A sweep line algorithm for polygonal chain intersection and its applications

Sang C. Park, Hayong Shin**, Byoung K. Choi
*CAM Lab., Department of Industrial Engineering Korea Advanced Institute of Science & Technology (KAIST) 373-1 Kusong-dong, Yusong-gu, Taejon, Korea ** Chrysler Tech Center, Auburn Hills, Michigan, USA*

Key words: polygonal chain, intersection, sweep line algorithm

Abstract: Presented in this paper is a *sweep-line algorithm* for finding all intersections among *polygonal chains* with an $O((n + k)\cdot\log m)$ worst-case time complexity, where n is the number of line segments in the polygonal chains, k is the number of intersections, and m is the number of *monotone chains*. The proposed algorithm is based on the Bentley-Ottmann's sweep line algorithm, which finds all intersections among a collection of line segments with an $O((n + k)\cdot\log n)$ time complexity. Unlike the previous polygonal-chain intersection algorithms that are designed to handle special only cases, such as convex polygons or *C-oriented* polygons, the proposed algorithm can handle arbitrarily shaped polygonal chains having self-intersections. The algorithm has been implemented and applied to 1) testing simplicity of a polygon, 2) finding intersections among polygons and 3) offsetting planar *point-sequence* curves.

1. INTRODUCTION

A *polygonal chain*, or *chain* for short, is a connected sequence of line segments and a *polygon* is a chain which is closed (and non self-intersecting). Chain is a basic and commonly used geometric entity in many area, and finding intersections within a chain or among several chains is one of the fundamental operations in CAD/CAM as well as in computer graphics. Application examples include hidden-line removal for freeform surface display, freeform surface trimming, and point-sequence curve offsetting.

F. Kimura (ed.), *Geometric Modelling*

There are quite a few research results reported in the literature dealing with the *line-segments intersection problem*. Bentley and Ottmann[2] introduced a sweep-line algorithm to find all k intersections among n line-segments with an $O((n + k)\cdot\log n)$ time complexity. Chazelle and Edelsbrunner[4] presented an $O(n\cdot\log n + k)$ algorithm which is known to be optimal but complicated to implement. Mehlhorn and Naher[5] presented a robust implementation scheme for the Bentley and Ottmann's algorithm. There are also some research results directly dealing with the *polygonal-chains intersection problem*, but only for "special" polygon types such as rectangles[8], convex[6], simple[7], or C-oriented[9] polygons. To our best knowledge, there are no published algorithms to find intersections among general polygonal chains

The line-segments intersection algorithms may be applied to the polygonal-chains intersection problem, but there are some major differences between the two:

- The line-segments in a polygonal-chain are connected and linearly ordered, and this additional information plays a critical role when designing an efficient algorithm.
- A point shared by two adjacent line segments in a polygonal chain should not be reported as an intersection.

With a help of the above additional information, we will propose an efficient algorithm suitable for finding all intersections among chains, which is an extension of the Bentley-Ottmann sweep-line algorithm. The proposed algorithm will have $O((n + k)\cdot\log m)$ worst time complexity, where m is the number of *monotone chains* (to be described later) and is always smaller, often much smaller, than n the number of line segments. In a typical point-sequence curve offsetting application (e.g. contour-parallel type pocketing tool-path generation), for example, m could be as small as two while n could be as large as ten thousands. Recall that the time complexity of the Bentley and Ottmann's algorithm is $O((n + k)\cdot\log n)$.

Provided in the next section are basic terminology definitions and an informal description of the proposed algorithm, a formal description of the line sweeping algorithm is given in the section that follows. Applications of the algorithm are presented in the fourth section, followed by some concluding remarks in the final section.

2. DEFINITIONS AND OUTLINE OF THE ALGORITHM

Introduced in this section are some preliminary definitions, together with a brief outline of the proposed algorithm. A polygonal chain, or **chain** for short, is a sequence of connected line segments as shown in Figure 1-a. Throughout the paper, it is assumed that a *chain does not contain a consecutive collinear sequence of line-segments*: Consecutive collinear line-segments are merged into a single line-segment. The following definition of a monotone chain is borrowed from Preparata and Shamos[3].

Definition 1 (Monotone chain): A chain C is *monotone* with respect to a line L if C has at most one intersection point with a line L° perpendicular to L (See Figure 1-b). The line L is called the *monotone direction* and the line L° becomes a *sweep line*.

