한국CAD/CAM학회 논문집 Vol. 17, No. 4, pp. 262-273. August 2012
Transactions of the Society of CAD/CAM Engineers DOI http://dx.doi.org/10.7315/CADCAM.2012.262

〈응용논문〉 ISSN 1226-0606

GPU를 이용한 삼각형 집합의 외경계 계산 알고리즘

경민호†
아주대학교 미디어학과

GPU Algorithm for Outer Boundaries of a Triangle Set

Min-Ho Kyung†
Department of Digital Media, Ajou University

Received 30 January 2012; received in revised form 18 June 2012; accepted 18 June 2012

ABSTRACT

We present a novel GPU algorithm to compute outer cell boundaries of 3D arrangement subdivided by a given set of triangles. An outer cell boundary is defined as a 2-manifold surface consisting of subdivided polygons facing outward. Many geometric problems, such as Minkowski sum, sweep volume, lower/upper envelop, Bool operations, can be reduced to finding outer cell boundaries with specific properties. Computing outer cell boundaries, however, is a very time-consuming job and also is susceptible to numerical errors. To address these problems, we develop an algorithm based on GPU with a robust scheme combining interval arithmetic and multi-level precisions. The proposed algorithm is tested on Minkowski sum of several polygonal models, and shows 5-20 times speedup over an existing algorithm running on CPU.

Key words: 3D arrangement, GPU, Minkowski sum, Outer boundary, Robustness, Triangle set

1. 서 론

본 논문에서는 많은 삼각형들로 이루어진 집합의 외경계 표면을 강건(robust)하게 계산하는 GPU(graphics processing unit) 기반 알고리즘을 제안한다. 입력 삼각형들은 3차원 공간을 여러 개의 공간 셀들로 분할한다. 이러한 공간 셀들의 경계들 중에 삼각형의 외부 면으로만 구성된 경계들을 외경계라고 정의한다. 여기서 입력 삼각형의 외부 면은 법선 벡터 방향을 보는 면이고 내부 면은 반대편을 의미한다. Fig. 1에서와 같이 공간 상에 배치된 삼각형 집합(녹색 선분들)이 주어졌을 때 외경계는 파란색 경계(점선)에 해당한다.

삼각형 집합의 외경계 표면은 다양한 3차원 기하 문제들을 계산하는데 매우 유용하다. 예를 들어, 로봇 경로 계산, 오프셋 곡면, 3차원 모핑, 최단 관통 깊이 계산 등에 응용되는 Minkowski sum 경계는 합 삼각형(sum triangles) 집합의 외경계에

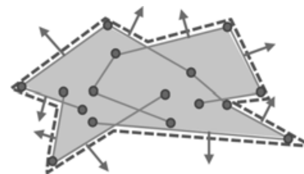


Fig. 1 Outer boundary of a triangle set (dotted lines in blue). Each green line segment represents a 3D triangle with a normal vector denoted by a red arrow.

†Corresponding Author, kyung@ajou.ac.kr
©2012 Society of CAD/CAM Engineers

해당한다[1]. 또한, 충돌 검사와 강체 모델링에 유용하게 사용되는 스윕 볼륨의 표면 또한 삼각형 집합의 외경계로 구해질 수 있다. 즉, 3차원 모델의 각 모서리들을 스윕 경로에 따라 이동시켜 룰드 곡면(ruled surface)를 만들고, 이 곡면을 삼각형화 한다. 여기서 나온 삼각형 집합들의 외경계를 구하면 스윕 볼륨 표면이 구해진다[2]. 이 외에도 3차원 모델 간의 불(Boolean) 연산[3,4], 엔벨로프(envelope) 표면 계산[5] 등도 삼각형 집합의 외경계로 구해질 수 있다.

이와 같이 다양한 응용 분야가 있지만, 삼각형 집합의 외경계 표면을 효율적이며 강건하게 계산하는 것은 쉽지 않다. 먼저 삼각형들에 의해 나누어지는 공간 셀을 구해야 하는데, 이를 위해서는 삼각형들의 3차원 배치(3D arrangement)을 계산해야 한다. 3차원 배치의 이론적 계산 복잡도는 $O(n^3)$이다[6]. 실제 예제의 복잡도는 이 보다 낮지만 여전히 매우 많은 처리량을 필요로 한다. 동시에 이 정도의 복잡도는 3차원 배치 데이터의 크기를 매우 빠르게 증가시키므로, 이를 저장하기 위한 저장 공간도 심각한 문제가 될 수 있다.

더욱 어려운 문제는 강건성을 확보하는 것이다. 3차원 배열을 구하기 위한 모든 기하학적 판단들은 수치적인 계산을 통해 이루어진다. 수치 계산은 부동소숫점 연산을 사용할 경우 필연적으로 수치 오차를 포함하게 된다. 이러한 수치 오차는 잘못된 기하학적 판단을 낳게 되고, 결과적으로 논리적인 오류를 만들어 올바른 결과를 얻을 수 없게 된다[7]. 특히, 3차원 배치 계산에서는 이러한 수치 오차에 특히 민감한 불량 조건(ill-conditioned) 경우들이 적지 않게 발생하기 때문에 강건성 확보가 쉽지 않다.

본 논문에서는 이 두 가지 난제들을 다음과 같이 해결하려고 한다. 3차원 배치의 점근적 복잡도는 낮출 수 없지만, 병렬 처리와 가지치기(pruning)을 통한 데이터 정리로 계산 시간과 저장 공간을 단축시킬 수 있다. 최근 병렬 처리에 사용할 수 있는 하드웨어로 일반 소비자용 GPU가 많은 각광을 받고 있다. 그 동안 반도체 기술의 발달과 컴퓨터 그래픽 분야의 수요 확대로 GPU의 성능은 과거 슈퍼컴퓨터에 필적하는 수준까지 발전하였다. 우리는 대부분의 기하 계산들을 GPU 상에서 처리하여 계산 성능을 획기적으로 높이려고 한다.

메모리 사용량을 낮추기 위해는 외경계에 포함되지 않는 기하 요소들을 최대한 찾아 제거하여 불필요한 데이터를 최소화하려 한다. 이러한 기하 요소들을 완벽하게 찾아 내는 것은 어려운 일이지만, 상당수의 불필요한 기하 요소들은 간단한 기하학적 추론을 통해 효율적으로 찾아낼 수 있다. 예를 들어, 한 에지가 삼각형과 교차한다면, 교차점을 중심으로 에지의 한 쪽은 삼각형의 외부 다른 쪽은 내부에 속하게 된다. 이 때 내부에 속하는 선분은 외경계에 들어 갈 수 없으므로 제거될 수 있다. 본 논문에서는 이와 같이 내부에 속하는 선분과 정점을 결정하는 규칙들을 정하고 이들을 이용하여 많은 기하 요소들을 가지치기하려고 한다.

