当前的关键目标，信息栏也会实时输出对应的结果。

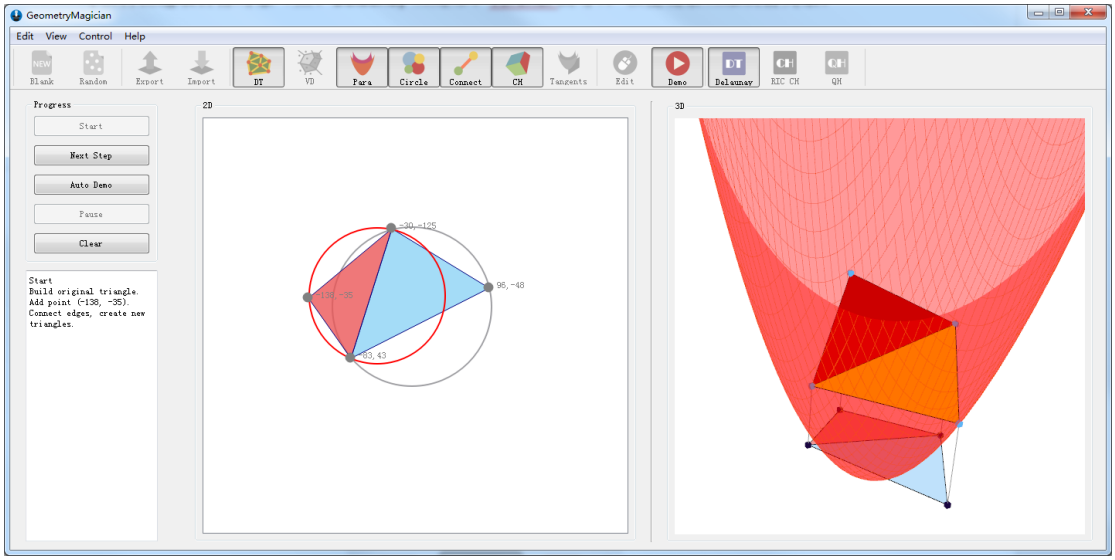


图 18

程序一共实现了四种算法，其中利用 BeachLine 实现的 Voronoi 算法为默认开启，其用于计算 Voronoi 剖分。此外，用户可以在随机增量 Delaunay 剖分算法、随机增量三维凸包算

法、Quick Hull 三维凸包算法之间切换。

图 19

若选择 Delaunay 算法，则程序会根据二维点集计算对应的 Delaunay 三角剖分，并投影到三维空间中，绘制凸包的下包络。若选择 RIC CH 或 QH 算法，则程序会根据三维点集计算对应的三维凸包，并投影到二维平面中，绘制 Delaunay 剖分。

用户可以随意切换对象的显示与隐藏，分别为：Delaunay 剖分三角形、Voronoi 面、3D 抛物面、三角形外接圆、原点-提升点连接线、3D 凸包、抛物面切平面。其中，只有在开启 DT 的情况下才可以开启 CH，只有在开启 VD 的情况下才可以开启 Tangents。

图 20

此外，可以通过“Blank”新建空白场景，“Random”新建随机场景（随机产生 n 个点）。也可以通过导入和导出将当前场景保存和读取。

Help 中的“帮助”和“关于”提供了必要的信息。

图 21

5.2 性能测试

实验中，我们分别针对所用到的三个算法（Convex Hull 3D、Delaunay Triangulation、Voronoi Diagram）的核心部分对不同规模的数据做多次的运算，最后得出下表。

表格 1

点数	CH 运行时间(ms)	DT 运行时间(ms)	VD 运行时间(ms)
1000	274.4	12.0	8.2
2000	571.6	24.8	15.0
3000	1015.0	45.4	24.0
4000	1652.8	56.4	27.2
5000	1975.4	70.0	44.0
6000	2431.4	87.0	50.6
7000	2821.0	129.6	72.8
8000	2995.4	143.6	72.2

9000	3210.6	140.8	95.4
1000	3734.0	172.2	112.2

表格 1 中，对不同规模的数据分别给出了三种算法运行时间。其中与其余两者不同的是 Convex Hull 算法计算的是三维空间中凸包。测试数据都是外部随机生成（均匀分布）的不同点。将上表内容转化成下图：

