计算几何课程报告

直骨架的研究与实现

曹一溪
2011210842
计研 111
caoyixi@gmail.com

吴敏
2011210824
计研 111
wumalbert@gmail.com

杨昊
2011210799
计研 111
yangh2007@gmail.com

1. 简介

1.1 直骨架(Straight Skeleton)

[Aichholzer et al., 1995] 首先针对简单多边形提出了直骨架的概念。我们可以将其看成是一个波传播的过程。考虑一个简单多边形，多边形的每一条边，以相同的速度，沿着其法线方向运动，同时其端点言端点所在角的内角角平分线方向运动。在运动的过程中，将会发生两种事件：

a. 边事件(Edge event): 在运动的过程中，一条边可能会消失，使得原本与该边相连的两边相连。

b. 分裂事件(Split event): 即一条边分裂为两条边。例如当一个凹点运动到一条边上时，这条边将会分成两个部分，并且有新的边的连接关系产生。

我们追踪所有顶点的运动轨迹，便可得到一个简单多边形的直骨架。

此后，[Aichholzer et al., 1996]拓展到了平面上任意直线图形(Planar Straight-Line Graph, PSLGs)。图 1 给出了一个直骨架的图示。

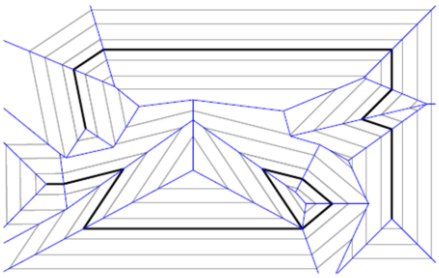


图 1：一个直骨架示例

1.2 摩托图(Motorcycle Graph)

在本文采用的计算直骨架的算法中，利用了摩托图。摩托图最初由[Eppstein and Erickson, 1999]提出，其定义一个"摩托"(motorcycle)为一个三元组$(\mathrm{p}, \mathrm{s}, \mathrm{t}^*) \in \mathbb{R}^2 \times \mathbb{R}^2 \times [0, +\infty)$,其中 p 为起点，$\mathrm{t}^*$为开始时间，s 为一个速度的向量。

当考虑 n 个"摩托"$\mathrm{m}_1, m_2, \cdots, m_n$, 其中$\mathrm{m_i} = (p_i, s_i, t_i^*)$,每一个摩托将会留下一条其对应的运动轨迹，并将在遇到其他摩托的轨迹是消亡。我们将这样形成的图形称为摩托图，如图 2 所示。

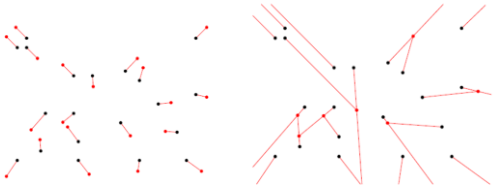


图 2：左图：n 个摩托(motorcycle)
右图：左图对应形成的摩托图

1.3 术语和记号

我们简记一个多边形 P 的直骨架为 S(P), 其摩托图为 M(P)。在 S(P)中，对于每一条 P 上的边 e 扫过的区域称为 face(e), 每个顶点的运动轨迹称为弧(arc), 弧的两个端点不属于 P 的点称为 node，如图 3 所示.

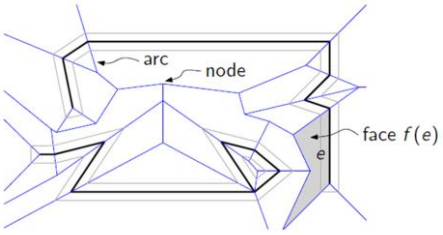


图 3：术语及记号示例

1.4 直骨架的应用

直骨架应用范围非常广泛，它可以代替 Vonoroi 图来解决很多问题，在一些 Vonoroi 图难以处理的领域，直骨架也可以发挥重要作用。

1.4.1 斜接等距曲线的求取

等距曲线（Offset Curves）的计算是数控机械加工（NC-machining）过程中所必需的工序。假设需要使用数控机床打磨出一个简单多边形的内腔P（如图 4 中粗黑线位置），那么圆形磨机的圆心必须沿着较P略有收缩的轨迹P'（如图 4 中细黑线位置）打磨方能保留P的边缘。数学意义上，通过 Minkowski 和的运算可以求出一种满足条件的等距曲线P'，即

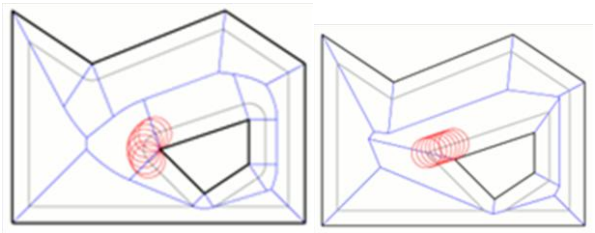


图 4：斜接等距曲线示例

应满足$P = Minkowski\ Sum\ P' + D_r$，其中$D_r$是以零点为圆心、以磨机半径为半径的圆。这种方法定义的P'被称为传统等距曲线斜线（Conventional Offset Curve）。基于P的 Vonoroi 图可以轻易得到它的传统等距曲线。

但是如图 4 左所示，若圆形磨机沿着传统等距曲线移动，在目标多边形的尖角处往往会打磨过长时间，很容易导致高温和磨损。Park 和 Chung（2003）为此提出了斜接等距曲线（Mitered Offset Curves）的概念，见图 4 右，以这种斜接等距曲线作为磨机圆心的轨迹便可以避免这些问题。基于多边形P的直骨架可以很容易地算出它的斜接等距曲线。

1.4.2 自动化的屋顶构建与地形建模

给定一个平面直线图（PSLG）G，若把时间作为第三维，将基于G生成直骨架的二维波阵前进过程映射到R^3空间中，那么这些波阵在R^3中扫过的点集便构成了G的地形$T(G)$。其中每一条波前边（Wavefront Edge）扫过的点集都是一张三维面片。

这种方法可以直接用于构造房屋屋顶，或者给地形图建模。构造屋顶时，以墙的布局图作为输入；而给地形图建模时，则以地形图上的河岸、海岸线作为输入。

图 5：自动化的屋顶构建与地形建模

1.4.3 折剪纸问题的求解

直骨架还能用于求解数学折纸（Mathematical origami）中的折剪纸问题（Fold and cut problem）。在这种折剪纸问题中你可以对一张纸进行任意次折叠，最后对折好的纸沿任一条直线剪掉一块，流程见图 6。那么你应该如何折叠这张纸以便最终能剪出想要的形状呢？这个问题也可以通过提取目标多边形的直骨架来求解。

1.4.4 其它应用

如果给出一个三维体在某一方向上的平行剖面图，那么可以利用直骨架来重建三维表面，这在医学成像领域用得很多。 此外直骨架还能用于计算所谓的城市 Voronoi 图（city Voronoi diagrams），可以用在地理位置信息系统（GIS）里。另外直骨架还被用于多边形分解、多边形轮廓匹配等领域。