마지막으로 알고리즘의 강건성을 보장하기 위한 방법으로 2-단계 구간 연산(interval arithmetic)과 다중정밀도 부동소숫점 연산(multiple-precision floating-point arithmetic)을 적용하여 수치 계산의 안정성을 높이려고 한다. 이 방법은 Milenkovic와 Sacks가 제안한 ACP(adaptively controlled perturbation)을 개선한 것이다[18]. ACP는 배정밀도 구간 연산과 다중정밀도 부동소숫점 연산을 사용하여 기하 판단 함수(predicate)의 값을 안전하게 결정하는 방법이다. 여기서 동일 평면 상의 삼각형들과 같은 특이한 경우(degenerate cases)들이 발생하는 것을 사전에 방지하기 위해 각 정점들의 좌표를 δ 범위 안에서 무작위로 교란시켜 놓는다. 기존 ACP는 다중정밀도 부동소숫점 연산을 사용하기 위해 MPFR 라이브러리를 사용하였다. MPFR 라이브러리에서는 임의 정밀도의 부동소숫점 수치 연산을 지원해 준다[23]. 이 라이브러리는 아직 GPU 상에서 구현되지 않았기 때문에 우리는 구간 연산만을 GPU에서 구현하고, 구간 연산으로 처리할 수 없는 불안전한 경우들만을 따로 모아 MPFR 라이브러리를 사용하여 CPU에서 계산하도록 하였다. 대부분의 입력 데이터에서 불안전한 경우들은 매우 드물게 발생하므로 MPFR 사용으로 인한 성능 저하는 우려할 만한 정도는 아니다. 그리고, 우리는 추가로 GPU에서의 구간 연산을 두 단계로 나누어 구현한다. 배정밀도 연산은 단정밀도 연산에 비해 GPU에서 4~5배 정도로 시간이 많이 걸리기 때문에, 첫 단계에서는 단정밀도 구간 연산으로 계산을 하고 여기서 실패할 경우 배정밀도 구간으로 바꾸어 좀 더 정밀하게 계산을 한다. 이러한 2-단계 구간 연산은 아직 ACP에서 시도되지 않은 방식이다.

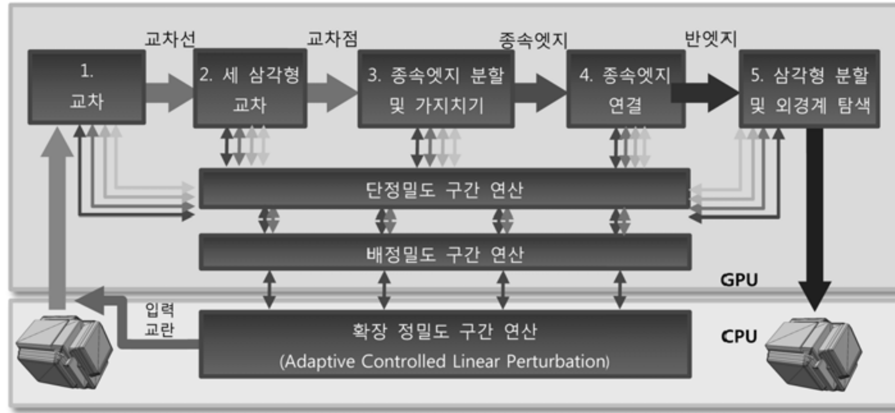


Fig. 2 Pipeline of outer-boundary algorithm

제안된 알고리즘은 Minkowski sum 계산에 적용하여 알고리즘의 성능과 강건성을 검증한다. Minkowski sum의 경계는 3차원 콘볼루션(convolution)으로부터 나온 합삼각형(sum triangles)의 외경계가 된다. 이 합삼각형들의 집합은 $O(n^2)$의 복잡도를 가지고 특이 경우들을 많이 포함하므로 빠르고 안정적으로 외경계를 구하는 것이 쉽지 않다. 우리는 제안된 알고리즘을 적용하여 합삼각형들의 외경계를 빠르고 안정적으로 구함으로써 기존의 연구 결과([1])에 비해 20배에 이르는 성능을 가지는 Minkowski sum 알고리즘을 구현하였다.

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구들에 관해 논의하고, 3장에서는 제안된 외경계 알고리즘에 관해 기술한다. 4장에서는 알고리즘 강건성을 위한 다단계 정밀도 방법을 설명한다. 5장에서는 Minkowski sum 알고리즘의 구현과 실험 결과를 설명하고, 6장에서 결론과 향후 연구에 관해 제시한다.

2. 관련 연구

계산 기하학 분야에서 평면 또는 공간의 기하 배치와 관련된 문제는 가장 핵심적이고 중요하게 연구되어온 주제라고 할 수 있다. 배치는 주어진 선, 면, 곡면 등의 기하요소들에 의해 나누어진 평면 또는 공간 분할로 정의된다. 배치의 복잡도와 계산 알고리즘에 관해서는 그 동안 많은 연구 결과들이 발표되어 왔는데, 그 중에서 중요한 연구 결과들은 참고문헌 [8], [11], [12]에 잘 정리되어 있다.

삼각형 집합의 외경계는 3차원 배치의 부구조(substructure) 중의 하나이다. 따라서 3차원 배치를 계산할 수 있다면 외경계는 쉽게 찾아낼 수 있다. Arnovo와 Sharir[6]는 삼각형 집합에 의한 3차원 배치의 한 공간 셀을 $O(n^{2+\varepsilon})$ 시간에 구하는 알고리즘을 제안하였고, de Berg 등[9,10]은 수직 분할과 점진적 랜덤 생성을 이용하여 같은 복잡도를 가지면서 좀 더 효율적인 알고리즘을 제안하였다. 이러한 알고리즘들은 수치 오차가 발생하지 않는 real RAM계산 모델에서 동작하는 것을 가정하고 있기 때문에 수치 오차가 발생하는 경우 올바른 결과를 얻을 수 없고, GPU와 같은 SIMD 구조를 갖는 하드웨어에서 병렬적으로 구현하기에는 지나치게 복잡한 단점이 있다.

지난 수 년간 GPU의 높은 성능을 이용하여 많은 계산량이 요구되는 문제들을 가속하는 연구가 활발히 진행되고 있다. 이러한 연구들은 주로 행렬연산, 이미지 처리, 3차원 렌더링 등과 같이 대량의 데이터에 작은 단위의 독립적인 연산이 일괄적으로 적용되는 문제들에서 수행되어 매우 성공적인 성능 향상을 얻어 왔다. 최근에는 복잡한 자료구조를 필요로 하는 기하 문제를 GPU로 가속하는 연구들도 시도되고 있다. Krishnamurthy 등은 매개변수 공간을 계층적으로 분할하여 NURBS 곡면의 실루엣 곡선, 점/곡면 최소 거리, 곡면/곡면 최소 거리 등을 구하는 GPU 알고리즘들을 제안하였다[13]. Li와 McMains는 3차원 그리드를 만들고, 두 다면체의 합 삼각형들이 포함되는 그리드 셀을 찾아 Minkowski sum 경계를 구하는 GPU 알고리즘을 제안하였다[14]. 이 외에도 많은 GPU 관련 연구가 있었지만, 3D 삼각형 집합에 대한 외경

계 계산을 GPU로 가속하는 연구는 본 논문에서 처음으로 시도하는 것이다.

기하 알고리즘을 유한정밀도 연산을 사용하는 일반 컴퓨터에서 강건하게 구현하기 위하여 여러 가지 방법들이 제안되었는데, 그 중에서 비교적 계산 시간과 저장공간의 사용에서 가장 효과적인 방법은 Halperin 등이 제안한 통제된 교란(controlled perturbation) 기법이다[15-17]. 기본적인 아이디어는 입력 기하 데이터를 사전에 정해진 δ 범위 안에서 무작위로 교란시켜 수치 오차에 영향을 받는 경우를 원천적으로 차단하는 것이다. 이 방법을 개선하여 계산 과정 중에 입력 데이터에 따라 δ를 최적으로 설정하는 통제된 선형 교란법(controlled linear perturbation)이 Sacks 등에 의해 제안되었다[1]. Milenkovic와 Sacks는 구간 연산과 다중정밀도 연산을 혼합한 새로운 통제된 교란법(ACP)을 제안하였고, 본 논문에서는 이 ACP를 GPU알고리즘과 결합하여 강건성을 확보하였다.

