한국 CAD/CAM 학회 논문집
제 4권 제 3호 1999년 9월 pp. 190-199

학술논문

점열 곡선의 꼬임을 효율적으로 찾는 알고리즘

박상철*, 신하용**, 최병규***

An efficient polygonal chain intersection algorithm

Sang C. Park*, Hayong Shin** and Byoung K. Choi***

ABSTRACT

Presented in this paper is an 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 sweep line algorithm[2]. 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 and singularities (tangential contact, multiple 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.

Key words : polygonal chain intersection algorithm, point sequence curve

1. 서 론

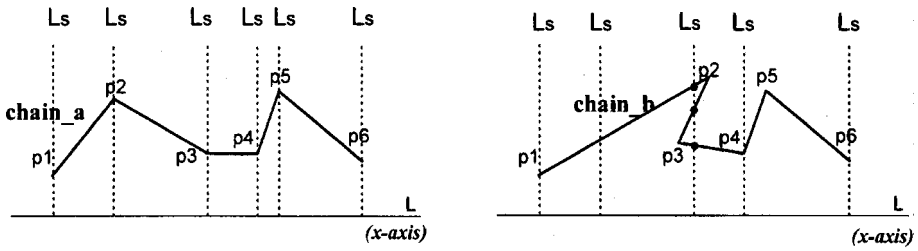
점열 곡선(polygonal chain)은 점들의 연속으로 정의되는 곡선을 말하며, 다각형이란 닫힌 점열 곡선을 의미한다. 점열 곡선은 가장 일반적이고 기본적인 기하 요소 중의 하나이며, 이들 사이의 꼬임을 구하는 것은 컴퓨터 그래픽스, CAD/CAM등에서 필수적인 오퍼레이션이다. 구체적인 적용 분야로는 자유곡면 렌더링에서의 hidden-line removal 혹은 점열 곡선 오프셋을 들 수 있다. 기존의 점열 곡선 꼬임 알고리즘들은 convex[5], simple[6], rectangular[7]그리고 C-oriented[8]등 특별한 부류의 점열 곡선에 대한 알고리즘이 대부분이다. 한편 평면상의 라인 세그먼트(line segment)들간의 꼬임을 구하는 알고리즘[2,4,9]들을 일반적인 점열 곡선 꼬임에 적용을 할 수도 있으나, 효율적이라고는 볼 수 없다. 왜냐하면 점열 곡선이라는 것은 라인 세그먼트들간의 연결정보를 포함하고 있는 것인데, 단순한 라인 세그먼트들간의 꼬임 문제로 보면 이러한 연결정보를 무시하는 것이므로 효율적이지 않은 것이다. 실제로 본 논문에서 제안된 점열 곡선 꼬임 알고리즘은 이러한 연결정보를 이용하여 효율화를 꾀하고 있다.

본 논문에서는 효율적인 점열 곡선 꼬임 알고리즘을 제안하고 있다. 구성은 다음과 같다. 다음 장에서는 제안된 알고리즘의 설명에 관련된 용어를 정의하고, 그 다음 장에서 제안된 알고리즘을 개략적으로 예를 들어 설명한다. 이를 기반으로 제안된 알고리즘을 체계적으로 기술하고 적용 분야 및 실험결과를 기술한다. 마지막으로 결론을 기술한다.

2. 용어정의

본 장에서는 제안된 알고리즘의 설명을 위해 몇 가지 용어 및 관련된 사항들을 설명하고자 한다. 점열 곡선이란 점들의 배열로써 정의되는 라인 세그먼트들의 연속으로 볼 수 있다. 본 논문에서는 주어지는 점열 곡선에 세 개 이상의 점들이 연속으로 같은 직선 상에 있는 경우는 없다고 가정한다. 세 개 이상의 점이 같은 직선에 연속으로 있는 경우에는 하나의

*학생회원, 한국과학기술원 산업공학과
**종신회원, Daimler-Chrysler Tech Center
***종신회원, 한국과학기술원 산업공학과



(a) monotone chain w.r.t. line L (b) not a monotone chain w.r.t line L

Fig. 1. A general chain and a monotone chain.

라인 세그먼트로 합칠 수 있다. 우선 monotone chain의 개념을 정의하고자 하며, 아래 정의는 Preparata and Shamos[3]로부터 빌려온 것이다.

정의 1. Monotone chain: 직선 L에 수직인 직선들을 L°라고 할 때, 여떤 chain C가 L°들과 한 번 이상 만나지 않을 때, chain C는 직선 L에 대한 monotone chain이라 한다. 이때 직선 L을 monotone direction이라 하고 L°를 스윕라인이라 한다. 이후로는 특별한 언급이 없는 한, X축을 monotone direction으로 보도록 한다.

정의 2. Extreme point: 하나의 chain에 속한 점들 중에서 X값이 local minimum인 점을 left-extreme point라 하고, X값이 local maximum인 점을 right-extreme point라 부른다.

다음은 X축을 monotone direction으로 보았을 때, 임의의 chain을 monotone chain으로 분할하는 방법을 간략히 기술 한 것이다. Monotone direction의 선정에 따라서 임의의 chain이 몇 개의 monotone chain으로 분할되는 지는 달라질 수 있으나 그 문제는 나중에 설명하도록 한다.