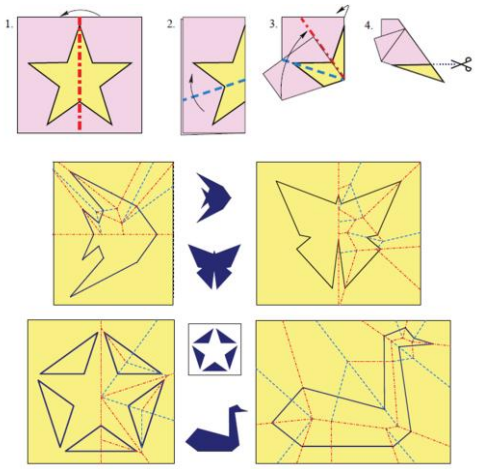


图 6：利用直骨架求解折剪纸问题

2. 本文工作

本小组此次课程设计的工作便是基于[S.Huber and M.Held, 11]所提出的算法。工作涉及内容涵盖了 S. Huber 博士论文的大部分工作。在本次课程设计中，我们将对其提出的直骨架算法予以实现。

该算法首先计算出平面直线图对应的摩托图，利用原有的平面直线图和摩托图生成一个新的平面直线图，然后模拟新的平面直线图的波前传播，得到新图的直骨架，也即是原图的直骨架。摩托图的引入降低了 split event 的时间消耗，从而保证该算法有较高的运行效率。

我们还实现了一个交互友好的界面以展示算法的实现过程以及计算结果。此外，我们还将把算法应用自动化的屋顶构建与地形建模上，将算法计算结果予以三维展示，以体现直骨架在解决实际问题中的重要作用。

3. 相关工作

[Aichholzer et al., 1995] 首先定义了简单多边形的直骨架，并给出了一个计算简单多边形的直骨架的算法，该算法采取离散事件模拟方法来模拟波前的传播，该算法为所有波前边所有可能的 edge event 维持一个优先队列，优先级为每个 edge event 的发生时间，每次取队列中的第一个事件，处理该事件并检测 split event 进行处理，之后更新整个队列，直到队列为空。对于有n个顶点的简单多边形，该算法的时间复杂度为$O(n^2\ log\ n)$。

[Aichholzer et al., 1996] 提出基于三角剖分计算平面直线图的直骨架方法，该算法能处理平面直线图，而不局限于简单多边形，它也是模拟波前的传播，但是利用了平面直线图的三角剖分，通过传播过程中三角面的拓扑变化来决定 edge event 和 split event。该算法在最坏情况下的时间复杂度为$O(n^3\ log\ n)$，作者提到对实际的数据其时间复杂度会降到$O(n\ log\ n)$，但没有给出具体的实验结果。

[Eppstein and Erickson, 99]提出了一个计算简单多边形直骨架算法，同时该算法可以计算平面直线图的直骨架。该算法提出一个能快速计算最近点对的数据结构，从而快速找出 split event，该算法的复杂度为$O(n^{1+\epsilon} + n^{8/11+\epsilon} r^{9/11+\epsilon})$（$r$

是凹点的数目，ϵ 为任意大于 0 的实数），该算法太难实现以至于实际很难达到这个理论上的复杂度。

[Cheng and Vigneron, 02,07] 利用直骨架与摩托图（motorcycle graph）的关系进行直骨架的抽取。该算法只能计算带洞的非退化的多平行的直骨架。算法的复杂度为 $O(n\sqrt{n}\,log^2 n)$，并且在波前传播中两个或两个以上的凹点不能相遇，否则出现不正确的结果，这个算法实现起来也太过复杂，到目前也没有被人实现。

[Felkel and Obdrzalek, 99]模拟波前传播给出了计算带洞简单多边形直骨架的算法和具体实现细节。该算法的时间复杂度为$O(nr + n\,log\,n)$，最坏情况下为$O(n^2)$。[Cacciola, 04] 实现了这个算法并集成到 CGAL 中。

[S.Huber and M.Held, 11]基于[Eppstein and Erickson, 99]和[Cheng and Vigneron, 02]，对摩托图进行扩展，并利用它改进了[Aichholzer, 98]的算法，给出了一个快速且容易实现的算法。该算法在最坏情况下的时间复杂度为$O(n^2\,log\,n)$，但实验结果显示，处理实际数据时复杂度可降为$O(nlogn)$，相比之下，[Cacciola, 04]中的算法的时间复杂度为$O(n^2)$。

4. 算法原理

本文实现算法首先计算出平面直线图对应的摩托图，利用原有的平面直线图和摩托图生成一个新的平面直线图，然后模拟新的平面直线图的波前传播，得到新图的直骨架，也即是原图的直骨架。

4.1 计算摩托图

为了快速地计算得到一个平面直线图 G 的直骨架 S(G)，我们需要快速地计算得到 G 中所有凹点对应的摩托图 M(G)。[Cheng and Vigneron, 07]所提出的摩托图生成算法的复杂度为 $O(n\sqrt{n}\log n)$，但是其算法较难实现。在本次试验中，我们采用了一个[Cheng and Vigneron, 07]的一个简化版本，我们采用一个简单的空间 Hash 使得算法的实现较为简单，并且能够获得较为高效的实际运行效率。

4.1.1 算法描述

算法的输入由两部分组成：一组摩托$M = \{m_1, m_2, \cdots, m_n\}$和一组墙边$W = \{w_1, w_2, \cdots, w_m\}$。墙边是指一些预先定义的线段，摩托若遇到墙边，将直接消亡。墙边实际对应平面直线图中的各条线段。

算法首先将所有摩托起始点和墙边的包围盒划分为一个$\sqrt{n} \times \sqrt{n}$的规则网格。算法在执行的过程中，需要维护两个几何哈希(Geometric Hash):H_M和H_W。H_M记录每一个摩托的轨迹，存储对应摩托当前经过的所有网格。H_{W_i}记录与墙边w_i相交的网格。

算法执行的过程是一个摩托运动的离散事件模拟机制，模拟过涉及两种事件：Crash Event 和 Switch Event。一个 Crash Event 是指一个摩托在运动过程中，撞上其他摩托的运动轨迹或一堵墙边而消亡。一个 Switch Event 是指一个摩托从一个网格进入另一个与其相邻的网格。所有的这些事件都被存储在一个优先队列 Q 中，按照发生时间的先后依次进行处理。实际上，对于每一个摩托 m，我们都维护一个二叉平衡搜索树C[m], 存储 m 将来可能发生的所有 Crash Event。