Without loss of generality, we can use the *x-axis* as its *monotone direction*. While traversing a chain, each of the local "extreme" points (with respect to their x-values) are marked either as a **left-extreme** point or a **right-extreme** point as

follows:

Definition 2 (Extreme point): A point in a chain is called a *left-extreme* (*right-extreme*) *point* if its x-value is locally minimum (maximum).

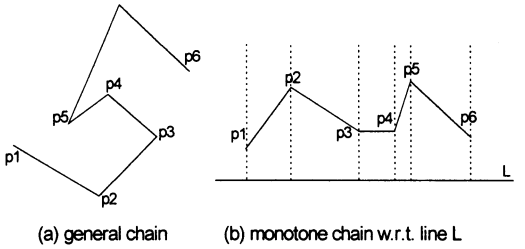

Figure 1 A general chain and a monotone chain

Construction of Monotone Chains

Shown in Figure 2 are local extreme points of a closed polygonal chain consisting of 17 points (or 17 line-segments): There are two left-extreme points, P0 and P11, and two right-extreme points, P6 and P14. For the moment, it is assumed that the chain contains no vertical line-segments. Then, the chain can easily be divided into monotone chains: Since a left-extreme point and a right-extreme point alternate, each sequence of line-segments starting from a left-extreme point to a right-extreme point (or vise versa) is identified as a monotone chain. During the process, it may be necessary to reverse the "direction" of some monotone chains so that each monotone chain starts from a left-extreme point and ends at a right-extreme point. Obviously, this **splitting operation** can be done in O(n) time.

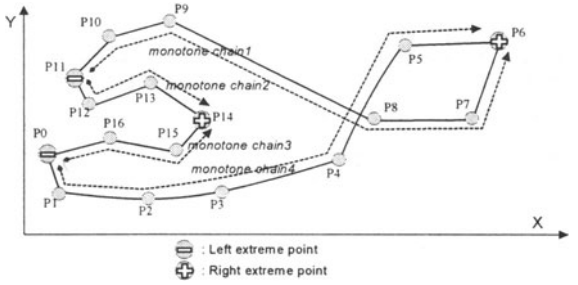

Figure 2 Extreme points and monotone chains

A vertical sweep-line is to be used in the proposed algorithm and it requires the chains contain no line-segments in parallel with the sweep-line, which is a fundamental limitation of the sweep-line method. However, a polygonal chain may contain vertical line-segments in which case the chain has to be rotated so that there are no vertical line-segments in the rotated chain. This **sweep-line validation** operation can be performed in O(n) time, where n is the number of line-segments in the chain.

Outline of the Monotone Chain Inter-section (MCI) Algorithm

The main phase of proposed polygonal-chain intersection(PCI) algorithm works on a set of monotone chains. The properties of a monotone chain that 1) it has

no self-intersections among its line segments and 2) its points are in an increasing order of x-values allow an efficient use of the sweep-line method. For a brevity of explanation, we introduce a few *data items*: *Vertex* (*v*); *Monotone-chain* (*MC*); *Front-vertex* (*fv*); *Active-chain-list* (*ACL*); *Sweeping-chain-list* (*SCL*); *Output-vertex-list* (*OVL*).

- **Vertex** (*v*) represents a point in a monotone chain and consists of its type (*v.type*), position (*v.pos*) and a pointer (*v.mc*) to the monotone-chain(s) to which it belongs. Initially, the type of a vertex is either *left-most*, *internal*, or *right-most* depending on its location in a monotone chain. An existing or newly created vertex corresponding to an intersection point becomes an *intersection* vertex.

- **Monotone-chain** (*MC*) is a sequence of vertices (pointers to vertices). Initially, an *MC* does not contain intersection vertices. As the algorithm progresses, the vertex located right after the sweep-line is designated as the **front-vertex** (*MC.fv*) and the newly generated intersection vertices are added. The i^{th} *MC* is denoted as MC_i.

- **Active-chain-list** (*ACL*) is a list of "active" *MC*s (i.e. the ones that have not been completely processed yet) ordered by the x-value of their front-vertex (*MC.fv.x*).

- **Sweeping-chain-list** (*SCL*) is a list of *MC*s being crossed by the sweep line. The *MC*s in the *SCL* are ordered by their y-values at the "sweep line crossing" point.