■ 임의의 chain을 monotone chain으로 분할

Fig. 2에 그려진 chain은 17개의 점으로 이루어져 있으며, 두 개의 left-extreme points(P0, P11)와 두 개의 right-extreme points(P6, P14)를 포함하고 있다. 일단 주어진 chain이 X축에 수직인 세그먼트를 포함하지 않는다고 가정하면 monotone chain으로 분할하는 일은 매우 간단하다. Chain을 한 방향으로 따라가면 left-extreme point와 right-extreme point가 번갈아 나온다는 것을 알 수 있다. 이때 각각의 left-extreme point에서 출발하여 right-extreme point에서 끝나는 한 조각이 X축에 대한 하나의 monotone chain이 되는 것이다. Monotone chain의 방향은 항상 X값이 증가하는 방향으로 설정한다. 그러면 Fig. 2에서 주어진 chain은 4개의 monotone chain으로

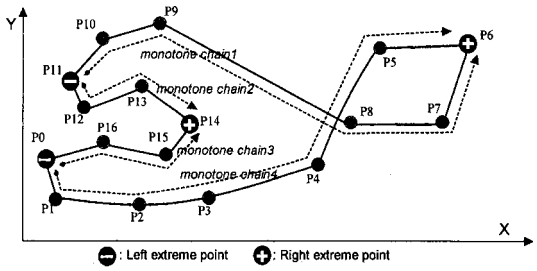


Fig. 2. Extreme points and monotone chains.

분할된다. 쉽게 알 수 있듯이 이렇게 임의의 chain을 monotone chain으로 분할하는 것은 점의 개수가 n이라 할 때 $O(n)$ 시간에 가능하다.

위에서는 주어진 chain에서 수직 세그먼트가 없다고 가정했는데 실제로 있다고 하여도 chain을 rotate함으로써 수직 세그먼트는 피할 수 있다. 스윕라인과 평행한 세그먼트가 존재하지 않는 rotation angle은 스윕라인에 가장 가까운 각도를 가지는 점을 $O(n)$ 시간에 찾음으로써 알아 낼 수 있다.

3. 제안된 알고리즘의 개략적 기술

제안된 점열 곡선 꼬임(PCI) 알고리즘은 입력으로 monotone chain들의 집합을 받아들인다. 본 논문에서는 제안된 알고리즘의 효율성을 위해서 다음과 같은 monotone chain의 특성을 이용하였다: 1) 하나의 monotone chain내에서는 자체 꼬임이 없다, 2) 하나의 monotone chain에 속한 모든 점들은 X값이 증가하는 방향으로 sorting 되어 있다. 이 두 가지 특성이 스윕라인 알고라즘의 효율적인 적용을 가능하게 하는 것이다. 제안된 알고리즘은 monotone direction (X축)에 수직인 스윕라인 (Y축)을 정의한 후, monotone chain들의왼쪽에서 오른쪽으로 지나가면서 꼬임을 구하는 방식이다. 설명의 효율성을 위해서 다음과 같은 몇 가지 data item을 정의하고자 한다.

• **Vertex (v):** monotone chain상의 한 vertex를 나타내는 것으로써, vertex의 type(v.type), 위치 정보(v.pos) 그리고 vertex가 속한 monotone chain을 가리키는pointer(v.mc)를 가지고 있다. 초기의 vertex type은 위치에 따라 left-most, internal 혹은 right-most중에 한 가지이다. (monotone chain에서 첫 번째 vertex는 *left-most* type이고, 맨 마지막 vertex는 *right-most* type이고 나머지는 모두 *internal* type이 된다.) 그리고 꼬임을 찾기 위해 제안된 알고리즘이 진행하는 도중에 꼬임이 발견되면 꼬임에 해당하는 점은 *intersection* type vertex가 된다. 그리고 intersection vertex인 경우는 소속된 monotone chain이 여러 개이므로 v.mc는 여러 개의 monotone chain들을 가리키게 된다.

• **Monotone chain (MC):** MC는 vertex들의 linked-list로 볼 수 있으며, 초기에는 intersection type vertex를 포함하지 않다가 꼬임이 발견되면 해당하는 위치에 삽입하게 된다. 알고리즘이 진행함에 따라 스윕라인은 X값이 증가하는 쪽으로 움직이게 되는데, 어느 시점에서 스윕라인 직후에 있는 vertex를 그 monotone chain의 front-vertex(MC.fv)라고 한다. 나중에 설명되겠지만 꼬임은 항상 front-vertex의 직전에 삽입된다.

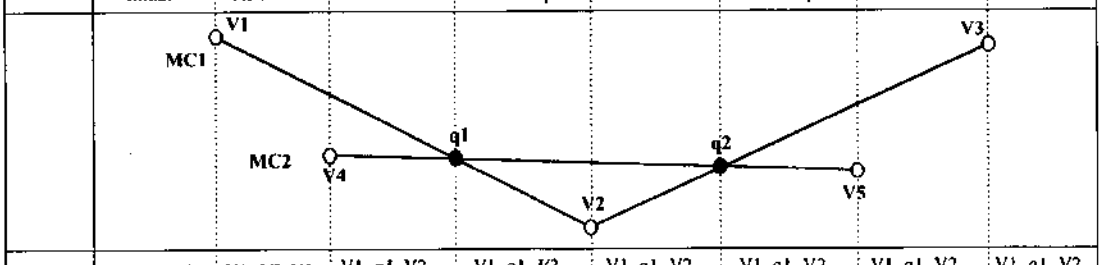
• **Active-chain-list (ACL):** ACL은 "active" MC들의 list이다. "active"라는 의미는 아직 processing이 완결되지 않음을 뜻한다. 스윕라인이 MC의 maximum X값을 지나 쳤을 때 processing이 완결된 것이다. ACL은 항상 MC들의 front-vertex(MC.fv)의 X값이 증가하는 방향으로 MC들을 sorting해서 유지한다.

