Delaunay 三角网格中节点删除算法的比较

Comparisons of algorithms for removing vertexes on Delaunay Triangulation

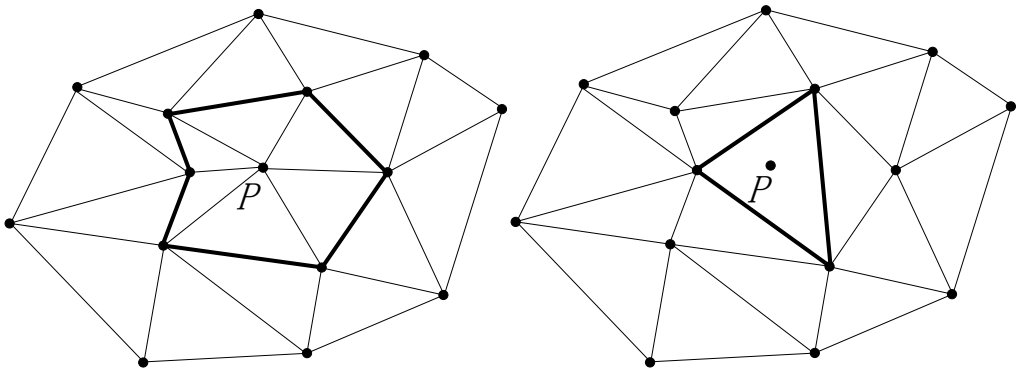
(计算几何课程实验报告)

小组成员：宋海川徐鑫陆洋

目录

1 实验背景

Delaunay 三角化是计算几何领域中最重要的结构之一。因此，已经有非常多的文章研究了如何构造 Delaunay 三角化。而在本次实验中，我们关注的是如何在二维的 Delaunay 三角化中删除一个或者多个点的问题。如图 1.1 所示，从原始 Delaunay 三角网格中删除一个节点 P 之后，能够对原始三角化进行处理，使得对余下的点集仍然满足 Delaunay 性质。



（a） （b）

图 1.1 在 Delaunay 三角化中删除一个节点：（a）原始三角化；（b）删除点 P 之后的三角化

到目前为止，已经有很多篇文章针对该问题进行了研究。在这些文章中所提出算法的复杂度一般都与待删除节点的度数 d 有关。只有很少的方法达到了线性的时间复杂度 $O(d)$，其中最具代表性的是 Aggarwal 等人[1]和 Chew[3]。[1]提出了一种在线性时间内计算二维 Voronoi 图的方法，该方法也可以在线性时间内删除 Voronoi 图中的一个节点。由于二维点集的 Voronoi 图与 Delaunay 三角化为对偶关系，从 Voronoi 图得到对应的 Delaunay 三角化需要线性的时间 $O(n)$。因此，[1]也间接的提出了一种在 Delaunay 三角化中删除一个点的线性算法。然而，虽然其复杂度较低，但是该方法的结论非常复杂并且难以实现[5]。随后，Chew[3]提出了一种具有随机化复杂度的方法，该方法的结论更加简单[5]。然而，该方法仍然非常复杂，并且在处理度数较低的节点的时候（对于二维的情况，在不限定点集分布的情况下，d 的期望值为 6[4]，因此对于低度数节点的处理非常重要），该方法的效率并不高[5]。因此后人的研究主要集中在如何降低算法的复杂程度以及对低度数节点的处理上。

从 Delaunay 三角化中删除一个节点后，与该节点相关的节点集会产生一个多边形空洞，如图 1.1(a)中粗线框构成的多边形。而空洞内部的三角化不会影响其外部的 Delaunay 性质[5]，因此可以只处理该空洞来达到删除节点后调整的效果。Heller[6]观察到了这一点，并将多边形分成多个 Ear。通过判断每个 Ear 是否是 Delaunay Ear 来重新对该多边形进行 Delaunay 三角化。该算法的复杂度为 $O(d\log d)$。然而，[6]中判断一个 Ear 为 Delaunay Ear 的准则是错误的[4]。因此，Deviller[4]通过结合 Shelling 和对偶的思想，提出了一种新的判断准则，并提出了一种复杂度为 $O(d\log d)$ 的方法。后来 Deviller[5]提出，在程序实现的时候，对于度数较低的节点可以进行特殊处理，已提高运行效率。他通过引入决策树，对度数为 3~7 的节点删除进行了处理，减少了程序访问堆栈的次数，从而达到了提高运行效率的目的。

在本次实验中，我们将会对该问题进行研究。通过比较几种不同的算法，来分析它们各自的优缺点以及适用范围。

2 实验内容

在本次实验中，我们分别实现了 Ear Queue 方法[4]、Triangulate and Sew 方法和基于决策树的低度数节点删除方法[5]。另外，实现 DT 分割的算法是基于边翻转的递增算法。

