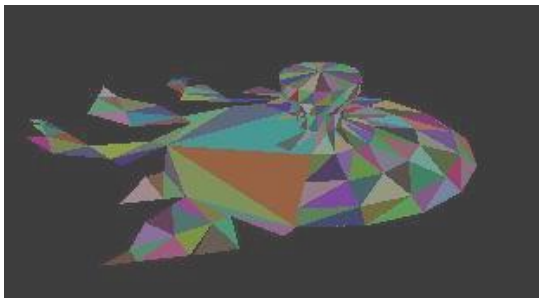
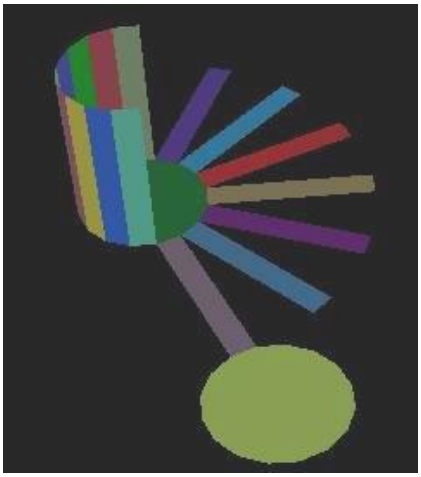


Unfolding Problem

李召荣

吴宇钦

方乐



目录

Unfolding Problem

一·问题描述与算法前提：

Folding 和 Unfolding 是近年来计算几何中一个很有趣的问题。我们的问题描述如下：

任意给定一个空间连通的二维多边形的集合（物体），只要满足一些条件，就可以用一个平面简单多边形通过有限次折叠得到。从空间到平面的这一反变换过程，可以称为 Unfolding。我们的任务，就是实现这一过程，对空间物体给出相应的平面简单多边形和折叠痕迹。

在 Unfolding 中，这类问题可以称为 Polygons Unfolding，因为构成物体的每个多边形都是不可拆分的。文献[4]中，对这种问题的算法复杂度做了分析，指出复杂度不能用多项式形式或指数形式表达。

利用穷举法能获得一个物体所有可能展开的方案，在后面我们将对它的复杂度进行分析。但即使使用穷举法，某些物体也是不可展开的，我们称之为不可展（Ununfoldable）。很早就有人研究物体的可展性。对于光滑曲面的可展性，可以根据微分几何的知识进行判断（文献[6]）。而对于一般物体，也存在很多反例。如文献[2]，[5]，[7]，分别从几个方面阐述了不可展的情况并举了反例。

首先，如果物体有凹多边形面，则很可能是不可展的。如图 1.1。由于凹多边形给分析增加了很多困难，在具体的计算中往往都对面做了限制，必须为凸多边形。在本文中，采用一种凹多边形分割的算法，将输入的凹多边形分割成若干凸多边形进行计算。

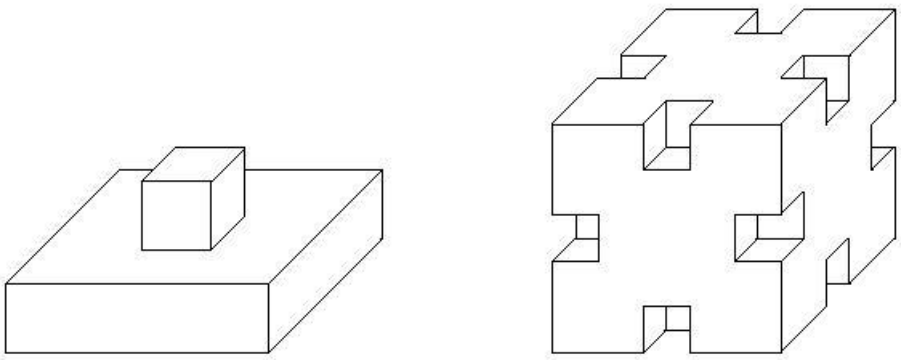


图 1.1　不可展类型 A（凹面）

在保证所有面是凸多边形的基础上，文献[1]指出可以 Unfold 的物体要满足的三个条件：

1）不超过两个面共享一条边；

2）相邻面方向性一致；

3）面不可重叠，也不可交叉。

其中，相邻面的方向性一致意味着可以对整个空间物体定义“外面”和“内面”，并给每个面一个法向量。

但这三个条件并不是充要条件。在文献[5]中，对三角形面包围的空间体可展性做了分析。即使是在这种最简单的情形下，可展性也不一定能保证。如图 1.2 举出了两个不可展的