• **Sweeping-chain-list (SCL):** SCL은 현재 스윕라인에 걸치는 MC들의 list이다. 그러므로 SCL에 속한 모든 MC들은 항상 스윕라인과 하나씩의 교점을 가지게 되는데, 그 교점의 Y값이 감소하는 방향으로 MC들을 sorting해서 유지한다.

• **Output-vertex-lsit (OVL):** OVL은 스윕라인이 지나가면서 발견되는 모든 intersection vertex들을 담고 있다.

Fig. 3은 monotone chain들간의 꼬임을 구하는 알고리즘의 진행 방식을 예를 들어 보여주고 있다. 초기에 스윕라인은 v1의 직전에 위치하게 된다. 그리고 알고리즘의 진행상태를 나타내는state variable들은 Fig. 3에 나타나 있듯이, MC1, MC2, ACL, SCL 그리고 OVL이다. Fig. 3에서는 두 개의 꼬임 q1, q2가 있고 각 MC에서 사각형으로 둘러싸인 vertex들은 그 시점의 front-vertex를 뜻한다. 그러면 아래에서 알고리즘의 진행을 매 단계 설명함으로써 제안된 알고리즘의 개략적인 특성을 보이려 한다. 제안된 알고리즘의 formal한 기술과 다양한 degeneracy 경우에 대한 설명은 다음 장에 나타나 있다.

1. Initial: 초기에 MC1과 MC2의 front-vertex는 각자의 첫 번째 vertex이다. 각 front-vertex들의 X값



	Initial	After V1	After V4	After q1	After V2	After q2	After V5	Final
MC1	***V1***, V2, V3	V1, ***V2***, V3	V1, ***q1***, V2, V3	V1, q1, ***V2***, V3	V1, q1, V2, ***q2***, V3	V1, q1, V2, q2, ***V3***	V1, q1, V2, q2, ***V3***	V1, q1, V2, q2, V3
MC2	***V4***, V5	***V4***, V5	V4, ***q1***, V5	V4, q1, ***V5***	V4, q1, ***q2***, V5	V4, q1, q2, ***V5***	V4, q1, q2, V5	V4, q1, q2, V5
ACL	MC1, MC2	MC2, MC1	MC2, MC1	MC1, MC2	MC1, MC2	MC2, MC1	MC1	Φ
SCL	Φ	MC1	MC1, MC2	MC2, MC1	MC2, MC1	MC1, MC2	MC1	Φ
OVL	Φ	Φ	Φ	q1	q1	q1, q2	q1, q2	q1, q2

Fig. 3. Progress of the Monotone-chain-intersection algorithm.

을 기준으로 sorting하여 active-chain-list(ACL)에 넣는다. 초기에는 sweeping-chain-list(SCL)과 output-vertex-list(OVL)는 비어있다.

- MC1={v1, v2, v3|MC.fv=v1};
  MC2={v4, v5|MC.fv=v4};
- ACL={MC1, MC2}; SCL=Φ;
  OVL=Φ;

1) ACL의 첫 번째 MC(MC1)의 front-vertex를 선택한다.(v1≡MC1.fv);

2) MC1의 front-vertex를 다음 vertex로 바꾼다. (MC1.fv=v2);

3) MC1의 front-vertex가 달라졌으므로 ACL에서 위치 재 선정. ACL={MC2, MC1};

2. After v1: 스윕라인이 v1을 지나감에 따라, MC1은 SCL에 삽입된다. 왜냐하면 v1이 left-most ver-tex이므로, MC1이 SCL과 교차하게 되었기 때문이다. 이로써 state variable들은 다음과 같이 변한다.

- MC1={v1, v2, v3|MC.fv=v2};
  MC2={v4, v5|MC.fv=v4};
- ACL={MC2, MC1}; SCL={MC1};
  OVL=Φ;

1) ACL의 첫 번째 MC(MC2)의 front-vertex를 선택한다.(v4≡MC2.fv);

2) MC2의 front-vertex를 다음 vertex로 바꾼다. (MC2.fv=v5);

3) MC2의 front-vertex가 달라졌으므로 ACL에서 위치 재 선정. ACL={MC1, MC2};

3. After v4: 스윕라인이 v4을 지나감에 따라, MC2는 SCL에 삽입된다. 왜냐하면 v4가 left-most ver-tex이므로, MC2가 SCL과 교차하게 되었기 때문이다. 이때 MC2는 스윕라인과 교차하는 점의 Y값을 고려하여 SCL에 삽입되는 자리를 정하게 된다. 이로써 state variable들은 다음과 같이 변한다.

- MC1={v1, v2, v3|MC.fv=v2};
  MC2={v4, v5|MC.fv=v5};
- ACL={MC1, MC2};
  SCL={MC1, MC2}; OVL=Φ;

이때 SCL에 새로이 MC2가 추가되었으므로, MC2의 위/아래에 위치한 MC들과 교차 test를 하게 된다. 두 MC간의 교차 test는 항상 각 MC의 front-vertex에 연결된 직전 세그먼트들끼리만 수행한다. 우선 MC2와 교차 test를 할 MC를 찾아보면, SCL에 MC2 아래에는 아무 것도 없고 위에는 MC1이 있으므로, MC1밖에 없다는 것을 알 수 있다. MC2와 MC1의 교차 test는 각각의 front-vertex 직전에 연결된 세그먼트(v1, v2 & v4, v5)간에 일어난다. 이 때 꼬임 q1을 찾을 수 있게 되며 q1을 각각의 MC에 삽입하고, 각각의 MC들의 front-vertex를 q1로 바꾼다. 이로써 state variable들은 다음과 같이 변한다.

