GPU를 이용한 대량 삼각형 교차 알고리즘

경민호[○,a] 곽종근[b] 최정주[a]

[a]아주대학교, [b]UDMTEK

kyung@ajou.ac.kr, kwak@udmtek.com, jungju@ajou.ac.kr

Robust GPU-based intersection algorithm for a large triangle set

Min-Ho Kyung[○,a] Jong-Geun Kwak[b] Jung-Ju Choi[a]

[a]Ajou University, [b]UDMTEK

요 약

삼각형간의 교차 계산은 많은 3차원 기하 문제들을 해결하는데 있어서 기본적으로 요구되는 연산 과정이다. 본 논문에서는 대량의 삼각형 집합 안에서의 교차 계산을 효율적이며 강인하게 처리할 수 있는 GPU 알고리즘을 제안한다. 이 알고리즘은 k-d 트리의 구성, 삼각형쌍 생성, 정확한 교차 계산을 모두 GPU에서 처리한다. 여기서 사용되는 k-d 트리에서는 분할 과정 중에 삼각형들의 복사가 많이 발생한다. 이렇게 복사된 삼각형들로 인하여 중복된 삼각형쌍들이 많이 생성되는데, 이러한 중복 삼각형쌍들을 효율적으로 제거하기 위하여 분할 인덱스를 도입하였다. 분할 인덱스는 간단한 논리곱 연산만으로 중복 여부를 효과적으로 판단할 수 있다. 수치적 강인성을 높이기 위하여는 부동소숫점 필터링을 통해 불안전한 삼각형쌍들을 분리하고, CLP(controlled linear perturbation)를 이용하여 CPU쓰레드에서 처리하도록 하였다. 제안한 알고리즘은 기존의 민코스키합 알고리즘의 합삼각형 교차계산에 적용하여 효율성과 강인성을 입증하였다.

Abstract

Computing triangle-triangle intersections has been a fundamental task required for many 3D geometric problems. We propose a novel robust GPU algorithm to efficiently compute intersections in a large triangle set. The algorithm has three stages:k-d tree construction, triangle pair generation, and exact intersection computation. All three stages are executed on GPU except, for unsafe triangle pairs. Unsafe triangle pairs are robustly handled by CLP(controlled linear perturbation) on a CPU thread. They are identified by floating-point filtering while exact intersection is computed on GPU. Many triangles crossing a split plane are duplicated in k-d tree construction, which form a lot of redundant triangle pairs later. To eliminate them efficiently, we use a split index which can determine redundancy of a pair by a simple bitwise operation. We applied the proposed algorithm to computing 3D Minkowski sum boundaries to verify its efficiency and robustness.

키워드: 삼각형 교차, GPU, k-d 트리, 강인성, 부동소숫점 필터링, CLP

Keywords: triangle intersection, GPU, k-d tree, robustness, floating-point filtering, CLP

1. 서론

3차원 공간에서의 삼각형 교차계산은 기하 알고리즘들에서 자주 요구되는 기본적인 계산 과정 중 하나로, 특히 중요한 작업 중 하나인 3차원 배치 구조(3D arrangement) 계산에서 필수적으로 요구된다. 3차원 배치 구조는 연결된 다각형들에 의해 분할된 공간 정보를 표현하는 자료 구조로 3차원 민코스키합, 스윕 볼륨 경계면, 다면체의 불 연산(Boolean operation) 등을 구하는데 유용하다. 따라서, 삼각형의 고속 교차계산은 이러한 알고리즘들의 성능을 높이는데 큰 기여를 할 수 있다.

삼각형 교차 계산을 효율적으로 하기 위한 일반적인 접근 방법은 공간 분할 또는 바운딩 볼륨 계층 구조(BVH)등을 이용하여 교차 가능성이 높은 삼각형 쌍들을 먼저 추려내고, 다

투고일 : 2011.05.26 / 심사일 : 2차 2011.08.25 / 게재확정일 : 2011.08.26

음으로 각 삼각형 쌍들의 교차를 정밀하게 계산하는 것이다. 삼각형 교차 알고리즘이 활발히 연구되어 온 분야 중 하나인 충돌 검사 연구에서는 빠른 생성 시간과 탐색 시간을 이유로 주로 BVH를 선호하는 경향이 있다. 일반적으로 충돌 검사는 충돌하기 시작하는 시점을 찾는 것이 목적이므로 주로 삼각형 교차가 없거나 매우 적은 횟수로 발생하는 상황에서 수행된다. 하지만 삼각형 교차 발생율이 매우 높은 경우에는 BVH 탐색 시간이 급격하게 증가하여 전체 성능이 공간 분할 방식보다 느려지는 단점이 있다. 일반적인 기하 알고리즘에 적용하기 위해서는 삼각형 교차 발생율이 높은 상황에도 효율적으로 대처할 수 있어야 하므로, BVH 방식보다는 삼각형 수에 따라 계산시간이 일정하게 증가하는 공간 분할 방식을 사용하는 것이 더 적합하다고 할 수 있다.