下面简要介绍一下各个算法的原理与特点。

2.1DT 分割算法

本次实验中，DT 分割是所有后续算法的基础。我们采用的算法是基于边翻转的随机递增算法。其实现框架如下。

1. 寻找近似无穷远的三个点，构造初始网格。具体方法是找到所有点集的矩形包围盒，将包围盒放大 100 倍，然后构造三角形，使之覆盖包围盒，并将其作为初始网格。
2. 依次将每个点加入到网格中。当前网格中的每个 FaceHandle，都对应一个桶（实现方式是 std::vector），存储在当前面内的所有尚未处理的点。程序同时维护一个数组 FaceHandle[]，存储对应序号的点当前所在的面片。这样，对每个面片，可快速找到位于其内的所有点；对每个点，也可找到其所属的面片。对于新加入的点，将原面片一分为三，然后进行边翻转调整，直到满足 DT 分割条件为止。桶结构和数组也进行同步更新。
3. 删除添加的三个无穷远点，完成 DT 三角分割。

输入点集数为n，则实现算法的期望时间复杂度是$\Theta(n\ln n)$。

2.2Ear Queue 方法

首先引入多边形 Ear 的概念：

定义 1：*在多边形H 中，对于顺序连接的三个顶点$q_iq_{i+1}q_{i+2}$，如果线段q_iq_{i+2}落在H 内部，并且不与H 的其他边界相交，那么就称$q_iq_{i+1}q_{i+2}$ 为H 的一个 Ear。*

正如上一章中介绍的，从 Delaunay 三角化中删除一个节点后，与该节点相关的节点集会产生一个多边形空洞。[4]使用了多边形 Ear 的概念，并将空洞多边形分成若干个按照逆时针排列的 Ear。

定理 1：*假设多边形$H=\{q_0,q_1,...,q_{d-1},q_d=q_0\}$ 以及点p，使得线段q_iq_{i+1}属于点集$\{q_0,q_1,...,q_{d-1},p\}$ 的 Delaunay 三角化。如果$\left|power\left(p,circle\left(q_i,q_{i+1},q_{i+2}\right)\right)\right|$ 是最小的，那么线段q_iq_{i+2} 是点集$\{q_0,q_1,...,q_{d-1}\}$ Delaunay 三角化中的一条边。*

其中，$power\left(p,circle\left(q_0,q_1,q_2\right)\right)$可以在线性时间内使用如下的公式计算：

$$power\left(p, circle\left(q0, q1, q2\right)\right) = \frac{\begin{vmatrix} x_{q0} & x_{q1} & x_{q2} & x_p \\ y_{q0} & y_{q1} & y_{q2} & y_p \\ x_{q0}^2 + y_{q0}^2 & x_{q1}^2 + y_{q1}^2 & x_{q2}^2 + y_{q2}^2 & x_p^2 + y_p^2 \\ 1 & 1 & 1 & 1 \end{vmatrix}}{\begin{vmatrix} x_{q0} & x_{q1} & x_{q2} \\ y_{q0} & y_{q1} & y_{q2} \\ 1 & 1 & 1 \end{vmatrix}} \quad (1)$$

根据**定理 1**，计算删除一个节点，只需要不断计算各个 Ear 的 $power()$ 函数，取出其中函数值最小的 Ear，并将它对应的边加入到原始三角化中。

[4]中该算法的伪代码如下：

Algorithm *Delete(DT(S),p)*

1. Let $q_0, q_1, ..., q_{d-1}$ be the vertices incident to p in *DT(S)*
   In CCW order around p;
2. Let Q be a priority queue;
3. **for** i =0 **to** $d-1$
4. **do** $ear \leftarrow q_i q_{i+1} q_{i+2}$;
5. **if** $counterclockwise\left(q_i q_{i+1} q_{i+2}\right)$
6. **then** $p \leftarrow \infty$;
7. **else** $p \leftarrow -power\left(p, ear\right)$;
8. $Q.insert\left(p, ear\right)$;
9. **while** $Q.size() > 3$
10. **do** $ear \leftarrow Q.minimum()$;
11. create triangle ear and link it to its two existing neighbors;
12. $ear0 \leftarrow ear.previous$;
13. $ear1 \leftarrow ear.next$
14. $ear0.vertex(2) \leftarrow ear.vertex(2)$; $ear0.next \leftarrow ear1$;
15. $ear1.vertex(0) \leftarrow ear.vertex(0)$; $ear1.previous \leftarrow ear2$;
16. $Q.delete\left(ear\right)$;

17. $Q.modify-priority(ear0);\quad Q.modify-priority(ear1);$

18. $ear \leftarrow Q.minimum();$

19. create triangle ear and link it to its three existing neighbors;