- MC1={v1, q1, v2, v3|MC.fv=q1};
  MC2={v4, q1, v5|MC.fv=q1};
- ACL={MC1, MC2};
  SCL={MC1, MC2}; OVL=Φ;

1) ACL의 첫 번째 MC(MC1)의 front-vertex를 선택한다.(q1≡MC1.fv);

2) MC1의 front-vertex를 다음 vertex로 바꾼다. (MC1.fv=v2);

3) MC1의 front-vertex가 달라졌으므로 ACL에서 위치 재 선정. ACL={MC2, MC1};

4. After q1: 스윕라인이 intersection vertex인 q1을 지나감에 따라 다음과 같은 과정을 수행한다: 1) q1에서 MC1과 교차하는 MC인 MC2를 가져온다, 2) MC2의 front-vertex를 다음 vertex로 바꾼다. (MC2.fv=v5), 3) MC2의 front-vertex가 달라졌으므로 ACL에서 위치 재 선정. ACL={MC1, MC2} 그리고 4) SCL에서 MC1과 MC2의 위치를 바꾼다, 왜냐하면 intersection vertex q1을 지나면서 스윕라인과 교차하는 Y값의 순서가 뒤바뀌기 때문이다. 이로써 state variable들은 다음과 같이 변한다.

- MC1={v1, q1, v2, v3|MC.fv=v2};
  MC2={v4, q1, v5|MC.fv=v5};
- ACL={MC1, MC2};
  SCL={MC2, MC1}; OVL={q1};

1) ACL의 첫 번째 MC(MC1)의 front-vertex를 선택한다.(v2≡MC1.fv);

2) MC1의 front-vertex를 다음 vertex로 바꾼다. (MC1.fv=v3);

3) MC1의 front-vertex가 달라졌으므로 ACL에서 위치 재 선정. ACL={MC2, MC1};

5. 이하 생략...

4. Monotone chain intersection algorithm

본 장에서는 monotone chain intersection 알고리즘을 formal하게 설명한다. 우선 꼬임의 정의가 application 마다 달라질 수 있으므로, 다양한 종류의 꼬임을 살펴보도록 한다. 본 논문에서는 Fig. 4와 같이 7가지의 꼬임을 분류하여 각각의 경우에 대해 report 대상이 되는 proper intersection을 정의하였다.

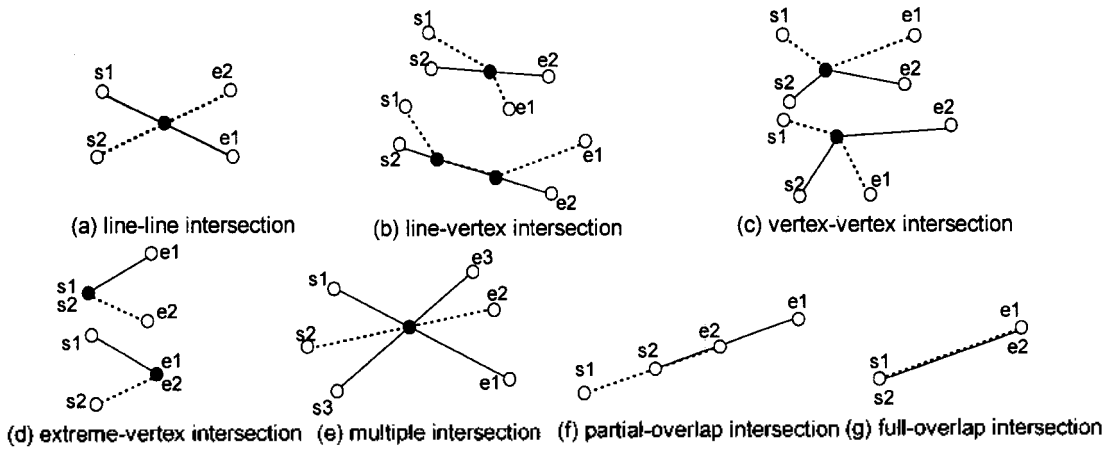


Fig. 4. Intersection types.

Fig. 4의 처음 세 가지 경우(line-line, line-vertex and vertex-vertex intersection)는 항상 proper intersection으로 취급하여 report하도록 한다. 네 번째 extreme-vertex intersection은 두 개의 monotone chain이 첫 점들에서 만나거나 끝점들에서 만나는 경우이다. 이 때는 만약 만나는 vertex가 같은 점이라면 (같은 다각형의 extreme vertex인 경우) 꼬임으로 보지 않고 그렇지 않다면 proper intersection으로 report한다. 세 개이상의 세그먼트가 한 점에서 만나는 경우는 multiple intersection이라 하고 proper intersection으로써 report한다. 마지막 두 가지, partial- and full-overlap intersection은 proper intersection으로 취급하지 않는다. 물론 필요에 따라 위에서 언급된 proper intersection 이외의 intersection도 필요한 경우가 있을 것이다. 이러한 필요를 위해서도 제안된 알고리즘을 조금만 수정하면 충분히 가능하다.

Monotone chain intersection 알고리즘(MCI-알고리즘)의 효율적인 설명을 위해서 몇 가지 함수들을 설명하도록 한다.