3. 외경계 알고리즘

3.1 문제 정의

주어진 삼각형들의 집합을 $T=\{t_0,t_1,\ldots,t_{n-1}\}$ 라고 하자. 각 삼각형 t_i는 법선 벡터에 의해 방향이 정해져 있다. 즉, 법선 벡터 방향의 면은 외부면, 반대 방향의 면은 내부면으로 정한다. T는 3차원 공간을 유한 개의 셀들로 나누게 된다. 각 셀 C의 경계를 ∂C 라고 할 때, ∂C 는 삼각형의 부다각형(subpolygon) $p_j \in t_i$ 로 구성된다. 만일 ∂C 의 각 부다각형 p_j가 t_i의 외부면과 같은 방향이고 서로 연결되어 2-다양체(2-manifold)를 구성한다면, 이 ∂C 를 본 논문에서는 외경계라고 정의한다.

두 다면체의 Minkowski sum 경계는 합삼각형들의 외경계가 된다. 왜냐하면 Minkowski sum 경계는 2-다양체이며 합삼각형들의 외부면과 같은 방향인 부다각형들로 구성되기 때문이다. 마찬가지로 한 다면체를 이동시켜 얻어지는 스윕 볼륨도 2-다양체이면서 표면을 구성하는 다각형들이 스윕 삼각형 외부면의 일부가 되므로 스윕 삼각형들의 외경계가 된다. 이외에도 다면체 간의 불 연산과 엔벨로프 등도 삼각형 집합의 외경계가 된다.

3.2 GPU 알고리즘

우리는 삼각형 집합의 모든 외경계를 찾는 것을 목표로 한다. 외경계를 구하는 기본적인 아이디어는 먼저 삼각형 집합의 배치를 구하고, 에지를 통해 연결된 외부 부다각형들을 탐색하여 2-다양체를 구성하는 집합을 찾아낸다. 이를 단계적으로 세분화하면, 1. 삼각형 교차, 2. 세 삼각형 교차, 3. 엣지 분할과 가지치기, 4. 엣지 연결, 5. 삼각형 분할 및 외경계 탐색으로 나눌 수 있다. 단계 1-4까지는 삼각형 집합의 배치를 구하는 과정이고, 마지막 단계가 부다각형 연결 관계를 따라 외경계를 탐색하는 과정이다. 전체 알고리즘의 구조도는 Fig. 2에 도시되어 있다. 다음에서 각 단계에 대한 자세한 설명을 하겠다.

3.2.1 삼각형 교차

배치에 포함되는 모든 기하 요소들의 분할은 삼각형들 간의 교차에 의해 발생하게 된다. 따라서 주어진 삼각형 집합 내에서 교차하는 삼각형 쌍들을 모두 찾아 내는 것이 우선적으로 필요하다. 교차 삼각형 쌍들을 찾는 기존의 GPU 알고리즘은 두 가지가 있다. 먼저 입력 삼각형들이 낮은 밀도로 분포할 경우 교차 횟수가 적기 때문에 BVH (bounding volume hierarchy)를 이용한 탐색이 가능하다. 실시간 충돌 검사 분야에서 많이 사용되는 방법으로 효율적인 GPU 알고리즘이 개발되어 있다. 하지만 입력 삼각형들이 밀집되어 있는 경우에는 탐색할 BV쌍의 수가 탐색 레벨의 제곱에 비례해서 증가하기 때문에 성능이 매우 떨어지는 단점이 있다.

밀집된 삼각형 집합에 대해서는 계층적 공간 분할을 이용하여 적은 공간에 모여 있는 삼각형 집합들을 구하고, 이 각각의 집합 안에서 교차 쌍을 찾는 것이 더 효율적이다. 경민호 등은 [19]에서 이러한 밀집된 삼각형 집합에서의 교차를 효율적으로 탐색하는 GPU 알고리즘을 제안하였다. 우리는 이 알고리즘에서 불안전한 삼각형 쌍들을 처리하는 부분을 ACP를 사용하도록 수정하여 삼각형 교차 계산에 적용하였다.

이 삼각형 교차 알고리즘의 유사코드는 Fig. 3에 기술되어 있다. 모두 5단계로 나누어진다. 단계 1에서는 입력 삼각형 집합을 하나의 노드(T_0)로 만들어 노드 리스트 T에 삽입하고, 초기 분할 축을 설정한다. 단계 2는 반복적으로 에 있는 각 노드들을 k축 방향의 중간에서 분할하여 두 개의 자식 노드들로 세분화해 간다. 노드의 삼각형 수가 m개

1. $\mathbf{T} \leftarrow \{T_0\}$; $k \leftarrow x - \mathbf{axis}$;
2. While $\mathbf{T} \neq \emptyset$ do
   Parallel segmented scan on $\mathbf{T}$ to find $r_i = (\min(T_i), \max(T_i))$ for every $T_i \in T$.
   GPU thread for $T_i \in T$:
      Subdivide T_i at midpoint of r_i into child nodes $T_{i,L}$ and $T_{i,R}$, stored into $\mathbf{T}'$.
   GPU thread for $T_i' \in \mathbf{T}'$:
      If T_i' contains less than m polygons then
         move T_i' from $\mathbf{T}'$ to leaf node list $\mathbf{L}$.
   $\mathbf{T} \leftarrow \mathbf{T}'$; $k \leftarrow$ ***next split axis***;
3. GPU thread for leaf node $T_i \in \mathbf{L}$:
   For two triangles $t_a, t_b \in T_i$,
      if $\mathbf{sp}(t_a) \wedge \mathbf{sp}(t_b) = \mathbf{0}$ then
         insert (t_a, t_b) into $\mathbf{P}$.
4. GPU thread for each $(t_a, t_b) \in P$:
   Compute an intersection edge.
   If unsafe, then insert (t_a, t_b) into $P_{\mathbf{unsafe}}$.
5. For $(t_a, t_b) \in P_{\mathbf{unsafe}}$,
   compute an intersection edge on CPU.

Fig. 3 GPU algorithm of triangle intersection

이하인 경우 분할을 중지하고 리프 노드 리스트 L로 옮긴다. 단계 3과 4에서는 L에 포함된 각 리프 노드의 삼각형들에서 교차하는 삼각형 쌍들을 찾아 교차선을 계산한다. $sp(t_a)$는 t_a가 분할되는 과정에서 왼쪽과 오른쪽 자식 노드에 모두 포함되는 레벨들을 인코딩한 정수이다. 따라서, $sp(t_a) \wedge sp(t_b) \neq 0$이라면 t_a와 t_b가 왼쪽 자식 노드에서 유래된 다른 리프 노드에도 동시에 포함되기 때문에 현재 리프 노드에서는 P에 삽입하지 않는다([19] 참조). 두 삼각형이 거의 붙어 있거나 같은 평면 상에 놓인 경우 수치적으로 불안정하므로 리스트 P_{unsafe}에 별도로 보관한다. P_{unsafe}에 보관된 삼각형 쌍들은 CPU 상에서 다중정밀도 구간 연산을 사용하여 교차 여부를 계산한다. 자세한 내용은 4장에서 설명하겠다.

이 알고리즘은 계층적으로 이분 분할을 하므로 평균 시간 복잡도는 $O(n\log n + p)$ 가 되고, 최악의 경우는 $O(n^2)$가 된다. 여기서 p는 교차 삼각형 수를 의미한다.

3.2.2 세 삼각형 교차

공간 상에서 세 삼각형의 교차는 드물지 않은 일반적인(generic) 경우로 최악의 상황에서 $O(n^3)$번의 교차가 발생할 수 있다. 따라서 삼각형 집합의 배치 계산에서 가장 시간이 많이 걸리는 과정이라고 할 수 있다.

세 삼각형 교차를 효율적으로 계산하기 위한 방법으로 3.2.1에서와 마찬가지로 공간 분할을 사용할 수 있다. 세 삼각형의 교차점은 세 교차선의 교차점으로 볼 수도 있다. 따라서 앞에서 구한 교차선들을 계층적 공간 분할을 이용하여 작은 공간 안에 모여 있는 교차선들의 집합들로 세분화한 후에 각각의 집합 내에서 교차하는 교차선들을 찾아내는 것이다.