例子，可以称它们的形状为“尖刺”。

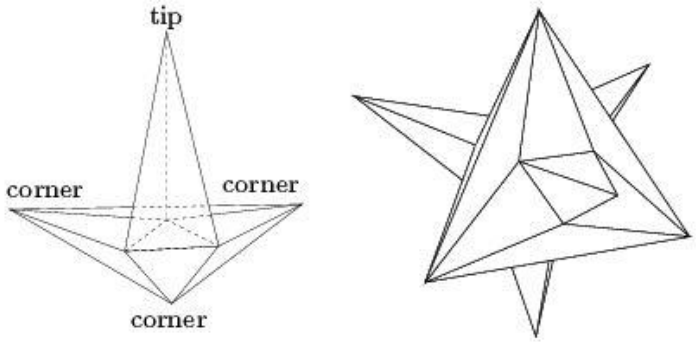


图 1.2　不可展类型 B（尖刺）

如果面都是凸的，还有一种不可展的情况，如图 1.3，可以称这种形状为“凹体”。当没有下底面时，图中的凹体是无法展开的。

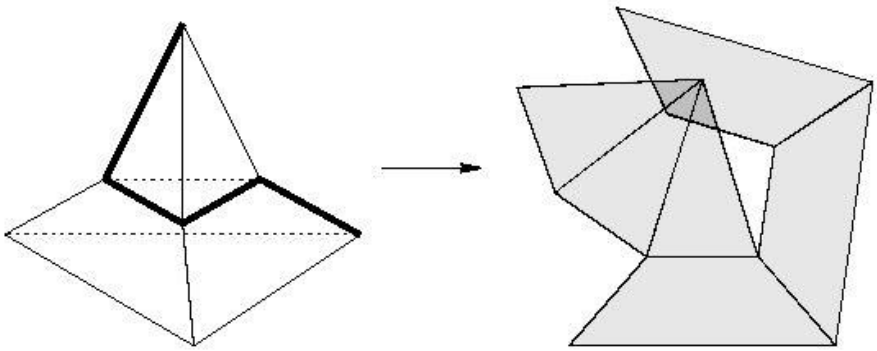
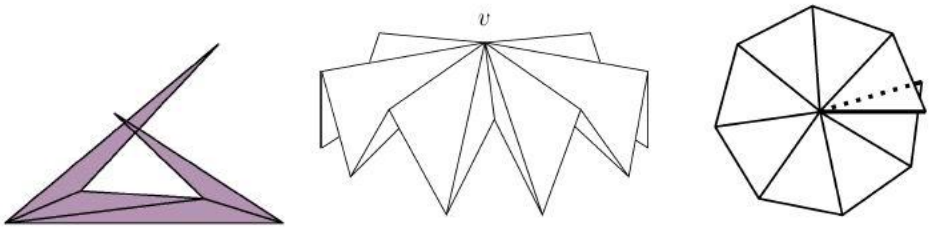


图 1.3　不可展类型 C（凹体）

上面举的例子中，每个面都与多个面相邻。如果每个面都只和最多两个面相邻，那么可称之为条带（Band）。文献[7]中对条带的可展性做了详尽的分析，并给出了算法。如图 1.4（A）在空间可以视为螺旋带，但要展到平面上却是不可能的。图 1.4（B）给出了另一个更直观的例子。总的说来，条带的展开方式比较单一，一旦发生冲突（collision）就很难调整。

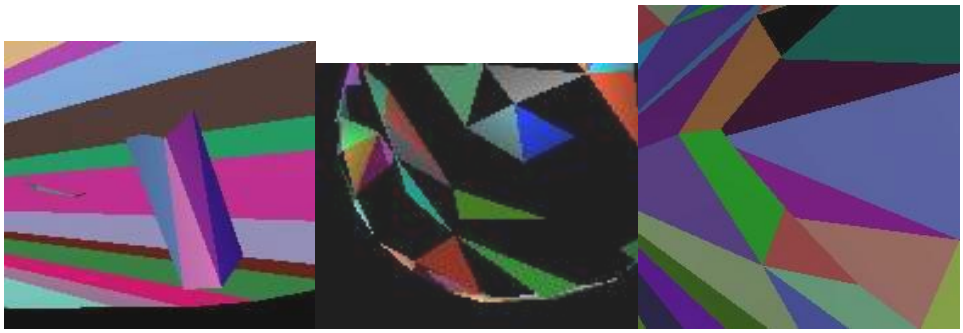


（A）　　（B）

图 1.4　不可展类型 D（条带）

我们的程序支持 3DS 文件，但并非所有的 3DS 文件都能展开。但在 3DS 文件的标准中，

面并不需要满足上面的性质，所以出现了很多由于面的划分不好而不可展的例子。如图 1.5 就是几种不可展情形的例子。



（A）不同 Object 间无相邻边　　（B）孤立面（体）　　（C）相邻边不满足条件

图 1.5 3DS 文件中不满足可展条件的情况

我们的程序假定物体是可展的并试图做 Unfolding。如果碰到不可展的情况，在深度搜索和组合算法中会穷举完所有情况，而启发参数法会一直迭代搜索下去。

二．数据结构：

空间物体和平面图形的表示方式是需要考虑的主要数据结构。

在空间采用 DCEL 储存各个面的信息。按照前面的条件给每个面定义一个法向量，按照右手定则确定边的旋向。

对于平面图形（Folding net），我们关心的就是面之间的相邻关系。用一般的树来表示这一关系。

另外，为了显示和计算求交的方便，对于平面展开的图形也需要用一组 DCEL 结构来储存信息。为了描述平面图形的边界，定义平面展开图形的外部也为一个多边形。

详细的数据结果见附录。

三．核心算法：

1．组合枚举算法

该算法利用组合数学里的一个定理（Cayley）：n 个有标号 1，2，…，n 的顶点的树的数目等于 n^{n-2}。如果把我们的物体的面看作是标有序号的顶点，则原来的物体就可以看成是一个连通图，展开算法就是将一个连通图展开成一棵生成树，要求面与面不能重合。因此我们的问题就可以对应到该定理上来。该定理的一个构造型证明过程实质上给出了我们问题的一个算法。

采用一一对应的办法来证明，证明如下：

假定 T 是其中的一棵树，树叶中标号最小者设为 a_1，a_1 的邻接点为 b_1。消去点 a_1 和

边(a_1,b_1)，点b_1便成为余下的树的树叶。在余下的树中继续寻找标号最小的树叶，设为a_2，a_2以边(a_2,b_2)与b_2想连接，再消去a_2及边(a_2,b_2)。继续以上的步骤 n-2 次，直到最后剩下一条边为止。于是一棵树对应一个序列

$$b_1,b_2,...,b_{n-2}$$

$b_1,b_2,...,b_{n-2}$是 1 到 n 中的数，且并不是一定不相同。

反之，任给一个序列

$$b_1,b_2,...,b_{n-2} \qquad (1)$$

其中$1\le b_i \le n, i=1,2,...,n-2$，便可找到与之对应的树$T$，方法如下：从序列

$$1,2,...,n \qquad (2)$$

中找到第一个不出现在$b_1,b_2,...,b_{n-2}$中的数，这个数显然就是a_1，同时找出树T 的边(a_1,b_1)。从序列（1）和序列（2）中分别消去b_1和a_1，在余下的序列中继续以上的步骤 n-2 次，直到序列（1）为空为止。这时序列（2）中剩下的两个数设为a_k和b_k。边(a_k,b_k)也是树T的最后一条边。于是便得到树T。