- Monotone chain (MC) 함수들:
  - advance-fv-MC (MCi): MCi의 front-vertex를 다음 vertex로 바꾼다.
  - insert-fv-MC(MCi, v): MCi의 front-vertex 전에 v를 삽입한다.
- Active-chain-list (ACL) 함수들:
  - reposition-mc-ACL (MCi): MCi의 frontvertex의 X값을 기준으로 ACL내에서 다시 위치를 잡는다.

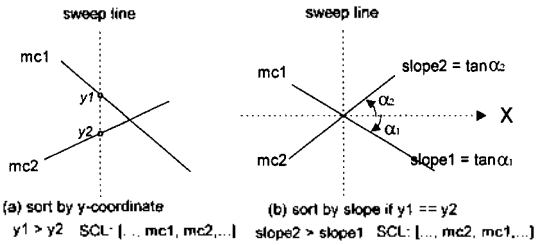


Fig. 5. Sorting of MCs in the SCL.

Sweeping-chain-list (SCL) 함수들:

- insert-mc-SCL (MCi): MCi를 SCL에 삽입한다. 이때 위치는 스윕라인과 교차하는 점의 Y값을 기준으로 선정한다(Fig. 5-a). 만약 같은 Y값을 가지는 것이 있으면, 그점에서의 slope로 결정한다(Fig. 5-b).
- swap-mc-SCL ({MCi}): 주어지는 {MCi}에 속한 MCi들은 항상 스윕라인과의 교점이 동일해야 한다. 이때 각 MC들의 slope를 고려해서 SCL내에서 위치를 재배치한다(Fig. 5. 참조).
- get-mc-SCL (p, {MCi}): 주어지는 점 p는 항상 스윕라인과 같은 X값을 가져야 한다. 이때 SCL에 속한 MC들 중에서 스윕라인과의 교점이 p와 같은 MC들을 {MCi}에 담아서 출력해 준다.

- 일반적인 list 관련 함수(L=MC 혹은 SCL)
  - prev(L, e): List L에서 element e의 직전 element를 출력한다. 없으면 NULL을 출력.
  - next(L, e): List L에서 element e의 직후

element를 출력한다. 없으면 NULL을 출력.

이러한 기본 함수들을 바탕으로 주어진 monotone chain들의 모든 꼬임을 구하는 MCI-알고리즘을 formal하게 기술하도록 한다. MCI-알고리즘의 입력은 m개의 monotone chain으로 이루어진 집합{MCi, i=1, ..., m}이 되고 출력은 모든 꼬임 들이다. 그리고 일단 세 개이상의 monotone chain이 한 점에서 만나는 multiple intersection은 없다고 가정한다. 나중에 이러한 multiple intersection도 모두 고려하여 구하는 알고리즘이 설명된다.

MCI-알고리즘(monotone-chain-intersection)

```
//multiple intersection이 없는 경우//
1) Initialzie:
   ACL={MCi, i=1, ..., m}; //Active-chain-list//
   SCL=Φ; //Sweeping-chain-list//
   OVL=Φ; //Output-vertex-list//
   for all MCi reposition-mc-ACL(MCi);
2) while ACL is not empty do {
   2-0) MCa=the first monotone chain in ACL;
        v=MCa.fv;
        advance-fv-MC(MCa);
        reposition-mc-ACL(MCa);
   2-1) case v of left-most-vertex {
        insert-mc-SCL(MCa);
        find-intersection(MCa, prev(SCL, MCa));
        find-intersection(MCa, next(SCL, MCa));
        }
   2-2) case v of internal-vertex {
        find-intersection(MCa, prev(SCL, MCa));
        find-intersection(MCa, next(SCL, MCa));
        }
   2-3) case v of right-most-vertex {
        MCp=prev(SCL, MCa);
        MCn=next(SCL, MCa);
        remove MCa from SCL and from ACL;
        find-intersection(MCp, MCn);
        }
   2-4) case v of intersection-vertex {
        MCb=the MC intersecting with MCa;
        // stored in v.mc //
        advance-fv-MC(MCb);
        reposition-mc-ACL(MCb);
        swap-mc-SCL({MCa, MCb });
        if (MCa= =prev(SCL, MCb)) then{
        find-intersection(MCa, prev(SCL, MCa));
        find-intersection(MCb, next(SCL, MCb));
        } else {
        find-intersection(MCb, prev(SCL, MCb));
        find-intersection(MCa, next(SCL, MCa));
        }
        add v to OVL;
        }
   }// end of while //
   3) Output OVL;
```

제안된 MCI-알고리즘의 시간복잡도는 $O((n+k)\cdot log\ m)$이다. 이때 n은 vertex의 개수, m은 monotone chain의 개수 그리고 k는 꼬임의 개수를 나타낸다. Step 1(initialization)에서는 m개의 monotone chain을 front-vertex의 X값으로 sorting해야 하므로 $O(m\cdot log\ m)$의 시간복잡도를 가진다. Step 2의 while loop은 정확히 $n+k$번 돌게 되고, loop의 시간복잡도는 $O(log\ m)$이다. 위 MCI-알고리즘에서 "find- intersection()"이라는 함수는 주어지는 monotone chain들의 꼬임을 구하는 함수이며, 두 개의 monotone chain들의 각 front-vertex의 직전 라인 세그먼트들끼리 꼬임을 구하게 된다. 만약 꼬임이 발견되면 발견된 intersection vertex를 양쪽 monotone chain에다 모두 삽입하고, 삽입된 intersection vertex를 front-vertex로 바꾼다. "find-intersection()" 함수를 아래에 기술한다.