在实现过程中，对于删除节点后产生的多边形空洞（顶点按照删除节点的逆时针排列），我们对该多边形中的每个内角进行判断。如果一个内角中三个顶点构成一个逆时针旋转，则说明该角构成一个 Ear，这时使用式（1）计算其权重，并且保存在一个优先队列中；否则将该角的权重赋为无穷大，并同样保存在优先队列中。

然后，每一次都从该优先队列中取出权重最小的角，将其构成三角形并加入到原始结构中。同时，需要更新与该角相邻（在多边形空洞边界上相邻）的两个角对应的权重。为了能够在常数时间内找到相邻的两个角，我们同时还维护了一个双向链表，链表的正向按照逆时针存储各个角。

在实现过程中，我们发现该段代码存在着如下两处错误：

1、代码第 5 行中，判断三个顺序连接的点为非 Ear 应该使用 $clockwise(q_iq_{i+1}q_{i+2})$，而代码中使用的恰恰相反。

2、算法第 7 行中，不需要将 $power(p,ear)$ 的函数值取相反数。[4]原文中强调，因为当 p 在 ear 确定的圆内部时，$power(p,ear)$ 的函数值为负数，因此取相反数。但是我们在实验中发现，这样会造成各个 Ear 的排序出错，因此不需要取相反数。

由于该算法在 CGAL[2]中进行过实现，为了验证该算法的正确性，我们还特意对 CGAL 中的该功能进行了测试，并没有发现对应的问题。这说明，上面两处错误为作者在论文中的笔误，特此更正。

对于该算法的复杂度分析如下：

设 d 为所删除节点的度数，即与它有 Delaunay 边相连的节点个数。循环总共执行 $O(d)$ 次，在循环中 $power(p,ear)$ 和 $clockwise(q_iq_{i+1}q_{i+2})$ 可以在 $O(1)$ 时间内完成，因此算法的整体复杂度主要取决于对于优先队列的操作。我们使用了 C++标准模板库中的优先队列 PriorityQueue，并且制定其使用堆实现，因此该优先队列的所有操作均可以在 $O(\log d)$ 时间内完成。因此综体算法的复杂度为 $O(d\log d)$，其中 d 为所删除节点的度数。

另外，由于**定理 1** 只有在多边形空洞封闭时候（$H=\{q_0,q_1,...,q_{d-1},q_d=q_0\}$）才成立。而只有当待删除节点为原始 Delaunay 三角化中的内部节点时，**定理 1** 才成立，从而才能使用。如图 2.1 所示，如果要删除边界点 ***P***，那么形成的多边形空洞图中 ***AB*** 这线段右侧的空间，并不构成一个封闭的区域，不满足**定理 1** 成立的条件。如果要强行构成一个封闭区域，即将 ***AB*** 使用直线段连接起来，那么就有可能形成如图 2.1 中所示的情况，即线段 ***AB*** 与其他的边界有交点，这时如果使用 Ear Queue 方法计算，则会将线段 ***AB*** 加入到原始三角化中，导致错误。

因此，使用 Ear Queue 方法，只能够删除内部节点，对于边界点并不能够保证其正确性，这是由算法本身导致的。如果要删除边界点，则需要在删除之后对整个点集重新计算一遍 Delaunay 三角化。

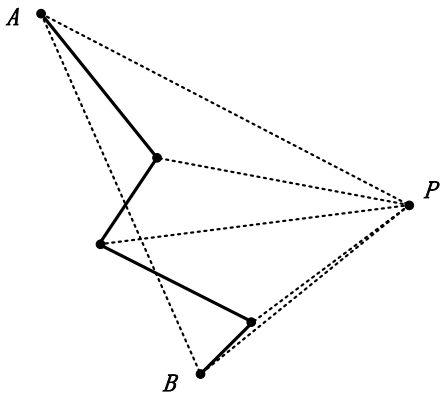
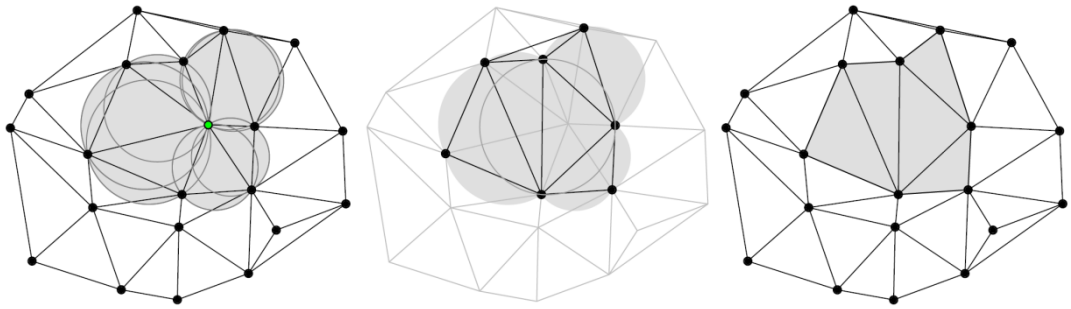