두 번째 방법으로는 공간 분할을 사용하지 않고, 각 삼각형에 대하여 그 위의 모든 교차선들 간에 교차 검사를 수행하고 교차점을 찾는 것이다. 이 방법의 시간 복잡도는 각 삼각형 위의 교차선이 k개 이하일 때 $O(k^2n)$가 된다. 따라서, 삼각형들의 밀집도가 높아 $k \rightarrow n$이 된다면, $O(n^3)$이 된다.

위 두 방법을 비교하면 계층적 분할 방법이 평균적으로 더 효율적일 것으로 생각된다. 하지만 실제 실험에 의하면 두 번째 방법과 수행 시간의 차이가 거의 없고, 밀집된 삼각형 집합에 대하여는 오히려 계산 시간이 급격히 증가하는 것을 발견할 수 있었다. 그 이유는 계층 분할을 GPU로 수행하는데 필요한 부가 비용이 두 번째 방법에 비하여 매우 높고, 밀집된 삼각형 집합에서는 교차선 집합이 교차 검사에 적합한 일정 크기 이하로 되기까지 상당히 많은 분할을 필요로 하기 때문이다. 따라서, 우리는 세 삼각형 교차 계산을 위하여 두 번째 방법을 선택하였다.

1. $\mathbf{S} \leftarrow \{(t_a, t_b) | t_a \cap t_b \neq \emptyset\}$
2. $\mathbf{S_2} \leftarrow \{(t_a, t_b), (t_b, t_a) | (t_a, t_b) \in S\}$
3. Sort $\mathbf{S_2}$ by using GPU radix sort.
4. Put all $(t_a, t_b) \in S_2$ into group G_{t_a}.
5. GPU thread for each G_{t_a}:
   For each (t_a, t_b), $(t_a, t_c) \in G_{t_a}$,
      compute intersection of t_a, t_b, and t_c,
      if they are an unsafe triple then
         inset (t_a, t_b, t_c) into $T_{\mathbf{unsafe}}$.
6. For each $(t_a, t_b, t_c) \in T_{\mathbf{unsafe}}$,
   compute intersection on CPU.

Fig. 4 GPU algorithm of three triangle intersection

Fig. 4는 세 삼각형 교차 알고리즘을 유사코드로 보여주고 있다. 단계1에서는 교차 삼각형 쌍들의 집합 S를 만들고, 각 삼각형 쌍들을 자신과 뒤바뀐 쌍으로 만들어 집합 S_2에 넣는다(단계2). 단계3에서는 GPU radix sort를 사용하여 S_2의 삼각형 쌍들을 정렬하고, 여기서 첫 번째 삼각형이 같은 쌍들끼리 묶어서 하나의 그룹으로 만든다. 다음 각 GPU 쓰레드에서 한 그룹에 포함된 삼각형 쌍들의 조합에 의해 구해진 세 삼각형 t_a, t_b 그리고 t_c에 대하여 교차 검사를 한다. 교차하는 경우 세 삼각형의 교차점을 출력한다. 만일 이 세 삼각형이 가깝게 접해 있거나 한 평면 위에 놓은 있는 경우 T_{unsafe}에 추가한다. T_{unsafe}의 (t_a, t_b, t_c) 는 단계6에서 다중 정밀도 구간 연산에 의해 정밀한 교차 검사를 받게 된다.

3.2.3 엣지 분할과 가지치기

교차선과 교차점이 모두 구해졌으므로 다음으로 엣지 분할을 할 차례이다. 삼각형 집합의 배치에 포함되는 엣지는 두 종류가 있다. 삼각형을 구성하는 엣지와 두 삼각형의 교차선이다. 이 두 종류의 엣지들은 서로 다른 종류의 교차점들에 의해 분할된다. 먼저 삼각형 엣지는 교차선의 한 끝점인 엣지/삼각형 교차점에 의해 분할된다(Fig. 5). 반면에 교차선을 분할하는 점들은 세 삼각형의 교차점이 된다(Fig. 6). 이 교차점들은 이미 3.2.1과 3.2.2에서 모두 구해졌기 때문에 여기서는 엣지의 종류에 따라 이 교차점을 모으고 정렬하기만 하면 된다.

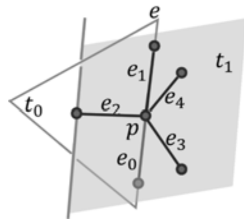


Fig. 5 Subedges made by intersection point p on edge e of triangle t_0

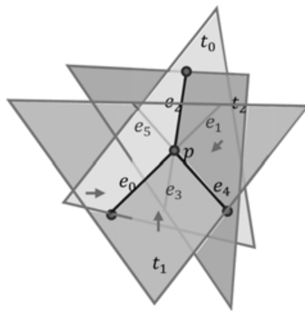


Fig. 6 Six subedges made by intersection point p of three triangles t_0, t_1, t_2

엣지/삼각형 교차점들은 교차하는 엣지에 따라 모아주고, 교차선의 교차점들은 교차하는 세 교차선으로 모아주면 된다. 먼저 교차점 p_i가 놓여 있는 엣지 $e_j = (t, h)$ 위에서의 위치값을 다음과 같이 부여한다:

$$l_i = (p_i - t) \cdot (h - t)$$

세 삼각형의 교차점은 세 개의 교차선에 대하여 각각 l_i를 구하게 된다. 다음 각 교차점의 정렬 키를 $k_i = (e_j, l_i)$ 으로 정의하고, GPU radix sort를 이용하여 교차점들을 정렬한다. 정렬된 교차점들은 같은 엣지 위에 있는 점들이 위치값의 증가하는 순서로 모이게 된다. 다음 각 엣지 위의 교차점들의 순서에 따라 엣지를 분할하여 종속엣지(subedge)들을 생성한다. 즉, 엣지 (t, h) 위의 교차점 $p_0, \ldots, p_{m-1}$ 들이 온다면, 생성되는 부엣지들은

$$(t, p_0), (p_0, p_1), \ldots, (p_{m-2}, p_{m-1}), (p_{m-1}, h)$$

가 된다.

이 부엣지들로부터 삼각형 분할을 구하기 위해서는 이들의 연결 관계를 표현할 수 있는 자료 구조가 필요하다. 우리는 평면 위의 배치를 표현하는데 많이 사용 하는 반엣지 자료구조(half-edge data structure)를 확장하여 사용한다. 그런데, 앞의 서론에서 언급한대로 삼각형 집합의 배치 복잡도는 $O(n^3)$으로 매우 높기 때문에 배치 데이터를 완전하게 저장하기 위해서는 많은 저장 공간이 필요하다. 이를 피하기 위해 불필요한 종속엣지를 가려내는 가지치기를 먼저 수행한다.

가지치될 종속엣지들은 삼각형의 내부면 공간에 존재한다고 확실히 판단되는 종속엣지들이다. 이들은 외경계에 포함되지 않으므로 안전하게 제거될 수 있다. 이러한 엣지들을 내부 종속엣지라고 부르겠다. 내부 종속엣지의 판단은 먼저 교차하는 삼각형의 방향을 통해 결정하게 되고, 다음 여기에 연결된 다른 내부 종속엣지들을 연쇄적으

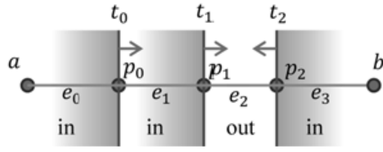


Fig. 7 Subedges made by triangle/edge intersetions

로 결정하는 방식으로 진행한다. 내부 종속엣지들은 각 교차점에서 독립적으로 결정되므로, 하나의 GPU thread에 하나의 교차점을 할당하여 처리하도록 하였다.