Function find-intersection (MC1, MC2)

```
//L1, L2: line-segments defined by their
 start-vertex (.sv) and end-vertex (.ev)//
1. If (MC1Null) or (MC2Null)
   //No line-segment left in the MC//
    then return;
2. //Retrieve the two front line-segments
   (L1 and L2)//
   L1.ev=MC1.fv; L1.sv=prev(MC1, MC1.fv);
   L2.ev=MC2.fv; L2.sv=prev(MC2, MC2.fv);
3. If (L1.sv≡L2.sv) or (L1.sv≡L2.ev) or
      (L1.ev≡L2.sv) or (L1.ev≡L2.ev)
   then return;
   //An extreme-vertex intersection is ignored//
4. If L1 and L2 intersect at p
   then {
        create a vertex v;
        v.pos=p; v.type=intersection-vertex;
```

```
        v.mc={ MC1, MC2};
        insert-fv-MC(MC1, v);
        reposition-mc-ACL(MC1);
        insert-fv-MC(MC2, v);
        reposition-mc-ACL(MC2);
    } else return;
```

위에서 제안된 MCI-알고리즘은 multiple intersection이 없다고 가정한 상태에서 기술되었는데, MCI-알고리즘의 Step 2-4를 다음과 같이 고치면 multiple intersection도 함께 찾을 수 있다.

```
Multiple-intersection-module:
2-4) case v of intersection-vertex {
  2-4-1) get-mc-SCL(v.pos, S);
         T=S-{MCa};
  2-4-2) for each MCj in T do {
         v.mc←MCj; //v.mc에 MCj를 등록//
         insert-fv-MC(MCj, v);
         advance-fv-MC(MCj);
         reposition-mc-ACL(MCj);
        } //end of for//
  2-4-3) swap-mc-SCL(S);
       //s0, s1: first and last elements of S//
  2-4-4) find-intersection(s0, prev(SCL, s0));
         find-intersection(s1, next(SCL, s1));
  2-4-5) add v to OVL;
}//end of case//
```

MCI-알고리즘의 Step 2-4를 multiple intersection module로 대치하면, 완전한 MCI-알고리즘을 얻을 수 있다. Multiple intersection module의 시간복잡도는 명백히 step 2-4-3(swap-mc-SCL(S))에 제약받게 되는데, 그것의 시간복잡도는 $O(s \cdot log\ s)$이다. 이때 s는 S에 속한 monotone chain들의 개수이다. Multiple intersection은 s개 보다 많은 proper intersection이 이루어진 것이므로 전체적인 MCI-알고리즘의 시간복잡도는 영향받지 않는다.

5. 제안된 알고리즘의 응용분야 및 실험결과

이미 설명된 MCI-알고리즘을 이용하여 최종적으로는 점열 곡선 꼬임 알고리즘 (PCI-알고리즘)을 다음과 같이 구성할 수 있다. n은 라인 세그먼트의 개수이며, m은 monotone chain의 개수 그리고 k는 꼬임의 개수라 하자.

1. 점열 곡선을 monotone chain들로 분할: $O(n)$
2. Monotone chain들을 입력으로 하여 MCI-알고리즘을 수행한다. $O((n+k) \cdot log\ m)$
3. 만약 입력 점열 곡선을 회전 변환하였으면 다시 반대로 회전 변환한다.

본 장에서는 제안된 PCI-알고리즘의 세 가지 응용분야를 설명한다. 주어진 다각형의 꼬임 여부 판별, 다각형들 사이의 꼬임 찾기 그리고 점열 곡선의 오프셋에 활용될 수 있다.

• 다각형의 꼬임 여부 판별

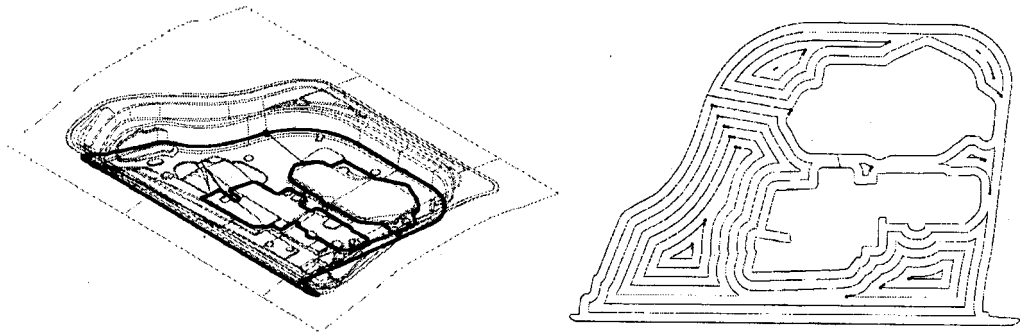
많은 다각형 관련 알고리즘이 입력으로 꼬임 없는 다각형(simple polygon)을 요구하는 경우가 많은데, 그러한 경우에 입력 다각형이 꼬임이 없는지 check하는데 유용하게 쓰일 수 있다. 입력되는 다각형을 monotone chain으로 분할한 후 첫 번째 꼬임이 발견될 때까지만 MCI-알고리즘을 수행하면 되므로 시간복잡도는 $O(n \cdot log\ m)$이 된다. 이것은 일반적으로 알려진 꼬임 여부 판별 알고리즘1 $O(n \cdot log\ n)$보다 좋으며, 특히 n이 커질수록 그 차이는 확실해 진다.