공간 분할에는 regular grid, k-d 트리, octree, BSP 트리 등의 다양한 방법들이 존재한다. 본 연구에서는 삼각형의 밀집도에 따라 적응적으로 공간을 분할하고 GPU 구현이 비교적 쉬운 k-d 트리를 사용한다. k-d 트리는 주어진 삼각형 집합을 x-, y-, z-축 방향으로 차례로 공간을 분할해 나가는 계층 구조이다. k-d 트리가 생성되면 리프 노드에는 같은 공간셀에 들어가는 삼각형들이 저장된다. 따라서, 이 삼각형들을 조합하여 정밀한 교차 계산으로 처리할 삼각형 쌍들을 만들어 낼 수 있다. 그런데, k-d 트리에서는 같은 삼각형들이 여러 개의 리프 노드에 동시에 포함될 수 있는데, 이러한 삼각형들은 중복된 삼각형쌍들을 낳게 된다. 중복 삼각형 쌍들은 계산 시간과 저장공간을 상당히 낭비하므로, 효율적으로 제거하는 방법이 필요하다. 본 논문에서는 분할 인덱스라는 플래그를 사용하여 중복 삼각형 쌍들을 효율적으로 제거하는 방법을 제시한다.

삼각형의 교차 계산은 유한정밀도 연산을 사용한다. 따라서 반올림 오차를 항상 내포하고 있고, 경우에 따라(ill-conditioned case) 반올림 오차가 크게 증폭되어 결과값에 심각한 영향을 미칠 수도 있다(그림 1). 이러한 수치 오차는 기하 문제에서 알고리즘이 제대로 동작할 수 없는 논리적인 모순을 발생시키게 된다. 수치 오차를 피하기 위하여 계산기하 분야에서는 정수에 기반한 정확한 계산(exact computation) 방법들이 연구되어 왔다. 정수 기반 연산은 일반적으로 유한정밀도 연산에 비해 수십배에서 수백배 느리고, 추가적으로 요구되는 저장 공간으로 인하여 대용량의 데이터 처리에 부적절하다. 제안된 알고리즘에서는 부동소숫점 필터링(floating-point filtering)과 제어된 선형 교란법(controlled linear perturbation)을 사용하여 알고리즘의 강인성 문제를 해결한다.

제어된 선형 교란법(controlled linear perturbation, CLP)은 수치 계산값이 반올림 오차로 인해 논리적 판단에 영향을 줄 수 있는 불안전한 상황이 발생하지 않도록 입력 데이터를 최소한으로 변형시키는 방법이다. 입력 데이터를 변형하므로 결과의 정확성이 희생되지만, 알고리즘의 안정성을 보장할 수 있는 장점이 있다. 부동소숫점 필터링은 부동소숫점 연산의 결과값이 불안전한 경우에 이러한 계산을 따로 분리하여 더 높은 정밀도 연산으로 처리하도록 하는 방법이다. 본 연구에서는 GPU에서의 계산 과정에 필터링을 적용하여 불안전한 삼각형 쌍들을 분리해 내고, 이 불안전한 쌍들은 별도로 CPU에서 CLP를 이용하여 처리되도록 하였다.

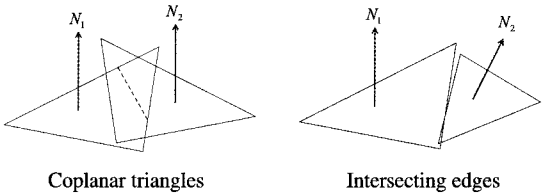


그림 1: 삼각형 교차에서 예외적인 경우들.

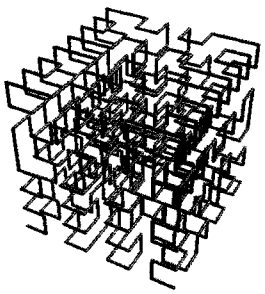


그림 2: 3차원 Hilbert 곡선. (출처:Wolfram Demonstractions Project)

GPU를 이용한 계산에서 데이터의 입출력은 전체 성능에 매우 큰 영향을 미친다. 우리는 교차 계산 전에 삼각형 데이터를 3차원 Hilbert 곡선(그림 2)의 순서로 정렬하여 공간상에서 가까운 거리에 있는 삼각형들이 연속적으로 정렬되도록 하였다. 이러한 정렬은 GPU쓰레드들이 처리하는 데이터의 참조지역성(locality of reference)를 높여 주어 GPU의 메모리 대역폭이 최대한 활용될 수 있도록 해준다.

우리는 제안된 삼각형 교차 알고리즘을 민코스키합 계산(그림 8)에 적용하여 계산의 정확성과 성능 향상을 검증하였다. 3차원 민코스키합 계산에는 n개의 삼각형으로 구성된 두 개의 다면체에 대하여 $O(n^2)$개의 합 삼각형(sum triangle)이 생성되고, 최대 $O(n^4)$의 합 삼각형 교차가 발생한다. 또한 수치 오차에 매우 민감하기 때문에 강인한 알고리즘을 개발하기가 어려운 문제이다. 따라서, 본 논문에서 제안하는 GPU 알고리즘의 성능과 강인성을 확인하는데 적합한 검증 예제라고 할 수 있다.