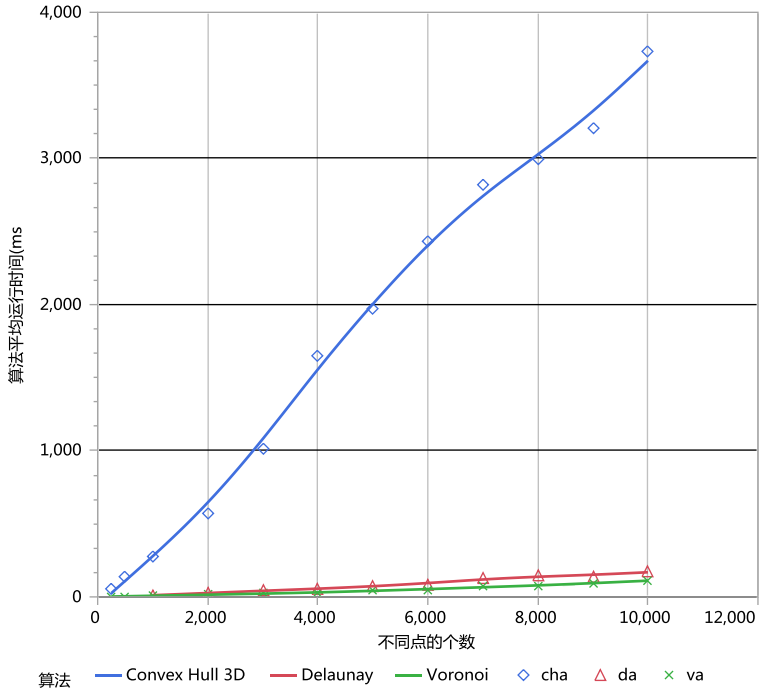


图 22

其中个值得说明的是，我们程序中在计算 Voronoi 面的时候是由所得到的边遍历得到的 Voronoi 图的面，由于没有足够时间进行优化，这一步操作所需要的时间复杂度为$O(n^2)$。另外一点需要说明的是为了能够方便地将三种算法放在同一张图表内显示，我们将数据规模限制为 10,000 以下。实际中我们所实现的三维凸包算法和 Voronoi 图算法的计算规模分别能达到十万和百万级别。如下图所示。

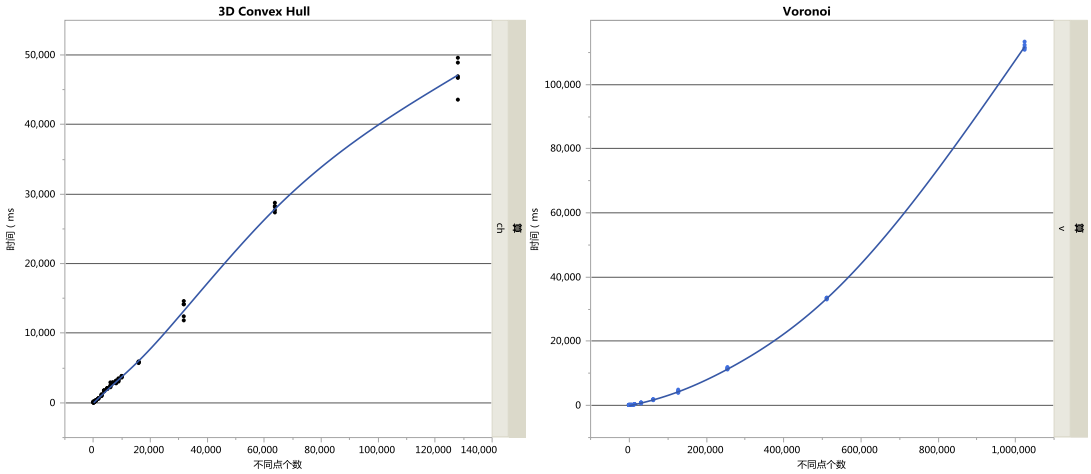


图 23

6 总结

6.1 问题总结

1)为了设计符合 MVC 模型的框架，我们在程序类的设计上耗费了不少功夫。首先，前台对应每一种对象有其对应的显示类，而后台也有着其对应的计算类，这些类之间的转换给程序的流畅运行带来了挑战。最终，我们通过设计“信使类”，其进行最高层次的抽象来表达每一种对象，而计算类和显示类都基于此进行扩展，从而在一定程度上缓解了这个问题。另一方面，由于 2D 和 3D 区域的显示和数据要保持高度的同步，在数据的计算和传递方面，也为我们带来了难题，最终，我们采用了“一份数据，多份显示”的方案，通过妥善维持一份数据副本，各个显示端和计算端分别读取的方式，解决了同步的问题。