- **Output-vertex-list** (*OVL*) contains all intersecting-vertices found while sweeping a vertical sweep-line from left to right.

Depicted in Figure 3 is the progress of the monotone chain intersection algorithm. Initially, the sweep-line is located right before (actually in contact with) v_1. The major "state variables" appearing in Figure 3 are MC_1, MC_2, *ACL*, *SCL*, and *OVL*. In the figure, q_1 and q_2 are intersection-vertices and the vertices enclosed by a rectangle are "current" front-vertices. Given below are step-by-step changes in the state variables as the algorithm progresses:

1. **Initial:** Initially, there are two monotone chains, MC_1 and MC_2, stored in the active-chain-list (ACL), and both the sweeping-chain-list (SCL) and OVL are empty:
   - $MC_1 = \{v_1, v_2, v_3 | MC.fv = v_1\}$; $MC_2 = \{v_4, v_5 | MC.fv = v_4\}$;
   - ACL = $\{MC_1, MC_2\}$; SCL = Φ; OVL = Φ;

   1) Select the front-vertex of the first MC in ACL ($v_1 \equiv MC_1.fv$) ;
   2) Advance the front-vertex of the selected MC by one ($MC_1.fv = v_2$) ;
   3) Reposition MC_1 in ACL w.r.t. its front-vertex's x-value: ACL = $\{MC_2, MC_1\}$;

2. **After v_1:** As the sweep-line crosses the selected vertex v_1, MC_1 is inserted into SCL since v_1 is the *left-most* vertex of MC_1. Thus, the state variables are updated as:
   - $MC_1 = \{v_1, v_2, v_3 | MC.fv = \mathbf{v_2}\}$; $MC_2 = \{v_4, v_5 | MC.fv = v_4\}$;
   - ACL = $\{\mathbf{MC_2}, \mathbf{MC_1}\}$; SCL = $\{\mathbf{MC_1}\}$; OVL = Φ;

   1) Select the front-vertex of the first MC in ACL ($v_4 \equiv MC_2.fv$) ;
   2) Advance the front-vertex of the selected MC by one ($MC_2.fv = v_5$) ;

3) Reposition MC_2 in ACL w.r.t. its front-vertex's x-value: ACL = {MC_1, MC_2};

3. **After v_4:** As the sweep-line crosses the selected vertex v_4, MC_2 is inserted into SCL since v_4 is the *left-most* vertex of MC_2. Thus, the state variables are updated as:
   - MC_1 = {v_1, v_2, v_3| $MC.fv = v_2$}; MC_2 = {v_4, v_5| $MC.fv = \mathbf{v_5}$};
   - ACL = {$\mathbf{MC_1}$, $\mathbf{MC_2}$}; SCL = {MC_1, $\mathbf{MC_2}$}; OVL = Φ;

   Now 1) the two "sweeping" line-segments, v_1v_2 and v_4v_5, are intersected with each other, 2) the intersection point q_1 is inserted into MC_1 and MC_2, and 3) the front-vertices of MC_1 and MC_2 are changed to q_1. Thus, the state variables are updated as:
   - MC_1 = {v_1, $\mathbf{q_1}$, v_2, v_3| $MC.fv = \mathbf{q_1}$}; MC_2 = {v_4, $\mathbf{q_1}$, v_5| $MC.fv = \mathbf{q_1}$};
   - ACL = {MC_1, MC_2}; SCL = { MC_1, MC_2}; OVL = Φ;

   1) Select the front-vertex of the first MC in ACL ($q_1 \equiv MC_1.fv$) ;
   2) Advance the front-vertex of the selected MC by one ($MC_1.fv = v_2$) ;
   3) Reposition MC_1 in ACL w.r.t. its front-vertex's x-value: ACL = {MC_2, MC_1};

4. **After q_1:** As the sweep-line crosses the selected vertex q_1, 1) get MC_2 which is intersecting with MC_1 at the *intersection-vertex* q_1, 2) advance the front-vertex of MC_2 by one ($MC_2.fv = v_5$), 3) reposition MC_2 in ACL w.r.t. its front-vertex's x-value (ACL = {MC_1, MC_2}), and 4) swap MC_1 and MC_2 in SCL. Thus, the state variables are updated as:
   - MC_1 = {v_1, q_1, v_2, v_3| $MC.fv = \mathbf{v_2}$}; MC_2 = {v_4, q_1, v_5| $MC.fv = \mathbf{v_5}$};
   - ACL = {MC_1, MC_2}; SCL = { $\mathbf{MC_2}$, $\mathbf{MC_1}$}; OVL = {$\mathbf{q_1}$};