算法开始执行时，首先对每一个墙边w，计算与其相交的网格，即初始化H_W。随后，对于每一个摩托 m,我们调用一次 insertMC(m)函数，以初始化相应的数据。随后，算法进入一个循环，在每轮循环中，算法从优先队列 Q 中抽取一个事件 e，并调用函数 handle(e)，进行处理，直至队列 Q 为空。

insertMC(m)函数和 handle(e)函数描述如下(简记事件 e 对应的摩托为 m)：

insertMC(m): 算法创建一个空的 m 对应的二叉平衡搜索树C[m]，并向优先队列 Q 中插入一个 m 的 Switch Event e，事件 e 的发生时间为 m 的开始时间。

handle(Switch Event e): 我们首先在 e 对应的网格 c 中，记录摩托 m 将会经过该网格，并检查 m 是否会进入新的邻近的网格，如果存在，则向优先队列 Q 中插入一个新的 Switch Event 事件。接着，我们检查每一条经过网格 c 的墙边是否与 m 在该网格内相交，若相交，向优先队列 Q 中添加最早的一个 Crash Event。然后，我们清空 C[m], 检查当前已经记录的会经过网格 c 的摩托m′是否与 m 在网格 c 中相交，若相交且 m 到达交点的时间晚于m′，则向 C[m]中添加相应的 Crash Event，否则向C[m′]中添加相应的 Crash Event。需要注意的是，若该事件添加到C[m′]之后，最终成为C[m′]中最早的事件，我们同样需要将该事件添加到优先队列 Q 中。最后，若C[m]不为空，我们将 C[m]中最早的事件添加到优先队列 Q 中。

handle(Crash Event e): 我们首先将 m 标记为已经消亡，并清空 C[m]，记录 m 的轨迹信息。随后，我们清空 Q 中所有与 m 相关，且不可能发生的事件(或不清楚 Q 中的相应事件，而在每次事件处理时，先检查该事件是否仍然可能发生)。最后，在 m 所处的网格 c 中，我们清除网格 c 中其他摩托m′对应的C[m′]中与 m 相关且不可能发生的事件，若原先的堆顶事件被删除，则需要添加新的堆顶事件到优先队列 Q。

4.1.2 算法复杂度分析

在后面的讨论中，为了简便，我们不讨论墙边的影响。事实上，在本文讨论的环境，即在多边形的直骨架生成过程中计算摩托图，墙边有 O(n)条，而墙边对时间复杂度的影响主要体现在 Switch Event 的处理中，在复杂度分析上可以等效为有 2n 个摩托。

在算法的执行过程中，insertMC 操作，准确地执行了 n 次，在每次操作中，需要向优先队列 Q 中插入一个元素。如果我们采用类似最小堆的结构实现 Q，每次的插入操作消耗的时间复杂度为$O(\log n)$。所以，该步操作总的时间复杂度为$O(n \log n)$。而在事件模拟的过程中，我们有 O(n)个 Crash Event，并且最多有$O(n\sqrt{n})$个 Switch Event。在每个 Crash Event 中，我们需要更新H_M，在最坏情况下(所有摩托都在一个格点中)，我们需要耗费$O(n^2 \log n)$才能完成这步操作。而在每个 Switch Event 中，我们需要检查一个摩托与其他的碰撞情况，最坏需要检查O(n)个。而在每次检查中，若相碰，则需要插入相应事件到优先队列 Q，插入操作耗费时间O(logn)。故 Crash Event 的处理，最坏情况下的时间复杂度为$O(n^2\sqrt{n}\log n)$。综上，最坏情况下，整个算法的时间复杂度为：

$$O\left(n \log n + n^2 \log n + n^2\sqrt{n}\log n\right) = O(n^2\sqrt{n}\log n)$$

但是，我们需要注意的是，在 Crash Event 操作中，最坏情况需要所有摩托均在同一个格点，这在摩托均匀分布的情况下，是很难发生的。在平均意义下，一个格点只包含 O(1)

个摩托，故此时，处理 Crash Event 的时间复杂度为$O(n\log n)$。同样的，处理 Switch Event 时，最坏情况需要所有摩托穿越$O(\sqrt{n})$个格点，并且各个格点包含 O(n)个摩托。这在实际情况中，同样是很难发生的。在平均意义下，一个格点只包含 O(1)个摩托，一个摩托只穿过 O(1)个格点，故此时，处理 Switch Event 的时间复杂度为$O(n\log n)$。综上，在摩托均匀分布在网格中时，算法总的时间复杂度为：

$$O(n\log n + n\log n + n\log n) = O(n\log n)$$

由此，我们得到了一个最坏情况下时间复杂度较高，但是期望的时间复杂度较低且易于实现的摩托图计算算法。

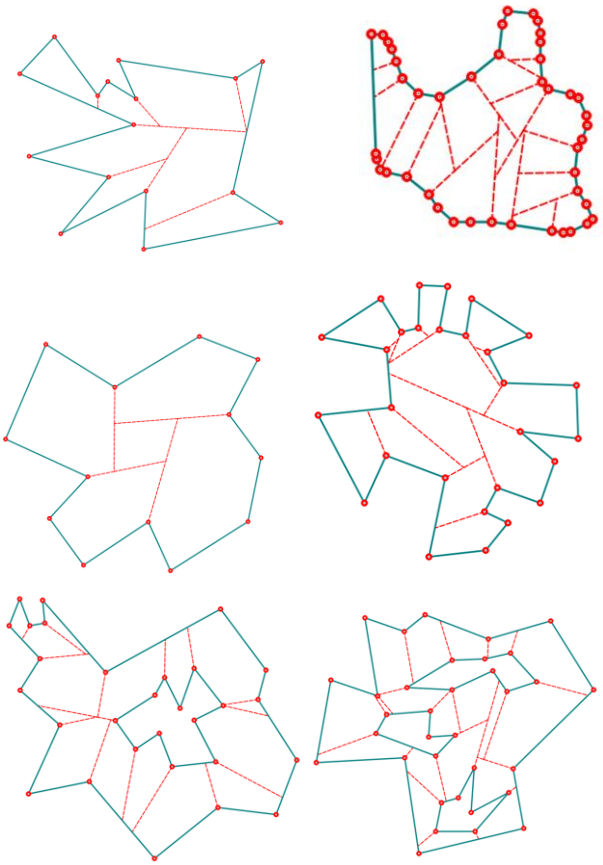


图 7：摩托图计算结果示例

4.1.3 实验结果

我们在 VS2010 上实现了上述小节提到的摩托图生成算法。算法的输入是一个多边形，多边形的凹点将生成摩托，多边形的各条边对应摩托图生成算法中的墙边。图 7 展示了一些计算的结果，其中最后两个示例为带内环多边形的示例。