2)在计算过程中，进行 toLeft 测试、inCircle 测试等时，会遇到浮点数运算的误差问题。特别是当点落在边界上时，误差的出现可能会带来致命的错误。为了解决这个问题，我们进行了两方面的改进：1、由于输入面板较大，可以用坐标来表达点的位置，因而在程序中出现的实际点的坐标都为整数，从而避免了这个问题。2、对于潜在的浮点数运算情况，我们采用小阈值（1e-6）的方法，将近似于 0 的结果直接归约为 0，从而在一定程度上缓解了这个问题。

3)在显示端，由于我们同时采用了 QT 和 OpenGL，其带来的不一致也对我们程序的框架带来了影响，导致结构不统一，并进一步影响到了接口的设计。但我们仍然通过“信使类”的引入，通过中间层起到了中转的作用，最终解决了这一问题。

6.2 实验收获

本次大实验，我们涉及的内容涵盖了计算几何的几个核心领域：Delaunay 剖分、Voronoi 图和几何变换。整体而言，既包含算法的设计，又包含理论的分析；既包括效率的追求，又包括视觉效果的展示，是一份相当有分量和有挑战性的工作。

在工作之初，由于目标相对比较宏大，计划的任务也很多，我们在设计和讨论上花费了大量的功夫。为了尽最大可能减少后期重构的成本和设计缺陷带来的额外工作，前期的充分准备是必要的，这也大大增强了我们的架构能力和整体意识，对我们软件工程能力的提升也是十分显著的。

在实现阶段，大量的工作量和设计量着实带来了很大的考验，特别是需要结合 C++、Qt、OpenGL，而且我们没有充足的使用经验，所以在编码时遇到了很大的挑战。但这也对我们的分工协作能力和学习能力起到了很大的锻炼作用，我们将工作分解为：架构、前端、后端、交互几大方面，解耦后分别由不同组员负责，从而实现了各自的高效率工作，也充分发挥了各人的长处。

最后，在大实验的后期，如何设计一种好的展示方式和视觉效果成为最重要的问题。得益于长久以来受到的人机交互和美术等专业的训练，我们充分发挥优势，将众多的功能以科学的方式组合和展现出来，最终实现了较好的交互和视觉效果，完成了一份满意的大作业。

7 参考文献

[1] Okabe, A., Boots, B., Sugihara, K., & Chiu, S. N. (2009). Spatial tessellations: concepts and applications of Voronoi diagrams (Vol. 501). John Wiley & Sons.

[2] Lawson, C. L. (1972). Transforming triangulations. Discrete mathematics,3(4), 365-372.

[3] Barnhill, R. E. (1977). Representation and approximation of surfaces.Mathematical software, 3, 69-120.

[4] Sibson, R. (1978). Locally equiangular triangulations. The computer journal,21(3), 243-245.

[5] Chand, D. R., & Kapur, S. S. (1970). An algorithm for convex polytopes.Journal of the ACM (JACM), 17(1), 78-86.

[6] Clarkson, K. L., & Shor, P. W. (1989). Applications of random sampling in computational geometry, II. *Discrete & Computational Geometry*, *4*(1), 387-421.

[7] Guibas, L. J., Knuth, D. E., & Sharir, M. (1992). Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, *7*(1-6), 381-413.

[8] Barber C B, Dobkin D P, Huhdanpaa H. The quickhull algorithm for convex hulls[J]. ACM Transactions on Mathematical Software (TOMS), 1996, 22(4): 469-483.

[9] Dirk Gregorius. Implementing Quickhull.

[10] Timothy M. Chan. A minimalist's implementation of the 3-d divide-and-conquer convex hull algorithm.