这就证明了 n 个标号顶点的树和 n-2 个数

$$b_1,b_2,...,b_{n-2}$$

一一对应，其中$1\le b_i \le n, i=1,2,...,n-2$。根据乘法法则，序列（1）的数目为$n^{n-2}$，定理得证。

借用该证明过程的后半部分得到的算法实现程序见：

```
void CUnfoldProcess::UnfoldPermu()
```

算法时间复杂度为$O(n^{n-2})$,空间复杂度为$O(n)$

2．深度优先算法

采用回溯策略得到的深度优先算法框架如下：

```
Algorithm   UnfoldDFS
    for 所有面 A
        加 A 进 foldingNet 中
        UnfoldStep(foldingNet,1)
    end for
```

```
end

Subroutine UnfoldStep(foldingNet,m)
    if m = 物体面数
        return
    for foldingNet 的所有边界边
        找到可加入到此处的面 F
        判断在该边界边处加入面 F，是否与 foldingNet 相交
        if  不相交
            将该面加入 foldingNet;
            更新可通过 foldingNet 的边界边加入到 foldingNet 中面的集合;
            UnfoldStep(foldingNet,m+1)
            将该面从 foldingNet 中拿走
        end if
    end for
end
```

该算法的时间复杂度约为指数级（$O(n^n)$），空间复杂度为$O(n)$

该算法实现程序见：

```
void CUnfoldProcess::UnfoldDFS()
BOOL CUnfoldProcess::UnfoldStep(CFaceList &faceList, int m)
```

3．物空间算法

以上两个算法的时间复杂度都高达指数级，当面数较大时，就不能实用，因而只有理论上的意义。因而寻求启发式搜索算法，下面的物空间算法就利用到了某些启发信息。

该算法通过对不断改变加入面的顺序，将多面体物体拆分成一张 folding net,加入的面要求与与展开的 folding net 邻接，但不能相交。该算法利用一个启发函数来指导面的加入顺序，思想是：将碰撞面尽量放到数的叶子结点中，因为直观上，叶子处空间更大，因而碰撞的机会更小；同时将怎么放也放不下的面尽量往根部放。

在加入面到 foldingNet 过程中，总是选取权值最小的面。因此，算法朝叶子移动的过程中，权值递增。在每次试着生成一个 foldingNet 的之后，更新面的启发因子（定义如下）。

- FactorA：
  该面作为发生碰撞的两个面的共同祖先的次数。
- FactorB：
  该面已经在 foldingNet 中，在加入其它面的过程中，与之碰撞的次数
- FactorC：
  该面在加入 foldingNet 中，与 foldingNet 中面相交的次数。

启发函数的定义则有多种模式，孰优孰劣不好下结论。经验上，比较好的几种列在下面，其中 a,b,c 分别为 FactorA，FactorB，FactorC 的权重，也是一些经验值。

- Scheme1

$$Pri_{face} \leftarrow Pri_{face} + a \times FactorA + b \times FactorB - c \times FactorC$$

该模式每次循环允许权值发生大的变化，因而理论上能够很快接近目标。缺点是可能 overshoot。

- Scheme2

$$Pri_{face} \leftarrow Pri_{face} + a \times \begin{cases} 0 & if\ FactorA = 0 \\ 1 & if\ FactorA > 0 \end{cases} + b \times \begin{cases} 0 & if\ FactorB = 0 \\ 1 & if\ FactorB > 0 \end{cases} - c \times \begin{cases} 0 & if\ FactorC = 0 \\ 1 & if\ FactorC > 0 \end{cases}$$

该模式只允许权值每次每个权因子变换 1 个单位，因而理论上比较慢才能接近目标，但是能够更彻底地尝试不同的排列。

- Scheme3 and 4

分别对应 Scheme1 和 Scheme2，只不过只对那些怎么放也放不下的面更新权值时考虑 FactorC。

算法框架如下：

```
Repeat
    清除当前的 foldingNet
    根据某种策略选取一个初始面，将它加入到 foldingNet 中
    while 存在可以加到 foldingNet 中的面，它与 foldingNet 中的某个面临接
          在这些面中选择一个权值最小的面
          计算该面的投影
          if 该面的投影与 foldingNet 相交 then
                更新 FactorA，FactorB，FactorC
                将该面放回 unplaced faces 表中
                将该面的可加入位置计数减 1
           else
                将该面加入到 foldingNet 中
                同时更新能够加入到 foldingNet 中的面的集合
          endif
    endwhile
    根据计算得到的 Factors 来更新面的权值
until 原物体的所有面都加入到 foldingNet 中
输出 foldingNet
```

该算实现程序见：

```
void CUnfoldProcess::Unfold()
```

四．细节算法

（1）判断多边形是否相交

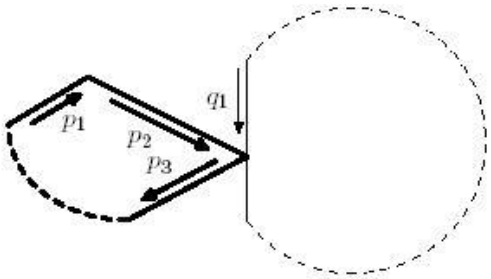


图 4.1.1

对于两个多边形是否相交，我们采用自己写的算法，对每两条边比较分析，一旦判断相交即退出算法。如果遍历完所有的边没检测到相交，则不相交。算法复杂度是 m*n，其中 m 和 n 分别是两个多边形的边数目。但由于新加入 folding net 的多边形边数不会太多（即原物体中每个面的边数目不会太多），算法复杂度可以认为是线性的。