4.2 计算直骨架

4.2.1 算法描述

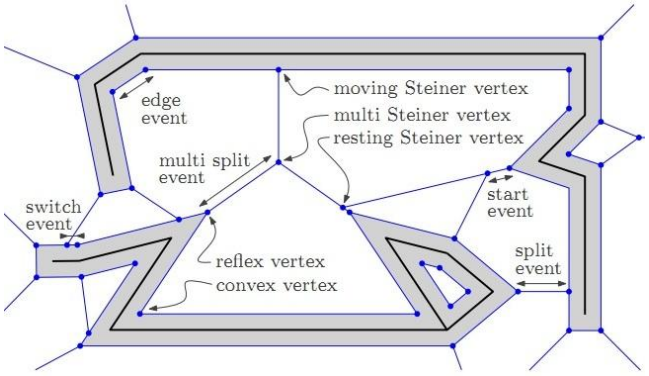


图 8：扩展波前的点的类型以及事件类型

利用原始多边形和由它导出的摩托图，我们构建一个平面图，即为扩展的波前，t 时刻的扩展波前记为$W^*(G,t)$。扩展波前中的点进行标记，原多边形的凸点记为 Convex Vertex，原多边形的凹点记为 Reflex Vertex（即为摩托图中摩托的起点），摩托与原多边形的边相撞的点记为 Moving Steiner Vertex，多辆摩托相撞的点记为 Multi Steiner Vertex，其它的点记为 Resting Steiner Vertex（如图 8 所示）。

在整个算法中，我们对波前中的边进行离散事件模拟，即维护一个最小优先级队列，波前中所有的边构成队列的元素（事件），边的消失时间为对应事件的优秀级。我们从初始的拓展波前$W^*(G,0)$开始，将$W^*(G,0)$中的每一条边 e 插入优先队列 Q 中。Q 按事件发生的时间（边的消失时间）先后顺序排列。

完成初始操作之后，我们从 Q 中一个一个地抽取事件，并对其进行处理，修改拓展波前，并维护优先队列 Q，直到队列为空，算法执行完毕。下面将具体讨论不同的事件类型的处理过程：

Edge Event: 两个凸点 u，v 间的边 e 消亡，如图 9(a) 所示。我们将 e 从原图中删除，并将 u,v 点合并为一个新的凸点 w。与点 w 相连的两条边将决定 w 的速度，因此，我们需要更新与 w 相连的两条边的消亡时间，并在 Q 中修改相应的项。当 w 的两条边平行的时候，如果按这两条边来求 w 得速度，w 的速度将趋向于无穷大，这将会导致后续的计算出现问题。我们对这种情况做特殊处理，将 w 的速度设为 0。

最后，我们需要向直骨架中添加两条边 vw 和 uw。

Split Event: 一个凹点 u 和一个 moving Steiner 点 v 之间的边 e 消亡，如图 9(b) 所示。我们可以注意到，v 点是凹点 u 在摩托图中消亡的对应点。我们分别记u_l, v_l为与 u,v 相连的且在$\overline{uv}$左边的点，同样有u_r, v_r。我们删除边 e，并合并点u_l和点v_l为一个新的凸点w_l。w_l的速度由与其相连的两条边决定。此处，我们需要检测三种特殊情况：(i) w_l相连的两条边平行，此时w_l的速度将设为 0。(ii) u_l, v_l, e 构成一个三角形，Split Event 事件后，这个三角形消失。(iii) e 在原来的多边形的某条边上，如果 b 所示。同样地，我们对u_r, v_r做相似的操作。

最后，我们需要向直骨架中添加一条边 uv，在特殊情况(ii)中还需要为凸点增加相应的直骨架。

Start Event: 一个 resting Steiner 点 v 和一个凹点或一个 Moving Steiner 点 u 之间的边 e 消亡，如图 9(c) 所示。v 将成为一个 Moving Steriner Vertex，而 u 的类型和速度不变，但是我们需要更新 v 的速度，并更新 v 的邻边的消亡时间。此外，有两种特殊情况需要特殊处理：(i) u 的邻边、v 的一条邻边和边 e 构成一个三角形，Start Event 发生时该三角形消失。(ii) v 的邻边中没有另外一条与 e 平行的边，即 u 发出的摩托因碰到其他摩托的轨迹而消失，原则上其他摩托会先到达 v 点，但这时候 Start Event 发生了，说明这两辆摩托同时到达 v 点。

一般情况下 Start Event 不会向直骨架中贡献一条边，特殊情况 (i) 中三角形消失，其中的凸点会给直骨架增加一条边。

Switch Event: 一个凸点 u 和一个凹点或一个 moving Steiner 点 v 之间的边 e 消亡，如图 9(d) 所示。v 点将从 u 的一条邻边跳到 u 的另一条邻边上，u 和 v 的速度都将发生改变，需要更新 u、v 的邻边的消失时间。有一种特殊情况需要考虑：u 的一条邻边、e 和 v 的另一条不在原多边形上的邻边构成一个三角形，Switch Event 发生时，该三角形消失。

如果 Switch Event 发生在凸点与凹点之间，需要为直骨架增加边，如果 Switch Event 发生在凸点与 Moving Steiner 之间，不需要为直骨架增加边。在特殊情况中消失的三角形会增加直骨架边。

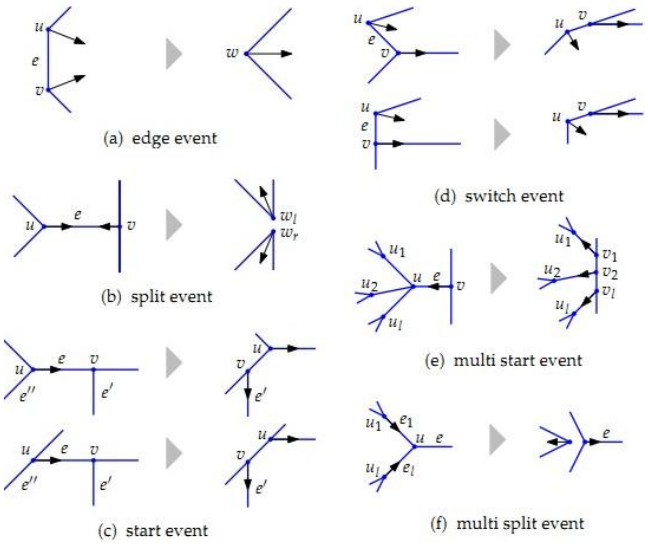


图 9 扩展波前中不同事件的处理方法

Multi Start Event: 一个 moving Steiner 点 v 和一个 multi Steiner 点 u 之间的边 e 消亡，如图 9(e) 所示。设其他与 u 相邻的顶点为$u_1, u_2, \dots, u_l$，相应的$u_1, u_2, \dots, u_l$与 u 形成的边为$e_1, e_2, \dots, e_l$，并且$e_1, e_2, \dots, e_l$按逆时针排序。u 对应多个摩托相撞点，e 边先消失，意味着这多个摩托不可能一起相撞，而是将 v 的多边形邻边进行分裂，即引入新的 Moving Steiner 点$v_1, v_2, \dots, v_l$，它们之间形成边$v_i v_{i+1}, (i = 1, \dots, i-1)$，另外原来与 v 相邻的两个多边形顶点与$v_1, v_l$相连，形成两条边。