   1) Select the front-vertex of the first MC in ACL ($v_2 \equiv MC_1.fv$) ;
   2) Advance the front-vertex of the selected MC by one ($MC_1.fv = v_3$) ;
   3) Reposition MC_1 in ACL w.r.t. its front-vertex's x-value: ACL = {MC_2, MC_1};

5. ...

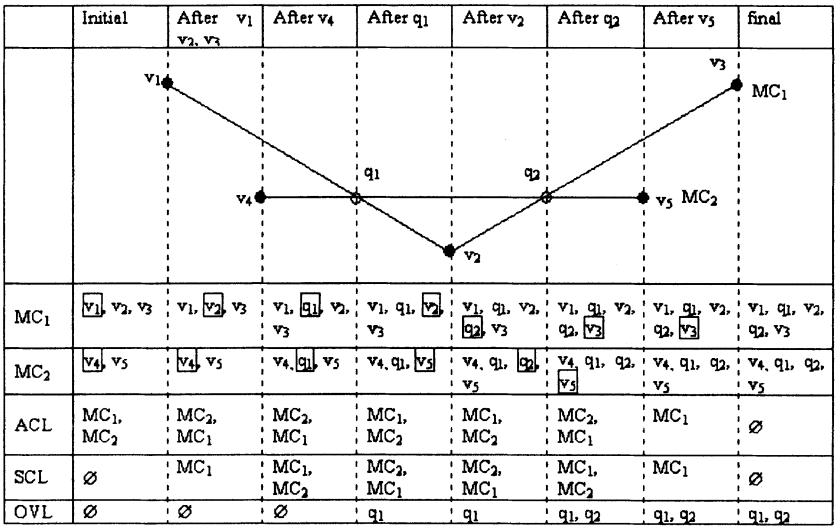

Figure 3 Progress of the monotone-chain-intersection algorithm

3. THE MONOTONE CHAIN INTER-SECTION ALGORITHM

Presented in this section is a formal description of the monotone chain intersection (MCI) algorithm which was informally described in the previous section. For a formal description, it is first required to provide a precise definition of an "intersection", which may vary among different applications.

One may identify that there are at least 7 types of intersections among monotone chains as illustrated in Figure 4. In the proposed MCI-algorithm, the first three types (i.e. *regular*, *double-vertex*, and *single-vertex* intersections) are always reported as **proper intersections**. The fourth one, the *extreme-vertex* intersection, which occurs when two monotone chains meet at their extreme vertex (i.e. start- or end-vertex), is not reported as a proper intersection if the two extreme vertices have the same identification (i.e. they came from the same point in a polygonal chain). Two or more proper intersections occurring at one point become a **multiple intersection**. The last two types, the *partial-* and *full-overlap* intersections (the hollow dots in Figure 4-f, g), are not reported as proper intersections.

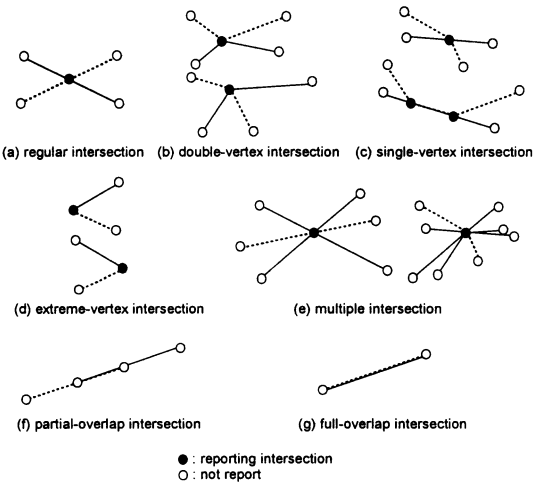

Figure 4 Proper intersections among monotone-chains (Solid dots)

Again, to simplify the explanations somewhat, a few primitive functions to be used in the *MCI-algorithm* are introduced:

- Monotone-chain(***MC***):
  - **advance-fv-MC**(MC_i): Advance the front-vertex of MC_i by one position (i.e. now the vertex next to the previous front-vertex position is the front-vertex).
  - **insert-fv-MC**(MC_i, v): If the existing front-vertex is not a right-most vertex and its position coincides with that of v **then** the vertex v replaces the existing front- vertex **else** it is inserted into MC_i right before the existing front-vertex position and is set to the front-vertex of MC_i.
- Active-chain-list(***ACL***):
  - **reposition-mc-ACL**(MC_i): Reposition MC_i so that all the monotone chains in the ACL are ordered by the x-values of their front-vertices. If the left-most

vertex of a monotone chain and the right-most vertex of another one have the same x-value, the left-most vertex comes first.

- Sweeping-chain-list(***SCL***):
  - **insert-mc-SCL(*MC*$_i$)**: Insert MC$_i$ into SCL such that the MCs are ordered by the y-values of the "sweep-line crossing" points (Figure 5-a). If they have the same y- value, their sequence is determined according to their "slopes" at the sweep-line crossing point (Figure 5-b).
  - **swap-mc-SCL({*MC*$_i$})**: For a set of monotone chains {*MC*$_i$} intersecting with the sweep-line at the same point, their sequence is reordered according to the "slopes" of the {*MC*$_i$} at the sweep-line crossing point (Figure 5-b).
  - **get-mc-SCL(p, {*MC*$_i$})**: Get all the monotone chains in the SCL that are passing through a given point **p** (and return them in {*MC*$_i$}).
- General list operation functions (L = MC or SCL):
  - **prev**(L, e): Return the previous element of "e" (if not exists, return Null).
  - **next**(L, e): Return the next element of "e" (if not exists, return Null).

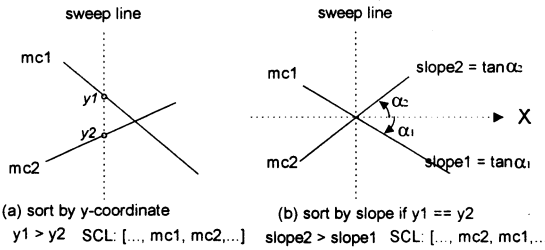

Figure 5 Ordering of Monotone-chains in the Sweeping-chain-list

Now we give a formal description of the MCI-algorithm to find all intersections among a set of monotone chains. The algorithm takes a set of *m* monotone chains {MC$_i$, *i=1..m*} as input, and produces a set of intersections as output. For the time being, it is assumed that *no more than two monotone chains intersect with each other* at any one point (which will be removed later). Further, recall that the input monotone chains have been constructed such that they contain *no vertical line segments*:

MCI-Algorithm (monotone-chain-intersection algorithm)

// No more than two monotone chains intersect with each other at any one point. //

0. Input: A set of monotone chains {MC$_i$, *i=1..m*} having no vertical line-segments;

1) Initialize:
   ACL = {MC$_i$, *i=1..m*}; //Active-Chain-List //
   SCL = Φ; // Sweeping-Chain-List //
   OVL = Φ; // Output-Vertex-List) //
   for all MC$_i$ reposition-mc-ACL(MC$_i$) ;

2) **while** ACL is not empty **do** {
   2-0) MC$_a$ = the first monotone chain in ACL;
   v = MC$_a$.fv;
   advance-fv-MC(MC$_a$);

```
      reposition-mc-ACL(MC_a);
  2-1) case v of left-most-vertex {
      insert-mc-SCL(MC_a);
      find-intersection(MC_a, prev(SCL, MC_a));
      find-intersection(MC_a, next(SCL, MC_a));
    }
  2-2) case v of internal-vertex {
      find-intersection(MC_a, prev(SCL, MC_a));
      find-intersection(MC_a, next(SCL, MC_a));
    }
  2-3) case v of right-most-vertex {
      MC_p = prev(SCL, MC_a);
      MC_n = next(SCL, MC_a);
      remove MC_a from SCL and from ACL;
      find-intersection(MC_p, MC_n);
      }
  2-4) case v of intersection-vertex {
      MC_b = the MC intersecting with MC_a;
              // stored in v.mc //
      advance-fv-MC(MC_b);
    reposition-mc-ACL(MC_b);
      swap-mc-SCL({MC_a, MC_b });
      if (MC_a == prev(SCL, MC_b)) then{
        find-intersection(MC_a,
            prev(SCL, MC_a));
        find-intersection(MC_b,
            next(SCL, MC_b));
      } else {
        find-intersection(MC_b,
            prev(SCL, MC_b));
        find-intersection(MC_a,
            next(SCL, MC_a));
      }
      add v to OVL;
    }
  } // end of while //
3) Output OVL;
```