图 2.1 删除边界点的情况

2.3Triangulate and Sew 方法

另一种常用的$O(d\log d)$方法是 Triangulate and Sew 方法[5]。该方法的主要原理如下：



（a） （b）（c）

图 2.2 [5]（a）DT_{big}与待删除节点的“花”；（b）DT_{small}与它的外接圆；（c）最终的三角化

假设DT_{big}表示原始 Delaunay 三角化，DT_{small}表示与待删除节点相关节点集的 Delaunay 三角化。那么在空洞的边缘便同时有两种三角化存在。

引入节点“花”的概念。在 Delaunay 三角化中，所谓一个节点的“花”，指的是在DT_{big}中与该节点关联的三角形的外接圆的并集。如图 2.2 所示，在DT_{small}在空洞内部三角形的外接圆一定落在删除节点的“花”的内部，因此不会包含任何的剩余节点，即不会影响到DT_{big}在空洞外部的 Delaunay 性质。

因此，我们需要做的只是排除掉 DT_{small} 中那些落在空洞外部的三角形，并且将剩余的三角形加入到 DT_{big}，这样便可以完成剩余节点的 Delaunay 三角化。

该算法的执行分为两步：首先，要计算得到 DT_{small}，例如可以采用随机化算法，那么在 $O(d\log d)$ 的期望时间里便可以完成；其次，需要将 DT_{small} 缝合到 DT_{big} 中，这可以在 $O\left(d^2\right)$ 的时间内完成。因此，总的复杂度为 $O\left(d^2\right)$。

虽然该算法的复杂度为 $O\left(d^2\right)$，但是由于其原理比较简单，并且实现方便。同时，与 Ear Queue 方法类似，Triangulate and Sew 方法可以处理边界点删除的情况，因此在实际应用中也经常被采用。在本次试验中，我们将该算法作为众多 $O\left(d^2\right)$ 算法（flipping，edge completion 等等）的代表进行实现。

2.4 基于决策树的低度数顶点删除方法

现在还未找到时间复杂度为O(d)的节点删除方法。但是，我们并不需要对算法复杂度太过关注，大多数的顶点度数都是很小的。因为对一个二维网格，其顶点的平均度数的最大不超过 6[4]。所以，很大一部分的顶点均是低度数节点。因而，对度数较低的顶点做针对的算法处理，可以提高整个程序的运行效率。我们实现的算法参考文献[5]，对度数 3-7 的顶点删除进行了处理。

1. 度数为 3 的顶点直接删除即可，然后如果出现空洞，用三角形面片缝补；
2. 度数为 4 的顶点处理方法仍比较简单，只要进行一次 inCircle 判断即可；
3. 度数为 5 的顶点处理仍然简单、直观。最多进行 3 次 inCircle 判断，就可以确定最后的结果。其对应的决策树如图 2.3 所示。

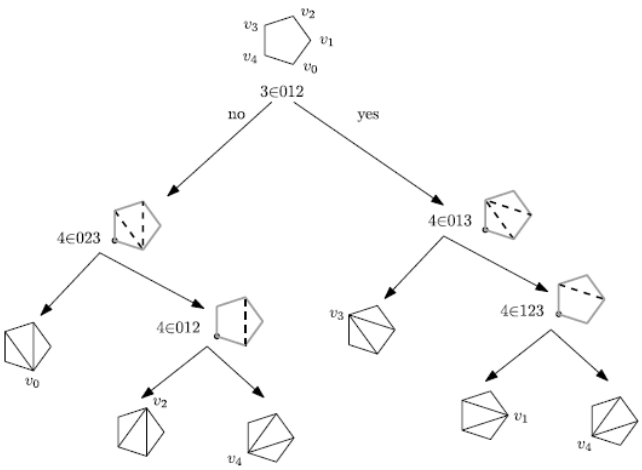


图 2.3 degree=5 的决策树

4. 度数为 6 的顶点处理开始变得复杂，很难直观地画出决策树。这时决策树的生成

可以借助分治算法（见文献[7]）。目前处理 Delaunay Triangulation 的最快算法就是分治算法，其基本思想对两部分子集分别完成 DT 分割，然后通过一系列 inCircle 检测，自下而上逐步将两个子集拼接起来。决策树生成方法如下，先把 6 个顶点分为两个三角形，然后利用分治算法中的合并思想，每进行一次 inCircle 检测，决策树对应建立一个中间节点。最终结果存储在叶子节点中。其最终生成决策树如图 2.4 所示。决策树深度为 7，即我们最多需要做 6 次 inCircle 检测。