Multi Start Event 不直接产生直骨架边。

Multi Split Event: 连接凹点$v_1, \dots, v_l$的与一个 multi Steiner 点 u 的边$e_1, \dots, e_l$同时消亡，如图 9(f) 所示，假设$e_1, e_2, \dots, e_l$逆时针方向排列。这个事件对应多个摩托同时相撞的情况，有可能发出新的摩托。设$v_1, \dots, v_r$对应的左右多边形邻边为$l_1, r_1 \dots l_l, r_l$，$l_i r_{i+1} (i = 1, \dots, l-1)$ 将决定一个新的凸点，如果l_1, r_l形成的角为凹角，此时这两条边决定一辆新的摩托；否则，它们构成凸角，不产生新的摩托。

Multi Split Event 中，$v_1, \dots, v_l$中的每一个凹点会增加一条直骨架边。

Remainning Event: 如果两个 Moving Steiner Vertex 构成的边 e 消失，直接删除相应的边以及顶点即可。该事件不会增加直骨架边。

除了上面七种事件外，其他的点的组合构成的事件不会发生。

4.2.2 数据结构

在多边形存在凹角和凸角，这与多边形的方向有关，即顺时针或者逆时针。因此，我们将整个直线图作为有向图处理。观察图 8 的扩展波前，我们看到 Convex Vertex 的度数（包含入度和出度）为 2，Reflex Vertex 的度数为 3，Moving Steiner Vertex 度数为 3，Resting Steiner Vertex 度数为 3，Multi Steiner Vertex 的度数大于等于 3。如果没有 Multi Steiner Vertex，我们完全可以为每个顶点存储三个边指针，分别指向与之关联的三条边，我们可以为 Multi Steiner Vertex 单独增加一个数组，保存所有与之关联的边，但是这样的话将会导致点的存储不一致。又由于算法的整个过程不涉及面的处理，而且边会动态变化，因此放弃了 DCEL 结构，并最终选择了有向图的十字链表结构。具体结构如下面的代码所示：

```
class PSLGVertex {
    bool mark; //顶点访问标记
    QPointF oriPosition; //顶点的起始位置
    QVector2D speed; //顶点的速度
    double startTime; //顶点的起始时间
    PSLGEdge* firstin;  //第一条入边
    PSLGEdge* firstout; //第一条出边
    PSLGVertexType type; //顶点类型
};
class PSLGEdge {
    bool mark; //是否被访问的标志
    PSLGVertex* tailvex; //弧尾，边的起点
    PSLGVertex* headvex; //弧头，边的终点
    PSLGEdge* tedge; //共弧尾的边
    PSLGEdge* hedge; //共弧头的边
    double vanishTime; //边的消失时间
    PSLGEdgeType type; //边的类型
    int heapIndex; //边在最小优先队列中的索引
};
class PriorityQueue {
    std::vector<PSLGEdge*> edges;
    int heapsize;
};
```

我们为每一个点记录他的第一条入边和第一条出边，同时，为了后面算法执行的高效性，我们规定这两条边都必须是多边形边；每一条边保存有起点、终点、共弧尾的边、共弧头的边，这些共弧头、共弧尾的边就是与相应顶点关联的摩托图边。

算法的核心是离散时间模拟，处理的对象是扩展波前中的边，因此我们使用最小优先级队列来模拟事件，其中队列中的元素为扩展波前中的边，优先级为边的消失时间。我们在基于动态数组 vector 建立了一个二叉堆作为我们的最小优先队列，在每一条边中保存它在最优队列中的索引值，这样可以快速定位一条边在最优队列中的位置，使得在事件模拟时可以快速维护优先级队列。

4.2.3 算法复杂度分析

首先需要构件初始的波前，即通过遍历多边形的顶点与边并建立顶点、边的关联关系构建十字链表结构。设多边形的顶点数位n，则边数 e 为$O(n)$，因此构造对应的十字链表结构所需要的时间为$O(n)$。

假设直线图中包含r个凹点，对应r辆摩托，第二步需要计算这r摩托构成的摩托图。假设摩托图已经事先由前面的算法给定了，紧接着需要将摩托图插入到初始波前中构成初始扩展波前$\mathrm{W}^*(G,0)$，记摩托图中有e_m条边，则将摩托图插入到初始波前的时间复杂度为$O(e_m)$。

下面分析e_m与r、n 的关系。摩托图中有三类点，Moving Steiner Vertex, Multi Steiner Vertex, Resting Steiner Vertex，他们的数目分别记为$r_{mov}, r_{mul}, r_{rest}$。每一辆摩托对应一条运动轨迹，如果一辆摩托 m_1撞在另一辆摩托 m_2的运动轨迹上，形成 Resting Steiner Vertex 点，该点将m_2的轨迹一分为二，如果多辆摩托同时相撞，将产生一辆新的摩托，对应一条新的轨迹。因此，摩托图中边的数目满足：

$$e_m = r + r_{mul} + r_{rest}$$

将摩托图插入到初始波前时，每一个 Moving Steiner Vertex 将会把它所在的多边形边一分为二，记初始扩展波前中边的数目为e_w，则有：

$$e_w = e + e_m + r_{mov}$$
$$= e + r + r_{mul} + r_{rest} + r_{mov}$$

设一共有 r' 辆摩托会同时相碰，则：

$$r' + r_{rest} + r_{mov} = r$$
$$e_w = e + 2r + r_{mul} - r'$$

对扩展波前构建完十字链表后，需要对所有的边建立优先级队列，建立二叉堆的时间为$O(e_w \lg e_w)$。

完成初始化后，需要对整个队列进行离散事件模拟。需要针对具体事件进行时间复杂度分析：

Edge Event 需要修改常数条边的消失时间，即更新队列中的常数项，这需要$O(\lg e_q)$的时间，e_q表示当前队列中的元素。处理完一次 Edge Event 后，队列中减少一条边。

Split Event 修改常数条边的消失时间，复杂度$O(\lg e_q)$。处理完毕后，队列中减少一条边。

Start Event 修改常数条边的消失时间，复杂度$O(\lg e_q)$。处理完毕后，队列中边的数目保持不变。

Switch Event 修改常数条边的消失时间，复杂度$O(\lg e_q)$。处理完毕后，队列中边的数目保持不变。

Multi Start Event 需要对l个摩托按角度进行排序，同时更新与这些摩托相邻的边的消失时间，复杂度$O(l \lg l + l \lg e_q)$。处理完毕后，队列中增加$(l-2)$ 条边。

Multi Split Event 需要对l个摩托按角度进行排序，同时更新与这些摩托相邻的边的消失时间，复杂度$O(l \lg l + l \lg e_q)$。处理完毕后，队列中减少l条边。