The time complexity of the MCI-algorithm is $O((n + k) \cdot \log m)$, while n is the number of vertices, m is the number of the monotone chains, and k is the number of intersections: The initialization step can be done in $O(m \cdot \log m)$ and the **while** loop in step 2 iterates exactly $n+k$ times, with each iteration executed in $O(\log m)$ time by using a heap structure for *ACL* and *SCL*. In the algorithm, the function "**find-intersection()**" attempts to find an intersection between two monotone chains (i.e. between the two line-segments, one from each MC, crossing the sweep-line). If an intersection is found, this point is inserted into both MC_1 and MC_2 to make it a front-vertex and then the $\{MC_i\}$ in ACL are reordered accordingly. Given below is the intersection function:

Function find-intersection(MC_1, MC_2)

// L1, L2: line-segments defined by their start-vertex (.sv) and end-vertex (.ev) //

1. **If** ($MC_1 \equiv$ Null) or ($MC_2 \equiv$ Null)
   then return;

2. // Construct two line-segments L_1 and L_2 //
   L_1.ev = MC_1.fv; L_1.sv = prev(MC_1, MC_1.fv);
   L_2.ev = MC_2.fv; L_2.sv = prev(MC_2, MC_2.fv);

3. **If** (L_1.sv $\equiv$ L_2.sv) or (L_1.sv $\equiv$ L_2.ev) or (L_1.ev $\equiv$ L_2.sv) or (L_1.ev $\equiv$ L_2.ev)
   then return; // An *extreme-vertex* intersection is not reported as an intersection point if the two vertices have the same id. (i.e. came from the same point) //

4. **if** L_1 and L_2 intersect at **p**
   then {create a vertex *v*;
   v.pos = **p**; *v.type* = *intersection-vertex*; *v.mc* = { MC_1, MC_2};
   insert-fv-MC(MC_1, *v*); reposition-mc-ACL(MC_1);
   insert-fv-MC(MC_2, *v*); reposition-mc-ACL(MC_2);
   } **else** return;

In order to handle the "multiple-intersection" case, Step 2-4 of the monotone-chain-intersection algorithm has to be modified as follows:

Multiple-intersection-module:

2-4) **case** *v* **of** *intersection-vertex* {
 2-4-1) get-mc-SCL(v.pos, S);
 // S: set of all MCs linked to *v* //
 T = S – {MC_a} ;
 // MC_a: the selected MC containing *v* //
 2-4-2) **for** each MC_j in T **do** {
 v.mc $\Leftarrow$ MC_j ; // link MC_j to *v* //
 insert-fv-MC(MC_j, *v*) ;
 advance-fv-MC(MC_j) ;
 reposition-mc-ACL(MC_j) ;
 } // end of for //
 2-4-3) swap-mc-SCL(S) ;
 // s_0, s_1: first and last elements of S //
 2-4-4) find-intersection(s_0, prev(SCL, s_0)) ;
 find-intersection(s_1, next(SCL, s_1)) ;
 2-4-5) add *v* to OVL;
} // end of case //

By replacing Step 2-4 in the MCI-algorithm with the multiple-intersection module given above, we have a complete algorithm to find all intersections among monotone chains. Obviously, the time complexity of the multiple-intersection module is bounded by Step 2-4-3 (**swap-mc-SCL**(*S*)) which has $O(s \cdot \log s)$ time complexity, where *s* is the number elements in S. Since the multiple-intersection contains more than $s \cdot \log s$ proper-intersections, the time complexity of our MCI-algorithm is not affected by this modification.

4. APPLICATIONS

There are many application areas where the proposed polygonal chain intersection (PCI) algorithm can be applied, with a little modification if necessary. The overall PCI-algorithm consists of the following three steps (n: number of line-segments; m: number of monotone chains; k: number of intersections) :

1. Construction of monotone chains from polygonal chains in $O(n)$ time.
2. Execution of the (modified) MCI-algorithm in $O((n + k)\cdot\log m)$ time.
3. Rotation transformation of the output intersection vertices if necessary.