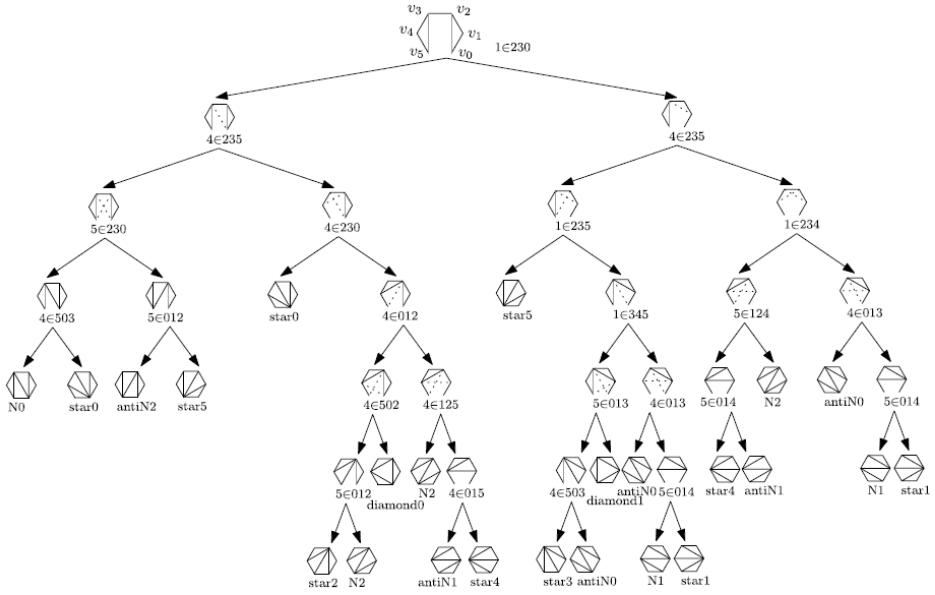


图 2.4 degree=6 的决策树

5. 度数为 7 的顶点处理方法类似，仍然是通过分治算法的思想建立决策树，将 7 个顶点分为一个三角形和一个四边形，然后进行合并操作。其决策树的深度为 11，其结构较复杂，这里不再列出。

理论上任何度数的顶点删除操作都可以通过构造决策树的方法完成。决策树方法的优点是占用的时间和空间资源都达到了最小。时间方面，该算法只需要进行最小次数的 inCircle 判断，就可以输出最终结果，没有任何冗余操作；空间方面，除了存储输入和输出数据，没有其他额外空间需求。但是，要实现决策树方法的需要人工代码量很大，而且随着度数的上升，决策树的规模呈指数上升，其实现也更加繁琐。因而我们只实现了度数最高为 7 的决策树算法。

2.5 特殊情况----共线的处理

这次实验中，我们需要处理的特殊情况共有三种，共点，共线，共圆。共点的情况只需要去除重复点即可。出现共圆情况时，我们将 inCircle 函数的返回值设为 false，即默认共圆是符合 DT 分割条件的，DT 分割可能不唯一，但不影响正确性。共线的情况相对复杂，我们在下面讨论。

2.5.1 DT 分割的共线问题

我们使用的算法是基于边翻转的随机递增算法。若存在三点共线，则在插入某点 P 时，会出现该点落在某两个三角形 ABC、ABD 共有的边 ABP 上。处理方法是，将该点看做落在

其中某一三角形 ABC 内，并将其一分为三：ABP、BCP、CAP。显然，出现了退化三角形 BPA。但是随后 inCircle 检测中，一定会检测到点 D 在 BPA 的外接圆内。回顾 inCricle 检测的实现：

$$\det = \begin{vmatrix} a.x & a.y & a.x^2 + a.y^2 & 1 \\ b.x & b.y & b.x^2 + b.y^2 & 1 \\ c.x & c.y & c.x^2 + c.y^2 & 1 \\ p.x & p.y & p.x^2 + p.y^2 & 1 \end{vmatrix} > 0.$$

上式 a,b,c 逆时针排列。当出现退化极限情况，即a, b, c共线形成向量$\overrightarrow{abc}$时，上式则等价于

$$\begin{cases} det = 0, & \text{p 和}\overrightarrow{abc}\text{共线} \\ det > 0, & \text{p 在}\overrightarrow{abc}\text{左侧} \\ det < 0, & \text{p 在}\overrightarrow{abc}\text{右侧} \end{cases}.$$

D 在$\overrightarrow{BPA}$左侧，因而$\det > 0 \Leftrightarrow inCircle = true$，一定会将边 AB 翻转到 PD。最终结果如下图所示，共线情况得到了正确的处理。特别的，当所有点共线时，生成的是一条线段，没有实际意义，因而对这种情况我们没有处理，结果仍然为孤立点。