2. 관련 연구

대량의 삼각형 교차 검사를 고속으로 처리하는 알고리즘은 충돌검사 분야에서 활발히 연구되어 왔고, 여기서는 주로 BVH(bounding volume hierarchy)를 이용한 탐색 방법을 사용하여 왔다. Gottschalk등[1]은 삼각형 집합을 OBB 트리로 구성하여 계층적 충돌 검사를 보다 효율적으로 수행하는 알고리즘을 제안하였다. 최근에는 동적으로 변형되는 모델의 충돌 검사를 위하여 생성 시간이 짧은 AABB 트리를 사용하는 추세이고, GPU 상에서 실시간으로 생성 및 탐색을 동시에 처리하는 알고리즘도 개발되고 있다 ([2, 3]). 위와 같이 BVH를 사용한 충돌 검사는 삼각형 교차율이 낮은 경우에는 $O(\log n)$에 근접한 탐색 성능을 보여 주지만, 삼각형들이 밀집되어 있고 교차율이 높은 경우 탐색 시간이 급격히 증가하는 단점이 있다.

k-d 트리는 원래 다차원 공간의 점 데이터를 저장하기 위한 이진트리로 공간을 계층적으로 분할하여 효율적인 데이터 검색을 가능하게 해준다[4]. 컴퓨터 그래픽스 분야에서는 광선 추적법의 광선/삼각형 교차 탐색을 가속하기 위하여 k-d 트리를 활발히 사용하고 있다[5]. Zhou등[6]은 k-d 트리의 구성부터 탐색까지를 모두 GPU에서 처리할 수 있는 알고리즘을 개발하였다. 우리는 Zhou등의 GPU k-d 트리 알고리즘에서 광선교차와 관련된 요소등을 제거하여 삼각형 교차 계산에 적합한 단순화된 알고리즘을 개발하였다.

모델링과 애니메이션 분야에서 다각형 메시의 자가교차(self-intersection)는 오랫동안 까다로운 문제로 여겨져 왔다. 이를 효율적으로 처리하기 위하여 Jung등[7]은 삼각형 메시의 오프셋 결과에서 발생하는 자가교차를 적응적 버킷을 이용하여 찾는 알고리즘을 제안하였다. Volino와 Thalmann[8]은 메시 표면의 법선 방향 분포를 이용하여 메시를 자가교차가 없는 패치로 분할하여 불필요한 교차 검사를 최소화하는 방법을 제안하였다. 이러한 알고리즘들은 효율적으로 메시의 자가교차를 찾을 수 있지만 단일 메시가 아닌 일반 삼각형 집합으로 확장하기는 어려운 한계가 있다. Campen과 Kobbelt[9, 10]는 BSP 트리를 이용하여 다각형 집합에서의 교차를 정확하고 강인하게 계산할 수 있는 알고리즘을 제안하였는데, 다각형 집합의 외곽경계면만을 구할 수 있고 계산시간이 많이 걸리는 한계가 있다.

3. 강인성 전략

기하 알고리즘의 강인성 문제는 유한 정밀도 연산에서 발생하는 반올림 오차로부터 기인한다. 반올림 오차는 기하 요소들의 상대적 위치 판단을 부정확하게 하여 알고리즘의 기하학적 논리가 모순되는 상황을 유발할 수 있다. 우리는 알고리즘의 강인성을 높이기 위하여 입력 데이터를 변형시켜 부정확한 판단 상황이 발생하지 않도록 하는 CLP(controlled linear perturbation)를 사용하였다. CLP는 Milenkovic등[11]에 의해 처음 제안되어 민코스키합 계산에 적용되었는데, 아직 GPU 상에서 동작하도록 구현되지 않았다. 따라서 우리는 먼저 GPU에서 부동소숫점 연산만을 사용하여 삼각형 교차계산을 시도하고, 그 중에 반올림 오차에 민감한 불안전한 삼각형 쌍들이 발견되면 이들을 별도로 CPU쓰레드로 가져와 CLP에 의해 처리하도록 하였다.

불안전한 삼각형쌍들은 부동소숫점 필터링에 의해 걸러낸다. 부동소숫점 필터링은 부동소숫점 연산으로 먼저 계산을 하고, 계산 결과가 반올림 오차 등의 요인으로 불안전한 것으로 판단되면 더 높은 정밀도로 다시 계산하는 방식을 말한다. 우리는 불안전한 삼각형쌍들을 CLP에 의해 처리하기 때문에 삼각형들의 변형까지 고려하여 불안전함을 결정해야 한다. 이를 위해 CLP에 의한 변형 한계를 설정하고 이 한계내에서 상계(upper bound) 분석을 하여 교차 결과가 영향을 받을 수 있는지를 판단하였다.

3.1 CLP

많은 기하 알고리즘들은 문제의 조합 성질을 결정하는 유한 개의 판단 함수 (predicates f_i)들로 구성되고 그 함수들의 부호에 따라 결과를 구하는 조합 문제들이다. 알고리즘의 입력을 벡터 $\mathbf{x}$로 나타낼 때, 조합 성질은 판단함수 $f_i(\mathbf{x})$의 값이 음수인 경우와 양수인 경우로 나누어 결정되고, $f_i(\mathbf{x}) = 0$인 경우는 예외적인 경우(degenerate cases)로 처리한다.