Remaining Event 更新常数条边的消失时间，复杂度$O(\lg e_q)$。处理完毕后，队列中减少 2 条边。

在最坏情况下，Multi Start Event 事件先发生，这部分的复杂度为：

$$\sum_{l=1}^{r_{mul}} l \lg l + l \lg e_q^l \le r' \lg r' + r' \lg(e_w + r')$$

之后队列中的元素个数为$e_w + r'$，处理完这个队列的复杂度为：

$$\sum_{e_q=1}^{e_w+r'} \lg e_q \le (e_w + r') \lg(e_w + r')$$

于是该算法总的复杂度为：

$$O(e_w \lg e_w) + r' \lg r' + r' \lg(e_w + r') + (e_w + r') \lg(e_w + r')$$

最坏情况下，$r' = r = n/2, r_{mul} = r'/2$，此时

$$\mathrm{e_w} = e + 2r + r_{mul} - r'$$
$$= e + 3r/2 = 7n/4$$

于是，这部分总的算法复杂度为：$O(n \lg n)$。

4.2.4 实验结果

我们在 VS2010 上，利用 C++实现上上述的直骨架算法。算法能够处理多种退化情况下的多边形，这也是目前 CGAL 所不具备的。附录 A 中给出了一些算法的运行结果。其中 A-C 为 CGAL 提供的测试数据，用于验证算法正确性。通过结果对比，本实验算法计算得到的直骨架是正确的；D-F 提供了利用本实验实现算法进行建模得到的多边形直骨架，及其对应的三维模型。可以看到，直骨架能够被应用于三维中来。

此外，我们还在一台装有八核 Intel i5-2400，8GB 内存并安装 Window7 64 位系统的实验机器上，将本文实现算法与 CGAL 的直骨架生成算法效率进行了对比，如图 10 所示。可以看出，本次实验实现的未经高度代码优化的算法，其在执行效率上至略微地慢于 CGAL 已经高度优化成熟的算法执行效率。

但是，本文所实现算法所能处理情况的多样性是 CGAL 不具备的。

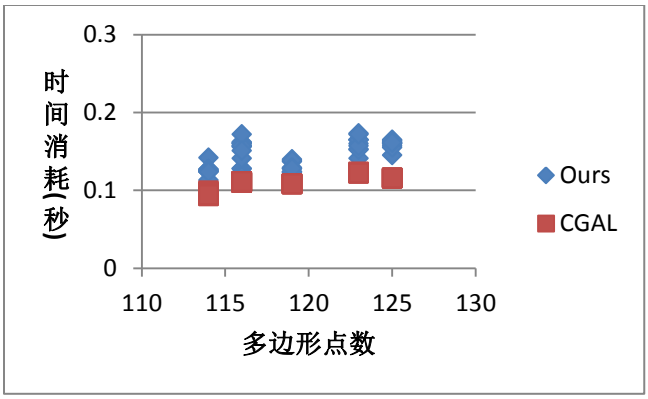


图 10：本文算法与 CGAL 运行速率对比

4.3 自动化的屋顶构建和地形建模

本次实验完成了平面图的直骨架抽取，同时实现了直骨架的一个应用——自动化的屋顶构建和地形建模。

自动化的屋顶构建过程伴随直骨架的生成而生成。把直骨架作为多边形的一个划分，则多边形的一条边对应一个剖分多边形，也就对应一个三维的多边形面，可以在算法的过程中实时记录每一条边对应的面的顶点及其连接关系。但是这个在十字链表的实现中有一定难度。同时，我们观察到一个现象，每一个多边形的边对应一个三维多边形面，这个面是边往前传播过程中扫过的区域，边消失时面就确定了。因此，我们可以“追踪”多边形边的传播过程， 它可能会被分裂成多条边，这多条边可能会逐一消失，对于每一次变化，边消失或边分裂，我们都为该边生成一个对应的三角面片，当这分裂出的边都消失时，意味着原来的多边形边消失了，这个过程中所有的三角面片拼接起来，就成为了原多边形边扫过的区域，也即原多边形边对应的剖分面。算法的运行结果请见附录 A。

5. 总结

本次试验，我们实现了 Stefan Huber 的博士论文的主要工作，实现了快速构建摩托图算法，和基于摩托图计算平面直线图的直骨架，并将该算法应用到自动化屋顶构建和地形建模的应用中。

总结这次试验，我们用一句话概括：过程很曲折，结果很成功。

虽然 Stefan Huber 博士在论文中对整体算法做了详细的理论阐述，并对每一个事件都给出了具体的处理方法，但是事件的处理方法不够详细，都只覆盖了基本的事件，没有对特殊情况进行分析和处理。而我们处理的对象是多边形，每个事件基本涉及到四五条边，这四五条边的连接关系可以有多种组合，因此对应多种特殊情况。由于作者在算法中没有对这些特殊情况进行处理，因此我们在初期实现中遇到了很多的 bug，最终对每一个 bug 进行跟踪调试，最终确定了每一种特殊情况的处理方法。由于属于特殊情况，因此对应的多边形也比较特殊复杂，这给 debug 带来了很大的工作量。整个实验的代码部分，我们花了三个多星期，大量的时间耗费在 bug 调试中，整个过程艰难曲折。我们还对其中的一些特殊情况咨询了作者本人，但作者本人也没有给予我们太多实质性的指导和帮助。但最终，我们把目前所遇到的 bug 都解决掉了，保证了算法的稳定性和结果的正确性，这一点来之不易，让我们甚感欣慰。同时，我们为实现了在这么多的时间内实现一个人的博士工作而感到自豪。

最后，感谢我们的组员曹一溪、吴敏、杨昊，感谢大家一起为大作业奋斗。也感谢 Stefan Huber 博士的回信，感谢邓老师精彩的授课。

6. 参考文献

[1] O. Aichholzer, D. Alberts, F. Aurenhammer, and B. Gärtner. Straight Skeletons of Simple Polygons. In *Proc. 4th Internat. Symp. of LIESMARS*, pages 114–124,Wuhan, P.R. China, 1995.

[2] O. Aichholzer and F. Aurenhammer. Straight Skeletons for General Polygonal Figures. In *Proc. 2nd Annu. Internat. Conf. Comput. Combinatorics*, volume 1090 of *Lecture Notes Comput. Sci.*, pages 117–126. Springer-Verlag, 1996

[3] O. Aichholzer and F. Aurenhammer. Straight Skeletons for General Polygonal Figures in the Plane. In A.M. Samoilenko, editor, *Voronoi's Impact on Modern Science, Book 2*, pages 7–21. Institute of Mathematics of the National Academy of Sciences of Ukraine, Kiev, Ukraine, 1998

[4] D. Eppstein and J. Erickson. Raising Roofs, Crashing Cycles, and Playing Pool: Applications of a Data Structure for Finding Pairwise Interactions. *Discrete Comput. Geom.*, 22(4):569–592, 1999

[5] .-W. Cheng and A. Vigneron. Motorcycle Graphs and Straight Skeletons. In *Proc. 13th ACM-SIAM Sympos. Discrete Algorithms*, pages 156–165, San Francisco,CA, USA, 2002.