먼저 한 삼각형 엣지에서의 교차점들을 보자. Fig. 7를 보면 삼각형 t_0, t_1, t_2가 엣지 $e = (a, b)$와 p_0, p_1, p_2에서 각각 만나고, 이 교차점들에 의해 종속엣지 e_0, e_1, e_2, e_3이 만들어진다. 여기서 e_0, e_1, e_3는 교차하는 삼각형의 내부면 바로 아래에 놓여 있기 때문에 외경계에 들어갈 수 없는 내부 종속엣지들이 된다. 세 삼각형의 교차점에서는 세 개의 교차선이 만나게 되고, 여기서 여섯 개의 종속엣지가 만들어지고 이 중에 항상 세 개의 내부 종속엣지가 포함된다. Fig. 6을 보면, e_1, e_3, e_5는 각각 t_2, t_1, t_0의 내부면 바로 아래에 놓여 있는 내부 종속엣지이다.

이와 같이 일차적으로 교차삼각형과의 상대적인 위치로 내부 종속엣지가 결정되고 나면 이 들과 연결된 종속엣지들을 따라 연쇄적으로 추가 내부 종속엣지들을 결정해 간다. 여기서는 세 가지 경우로 나누어 판단한다. 첫 번째는 삼각형 엣지의 종속엣지 (e_0, e_1)와 교차선의 종속엣지(e_2, e_3, e_4)가 연결된 경우이다(Fig. 5). 만일 e_0, e_1이 모두 내부 종속엣지라면 이 둘을 나누는 교차점 p 역시 내부에 위치하는 점이라고 할 수 있고, 따라서 여기 연결된 교차선의 종속엣지들도 모두 내부 종속엣지가 된다. 역으로 교차선의 종속엣지들이 모두 내부 종속엣지라면 e_0와 e_1도 내부 종속엣지가 되어야 한다. 두 번째 세 삼각형의 교차에서 내부에 포함되지 않는 세 개의 종속엣지(Fig. 6의 e_0, e_2, e_4) 중에 한 개라도 내부 종속엣지에 해당되면 나머지 두 개도 반드시 내부 종속엣지가 되어야 한다. 이러한 규칙에 의해 내부 종속엣지들을 추가하는 과정을 반복함에 따라 더 많은 내부 종속엣지들을 찾아낼 수 있다. 실험에 의하면 보통 두 번까지 상당 수의 내부 종속엣지들이 추가되고, 이후에는 추가되는 엣지들의 수가 급속히 감소한다. 따라서 두 번까지만 이 과정을 반복한다.

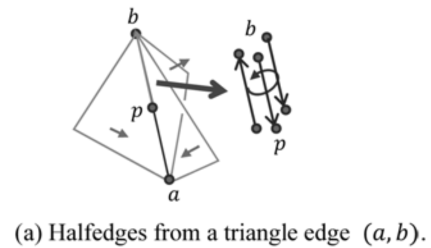
내부 종속엣지는 바로 제거되지 않고 반에지 자료구조를 생성할 때까지 GPU 버퍼에 그대로 남겨 둔다. 그 이유는 정확한 반에지 연결을 위해서는 각 교차점에 연결된 모든 종속엣지를 알고 있어야 하기 때문이다. 반엣지 자료구조가 생성된 후에 내부 종속엣지를 비롯한 모든 종속엣지들은 바로 삭제된다.

3.2.4 종속엣지 연결

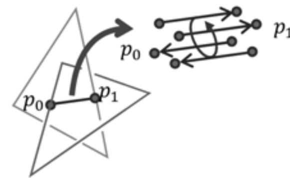
엣지 연결은 반엣지 자료구조의 생성 과정에 포함된다. 먼저 내부 종속엣지를 제외한 모든 종속엣지들에 대하여 반엣지들을 생성한다. 삼각형 엣지에서 나온 종속엣지들은 삼 각형 엣지를 공유하는 삼각형들의 개수만큼 반엣지들을 생성하고(Fig. 8(a)), 교차선에서 분할된 종속엣지들은 네 개의 반엣지들을 생성한다(Fig. 8(b)).

같은 종속엣지에서 나온 반엣지들은 서로 쌍반엣지(twin half-edge)가 된다. 평면 배치에서 반엣지들은 하나의 쌍반엣지만을 가지는 반면에 여기서는 여러 개의 쌍반엣지들이 나오게 된다. 따라서 같은 종속엣지에서 나온 반엣지들은 리스트로 연결하는데, 이 리스트의 순서는 작은 번호의 끝점에서 큰 번호의 끝점 ($P < b$, $p_0 < p_1$)으로 향하는 벡터를 축으로 반시계 방향으로 정한다.

다음은 정점과 교차점에 연결된 반엣지들을 분할면의 방향을 고려하여 서로 연결한다. 먼저 엣지/삼각형 교차점에서는 삼각형 엣지에서 온 반엣

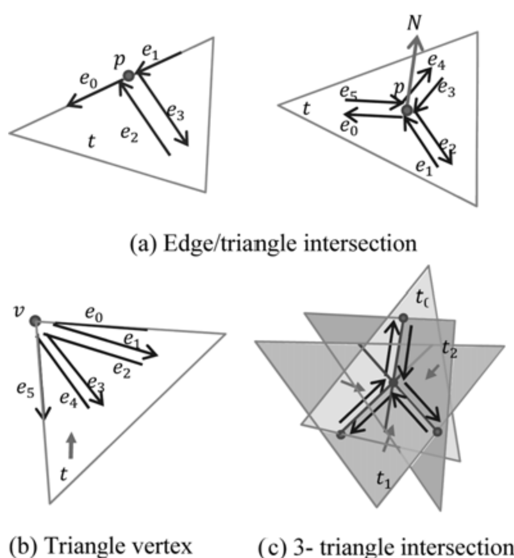
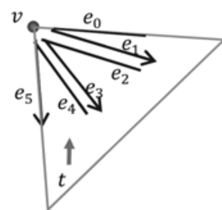
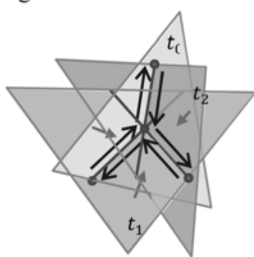


(a) Halfedges from a triangle edge (a, b).



(b) Halfedges from intersection segment (p_0, p_1).

Fig. 8 Halfedges generated from subedges

(a) Edge/triangle intersection

(b) Triangle vertex (c) 3- triangle intersection

Fig. 9 Halfedge connection

지 e_0, e_1과 같은 삼각형 t에 놓인 반엣지 e_2, e_3를 Fig. 9(a)의 왼쪽 그림과 같이 연결한다. 교차선에서 온 반엣지들은 오른쪽 그림과 같이 먼저 삼각형의 법선 벡터를 중심으로 시계방향으로 정렬하고, 순서대로 연결을 하면 된다. 삼각형 정점에서 만나는 반엣지들도 마찬가지로 법선벡터의 시계방향으로 정렬하고 순서대로 연결해 준다(Fig. 9(b)). 세 삼각형의 교차점에서 만나는 반엣지들은 같은 삼각형 위의 반엣지들을 반시계 방향 순서대로 연결해 준다(Fig. 9(c)).

3.2.5 삼각형 분할 및 외경계 탐색

반엣지들이 연결되고 나면 하나의 공간 그래프가 완성되고, 이 그래프에서 반엣지들로 연결된 루프는 삼각형의 외부면을 분할하는 다각형이 된다. 반엣지 루프를 찾기 위해 먼저 m개의 반엣지를 선택하고, 이들을 m개의 GPU threads에 각각 배정한다. GPU thread i에 배정된 반엣지 e_i에서부터 차례로 연결된 반엣지들을 따라가다 다시 e_i로 돌아오면 루프를 발견한 것이다.