输入：多边形 P 和 Q

输出：是否相交（必须有面积覆盖的部分才算相交，如图 4.1.1 和图 4.1.7 都是不相交的例子。）

预处理：对 P 和 Q 分别构造 DECL 结构，取 P 的内边和 Q 的外边。

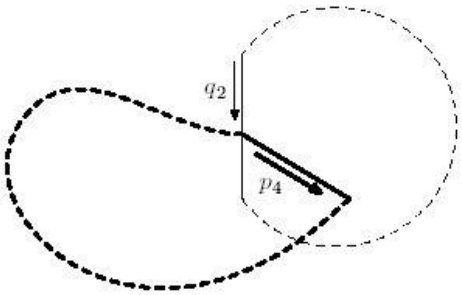


图 4.1.2

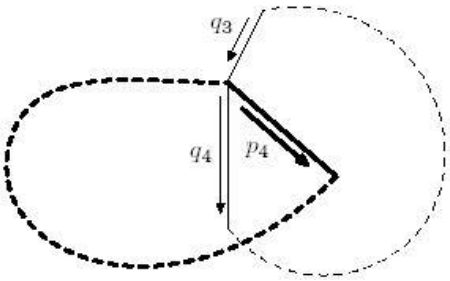


图 4.1.3

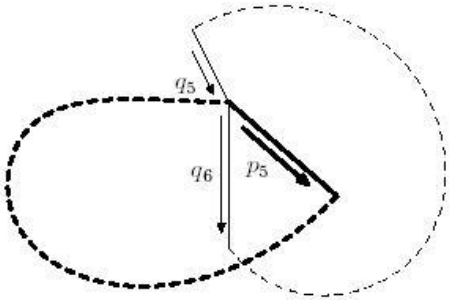


图 4.1.4

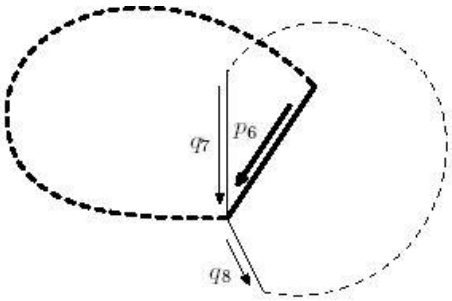


图 4.1.5

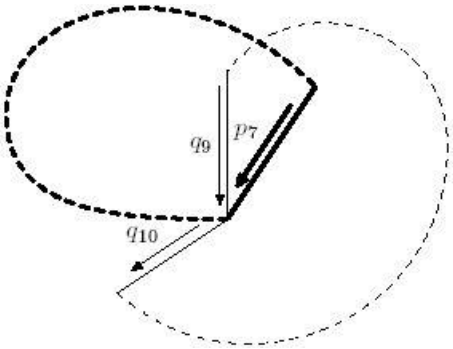


图 4.1.6

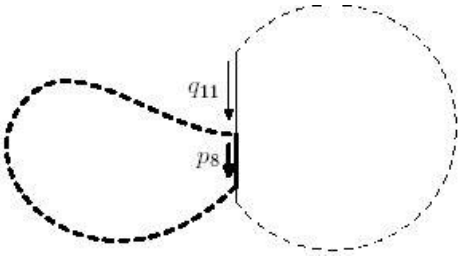


图 4.1.7

算法：

for each (p in P), (q in Q) 记 p、q 的起始点为 A_1，B_1，p、q 的结束点为 A_2，B_2,

if p 和 q 相交不在端点 then return 相交

if (仅相交于 A_1 且 A_2 位于 q 的左边)

或 (仅相交于 B_1 且 B_2 位于 q 的右边)

then return 相交 (图 4.1.2)

if (仅相交于 A_1 和 B_2 且 q 与 q->next 构成凸角 且 A_2 位于 q 和 q->next 的左边)

或 （q 相对与 p，类似的描述）

then retuen 相交（图 4.1.3）

if (仅相交于 A_1 和 B_2 且 q 与 q->next 构成凹角 且 A_2 位于 q 或 q->next 的左边)

或 （q 相对与 p，类似的描述）

then retuen 相交（图 4.1.4）

if (仅相交于 A_2 和 B_2 且 q 与 q->next 构成凸角 且 A_1 位于 q 和 q->next 的左边)

或 （q 相对与 p，类似的描述）

then retuen 相交（图 4.1.5）

if (仅相交于 A_2 和 B_2 且 q 与 q->next 构成凹角 且 A_1 位于 q 或 q->next 的左边)

或 （q 相对与 p，类似的描述）

then retuen 相交（图 4.1.6）

（2）最小包围圆

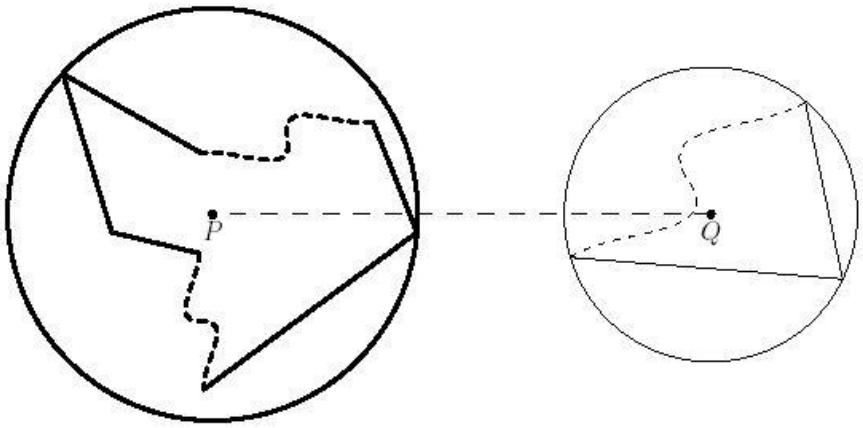


图 4.2 最小包围圆示意

为了减少多边形求交的运算次数，采用了最小包围圆算法，在每个多边形投影到平面时计算其最小包围圆。当需要判断多边形相交时，先考虑如果两个多边形对应的包围圆的圆心距离大于半径的和，则直接判断不相交。

采用文献[9]中讲述的随机增量法计算最小包围圆，第 125 页。

（3）凹多边形的凸划分

见文献[8]P54~55。本文对原算法做了一些修改。

凹多边形的凸划分算法

输入：多边形 P 的顶点 $p_1, p_2, \dots, p_n$ 按逆时针顺序排列（本程序中为空间多边形，要根据法向量来确定时针序）。

输出：凸多边形序列 W_i 及其顶点序列 $p_{11}, p_{12}, \dots, p_{1v1}$； $p_{21}, p_{22}, \dots, p_{2v2}$； … ； $p_{m1}, p_{m2}, \dots, p_{mvm}$。