판단 함수 f_i의 계산은 유한정밀도를 가지는 부동소숫점 연산으로 이루어지기 때문에 필연적으로 반올림 오차가 발생한다. 만일 함수의 절대값이 반올림 오차에 비해 충분히 크다면 반올림 오차는 함수값의 부호에 영향을 주지 않는다. 반대로 절대값이 반올림 오차에 비해 작다면 반올림 오차에 의해 함수값의 부호가 바뀔 수 있으므로 그 결과는 신뢰할 수 없게 된다. CLP는 이러한 불안전한 함수값이 발생하는 경우 입력 벡터 $\mathbf{x}$를 인위적으로 변형하여 함수값을 반올림 오차 범위 밖으로 이동시키는 역할을 한다.

$\mathbf{x}$의 변형은 다음과 같이 결정한다. 먼저 입력 벡터의 원래 값을 $\mathbf{x}_0$라고 하고, 변형시킬 방향을 $\mathbf{v}$라고 하자. $\mathbf{v}$는 알고리즘 시작 전에 무작위로 선택된 단위 벡터이다. 벡터 $\mathbf{x}$가 $\mathbf{v}$방향으로 길이 d만큼 변형된다면,

$$\mathbf{x} = \mathbf{x}_0 + d \cdot \mathbf{v} \tag{1}$$

와 같이 정의된다. 여기서 d는 초기에 0으로 설정된다. ϵ을 반올림 오차의 예상최대치라고 한다면, $|f_i(\mathbf{x})| > \epsilon$인 경우 $f_i(\mathbf{x})$의 부호는 반올림 오차에 의해 영향을 받지 않으므로 안전하다고 할 수 있다. 만일 $|f_i(\mathbf{x})| \leq \epsilon$라면 다음 조건을 만족하도록

d를 증가시켜 $|f_i(\mathbf{x})|$가 ϵ보다 커지도록 한다:

$$|f_i(\mathbf{x})| \simeq |f_i(\mathbf{x}_0) + \nabla f_i(\mathbf{x}_0) \cdot (d \cdot \mathbf{v})| \geq \epsilon$$
$$d \geq (\epsilon - s \cdot f_i(\mathbf{x}_0))/(s \cdot \nabla f_i(\mathbf{x}_0) \cdot \mathbf{v}).$$

여기서 s는 $f_i(\mathbf{x}) < 0$이면 -1, 아니면 1이다. 변형정도는 작을수록 좋기 때문에 d는 $(\epsilon - s \cdot f_i(\mathbf{x}_0))/(s \cdot \nabla f_i(\mathbf{x}_0) \cdot \mathbf{v})$ 로 설정하는 것이 좋지만 선형 근사에 의한 오차를 고려하여 이 값에 2를 곱하여 d값으로 설정한다.

3.2 부동소숫점 필터링

불안전한 판단함수는 교차결과를 뒤집을 수 있기 때문에 이러한 판단함수가 한번이라도 발생하면 삼각형쌍의 교차는 신뢰할 수 없다. 따라서, 부동소숫점 필터링은 교차 계산의 매 단계마다 판단함수에 적용되어 불안전성를 결정하게 된다.

판단 함수를 불안전하게 만드는 요소는 두 가지가 있다. 반올림 오차와 CLP에 의한 삼각형의 변형이다. 반올림 오차는 일반적인 유한정밀도 연산에서 발생하는 것으로 GPU에서 보장하는 연산 정밀도에 따라 결정된다. CLP에 의한 삼각형 변형이 미치는 영향은 변형 길이 d와 변형 방향 $\mathbf{v}$에 따라 결정된다. 여기서 문제는 먼저 실행되는 부동소숫점 필터링에서 가변적으로 변하는 d를 미리 알 수 없기 때문에 변형에 의한 불안전성을 정확히 예측할 수 없다는 점이다.

이 문제를 해결하기 위하여 우리는 d의 범위를 최대값 d_{max} 이내로 제한하는 방법을 사용한다. 이 변형 범위 안에서 판단함수값의 범위를 구할 수 있기 때문에 안전함을 보장할 수 있는 조건을 근사적으로 유도할 수 있다. 적절한 d_{max}는 입력 데이터에 따라 달라질 수 있는데, 실험 예제들에서는 10^{-8}을 사용하였다. 만일 CLP에서 d가 d_{max}보다 커진다면, d_{max}를 2배로 증가시키고 전체 교차 계산을 재시작한다.

판단함수 f의 안전함을 결정하는 조건은 다음과 같이 유도된다. 먼저 식 1의 $\mathbf{x}$에 대한 $|f|$의 최소값을 다음과 같이 근사한다:

$$|f(\mathbf{x})| = |f(\mathbf{x}_0) + \delta f(\mathbf{x})| \geq |f(\mathbf{x}_0)| - |\delta f(\mathbf{x})| \geq \epsilon$$
$$|f(\mathbf{x}_0)| - \epsilon \geq |\delta f(\mathbf{x})| \quad (2)$$

따라서 $|f(\mathbf{x}_0)| - \epsilon$이 $|\delta f(\mathbf{x})|$의 최대값보다 크면 $f(\mathbf{x})$는 안전하다고 할 수 있다. GPU에서 $|\delta f(\mathbf{x})|$의 최대값을 정확히 구하는 것은 어려우므로 근사적으로 $|\delta f(\mathbf{x})|$의 상계를 유도하여 사용하였다. 교차계산에 사용되는 판단함수의 상계는 4.3에 정리되어 있다.