이 때 방문한 반엣지들에 분할 다각형 ID를 부여하고 탐색을 마친다. 만일 중간에 반엣지 연결이 끊어진 경우가 발생하면 e_i가 포함된 다각형이 존재하지 않는 것을 의미하므로 탐색을 중단한다. 모든 GPU threads가 끝나고 방문하지 않은 반엣지가 존재한다면 이들을 모아 다시 탐색 과정을 반복한다.

모든 분할 다각형들이 구해지고 나면 외경계 탐색을 시작한다. 외경계는 2-다양체를 형성하는 분할 다각형들의 집합이다. 따라서, 쌍반엣지를 따라 연결된 분할 다각형들의 집합들을 구하고, 이 중에 2-다양체가 되지 않는 집합들을 제거하면 된다. 2-다양체 여부는 분할 다각형 엣지의 반시계 방향으로 첫번째 쌍반엣지로 간단히 판단할 수 있다. 쌍반엣지가 속한 다각형이 존재하지 존재하지 않는다면 현재 다각형 한 변에 인접한 다각형이 없으므로 2-다양체를 형성할 수 없다.

분할 다각형 집합은 병렬 union-find 알고리즘을 이용하여 구하였다. 먼저 집합은 트리 구조로 저장하고, 각 다각형이 소속된 집합은 트리의 루트 노드에 저장된 다각형을 참조하여 구별한다. 초기에는 각 분할 다각형을 루트 노드로 하는 하나의 단위 집합으로 만들고, 각 분할 다각형을 GPU threads에 각각 배정한다. 각 GPU thread i에서는 배정된 분할 다각형 p_i의 각 엣지를 통해 연결된 인접 분할 다각형 r을 찾고, 두 다각형이 포함된 집합들을 합집합한다. 합집합 연산은 한 집합의 루트 노드를 다른 집합의 루트 노드에 연결시켜 0(1) 시간에 할 수 있다. 합집합의 루트 노드는 두 집합에서의 루트 다각형의 ID가 작은 쪽으로 선택한다. 이 때 경로 압축(path compression)을 함께 수행하여, 트리의 높이가 낮게 유지되도록 해준다.

위의 병렬 union-find는 비동기적으로 수행되므로 하나의 2-다양체가 여러 개의 분할 다각형 집합으로 나올 수 있다. 예를 들어 Fig. 10과 같이 GPU thread i와 j가 sep(p_i)∪set(r_4)와 set(p_j)∪set(r_4)를 동시에 처리하는 경우가 발생할 수 있다. 만일 r_4가 r_0와 r_2보다 작다면 r_4는 r_0 또는 r_2로 연결되게 되고, 이 때 p_i, r_4, p_j는 서로 인접해 있음에도 불구하고 두 개의 다른 집합으로 나뉘게 된다. 이러한 집합들을 하나로 만들기 위해 위의 union-find 알고리즘을 한번 더 수행한다.

최종적으로 같은 집합에 소속된 분할 다각형들

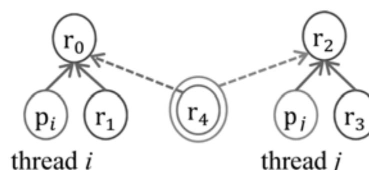


Fig. 10 GPU thread I and j attempt set(p_i)∪(r_4) and set(p_j)∪(r_4) at the same time

을 모아 출력하면 각 외경계를 구성하는 다각형들을 얻게 된다.

4. 알고리즘 강건성

3차원 배치와 같은 기하 알고리즘들의 해는 주어진 기하 요소들간의 조합 성질로부터 구해진다. 조합 성질은 함수값이 –1 또는 1인 유한 개의 판단 함수(predicate)들로 결정된다. 기하 문제에서는 주요 판단 함수들이 대수식의 부호로 정의되는데, 본 논문의 외경계 알고리즘에는 다음과 같은 판단 함수들을 사용한다:

$$f_0 = \mathrm{sign}(N\cdot(p-r))$$
$$f_1 = \mathrm{sign}(N\cdot(p-r)\times(q-r))$$
$$f_2 = \mathrm{sign}(x_0y_1-y_0x_1)$$
$$f_3 = \mathrm{sign}((p-r)\cdot(q-r))$$
$$p,q,r,N\in R^3,\ (x_0,y_0),(x_1,y_1)\in R^2$$

만일 이 판단함수들의 계산에서 sign(·) 안의 수식값이 0에 가까울 경우에는 계산 과정의 작은 반올림 오차에 의해서도 부호가 바뀔 수 있다. 오차에 의해 하나의 부호라도 바뀔 경우 전체 조합 성질에 오류가 생길 수 있어 잘못된 결과가 나오거나 최악의 경우 프로그램이 중단되는 사태가 발생할 수 있다.

본 논문에서는 이러한 문제를 해결하기 위해 Milenkovic와 Sacks가 제안한 ACP를 사용하였다[18]. ACP는 구간 연산(interval arithmetic)과 다중정밀도 연산을 함께 사용한다. 먼저 삼각형의 중첩과 같은 특이 경우들을 피하기 위해 기하 데이터를 $[-\delta,\delta]$ 범위 안에서 임의 교란(random perturbation)하는 전처리를 수행한다. 다음 판단함수마다 수식을 배정밀도(double precision)를 사용한 구간 연산으로 계산한다. 계산 결과는 참값이 a라면 구간 $[a-\varepsilon_l, a+\varepsilon_u]$으로 나오게 된다. 만일 $a-\varepsilon_l>0$ 또는 $a+\varepsilon_u<0$이면 수식의 부호는 반올림 오차에 영향을 받지 않으므로 안전하게 판단함수의 부호를 결정할 수 있다. 만일 이 구간에 0이 포함된다면 부호를 결정할 수 없으므로, 더 정밀한 구간을 얻을 수 있는 다중정밀도 연산으로 넘어간다. 먼저 다중정밀도 연산의 유효숫자를 250 bit로 설정하여 구간 연산을 행하고, 여전히 0을 포함한 구간이 나오면 유효숫자를 2배로 증가시켜 다시 재계산한다. 우리는 다중정밀도 구간 연산을 위해 Milenkovic와 Sacks에 의해 개발된 ACP library와 freeware인 MPFL을 사용하였다.

Table 1 Data size of input polyhedral models

	knot	torus	dragon	helix
Triangles	992	2,068	2,328	4,012

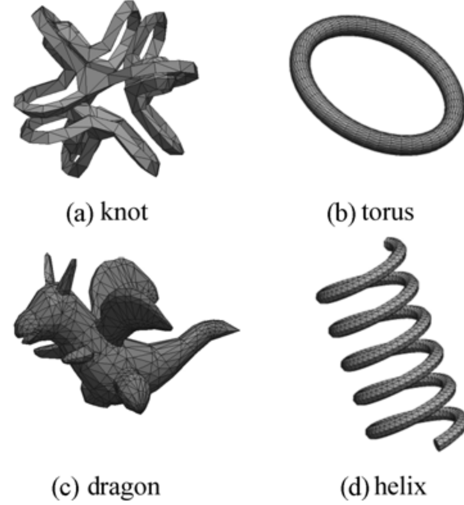


Fig. 11 Input polyhedral models

ACP는 원래 CPU에서 단일 thread 또는 multi-thread 프로그래밍에 맞게 개발되었다. 우리 알고리즘은 GPU에서 구동되기 때문에 ACP를 GPU에 맞게 수정해야 하지만, 아직 다중정밀도 연산이 GPU에서 지원되지 않으므로 완벽한 통합은 어렵다. 그래서 배정밀도 구간 연산까지만을 GPU에서 처리하고, 여기서 처리할 수 없는 판단함수들만을 따로 MPFL을 사용하여 CPU에서 처리한다. 그 결과는 다시 GPU로 전달되어 다음 과정에 사용되도록 한다.