• 다각형간의 꼬임 찾기

CAD/CAM에서 다각형들 사이의 꼬임을 효율적으로 구하는 것은 매우 기본적이고도 중요한 오퍼레이션이다. 영역 가공에서의 가공영역 모델링, 자유곡면으로 이루어진 3차원 물체의 불리언 오퍼레이션에 활용될 수 있다. 영역가공에서의 area 혹은 island curve들은 촘촘한 점들로 이루어진 다각형으로 표현되는 경우가 많은데, 촘촘한 점들로 이루어진 다각형들 사이의 꼬임을 빨리 구하는 것은 제안된 알고리즘에 매우 적합하다. 3차원 물체들간의 불리언 오퍼레이션은 이차원 다각형들 사이의 불리언 오퍼레이션으로 변화될 수 있다. 이때 trimming boundary는 surface/surface intersection으로 구할 수 있는데, 그것은 이차원 parameter domain에서의 점열 곡선으로 나타나게 된다. trimming surface를 서로 상대방에 대해서 분류하는 것은 이차원 parameter domain에서의 점열 곡선들간의 꼬임을 구하는 문제와 동일하다.

• 점열 곡선 오프셋(다각형의 꼬임 찾기)

이차원 점열곡선의 오프셋은 매우 다양한 적용분야를 가지고 있는 중요한 오퍼레이션이다. 기본적인 적용분야로는 NC pocket machining[11], VLSI circuit design 그리고 robot path planning등을 들 수 있다. 이차원 curve offset은 Voronoi diagram방식[10]과 direct offset방식[11]의 크게 두 가지 방향에서 연구되어 왔다. 이미 잘 알려진 대로 다각형의 Voronoi diagram은 divide-and-conquer 방식을 적용함으로써 $O(n \cdot log\ n)$



(a) Boundary polygons by SSI-operation (b) Offset the poygon

Fig. 6. Planar-curve offset example.

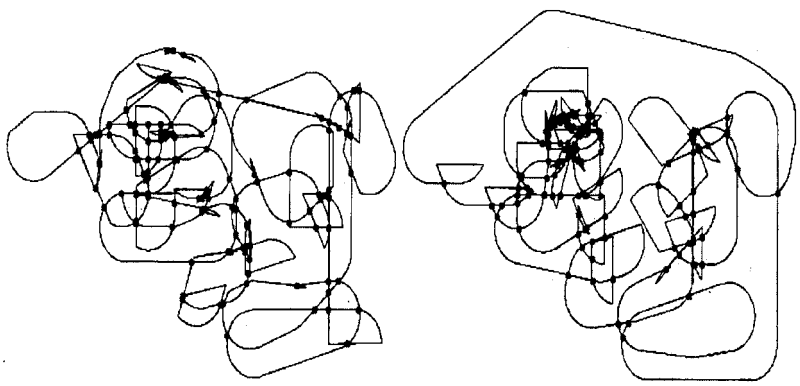
시간에 구할 수 있다. Direct offset 방식은 다각형의 각 세그먼트들을 offset하고 꼬임을 찾아내서 invalid 한 부분들을 제거함으로써 valid offset curve를 얻는 방식이다. Direct offset 방식이 개념상으로는 더 간단하지만 brute force로 구현하면 매우 많은 계산시간을 요구하게 된다. 특히 raw offset curve의 모든 self-intersection을 구하는 부분은 많은 계산을 요구하지만 아직 적절한 알고리즘이 제시된 바 없다. 우리는 제안된 PCI-알고리즘을 이용하여 많은 점으로 이루어진 다각형을 효율적으로 offset할 수 있는 알고리즘을 구현하였다. Fig. 6에는 다각형 offset의 예가 나타나 있는데, Fig. 6(a)에는 die-cavity surface를 평면으로 잘라서 생긴 점열 곡선이 두꺼운 선으로 그려져 있다. Fig. 6(b)에는 이차원 다각형 offset을 적용시켜 생긴 pocketing tool-path가 그려져 있다.

• 실험을 통한 성능분석

제안된 점열 곡선 꼬임(PCI) 알고리즘은 구현되었으며, Fig. 7에 그려진 두 가지 type의 꼬임 있는 다각형(non-simple polygon)에 대하여 실험이 수행되었다. Fig. 7(a)와 Fig. 7(b)에 그려진 두 다각형은 각각 74개와 26개의 monotone chain을 가진다(m1 =74, m2=26). 만약 연속적인 raw offsetting이 그 다각형들에 적용되면 monotone chain의 개수(m)는 변하지 않지만 라인 세그먼트의 개수(n)와 꼬임의 개수(k)는 변하게 된다. Table 1에 사용된 입력 다각형들은 이런 방식으로 만들어 졌다.

PCI-알고리즘은 C language를 이용하여 작성되었고 EWS에서 실험을 수행하였다. Table 1은 다양한 경우의 입력 다각형에 대한 수행시간을 보여주고 있다.

Fig. 8에서는 Type-1 다각형(Fig. 7(a))와 Type-2 다각형(Fig. 7(b))에 대하여 라인 세그먼트와 꼬임 개수의 합($n+k$)에 대해 수행시간을 그린 것이다. Fig. 8을 통하여 실제 계산시간은 $n+k$에 대해 선형적으로 움직인다는 것을 알 수 있다.



(a) Type 1 polygon
number monotone chains = 74

(b) Type 2 polygon
number of monotone chains = 26

Fig. 7. Non-simple polygons for performance test.