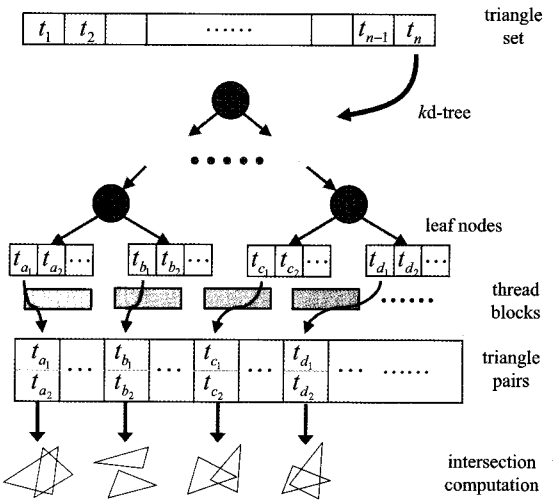


그림 3: 교차 알고리즘의 전체적인 구조. 주어진 삼각형 집합으로부터 k-d 트리를 만들고, k-d 트리의 리프 노드에 저장된 삼각형들을 조합하여 삼각형쌍들을 만든다. 마지막으로 삼각형쌍들의 교차를 계산한다.

4. GPU 알고리즘

삼각형 교차 계산을 위한 GPU 알고리즘은 세 단계로 이루어져 있다(그림 3 참조). 먼저 주어진 삼각형 집합에 대한 k-d 트리를 구성한다. 다음 k-d 트리의 리프 노드로부터 정확한 교차계산을 위한 삼각형 쌍들을 생성한다. 다음 각각의 삼각형 쌍들에 대한 정확한 교차 계산을 수행한다. GPU 교차 계산에서 처리되지 못한 불안전한 삼각형 쌍들은 CPU쓰레드로 보내져 CLP에 의해 다시 처리된다. 이 과정에서 변형 길이 d가 갱신되므로 입력 삼각형들을 $d \cdot \mathbf{v}$만큼 변형시킨 후 전체 교차점 좌표들을 다시 계산한다.

k-d 트리와 삼각형쌍 생성에는 삼각형의 AABB(axis-aligned bounding box)와 OBB(oriented-bounding box)가 사용된다. 이후의 CLP 변형에 의해 바운딩 박스 교차 결과가 영향을 받는 것을 피하기 위해 바운딩 박스들의 크기는 실제 크기보다 약 1% 정도 여유를 갖도록 설정하였다.

입력 삼각형 데이터가 불규칙하게 배열되어 있을 경우 GPU에서의 데이터 I/O시간이 매우 길어질 수 있다. 데이터 I/O의 효율성을 높여 주기 위하여 가까운 삼각형들이 저장공간에서 인접하게 배열되는 Hilbert 정렬을 전처리 과정으로 수행한다. Hilbert 정렬은 3차원 점들을 Hilbert 곡선 위로 대응시키고, Hilbert 곡선을 따라 순서대로 정렬하는 방법이다. 삼각형들을 정렬하기 위해서는 삼각형의 중심점을 구하고, 이 점들을 Hilber 곡선 위로 대응시키면 된다. 정렬 후에 각 삼각형들의 정렬 순서를 자신의 ID로 부여한다. 불규칙하게 배열된 삼각

형의 정점들도 마찬가지로 정렬한다. 전처리에서의 Hilbert 정렬은 4.2절에서의 삼각형쌍 정렬과 연결되어 연속된 쓰레드에서 처리되는 삼각형 데이터의 응집성을 높여 준다.

4.1 k-d 트리 구성

본 논문의 k-d 트리 구성은 공간 탐색 구조를 만들기 위한 것이 아니므로 매우 단순하게 구현되게 된다. 최종적으로 필요한 것은 리프 노드에 저장된 삼각형들의 집합들이므로 트리의 내부 노드들은 따로 저장할 필요가 없다. 또한 탐색 효율이 높은 트리를 만들기 위해 사용하는 SAH(surface area heuristic)도 적용하지 않는다. 물론 SAH가 좀 더 균형있고 고르게 나누어진 삼각형 집합들을 만들어 낼 가능성도 있지만, 이를 위해 추가로 들어가는 시간에 비해 잇점이 크지 않을 것으로 예상된다. 이와 같이 일반적인 k-d 트리 구성에 비해 단순화함으로서 기존 Zhou등[6]의 알고리즘에 비해 쉽게 구현할 수 있다.