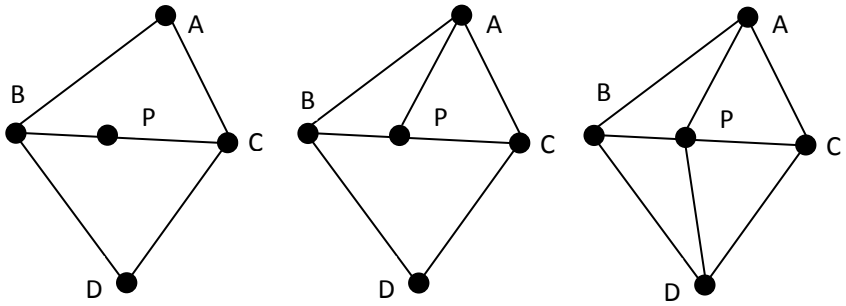


图 2.3.2 DT 分割共线情况处理

2.5.2 决策树方法中的共线问题

决策树方法的本质就是进行若干次 inCircle 检测，其共线问题的情况类似 DT 分割。若删除顶点出现的多边形空腔中存在共线的连续三点，也不需要担心算法结果中这三点会产生退化的三角形，因为在 inCircle 检测中一定会存在空腔多边形的点出现在这三点形成有向线段的左侧，导致 inCircle = true。

2.5.3Ear Queue 方法实现对于共线问题的处理

由于在删除节点之后，要判断多边形空洞中每个角是否为 Ear（即构成一个逆时针旋转），如果为 Ear 则需要使用式（1）计算其对应的权重。在判断一个角是否为 Ear 的时候，如果三个点共线，那么会导致使用式（1）计算权重时分母为零，导致程序崩溃。因此，可以首先判断是否三点共线，如果共线，那么认为这种情况不构成一个 Ear，即不需要计算其权重，便可以避免这种情况导致的错误。

另外，在程序结束之前，需要将优先队列中最后的一个三角形插入到 Delaunay 中，这时需要判断此三角形是否为退化三角形，即三个顶点共线，如果为退化情况，则不需要将该三角形插入到 Delaunay 中。

2.5.4 Triangulate and Sew 方法实现对于共线问题的处理

在删除一个节点之后，首先要对产生的多边形空洞调用 DT 三角化函数，根据 **2.5.1** 节中的介绍，本文中实现的 DT 三角化可以处理共线的情况。

另外，在 DT 三角化结束之后，需要将三角化结果缝合到原始结构中，对于落在多边形空洞内部的三角形予以保留，对于落在其边界（退化三角形）以及外部的三角形不进行处理。这样便可以对该退化情况进行处理。

3 实验过程

本次实验的环境如下：

操作系统：windows 7，32 位

硬件：内存 2G，处理器 Intel(R) Core(TM) i5 CPU M460 @2.53GHz 2.53GHz

编程语言/环境：C++ / VS 2010

相关库：Qt 4.7.4，OpenGL 2.0，OpenMesh 2.0.1

3.1 程序结构

图 3.1 为是程序实现中的类图，其结构是基于标准的 MVC（Model-View-Controller）模型，下表针对该结构进行了具体说明：

Model
类 Model 记录了主要的数据：点（points），三角网格（myMesh）； 其中三角网格（myMesh），使用了 OpenMesh 中的半边结构。
View
UI 模块中，类 MyGLWidget, VertextRemoval 分别继承了 Qt 中两个类，实现了用户界面以及交互事件；其中 MyGLWidget 继承了 QGLWidget，能够使用 OpenGL 画图；用于画图的数据来自类 Model。
Controller
Control 模块中的五个类都是对类 Model 中的 myMesh 进行操作： a. 类 DtProcess 中的方法 DelaunayTri 将 Model 中的点（points）变成 myMesh； b. 类 VrProcess, VrUsingEars, VrUsingSew 都有方法 vertexRemove，作用是删除 myMesh 中的一个点，并使之保持 Delaunay 三角化；其中 VerProcee 是决策树方法，VrUsingEars 是 Ear Queue 方法，VrUsingSew 是 Triangulate and Sew 方法 c. 类 Compare 用来比较上面三个算法的效率 Control + Demo 模块中，四个类都是 QThread 的子类，分别展示算法的过程 a. 类 VrProcessDemo, VrUsingEarsDemo, VrUsingSewDemo 分别是三种算法的展示 b. GeneralDemo 是整合了三种算法的展示，具体为 b1. 若节点的度数在 3~6 之间，使用决策树方法，否则进入 b2 b2. 若节点不在边界处，使用 Ear Queue 方法，否则进入 b3 b3. 使用 Triangulate and Sew 方法