In this section, we will present three applications of the PCI-algorithm: 1) simplicity test of a polygon, 2) intersection among polygons, and 3) offset operation of piecewise linear curve. Also presented in this section are test results of the algorithm for some non-simple polygons.

Simplicity test of a polygon

Simplicity test of a polygon is an important subject in many polygon-handling algorithms which require their input polygons to be simple ones. After splitting an input polygon into monotone chains, the MCI-algorithm is executed until an intersection is detected, which can be done in $O(n\cdot\log m)$ time. Note that the proposed algorithm has its worse case time complexity better than that of the Shamos and Hoey's intersection test algorithm[1] which has a time complexity of $O(n\cdot\log n)$.

Finding intersection among polygons

Finding intersections among given polygons is one of the basic operations in CAD/CAM. Examples include "area cutting" tool-path generation in regional milling and Boolean operations among 3D objects surrounded by (trimmed) freeform surfaces. In area-cutting tool-path generation, "areas" and "islands" are represented as polygons consisting of "dense" line-segments. Finding intersections among polygons consisting of large numbers of small line-segments is an ideal application area for the proposed PCI-algorithm. Boolean operations of 3D objects surrounded by trimmed freeform faces can be transformed into a 2D Boolean problem[18]. The trimming boundary is typically obtained from surface/surface intersection, which produces dense line-segment curves in a 2D parameter domain. Classifying a trimmed surface by another requires intersection of 2D parameter domain curves represented as polygonal chains.

Planar curve offsetting (self-intersections of a non-simple polygon)

Planar curve offsetting is an important geometric operation in many application areas including: NC pocket machining[15], VLSI circuit design, and robot path planning. In the literature, the "discrete" planar-curve offsetting problem has been approached from two difference directions: Voronoi-diagram method and direct offsetting method. In the first approach, the individual offset of a segment is trimmed to its intersections with the Voronoi diagram of the original curve. It is well known that the Voronoi diagram of a polygon can be obtained in $O(n\cdot\log n)$ time by applying a divide-and-conquer method or sweep line algorithm[15,16,17].

The direct offsetting method is based on a pair-wise intersection of individual offset segments to trim them into a valid offset curve. This approach is simpler in concept (and easier to implement) than the Voronoi diagram method, but it suffers

from its performance when the pair-wise intersection is implemented in brute force, which may take $O(n^2)$ time. With the help of the proposed intersection algorithm, the authors implemented a simple and relatively efficient procedure for offsetting dense polygons.

The *2D curve offsetting algorithm* consists of four steps: 1) "primitive" offsetting of the input curve, 2) self-intersection detection, 3) loop construction, and 4) removal of invalid loops. The proposed PCI-algorithm is used Step 2. In loop construction, the contour tracing method proposed by Vouzelaud[11] is employed, and both the orientation test[12,13] and circle test[13,14] are utilized in Step 4. Shown in Figure 6 is an example of planar-curve offset. Depicted in Figure 6-a are polygonal chains (denoted by "thick" curves) obtained by intersecting a die-cavity surface with a horizontal plane (It is an automobile stamping die). Shown in Figure 6-b are pocketing tool-paths at the "plane-step" which are obtained by successively applying the 2D-curve offsetting algorithm.

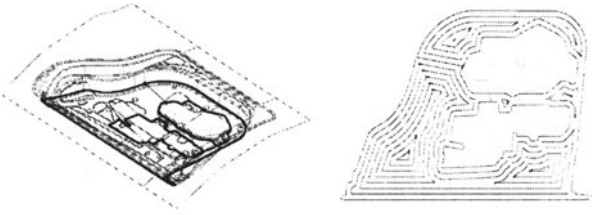

Figure 6 Planar-curve offset example

5. CONCLUSION AND DISCUSSION

Presented in this paper is a polygonal chain intersection algorithm having a worst case time-complexity of $O((n + k)\cdot\log m)$ which is a considerable improvement over the $O((n + k)\cdot\log n)$ time complexity of the Bentley-Ottmann's algorithm for a polygonal chain intersection problem. This gain has come from the use of connectivity information of a polygon: The proposed MCI-algorithm works on the monotone chains (instead of the line-segments) of a polygonal chain. Three major applications of the MCI-algorithm are simplicity test of a polygon, finding intersections among polygons, and offsetting planar curves. For a polygon-simplicity test, the time complexity of the MCI-algorithm becomes $O(n\cdot\log m)$ which is better that of the Shamos and Hoey's algorithm[1] which has a time complexity of $O(n\cdot\log n)$. In practice, number of monotone chains (m) is much smaller then number of line-segments (n).