约定：如图 4.3 方式确定每个凹点的 A、B、C 域。

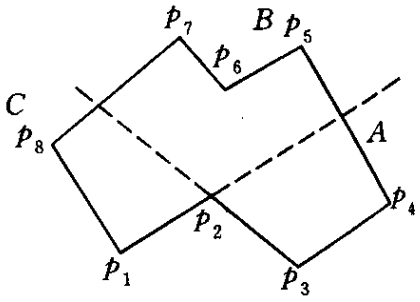


图 4.3　凹点的临域划分示意

预处理：

确定多边形 P 的凸凹顶点，并对凹点重新编号（逆时针），设为 $Q=\{q_1(=p_k), q_2, ..., q_l\}$

Call PPCP(P)

Subroutine PPCP(P) 凸划分的递归程序

if P 中无凹点 then return

在当前多边形 P 中找一个凹点 q_i，如 i=1

if q_i 的 B 域中无 P 的顶点 & P 边 $\overline{p'p''}$ 的两端点分别位于 q_i 的 A、C 域中 & $\overline{p'q_i}$、$\overline{p''q_i}$ 不交多边形的边 then

连接 p' 与 q_i，p'' 与 q_i，$\overline{p'q_i}$、$\overline{p''q_i}$ 分割 P 成三角形 $p'q_ip''$ 及两个多边形 P_1 和 P_2，Q:=Q-$\{q_i\}$，重排 P_1 和 P_2 中凹点编号。

Call PPCP(P_1)，PPCP(P_2)

else if q_i 的 B 域中无 P 的顶点 & q_i 的 A(C)域中有凹点 $q_{i'}$ & $\overline{q_iq_{i'}}$ 不交多边形的边 then

连接 $q_{i'}$ 与 q_i，$\overline{q_{i'}q_i}$ 分割 P 为 P_1 和 P_2，重排 P_1、P_2 中凹点编号。

Call PPCP(P_1)，PPCP(P_2)

else if q_i 的 B 域中有凹点 $q_{i'}$ & $\overline{q_i q_{i'}}$ 不交多边形的边 then

连接 $q_{i'}$ 与 q_i， $\overline{q_{i'} q_i}$ 分割 P 为 P_1 和 P_2，重排 P_1、P_2 中凹点编号。

Call PPCP(P_1)，PPCP(P_2)

else if q_i 的 B 域中有凸点 p' & $\overline{q_i p'}$ 不交多边形的边 then

连接 p' 与 q_i， $\overline{p' q_i}$ 分割 P 为 P_1 和 P_2，重排 P_1、P_2 中凹点编号。

Call PPCP(P_1)，PPCP(P_2)

End Subroutine

这里对周培德老师原来的算法做了一点改动，主要是因为原来的算法也许对某些特殊情形无法处理。已给周老师发了邮件询问，但未收到答复。

五．算例与结果分析

测试环境：P4 2.0G，1G RAM，Windows XP Professional。
注意：以下所有测试均为关闭“实时显示”后的结果。

1．三种算法的比较

因为三种算法没有统一的比较标准，所以执行比较它们的执行时间。

批注 [Jordan1]:
展开的效果？

模型	面数目	组合算法	深度搜索	物空间(参数 1 1 1)
立方体改.txt	11	26 秒	<1 秒	<1 秒
凹棒.txt	26	>2 小时	56 秒	<1 秒

如果模型更复杂，组合算法和深度搜索的执行时间都会急剧增大。大致说来：物空间算法在执行时间上远优于组合算法和深度搜索。而深度搜索可以认为是组合算法的优化版，因为它去除了很多不连通的无用组合。

2．物空间算法研究

（1）不同 Scheme 算例

三个数字是算法中权值 a,b,c。括号中的数字是循环迭代的次数。加有[d]的算例表示经过了面合并。

模型	Scheme 1	Scheme 2	Scheme 3	Scheme 4
CHAMPAGN.3DS	0 4 1 (15)[d] 0 4 1 (4175) 1 2 1(42)[d]	0 4 1(25)[d] 0 4 1(143)	1 4 3(32)[d] 1 4 3(104)	0 3 1(21)[d] 0 3 1(871)
茶壶盖.3DS	0 4 1(12)[d] 0 4 1(32)	0 4 1(18)[d] 0 4 1(68)	1 4 3(20)[d] 1 4 3(26)	0 3 1(11)[d] 0 3 1(20)
细杯子.txt	0 4 1(772)	0 4 1(9876)	1 4 3(38607)	0 3 1(665)

在选择模型上碰到了很多问题，一般的 3DS 模型大多是不可展的，而如果可展的又一般过于简单。最终这里选择了一个酒瓶和一个茶壶盖作为算例。

总体说来：

合并面后的结果比不合并要好。

不同模式之间不存在谁好谁差的问题，只是对于某个特定的算例，在模式选择上有区别。一般说来模式的选择和参数的选择对迭代次数影响还是比较大的。文献[1]中推荐了一些参数，经测试确实有相对比较好的效果。

这里关于合并面要特别提起一个问题，就是是否存在某个模型，在合并面之前是可以展开的，而合并了之后就不能展开？这是肯定存在的，可以想象对合并面之后的模型做后向操作，“反合并”的操作实际上就是对面做裁减，因此我们的 Polygons Unfolding 问题就变成了一般意义上的 Unfolding 问题。但是在我们的程序中对合并面做了一些限制，在合并的过程中不允许出现凹多边形的面。在这个前提下，找一个这样的例子很不容易。

（2）算法收敛研究

为了解释这个算法的收敛实质，需要对每次迭代，搜索队列的变化过程做跟踪分析。

以“细杯子.txt”为例，分析这个变化过程。每个面 k 在每个迭代步 i 中都对应一个优先值 pri(i,k)，这个优先值直接影响搜索队列。因此我们分析优先值的变化。若第 m 步得出展开解，那么定义某个迭代步 i 的优先值平均差值为

$$\overline{pri}(i)=\sqrt{\frac{\sum_{k}(pri(i,k))^{2}}{k}}$$

对两个不同模式参数时时的结果如下图所示。可以观察到模式和参数选取的不同仅影响到收敛的速率。而对于 Scheme 1，由于模式本身允许每次迭代可以有较大的变化，所以变化的速率也剧烈一些，这是和分析符合的。