먼저 N_j^i는 k-d 트리에서 레벨 i의 j번째 노드로 정의한다. 노드집합 $I^i = \{N_j^i | 1 \leq j \leq n\}$는 레벨 i의 노드를 모아 놓은 집합이다. 먼저 모든 삼각형들을 포함하는 루트 노드 N_1^1을 만들어 노드집합 I^1에 넣고, k-d 트리 구성을 시작한다. 레벨i에서 노드집합 I^i의 각 노드들을 분할하기 위한 분할면을 결정한다. 분할면의 방향은 $(i\%3)$에 따라 x, y, 또는 z 방향으로 결정되고, 분할면의 위치는 분할면의 방향을 따라 각 노드에 포함되는 삼각형들의 최소값과 최대값의 중간으로 결정한다. 분할면의 위치가 결정되면 이를 기준으로 각 노드 N_j^i를 왼쪽과 오른쪽 자식노드로 분할하여 I^{i+1}에 추가한다. 만일 삼각형들이 한쪽 노드로 몰리는 경우에는 현재 분할면이 적절한 분할면이 아니므로 N_j^i를 그대로 I^{i+1}에 추가하고 다음 레벨의 분할 방향으로 분할되도록 한다. 만일 연속적으로 다음 두 레벨에서도 분할에 실패하면 이 노드는 노드집합에서 제거하여 리프 노드 집합 L에 추가한다. 분할된 노드에 포함된 삼각형들의 수가 10개 이하면 역시 노드집합에서 제거하고 L에 추가한다. 현재 레벨의 노드 집합이 공집합이 되어 더 이상 분할할 노드가 없다면 k-d 트리 구성을 끝내고 다음 삼각형쌍 생성 단계로 넘어 간다. 노드의 최대, 최소값을 찾는 과정, 분할면에 의한 노드 분할은 모두 병렬 스캔 알고리즘을 사용하였다. 이러한 과정들은 I^i의 모든 삼각형들을 차례로 스캐닝해서 처리해야 하는데, 병렬 스캔 알고리즘은 이러한 데이터 스캐닝을 m개의 코어를 가진 GPU에서 $O(n/m + \log n)$시간 에 처리할 수 있다[12].

노드가 분할될 때 분할면에 걸쳐 있는 삼각형은 두 부분으로 나누어 지는 대신 양쪽 자식노드에 모두 포함된다. 즉, 레벨 i에서 삼각형 $t_a \in N_j^i$가 분할면에 걸쳐 있다면 t_a를 복사하여 t'_a을 만들고, t_a는 왼쪽 자식노드에 복사본 t'_a은 오른쪽 자식노드에 삽입한다. 이러한 복사로 인하여 여러 리프 노드에 동시에 포함되는 삼각형들이 발생하게 된다. 각각의 리프 노드들은 독립적으로 삼각형쌍들을 만들기 때문에, 공통된 삼각형들을 가지고 있는 리프 노드들은 중복된 삼각형쌍들을 낳게 된다. 삼각형들이 밀집되어 있을 수록 많은 삼각형 복사가 일어나고, 따라서 중복 삼각형쌍들의 수도 증가한다. 예를 들어 knot⊕helix 민코스키합 계산에서는 47,297,091개의 삼각형쌍들이 생성되는데, 이 중에 약 80%에 해당하는 37,997,734가 중복된 쌍들이었다. 이러한 중복 삼각형쌍들은 저장공간과 처리시간을 심각하게 낭비하므로 효율적으로 제거해야 한다. 이를 위해 우리는 분할 인덱스라는 플래그를 도입하였다. 분할 인덱스는 삼각형이 리프 노드까지 오는 경로 상에서 복사본으로 만들어진 레벨을 인코딩하는 정수로 분할로 인한 중복 삼각형쌍들을 완벽하게 구분할 수 있다. 분할 인덱스는 초기에 모든 삼각형들에 대하여 0으로 설정하고, 노드 분할에서 삼각형 t_a가 t'_a으로 복사될 때마다 t'_a에 대하여 다음과 같이 설정한다:

$$s(t'_a) \leftarrow s(t_a) \vee (1 << (i-1)).$$

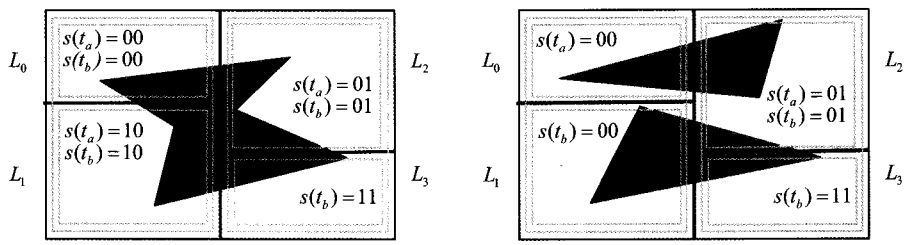
여기서 $s(t'_a)$는 t'_a의 분할 인덱스이다. 분할 인덱스를 이용한 중복 삼각형쌍 판단은 다음 절에서 자세히 설명하겠다.

4.2 삼각형 쌍 생성

이 단계에서는 L에 있는 각각의 리프 노드에 저장된 삼각형들을 조합하여 삼각형 쌍들을 생성한다. 리프 노드 L_i가 k개의 삼각형들을 가지고 있다면 ${}_kC_2$개의 쌍들이 만들어지고, 이 중에서 중복 쌍이 아니며 바운딩 박스가 교차하는 삼각형 쌍들만을 선택하여 출력한다.