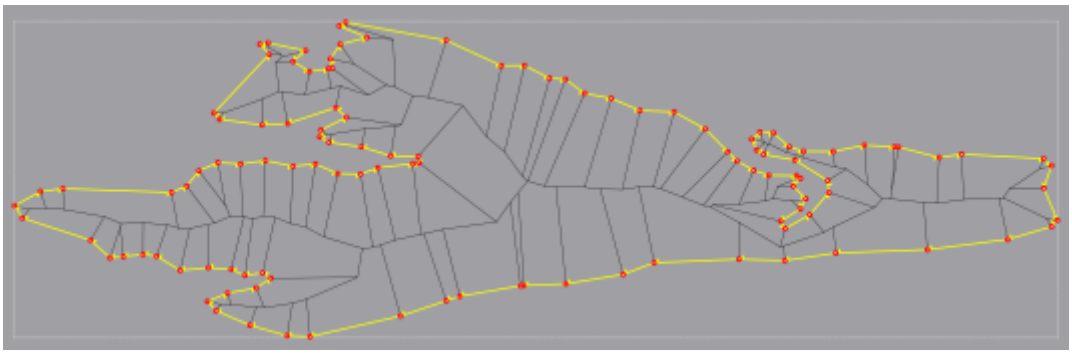
[6] S.-W. Cheng and A. Vigneron. Motorcycle Graphs and Straight Skeletons. *Algorithmica*,47:159–182, Feb 2007

[7] P. Felkel and Š. Obdržálek. Improvement of Oliva's Algorithm for Surface Reconstruction from Contours. In *Proc. 15th Spring Conf. Comput. Graphics*, pages 254–263, Budmerice, Slovakia, Apr 1999.

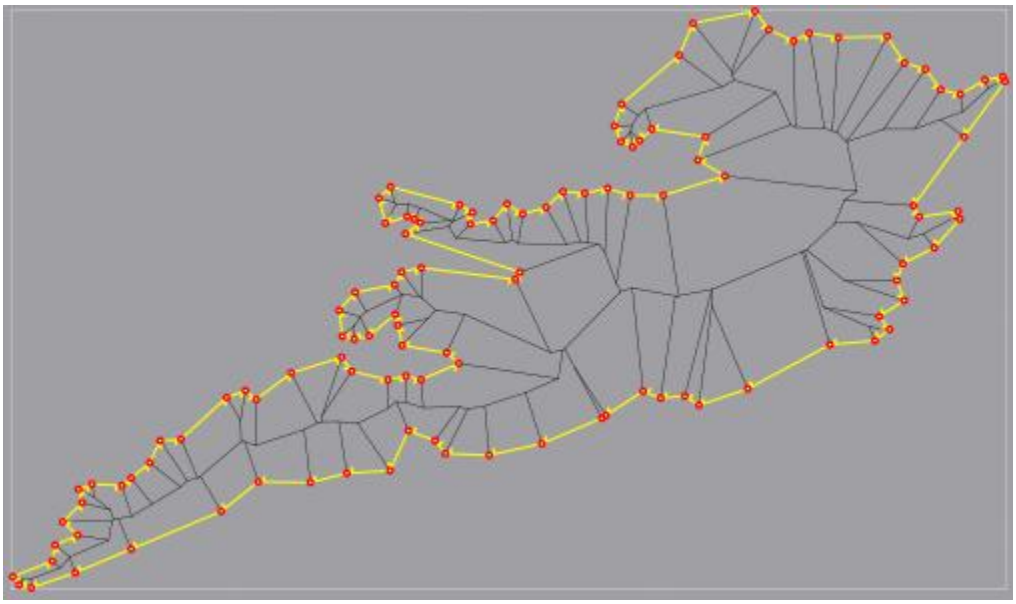
[8] S. Huber and M. Held. Theoretical and Practical Results on Straight Skeletons of Planar Straight-Line Graphs. In *Proc. 27th Annu. ACM Sympos. Comput. Geom.*, Paris, France, to be published 2011

[9] F. Cacciola. A CGAL Implementation of the Straight Skeleton of a Simple 2D Polygon with Holes. In *2nd CGAL User Workshop*, Polytechnic Univ., Brooklyn,New York, USA, June 2004