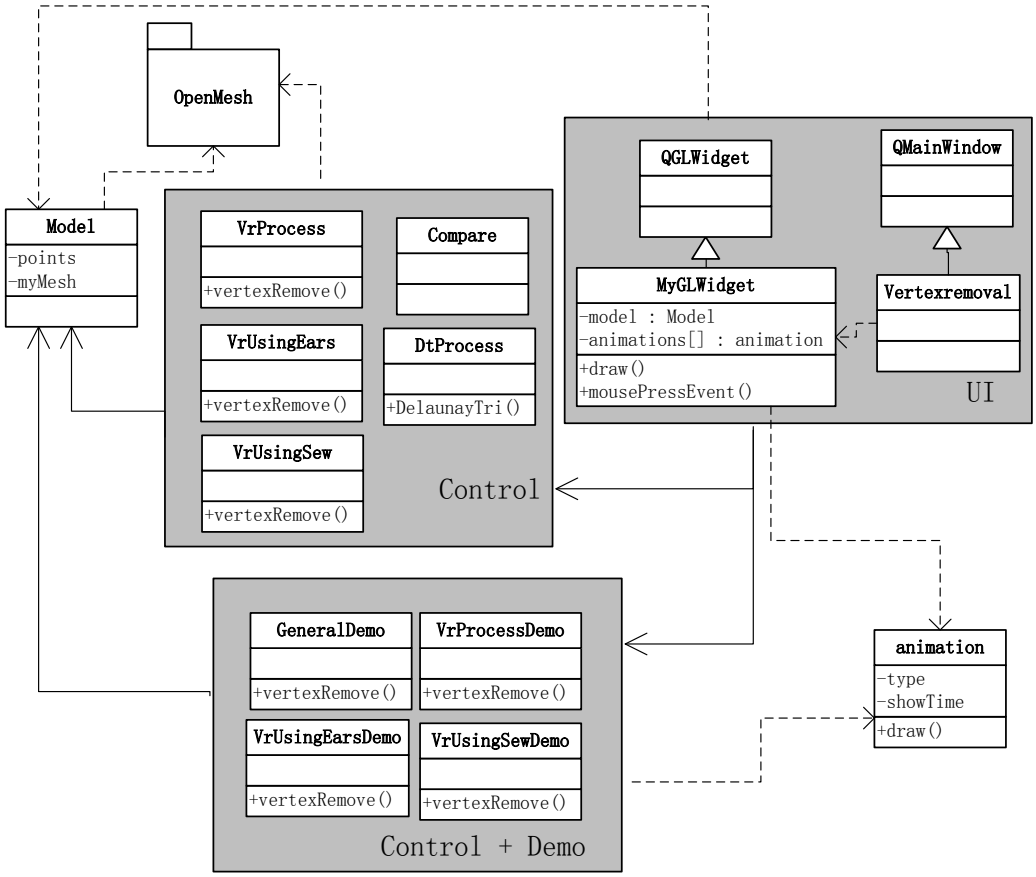


图 3.1 系统类图

3.2 问题及解决方法

a) 动画的设计

为了切合 MVC 的设计模型，使用一个类 animation 实现了动画，于是动画可以视为 MVC 模型中的 Model 的一部分。在 Control + Demo 模块中的方法运行时，会产生一系列的动画效果，可以分成点，线，面，圆等不同的种类，这些信息都保存在 MyGLWidget 中的属性 animations 中，然后再画出来。同时，每次从 Control + Demo 模块中发送一个新的动画请求时，会将之前的每个 animation 实例中的属性 showTime 减一，如果等于零就删除。

b) 浮点数误差的检测

在对算法进行测试的过程中，遇到了因为浮点数计算误差，产生的程序逻辑错误。这个错误是随机发生的，并且无法保存数据（因为把浮点数存入磁盘的过程中会发生误差），很难重现错误。后来的方法是先随机生成数据直接保存到 1000 个文件中，然后再从文件中读取数据做测试，直到发现了错误。

4 实验结果

实验中，我们分别对不同规模的数据做多次的节点删除操作，其中在使用三个算法删除点的过程中，需要保证删除节点的顺序一致，于是 三个算法每次执行的输入和输出始终保持一致。由于决策树方法和 Ear Queue 方法是有条件的，实验中先遍历所有的点，测试出一个合适的节点序列，使得按这个序列删除节点能够同时满足以上两个算法的条件。

实验统计多次结果后得到下表：

表 4.1

点的总数	删除点	决策树（秒）	Ear Queue（秒）	Tri Sew(秒)
100	81.9	0.0128	0.0229	0.1339
500	446.5	0.0711	0.1275	0.726
1000	901.8	0.1431	0.2602	1.4865
2000	1800.7	0.2877	0.523	2.9785
3000	2704.3	0.4356	0.7939	4.5089
4000	3610.6	0.5813	1.0521	5.9663
5000	4519.3	0.7207	1.3352	7.5652
6000	5428	0.8929	1.6532	9.3077
7000	6337.8	1.0192	1.9153	10.749
8000	7244.3	1.1817	2.1722	12.181