삼각형 쌍을 생성하는 것은 매우 단순한 알고리즘이지만, GPU에서 효율적으로 동작하도록 구현하기 위해서는 최적의 쓰레드 작업 할당과 결과 저장 방식을 찾아야 한다. 단순히 각 리프 노드를 개별 쓰레드에 할당한다면, 리프 노드마다 가지고 있는 삼각형들의 갯수가 다르므로 일부 쓰레드에만 계산량이 집중될 수 있다. 따라서 전체 GPU 코어들 중의 일부만 계산에 활용되게 된다. 우리는 작업량이 균형있게 배분되도록 먼저 리프 노드들을 삼각형 갯수 k에 따라 정렬시키고, 각 리프 노드를 $n_{threads}$개의 쓰레드들로 구성된 블럭에 할당시킨다. $n_{threads}$는 리프노드의 삼각형수가 $2^{b-1} < k \leq 2^b$라면 2^b로 결정된다. 이와 같이 리프 노드의 삼각형수에 따라 적절한 크기의 쓰레드 블럭을 할당함으로서 작업량이 GPU코어에 균형있게 배분되게 된다.

쓰레드 블럭에서 생성된 삼각형쌍들은 전역 버퍼에 저장된다. 문제는 저장되는 삼각형 쌍들의 갯수가 리프 노드마다 다르기 때문에 저장할 전역 버퍼의 저장 위치를 미리 정해 주기가 어렵다는 것이다. 물론 각 리프노드에 최대 삼각형 쌍의 갯수 ${}_kC_2$만큼 미리 저장 공간을 할당해 줄 수도 있지만, 이에 비해 실제 출력되는 삼각형쌍들이 매우 적기 때문에 저장 공간의 낭비가 심해진다. 이를 해결하기 위해 우리는 GPU의 최소 단위 연산을 사용하였다. GPU 최소 단위 연산은 다른 쓰레



(a) triangle pair (t_a, t_b) from N_0 (b) no triangle pair

그림 4: Split index에 의한 삼각형쌍 생성. (a) 리프 노드 N_0에서 $s(t_a) \wedge s(t_b) = 0$이므로 삼각형쌍(t_a, t_b) 생성. N_1과 N_2에서는 $s(t_a) \wedge s(t_b) \neq 0$이므로 만들지 않음. (b) 리프 노드 N_2에서 $s(t_a) \wedge s(t_b) \neq 0$ 이므로 삼각형쌍을 만들지 않음.

드의 간섭없이 한 번에 한 쓰레드만 실행하는 단위 연산으로, 그 중에서 atomicADD를 사용하면 전역 버퍼 저장가능 위치의 현재값을 읽어 오고 동시에 m만큼 증가시켜 삼각형쌍 m개를 저장할 공간을 선점할 수 있다.

삼각형 쌍을 생성하는 GPU 쓰레드 코드는 그림 5와 같다. 여기서 ID_{thread}는 현재 쓰레드의 ID, AABB(t)는 삼각형 t의 AABB, T_{sh}는 삼각형 저장을 위한 공유버퍼, P_{sh}는 삼각형쌍을 임시로 저장하는 공유버퍼, m은 P_{sh}의 최대버퍼크기를 나타낸다. 라인2-8은 쓰레드별로 삼각형쌍을 만들어 공유버퍼 P_{sh}에 임시로 저장하는 부분이다. 라인6은 삼각형 t_a와 t_b의 분할 인덱스를 논리곱 연산하여 0인지를 검사한다. 논리곱 연산 결과가 0이 아니라면 같은 분할면 p에 의해 복사된 복사본이므로 이 두 삼각형들의 쌍을 생성할 필요가 없다. 그 이유는 두 가지 경우로 나누어 생각해 볼 수 있다. 두 삼각형을 포함하고 $s(t_a) \wedge s(t_b) = 0$인 다른 리프 노드 L_i가 존재하는 경우(그림4a)와 그렇지 않는 경우(그림4b)이다. 전자라면 L_i에서 (t_a, t_b)를 생성하므로 현재 리프 노드에서는 이 삼각형쌍을 만들 필요가 없어진다. 후자라면 원본 삼각형들이 분할면 p에 의해 왼쪽 자식노드로 들어간 이후 다음 분할면에 의해 완전히 분리된 것을 의미한다. 따라서 두 삼각형이 교차하지 않는 것이 확실하므로 역시 삼각형쌍을 만들 필요가 없어진다. 이와 같이 분할 인덱스는 분할로 인한 중복 삼각형쌍들을 완벽하게 구분해 낼 수 있다. 분할 인덱스 검사를 통과하면 라인7에서 바운딩박스 교차검사를 하고, 교차하는 경우 삼각형쌍을 P_{sh}에 추가한다 (라인8).