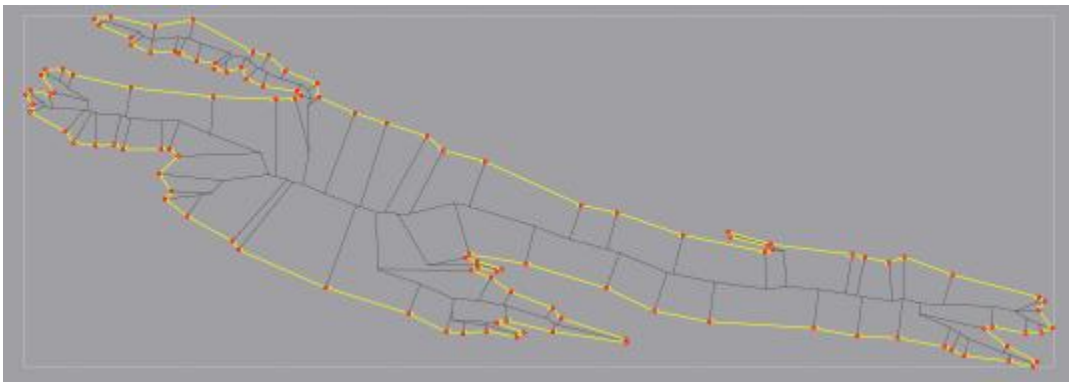
附录 A:直骨架生成结果



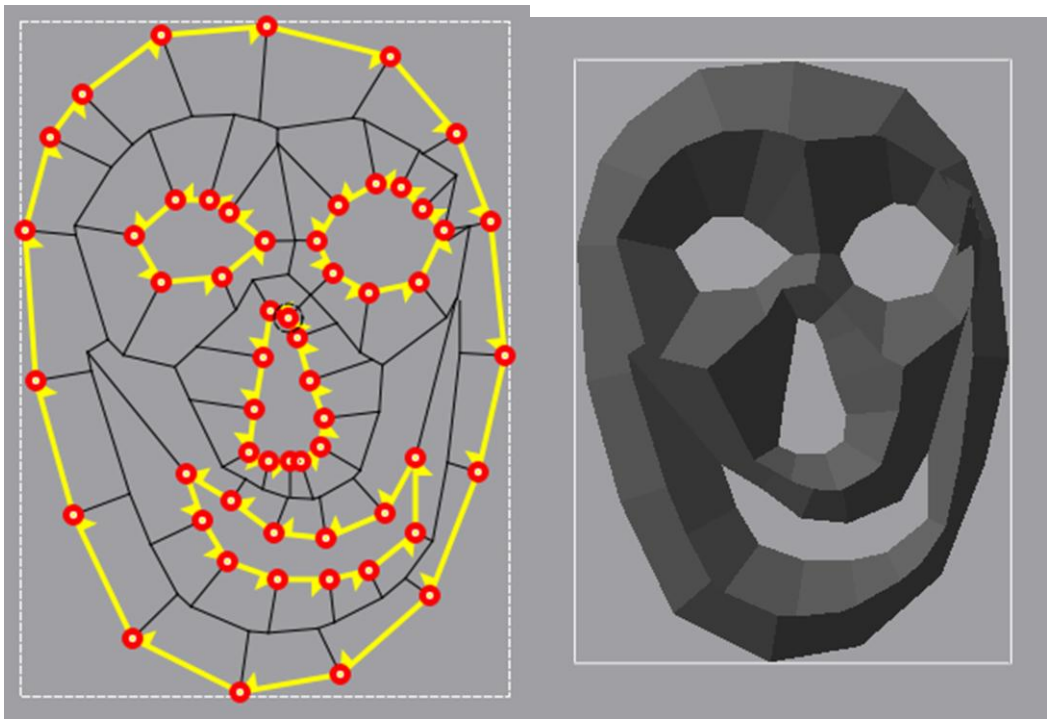
A: 复杂多边形(数据源自 CGAL，验证本算法正确性)



B: 复杂多边形(数据源自 CGAL，验证本算法正确性)

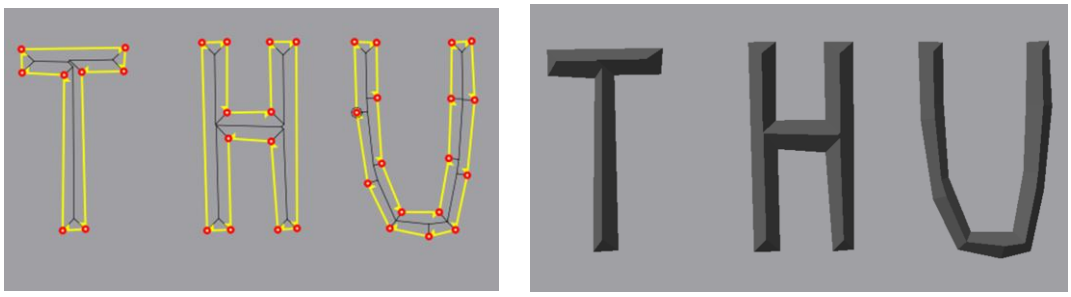


C: 复杂多边形(数据源自 CGAL，验证本算法正确性)



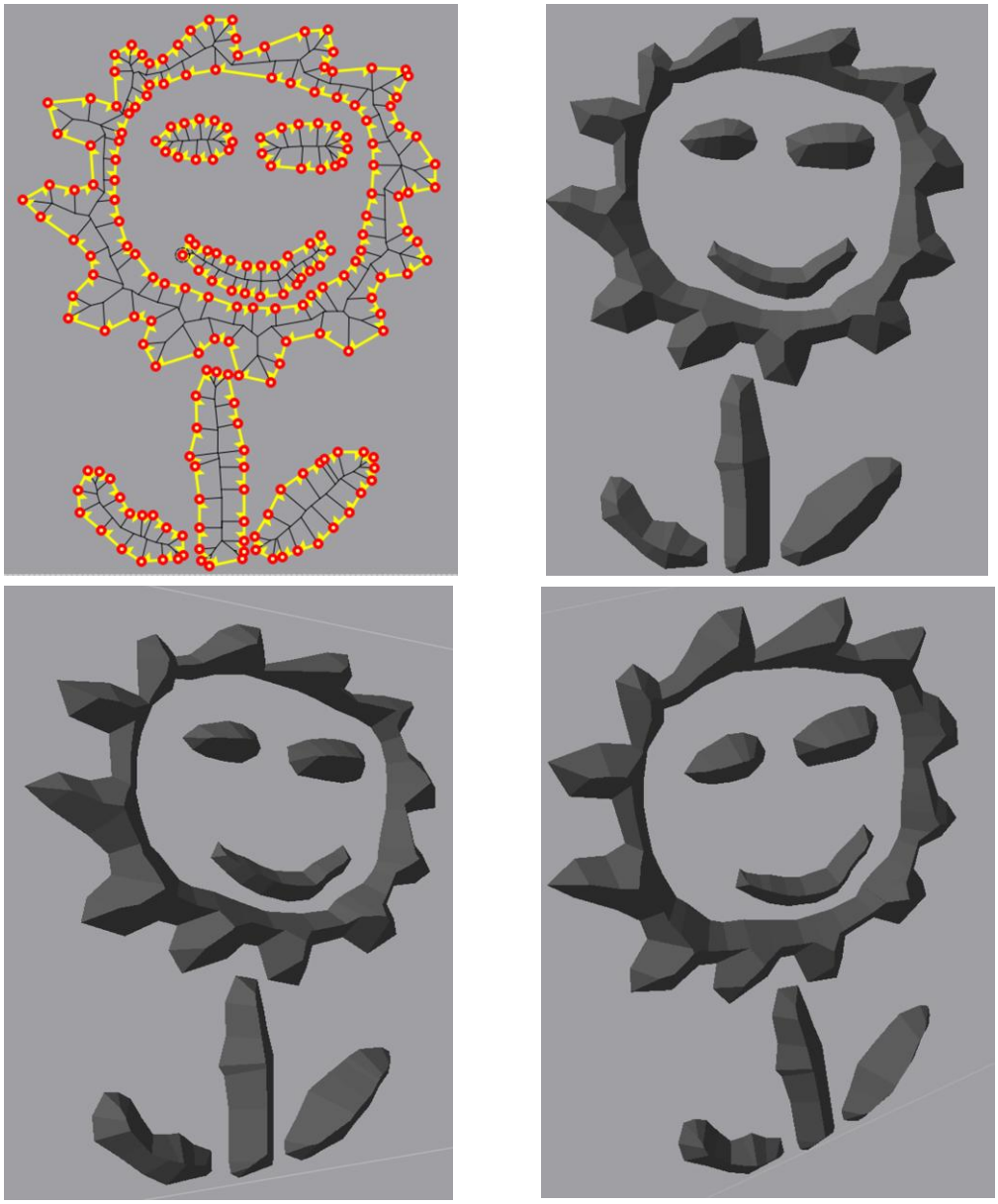
D：利用直骨架进行建模(人脸)

该输入多边形包含内环



E：利用直骨架进行建模(字符建模：THU)

很多 CAD 软件中的字符均需要以直骨架形式存储，本算法可被应用于该领域



F：利用直骨架进行建模(花的微笑)

输入带多个多边形，且多边形包含内环。

从不同角度查看三维模型