表 4.1 中，以第一行为例，表示实验中有 100 个点，删除其中的 81.9 个点（多次统计的平均值），三种算法消耗的时间分别为 0.0128 秒，0.0229 秒，0.1339 秒。为了更加直观地显示三种算法的执行时间，我们将表 4.1 转化为如下的折线图：

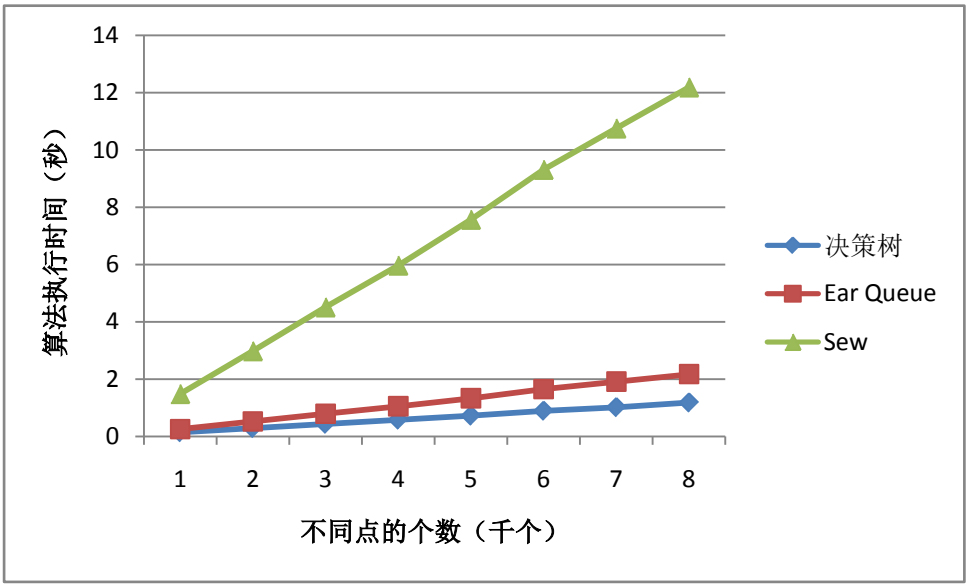


图 4.1 三种算法运行时间的比较

根据图 4.1 可以很明显地看出三种算法运行时间的差别。其中，决策树方法速度最快，

Ear Queue 方法次之，Triangulate and Sew 方法最慢。这也与报告中关于三种算法复杂的分析一致，即决策树方法与 Ear Queue 方法的复杂度均为$O(d\log d)$，而 Triangulate and Sew 的复杂度为$O(d^2)$。

另外，由于决策树方法只能处理度数为 3~7 的节点，Ear Queue 方法不能处理位于边界的节点，而 Triangulate and Sew 方法可以处理所有情况，所以可以考虑将这三种方法相互补充，于是我们的程序给出了一个 GeneralDemo 的选项，该方法就是整合了三种算法的一个展示，具体为：

a. 若节点的度数在 3~6 之间，使用决策树方法，否则进入 b
b. 若节点不在边界处，使用 Ear Queue 方法，否则进入 c
c. 使用 Triangulate and Sew 方法

5 参考文献

[1] A. Aggarwal, L.J. Guibas, J. Saxe, P.W. Shor, A linear-time algorithm for computing the Voronoi diagram of a convex polygon, Discrete Computational Geometry 1989; 4 (6): 591-604.

[2] CGAL Editorial Board, CGAL User and Reference Manual, 3.5 ed.; 2009.

[3] L.P. Chew, Building Voronoi diagrams for convex polygons in linear expected time, Technical report PCS-TR90-147, Department of Mathematics and Computer Science, DartmouthCollege, Hanover, NH: 1990.

[4] Olivier Devillers, On deletion in Delaunay triangulation, International Journal onComputational Geometry Application. 2002; 12: 193-205.

[5] Olivier Devillers, Vertex removal in two-dimensional Delaunay triangulation:Speed-up by low degrees optimization, Computational Geometry: Theory andApplications. 2010; 44: 169-177.

[6] M. Heller. Triangulation algorithms for adaptiveterrainmodeling.In Proc. 4th Internat.Sympos.Spatial Data Handling.1990; 163-174.

[7] F.P. Preparata, M.I. Shamos, Computational Geometry: An Introduction, 3rd ed., Springer-Verlag, October 1990.

[8] Rex A. Dwyer. A Faster Divide-and-Conquer Algorithm for Constructing Delaunay Triangulations.Algorithmica(1987)2: 137-151