For a given set of polygonal chains, number of intersections (k) is fixed. However, m may vary depending on the direction of the sweep-line, which means that there exists an "optimal" sweep-line direction that gives a minimum number monotone chains[19]. To suit a specific application purpose the definition of "proper intersection" may need to be changed, which may be easily accommodated by minor modifications in the algorithm.

REFERENCE

1. Shamos, M. I., Hoey, D. J. Geometric intersection problems. in Proc. 17th Annu. Conf. Foundation of Computer Science, pp 208-215, Oct 1976.

2. Bentley, J. L., Ottmann, T. A. Algorithms for reporting and counting geometric intersections. IEEE Transactions on Computers 28, 643-647, 1979.

3. Preparata, F. P., and Shamos, M. I. Computational geometry - An introduction. Springer Verlag, New York, 1985.

4. Chazelle, B., Edelsbrunner, H. An Optimal algorithm for intersecting line segments in the plane. Journal of the Association for computing machinery, Vol. 39, No. 1, January 1992, pp 1- 54.

5. Mehlhorn, K. and Naher, S., An Implementation of a sweep line algorithm for the straight line segment intersection problem. Technical Report No. MPI-I-94-160. Max-Planck-Institut fur Informatik, 1994.

6. Nievergelt, J., Preparata, F. P., Plane-sweep algorithms for intersecting geometric figures. Communications of the ACM, Vol. 25, No. 10, 1982, pp 739-747.

7. Edelsbrunner, H., Maurer, H. A. Polygonal intersection searching. Information processing letters, Vol. 14, No. 2, 1982, pp 74-79.

8. Bentley, J. L., Wood, D. An Optimal worst-case algorithm for reporting intersections of rectangles. IEEE Trans. Comput. C-30 (1981), 147-148.

9. Ralf Harmut Guting, Stabbing C-oriented polygons. Information processing letters, Vol. 16, No. 1, 1983, pp 35-40

10. Xue-Hou Tan, Tomio Hirata and Yasuyoshi Inagaki, The intersection searching problem for c-oriented polygons. Vol. 37, No. 4, 1991, pp 201-204

11. Vouzelaud, F. A., Bagchi, A. Offset of Two dimensional Contours : Finish Machining. Manufacturing Science and Engineering ASME, PED Vol. 64, 1993, pp 125 – 138.

12. Yong Seok Suh and Kunwoo Lee, NC milling tool path generation for arbitrary pockets defined by sculptured surfaces. CAD, Vol. 22, No. 5, pp 273 – 284.

13. Shi-Nine Yang and Ming-Liang Huang, A New Offsetting Algorithm Based On Tracing Technique. Proc. Second Symposium on Solid Modeling and Applications, pp 201 – 210.

14. Allan Hansen, Farhad Arbab, An algorithm for generating NC tool paths for arbitrary shaped pockets with island. ACM Transaction on graphics, Vol. 11, No 2. April 1992 , pp 152-182.

15. M Held, G Lukacs and L Andor, Pocket machining based on contour-parallel tool paths generated by means of proximity maps. CAD, Vol 26, No 3, March 1994.

16. Lee D. T., Medial axis transformation of a planar shape, IEEE Trans. On Pattern Analysis & Machine Intelligence, Vol 4, No 4, 1982 pp 363-369.

17. Fortune, S., A sweepline algorithm for Voronoi diagrams. Algorithmica, Vol 2, 1987 pp 153-174

18. David Dobkin, Leonidas Guibas, John Hershberger, and Jack Snoeyink. An Efficient algorithm for finding the CSG representation of a simple polygon. Computer Graphics, 22(4):31-40, 1988

19. M Held. A geometry-based investigation of the tool path generation for zigzag pocket machining. The Visual Computer, 1991, 7:296-308.