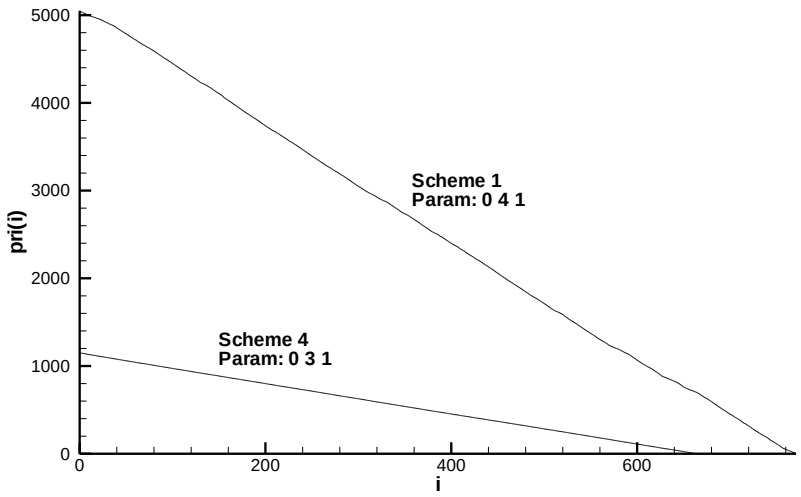


图 5.1 优先度平均差值变化

但这样的分析还不够直观，因为优先度和搜索队列之间还是有一定的差别。所以我们直接对搜索队列进行收敛分析。给每个迭代步 i 中的优先值数组 pri(i,k)排序，得到一个大小序列 sort(i,k)，表示面 k 的 pri 值在整个数组中的大小序号。这也就等于给出了搜索队列。同样定义序列平均差值为

$$\overline{sort}(i)=\sqrt{\frac{\sum_{k}(sort(i,k))^{2}}{k}}$$

同样两个不同模式参数的结果如下图。这更清楚地证明了迭代有趋于收敛的性质。但收敛速度和收敛形式和模式参数都有关。如 Scheme 1 允许每次迭代可以有较大的变化，所以在开始的一些迭代步中收敛非常明显。但这样大的变化对找到合适的解帮助却不大，因此在很长一段时间内收敛速度很慢。相对的 Scheme 4 则是比较稳妥的方法。当然，这和算例本身也有联系。