Table 1. Test results

Type1 (m1=74)			Type2 (m2=26)		
n1	*k1*	Time (sec)	*n2*	*k2*	Time (sec)
1212	105	0.0120	1718	28	0.0078
1377	135	0.0137	2211	32	0.0117
1661	165	0.0205	2952	48	0.0146
3135	63	0.0166	3260	68	0.0176
4401	91	0.0303	4386	86	0.0273
5373	136	0.0430	5368	82	0.0361
6187	203	0.0501	6193	68	0.0410

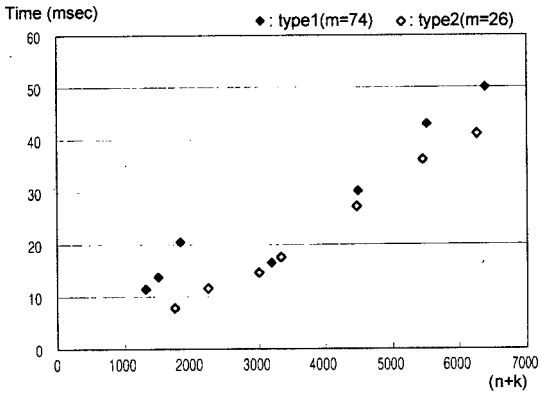


Fig. 8. Plotting of the test results.

6. 결론 및 토론

본 논문에서는 $O((n+k)\cdot log\ m)$의 시간복잡도를 가지는 점열 곡선 꼬임(Polygonal chain intersection) 알고리즘을 제안하였다. 실제 계산 시간은 $n+k$에 선형적으로 움직이는 것이 실험을 통하여 관찰되었다. 왜냐하면 monotone chain의 개수(m)이 일반적으로 라인 세그먼트의 개수(n)에 비해 매우 작기 때문이다. 뿐만 아니라 제안된 PCI 알고리즘은 tangential contact 혹은 multiple intersection과 같은 singularity가 존재하는 점열 곡선에 대해서도 안정적으로 작동한다.적용 분야로는 다각형의 꼬임여부 판별, 다각형들간의 꼬임 구하기 그리고 이차원 curve의 offset을 예로 들었다.

주어진 점열 곡선에는 꼬임의 개수(k)는 고정된 것이다. 그러나 monotone chain의 개수 m은 sweeping direction에 따라 달라질 수 있다. 즉 최소 개수의 monotone chain으로 분할을 가능하게 하는 optimal sweeping direction이 존재[10]한다는 의미이다. 또한 특정 application의 목적에 따라 proper intersection의 정의가 달라질 수도 있는데, 이것은 제안된 PCI-알고리즘을 조금만 수정하면 가능할 것이다.

참고문헌

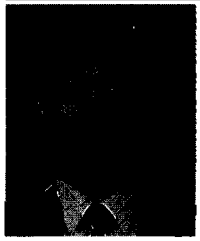
1. Shamos, M.I. and Hoey, D.J., Geometric intersection problems. *in Proc. 17th Annu. Conf. Foundation of Computer Science*, pp. 208-215, Oct 1976.
2. Bentley, J.L. and Ottmann, T.A., Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers* 28, pp. 643-647, 1979.
3. Preparata, F.P. and Shamos, M.I., *Computationalgeometry-An introduction*. Springer Verlag, New York, 1985.
4. Chazelle, B. and Edelsbrunner, H. An Optimal algorithm for intersecting line segments in the plane. *Journal of the Association for computing machinery*, Vol. 39, No. 1, pp. 1-54, January 1992.
5. Nievergelt, J. and Preparata, F.P., Plane-sweep algorithms for intersecting geometric figures. *Communications of the ACM*, Vol. 25, No. 10, pp. 739-747, 1982.
6. Edelsbrunner, H. and Maurer, H.A. Polygonal intersection searching. *Information processing letters*, Vol. 14, No. 2, pp. 74-79, 1982.
7. Bentley, J.L. and Wood, D. An Optimal worst-case algorithm for reporting intersections of rectangles. *IEEE Trans. Comput.* C-30, pp. 147-148, 1981.
8. Xue-Hou Tan, Tomio Hirata and Yasuyoshi Inagaki, The intersection searching problem for c-oriented polygons. *Information Processing Letters*, Vol. 37, No. 4, pp. 201-204, 1991.
9. Chan, T.M. and Simple, A., Trapezoid Sweep Algorithm for Reporting Red/Blue Segment Intersections. *Proc. 6th Canad. Conf. Comput. Geom.*, pp. 263-268, 1994.
10. Held, M. and A geometry-based investigation of the tool path generation for zigzag pocket machining. *The Visual Computer*, 7, pp. 296-308, 1991.
11. Allan Hansen, Farhad Arbab, An algorithm for generating NC tool paths for arbitrary shaped pockets with island, *ACM Transactions on graphics*, Vol. 11, No. 2, pp. 152-182, April 1992.

박 상 철

1994년 KAIST 산업공학과 학사
1996년 KAIST 산업공학과 석사
1996년~현재 KAIST 산업공학과 박사
관심분야: Geometric algorithms in CAD/CAM, CAPP, System modeling & simulation

신 하 용

1985년 서울대학교 산업공학과 학사
1987년 KAIST 산업공학과 석사
1991년 KAIST 산업공학과 박사
1991년~현재 미국 DiamlerChrysler의 advanced CAD/CAM engineer
관심분야: Geometric modeling, Tool path generation, Computational geometry



최 병 규

1973년 서울대학교 산업공학과 학사
1975년 KAIST 산업공학과 석사
1982년 미국 Purdue 산업공학과 박사
1982년~현재 KAIST 산업공학과 교수 및 KAIST CIM 연구센터장
관심분야: Surface Modeling, CAD/CAM, CAPP, 자동화 제조시스템 모델링 시뮬레이션