배정밀도 연산은 GPU에서 단정밀도 연산에 비해 4배 정도 더 많은 시간이 걸린다. 반면에 대부분의 판단 함수들은 단정밀도 연산으로도 충분히 정확한 결과를 얻을 수 있다. 따라서 우리는 먼저 단정밀도 구간 연산을 적용하여 대부분의 판단함수를 계산하고, 0을 포함하는 구간이 나올 경우 배정밀도 구간 연산으로 전환하여 다시 구간을 계산하도록 하였다.

5. 실험 결과

5.1 Minkowski Sum

두 다면체 A와 B가 주어져 있을 때, 이들 간의 Minkowski sum은 다음과 같이 정의된다:

$$M = A \oplus B = \{a+b | a \in A \text{ and } b \in B\}$$

Minkowski sum은 기하 계산을 포함하는 많은 분야에서 매우 응용된다. 예를 들어 로보틱스 분야의 전통적인 문제 중 하나인 충돌 회피 경로 계산 문제는 Minkowski sum이 계산된다면 쉽게 해결되는 문제이다([20] 참조). 로봇 A와 장애물 B가 주어졌을 때, 집합 $M = (-A) \oplus B$는 A가 B와 충돌하는 이동 위치를 정의한다. 따라서 M에 포함되는 위치만을 따라서 이동하면 A가 B와 충돌하지 않고 목표지점까지 갈 수 있게 된다. 이 외에도 형상의 오프셋 모델링, 두 물체 간의 투과 깊이(penetration depth) 계산, 3차원 몰핑 연산, 컴퓨터 애니메이션, 부품 레이아웃, 제품 조립 등의 다양한 문제들을 해결하는데 유용하게 사용될 수 있다.

우리는 제안된 외경계 알고리즘을 검증하기 위하여 삼각형 메시로 구성된 두 다면체 A와 B의 Minkowski sum 계산에 적용하였다. 먼저 다음과 같이 정의되는 A와 B의 콘볼루션을 구한다:

$$A*B = \{p+q | N_p \cap N_q \neq \emptyset, p \in A, p \in B\}$$

N_p와 N_q는 각각 p와 q의 법선 벡터 집합에 해당된다. $\partial(A \oplus B)$ 는 콘볼루션 $A*B$의 외경계에 포함되므로 ([21, 22]), 제안된 알고리즘을 적용하여 콘볼루션의 외경계를 구한다. 만일 콘볼루션의 외경계가 한 개의 연결 요소(connected component)라면, 이 것이 바로 $\partial(A \oplus B)$ 이 된다. 만일 여러 개의 연결 요소로 구성되어 있다면 이 중에서 $\partial(A \oplus B)$ 을 찾아내야 한다. 먼저 간단히 연결 요소의 한 점에서 법선 방향으로 ε만큼 떨어진 점 p를 구하고, $(t-A)$ 와 B와 충돌 검사를 한다. 충돌하지 않는다면 이 연결 요소는 $\partial(A \oplus B)$ 에 들어가는 외경계가 된다.

Table 2 Performance comparison

	Conv. (tris.)	Pure CPU (s)	GPU+ CPU (s)	Speedup
dragon⊕torus	31,150	0.66	0.111	×5.9
torus⊕helix	47,294	1.17	0.107	×10.9
knot⊕dragon	56,561	2.93	0.209	×14.0
knot⊕helix	108,220	9.26	0.448	×20.7

5.2 결과

알고리즘의 GPU 부분은 Nvidia의 CUDA와 병렬 radix sort 및 prefix-sum 이 지원되는 cudpp 라이브러리를 사용하여 구현하였고, 다중정밀도 연산은 GNU MPFR 라이브러리를 사용하였다. 실험은 Intel Core i7 3.07 GHz 시스템에서 Nvidia GeForce GTX580을 사용하여 수행하였다.

실험에 사용한 다면체 모델은 knot, torus, dragon, helix이고(Table 1참조), 이들을 조합한 네 가지 Minkowski sum을 구하는데 걸린 시간을 측정하였다. 입력 다면체 모델들은 먼저 교란 범위 $\delta = 10^{-8}$ 안에서 랜덤하게 교란된 후에 컨볼루션을 구하고, 우리의 외경계 알고리즘을 사용하여 외경계를 구하였다. 그리고, 그 결과를 CPU만을 사용하여 기존 알고리즘으로 계산한 결과와 비교하였다. Table 2의 첫번째 열은 두 모델의 컨볼루션 삼각형 개수를 보여주고 있다. GPU+CPU로 표기된 열은 우리가 개발한 알고리즘으로 계산한 시간이고, 이와 비교한 Pure CPU는 [18]의 Minkowski sum 결과에서 가져온 것이다. 위 표에서 알 수 있듯이 우리가 제안한 외경계 알고리즘은 GPU의 도움으로 5.9배에서 20.7배에 이르는 성능 향상을 얻을 수 있었다. 전반적으로는 콘볼루션에서 생성된 삼각형들이 많아짐에 따라 GPU 가속으로 인한 성능향상이 커지는 경향이 나타난다. 이러한 경향은 동시에 처리할 데이터가 많아짐에 따라 GPU의 병렬 코어 점유율이 높아지기 때문이다. dragon⊕torus에서 특히 성능 향상이 적었던 이유는 다중정밀도 구간 연산을 필요로 하는 판단함수들이 상대적으로 많이 발생했기 때문이다. 다중정밀도 구간 연산은 일반 배정밀도 연산에 비해 5~10배 정도 계산량이 많고, CPU에서 순차적으로 처리되기 때문에 GPU처리되는 판단함수들에 비해 50~100배 정도의 시간이 더 걸린다. 컨볼루션에 중첩된 삼각형들이 많으면 이러한 다중정밀도 구간 연산량이 늘어나 계산시간이 급격히 증가한다.

Fig. 12는 Minkowski sum 계산 결과를 보여주고 있다. (a)-(d)까지는 각각 dragon⊕torus, torus⊕helix, knot⊕dragon, knot⊕helix의 결과이다. (e)–(g)는 외경계가 올바르게 구해졌는지를 확인하기

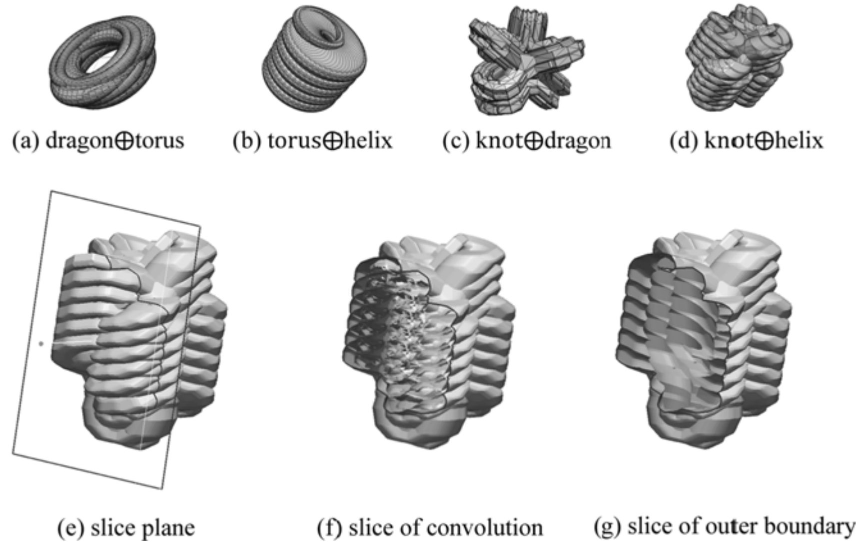


Fig. 12 Minkowski sum result. (a) dragon⊕torus, (b) torus⊕helix, (c) knot⊕dragon, (d) knot⊕helix. (e)⊕(g) slices of convolution and outer boundary of knot⊕helix

위해 knothelix의 컨볼루션 삼각형 집합(f)과 외경계 결과(g)의 단면을 비교한 것이다. 컨볼루션 삼각형 집합의 내부에서 교차하는 많은 삼각형들이 외경계 결과에서는 깨끗하게 제거되어 있는 것을 확인할 수 있다.