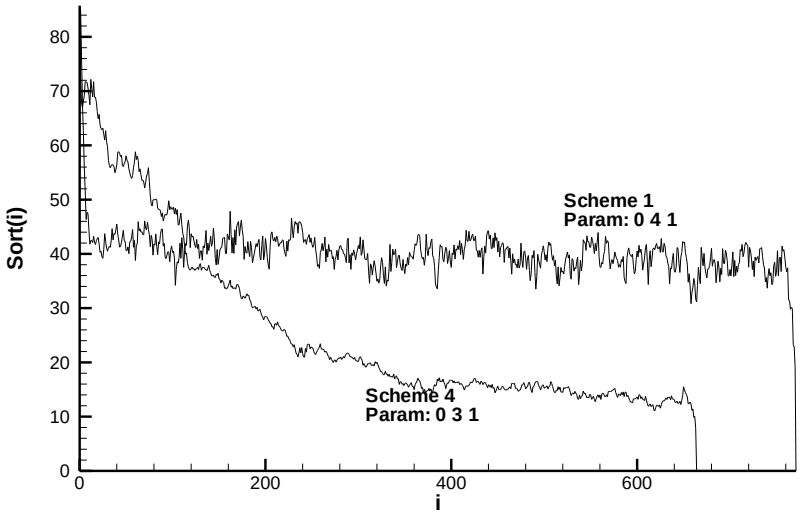


图 5.2 序列平均差值变化

由于时间有限，没有对各个模式和不同参数进行结果的测试分析。

六．程序的用户界面

为了使 unfolding 的算法更容易被检测和理解，我们程序实现以下三方面功能：

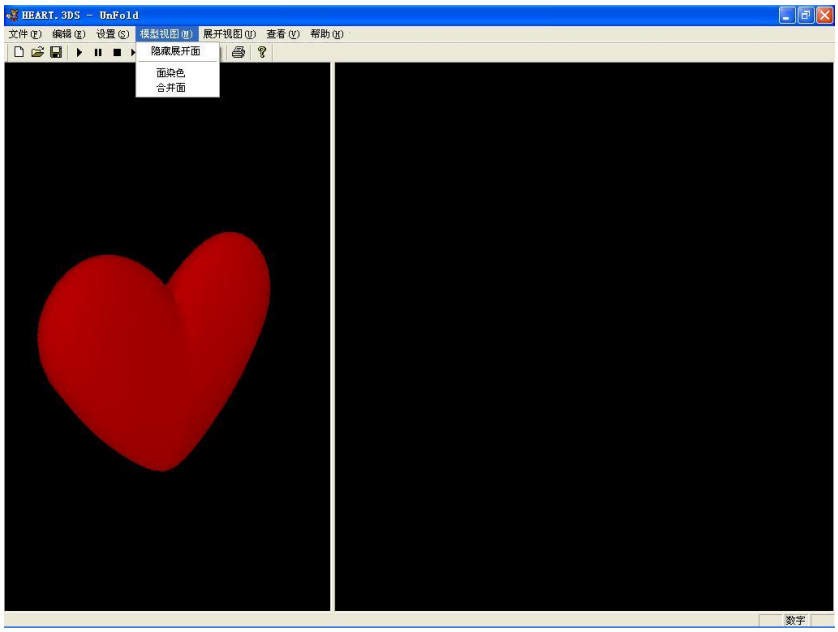


图 6.1 用户界面

（1）支持常用 3D 模型文件的读取：3ds 和 obj（ASCII），以及用户自己按一定格式编写的 txt 文档；

txt 自编写文档格式：

每行以关键字开头，后接相应数据。

关键字：v　顶点数据　　数据：3 个数表示 xyz 坐标

f　面数据　　数据：一个表示顶点数的数字后接各顶点编号

支持纹理坐标和面材质颜色

vt　顶点数据　　数据：顶点空间和纹理坐标

mc　面颜色数据　数据：一个表示对应面数后接各 rgb 值

（2）简单的三维交互功能

添加交互功能，使得对三维的观察更直接。功能简介：

左键拖动：旋转物体；

右键拖动：缩放物体；

shift＋左键拖动：平移物体；

左键双击：面显示和边显示切换；

右键双击：光照切换；

shift＋右键双击：单面显示和双面显示切换

（3）unfolding 结果的简单演示功能（如图 6.2）。

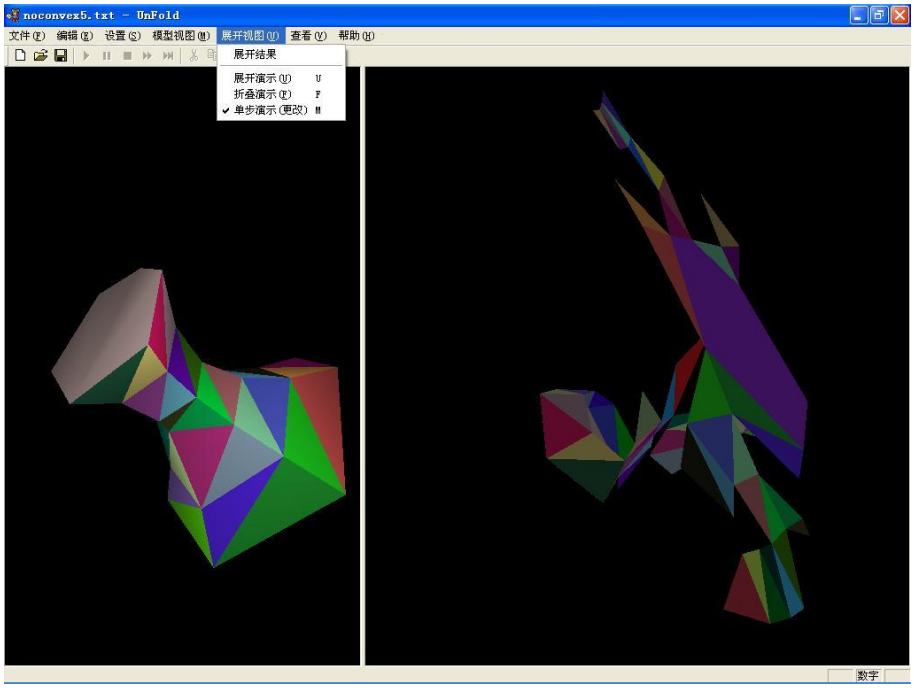


图 6.2 折叠演示

对 unfolding 结果得到的面的树结构，按深度方向进行逐步折叠，折叠的角度从原来 3D 模型各个面的法向量计算得到。

展开是折叠的逆过程。

可选单步、快速和慢速 3 种模式。

七．其他想说的

终于忙完了 ^_^

这个 project 的工作量远远超乎我们最初的想象。每个人都写了大量的程序，而且对于经常出现的 bug 也是绞尽了脑汁。为了加入控制单步，暂停等功能，采取了多线程的方式，但随之而来的是经常出现的冲突。即使加了互斥语句，长时间的实时显示也会引发一些错误。因为刚开始老师对我们的工作难度评价不高，所以大家都尽力往程序里面添加新的功能。首先是深度搜索和组合算法，然后是折叠和展开的演示，在算法上又增加了最小包围圈和凹多边形分割的部分。但最关键的问题还在于找不到很好的测试数据。我们自己编程画的模型往往太规则，而 3DS 文件满足可展条件且足够复杂的又非常难找到。因此最大规模的计算也就是那个细杯子模型中的一个算例，迭代了 38607 次，花了两天时间。