6. 결 론

본 논문에서는 3차원 공간에 분포하는 삼각형들에 의해 분할된 배치로부터 외경계를 구하는 알고리즘을 제안하였다. 주어진 알고리즘은 두 삼각형 교차, 세 삼각형 교차, 엣지 분할 및 연결, 삼각형 분할, 외경계 탐색의 과정을 거쳐 외경계를 찾게 된다. 이 과정들은 주로 GPU상에서 병렬적으로 처리된다. 알고리즘 과정 중에 많은 기하 판단함수가 발생하는데, 이러한 함수들의 결과를 수치적으로 강건하게 계산하기 위하여 다단계 정밀도 구간 연산을 사용하였다. 먼저 단정밀도 구간 연산으로 판단함수의 부호를 결정하고, 여기서 수치 오차로 실패할 경우 배정밀도 구간 연산을 시도하였다. 만일 여기서도 실패한다면 마지막으로 CPU에서 연산 정밀도를 증가시켜가며 구간연산을 시도하여 판단함수의 부호를 결정한다. 제안한 외경계 알고리즘은 Minkowski sum의 경계를 구하는 문제에 적용하여 기존 알고리즘과 비교하여 5~20배의 성능향상을 확인할 수 있었다.

앞으로의 연구 방향으로는 제안된 외경계 알고리즘을 스윕 볼륨 계산, 삼각형 집합의 엔벨로프 계산, 두 다면체의 불 연산 등에 적용하는 것이다. 이러한 계산들은 기하모델링에서 유용하게 사용되는 반면에 많은 계산 시간과 강건성의 문제가 여전히 실제 응용에서 장애가 되고 있다. 본 논문에서 개발한 외경계 알고리즘은 이 두 문제를 해결하는데 크게 기여할 것으로 예상된다.

다음으로 제안된 외경계 알고리즘은 현재 GPU가 가지는 온보드 메모리의 제한으로 인하여 처리할 수 있는 데이터에 한계가 있다. 3차원 배치는 최악의 경우 $O(n^3)$의 복잡도를 가질 정도로 많은 데이터가 발생할 수 있다. 예를 들어 helix⊕helix의 콘볼루션 삼각형 집합에서는 5,000,000개의 교차선이 발생하고, 21,000,000개의 세 삼각형 교차점이 나온다. 이 예제를 처리하는데는 약 6 GB 정도의 데이터가 발생하므로 현재 알고리즘으로는 이를 제대로 처리할 수가 없다. 우리는 삼각형들과 교차 데이터를 일정한 크기로 분할하여 처리하는 방법을 모색하고 있다.

감사의 글

이 논문은 2011년 정보(교육과학기술부)의 재원으로 한국연구재단의 기초연구사업 지원을 받아 수행된 것임(2011-0004030).

참고문헌

1. Sacks, E., Milenkovic, V. and Kyung, M.-H., 2011, Controlled Linear Perturbation, *Computer-Aided Design*, 43(10), pp. 1250-1257.
2. Abrams, S. and Allen, P., 2000, Computing Swept Volums, *Journal of Visualization and Computer Animation*, 11, pp. 69-82.
3. Tilove, R.B. and Requicha, A., 1980, Closure of Boolean Operations on Geometric Entities, *Computer-Aided Design*, 12(5), pp. 219-220.
4. Requicha, A., 1985, Boolean Operations in Solid Modeling: Boundary Evaluation and Merging Algorithms, *Proc. IEEE*, pp. 30-44.
5. Pach, J. and Sharir, M., 1989, The Upper Envelope of Piecewise Linear Functions and the Boundary of a Region Enclosed by Convex Plates: Combinatorial Analysis, *Discrete & Computational Geometry*, 4, pp. 291-309.
6. Arnovo, B. and Sharir, M., 1990, Triangles in Space or Building (and analyzing) Castles in the Air, *Combinatorica*, 10(2), pp. 137-173.
7. Halperin, D., 2002, Robust Geometric Computing in Motion", *The Journal of Robotics Research*, 21(3), pp. 219-232.
8. Pach, J. and Sharir, M., 2009, *Combinatorial Geometry and Its Algorithmic Applications: The Alcala Lectures*, AMS.
9. de Berg, M., Guibas, L.J. and Halperin, D., 1994, Vertical Decomposition for Triangles in 3-space, *Proc. the 10th Annu. ACM Symposium on Computational Geometry.*
10. de Berg, M., Dobrindt, K. and Schwarzkopf, O., 1994, On Lazy Randomized Incremental Construction, *Discrete & Computational Geometry*, 14(1), pp 261-286.
11. Agarwal, P. and Sharir, M., 1998, Arrangements and Their Applications, *Handbook of Computational Geometry*, Elsevier Sience Publishers, pp. 49-119.
12. Sharir, M., 2007, Arrangements in Geometry: Recent Advances and Challenges, *Proc. the 15th annual European conference on Algorithms*, Eilat, Isarael, pp. 12-16.
13. Krishnamurthy, Sara McMains, S. and Haller, K., 2009, Accelerating Geometric Queries using the GPU, *Proc. 2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*, pp. 199-210.
14. Li, W. and McMains, S., 2011, Voxelized Minkowski Sum Computation on the GPU with Robust Culling, *Computer-Aided Design*, 43(10), pp 1270-1283.
15. Halperin, D. and Shelton, C., 1998, A Perturbation Scheme for Spherical Arrangements with Application to Molecular Modeling, *Computational Geometry: Theory and Applications*, 10(4), pp. 273-288.
16. Halperin, D. and Leiserowitz, E., 2004, Controlled Perturbation for Arrangements of Circles, *International Journal of Computational Geometry and Applications*, 14(4), pp. 277-310.
17. Halperin, D. and Raab, S., 1999, Controlled Perturbation for Arrangements of Polyhedral Surfaces with Application to Swept Volumes, *Proc. 15th Annual ACM Symposium on Computational Geometry*, pp 163-172.
18. Milenkovic, V. and Sacks, E., 2012, Adaptive-Precision Controlled Perturbation, *submitted to Symposium on Computational Geometry.*
19. Kyung, M.-H., Keun, K.-J. and Choi, J.-J., 2011, Robust GPU-based Intersection Algorithm for a Large Triangle Set, *Journal of Korea Computer Graphics*, 17(3), pp. 9-20.
20. Lozano-Pérez, T., 1983, Spatial Planning: A Configuration Space Approach, *IEEE Transactions on Computers*, C-32(2), pp. 108-120.
21. Guibas, L.J., Ramshaw, L. and Stolfi, J., 1983, A Kinetic Framework for Computational Geometry, *Proc. 24th Annu. IEEE Symposium on Foundation of Computer Science*, pp. 100-111.
22. Kaul, A. and O'Connor, M.A., 1993, Computing Minkowski Sums of Regular Polyhedra, Technical Report RC 18891(82557), IBM T. J. Watson Research Center, Yorktown Heights, N. Y..
23. Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, Paul Zimmermann, 2007, MPFR: A Multiple-Precision Binary Floating-Point Library With Correct Rounding, *ACM Transactions on Mathematical Software*, 33(2).

경 민 호

1993년 포항공과대학교 전자계산학과 학사
1995년 포항공과대학교 전자계산학과 석사
2001년 퍼듀대학교 전자계산학과 박사
2002년~현재 아주대학교 미디어학과 교수
관심분야: 컴퓨터 그래픽스, CAD/CAM