小组的合作很愉快，李召荣负责核心算法部分，吴宇钦负责显示部分，方乐负责其他细节算法以及算例测试。尤其对于我们两个外系的学生来说是很大的考验。

谢谢邓老师给我们的指导！

附录：算法部分用到的数据结构

```
struct vector2
{
    float x,y;
};
struct vector3
{
    float x,y,z;
}

//--------------------------------
//  顶点类
//--------------------------------
class Vertex
{
public:
    Vertex();
    ~Vertex();
public:
    //顶点标号
    int id;

    //顶点坐标
    float x,y,z;

    //顶点法向
    vector3 norm;

    //该顶点关联的边
    Edge* inc_edge;
};

//--------------------------------
//  边类
//--------------------------------
class Edge
{
public:
    Edge();
    ~Edge();
public:
    //顶点标号
    int id;

    //孪生边
    Edge* twin;

    //起始顶点
    Vertex* org;

    //纹理
```

```
    vector2* texture;

    //改边与下一条边的夹角
    float arc;

    //边长
    float length;

    //邻接面
    Face* leftFace;

    //前一条边
    Edge* prev;

    //后一条边
    Edge* next;

    //当改边是foldingNet中的边时，才会用到。指向与 Object 中邻接的//边
    Edge *fitEdge;
};

//------------------------------
//  面类
//------------------------------
class Face
{
public:
    Face();
    ~Face();
public:

    //面标号
    int id;

    //若该面为普通面，指向任意一条与该面邻接的边；否则为NULL
    Edge* inc_edge;

    //若该面为边界面，指向任意一条与该面邻接的边；否则为NULL
    Edge* inner_edge;

    //面法向量
    vector3 norm;

    //标识该面是否可见
    BOOL bVisible;

    //材质
    int materialID;

    //最小包围圆的圆心
    Vertex center;

    //最小包围圆的半径
    float radius;

public:

    //生成该面的最小包围圆
    int initMiniDisc( int if2D = 1 );
};

//------------------------------
//  DCEL类
//------------------------------
class DCEL
{
public:
    DCEL();
    virtual ~DCEL();
public:

    //置空
    void MakeEmpty();

    //加入一个面
    void AddFace(Face *f);

    //加入一条边
    void AddEdge(Edge *e);

    //加入一个顶点
    void AddVertex(Vertex *v);
```

```
    //删除一个顶点
    void DeleteVertex(Vertex *v);
    //删除一条边
    void DeleteEdge(Edge *e);

    //删除一个面
    void DeleteFace(Edge *e);

    //拷贝函数
    DCEL* Copy();

    //清除已展开标志
    void EraseUnfold();

    //绘制
    void Draw(CDC *pDC);

    //检查是否满足 DCEL 约束
    void Check();

    //判断是否为空
    BOOL IsEmpty();

    //合并 DCEL；fittedEdge 为相邻
边
    void  Unite(DCEL  *proj,Edge
*fittedEdge);

    //判断两个 DCEL 是否相交
    BOOL Intersect(DCEL *proj);

    //顶点数
    int  numOfVerts;

    //边数
    int  numOfEdges;

    //面数
    int  numOfFaces;

    //显示在屏幕上的起始位置
    int orginX,orginY;

    //纹理列表
    vector <vector2*> textureList;

    //顶点表
    vector <Vertex*> vertList;
    //边表
    vector <Edge*> edgeList;

    //面表
    vector <Face*> faceList;
};

//--------------------------------
//  面节点类，定义 CFaceList 的节点
//--------------------------------
class CFaceNode
{
public:
    CFaceNode();
    CFaceNode(Face* f);
    ~CFaceNode();
public:

    //得到下一个节点
    CFaceNode* NextNode();

    //比较节点的权值
    BOOL  operator  <  (const
CFaceNode& fd);

    //计算权值
    void CalculatePriority();

    //计算平面投影
    DCEL* Projection();

public:

    //父节点指针
    CFaceNode *parent;

    //指向平面投影的指针
    Face* pro;

    //指向对应的面
```

```
    Face* face;

    //邻接计数，一个面可与多个面邻接
    int fitCount;

    //记录邻接面与之邻接的面
    Face* fittedFaces[MAX_INCIDENT_NUM];


    //记录邻接面与之邻接的边
    Edge* fittedEdges[MAX_INCIDENT_NUM];

    //记录与邻接面邻接的边，与fittedEdges对应
    Edge* fitEdges[MAX_INCIDENT_NUM];

    //启发因子
    int factorA,factorB,factorC;

    //权值
    int pri;

    //启发模式
    static int nScheme;

    //权重
    static int a,b,c;

    //标志该面节点是否加入到foldingNet中
    BOOL deleted;

    //下一个节点
    CFaceNode* next;
};

//--------------------------------
//  面节点链表（树），用在展开算法中
//--------------------------------
class CFaceList
{
public:
    CFaceList();
    virtual ~CFaceList();
public:

    //寻找怎么加入也加入不到foldingNet中的面
    CFaceNode* FindUnplacable();

    //更新面的启发因子
    void UpdateFaceFactors(CFaceNode *fd);

    //根据面标号得到对应的面节点
    CFaceNode*   Face2FaceNode(int fid);

    //标记可加入到foldingNet中的面，也即与foldingNet相邻的面
    void    MarkFitFaces(CFaceNode *fnd);


    //更新所有面节点的权值
    void UpdateAllPriority();

    //得到第一个面节点
    CFaceNode* GetHead();

    //判断是否所有的面都加入到foldingNet中
    BOOL IsAllDeleted();

    //从DCEL构造faceList
    void CreatFromDCEL(DCEL *obj);

    //恢复到初始状态
    void Recover();

    //删除面节点
    BOOL Delete(CFaceNode *fd);
```

```
    //获得权值最小并且还没加入但可以加入到 foldingNet 的节点
    CFaceNode* Min();

    //在头节点处插入面节点
    void AddHead(CFaceNode *fd);

    //置空
    void MakeEmpty();

private:

    //头指针和尾指针
    CFaceNode *first,*last;

    //节点数
    int fndNum;

    //存放面节点列表
    CFaceNode **f2fnd;

};

//------------------------------
//  展开处理类，主要实现 3 个展开算法
//------------------------------
class CUnfoldProcess
{
public:
    CUnfoldProcess();
    virtual ~CUnfoldProcess();
public:

    //深度优先求解的单步算法
    BOOL    UnfoldStep(CFaceList &faceList,int m);

    //深度优先算法
    void UnfoldDFS();

    //组合算法
    void UnfoldPermu();

    //启发式算法
    void Unfold();

    //设置需要用到的一些数据
    void SetOutputDCEL(DCEL *oup);
    void    SetNetView(CUFNetView *ppView);
    void    SetUFView(CUnFoldView *ppView);
    void SetInputDCEL(DCEL *inp);

private:

    //显示 foldingNet 的视图
    CUFNetView *pNetView;

    //显示展开对象的视图
    CUnFoldView *pUFView;

    //展开对象的 DCEL 表示
    DCEL* input;

    //foldingNet 的表示
    DCEL* output;
};
```

参考文献：

[1] From virtual to physical reality with paper folding, Shaun Bangay, Computational Geometry 15(2000) 161-174
[2] Unfoldable Polyhedra, Marshall Bern et al, 1999
[3] Unfolding Some Classes of Orthogonal Polyhedra, Therese Biedl et al, 1998
[4] Examples, Counterexamples, and Enumeration Results for Folding and Unfolding between Polygons and Polytopes, Erik Demaine et al, 2000
[5] Ununfoldable Polyhedra with Triangular Faces, Marshall Bern et al
[6] Unfolding Smooth Primsatoids, Nadia Benbernou et al, 2004
[7] Unfolding Polyhedral Bands, Greg Aloupis et al, 16th Canadian Conference on Computational Geometry, 2004
[8] 计算几何——算法分析与设计，周培德，清华大学出版社, 2000
[9] 计算几何——算法与应用，Mark de Berg et al