라인9-15는 P_{sh}가 충분히 채워졌을 때 저장된 삼각형쌍들을 전역 버퍼 P로 출력하는 부분이다. 라인11에서 atomicAdd($|P|$, $|P_{sh}|$)는 P_{sh}를 복사할 P의 위치를 얻기 위한 것으로, P의 현재 크기를 r에 쓰고 동시에 P의 크기를 $|P| + |P_{sh}|$로 갱신해 준다. 따라서 현재 쓰레드 블럭은 $P[r]$에서 $P[r + |P_{sh}| - 1]$ 사이의 전역 버퍼 공간을 선점하게 된다. 라인13에서는 P_{sh}의 삼각형쌍들을 확보된 전역 버퍼 공간에 차례로 복사하고, 라인14에서는 P_{sh}를 비워주어 다음 루프에서 다시 새롭게 채워지도록 한다.

```
1.  for i = 0 to ⌈k²/n_threads⌉ do
2.      p = i · n_th + ID_thread;
3.      (a, b) = (p/k, p mod k);
4.      (t_a, t_b) = (T_sh[a], T_sh[b]);
5.      if (a < k and a < b) and
6.        (s(t_a) ∧ s(t_b) = 0) and
7.        (AABB(t_a) ∩ AABB(t_b) ≠ ∅)), then
8.            insert (t_a, t_b) into P_sh;
9.      synchronizeThread();
10.     if |P_sh| + n_threads > m, then
11.       if ID_thread = 0, then
12.           r = atomicAdd(|P|, |P_sh|);
13.       output P_sh from P[r] to P[r + |P_sh| − 1];
14.       P_sh = ∅;
15.     endif
16. endfor
```

그림 5: 삼각형쌍을 생성하는 GPU 쓰레드 코드.

전역 버퍼 P의 삼각형쌍들에 대하여 다음 단계로 넘어가기 전에 한번 더 OBB 교차 검사를 하여 교차하지 않는 삼각형쌍들을 추가로 제거한다. OBB 검사는 삼각형쌍 생성 코드에서 분리하였는데, 그 이유는 OBB검사를 받을 삼각형쌍들이 쓰레드순으로 응집되어 있지 않고 또 교차 검사의 계산량이 상대적으로 크기 때문이다. 응집되지 않은 삼각형쌍들은 GPU코어들의 일부만을 활성화시키므로, 계산량이 많은 OBB검사를 함께 처리할 때 성능 저하가 크게 나타날 수 있다. 따라서 AABB검사를 통과한 삼각형쌍들이 모두 P에 저장된 후에 쓰레드들을 새로 생성하여 P에 저장된 삼각형쌍들의 OBB검사를 처리한다. OBB검사를 통과한 삼각형쌍들은 병렬 스캔 알고리즘에 의해 압축되어 다시 P에 저장된다.

4.3 삼각형/삼각형 교차 계산

삼각형 쌍들의 정확한 교차 계산을 GPU 상에서 효율적로 수행하기 위해서는 데이터의 응집성과 인접한 쓰레드간의 실행 유사성을 높이는 것이 중요하다. 데이터의 응집성은 교차 계산에 필요한 삼각형 데이터를 전역 버퍼에서 읽어오는데 걸

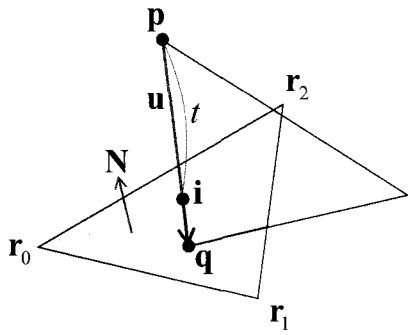


그림 6: 엣지 $\mathbf{pq}$와 삼각형 $\mathbf{r}_0\mathbf{r}_1\mathbf{r}_2$ 간의 교차. 교차점 $\mathbf{i} = \mathbf{p} + t\mathbf{u}$의 $t = (\mathbf{r} - \mathbf{p}) \cdot \mathbf{N} / \mathbf{u} \cdot \mathbf{N}$이다.

리는 시간을 단축시켜 준다. GPU는 기본적으로 다수의 GPU 코어가 동시에 한 개의 명령어를 처리하는 SIMD 구조로 되어 있다. 따라서 쓰레드들의 실행 과정이 유사할 수록 GPU코어의 사용 효율이 높아져 전체적인 성능이 향상된다. 이러한 점들을 고려하여 전역 버퍼 P의 삼각형쌍들을 다음 네 단계로 나누어 처리한다:

1. P의 삼각형쌍들을 삼각형ID에 따라 정렬한다.
2. 두 삼각형이 한 엣지를 공유하며 인접해 있는 경우 이 삼각형쌍을 제거한다.
3. 꼭지점을 공유하는 삼각형쌍들과 그렇지 않은 삼각형쌍들을 나누고, 꼭지점을 공유하는 삼각형쌍들에 대한 정확한 교차 계산을 수행한다.
4. 나머지 삼각형쌍들에 대한 정확한 교차 계산을 수행한다.

단계1의 삼각형ID에 의한 삼각형쌍의 정렬은 인접한 GPU 쓰레드들이 처리하는 삼각형 데이터가 최대한 응집되도록 만들어 준다. 이러한 응집성은 실험에 의하면 정렬을 하지 않은 경우에 비해 이후 단계의 성능을 최대 30%정도까지 향상시켜 준다. 그리고, 단계2-4의 인접 삼각형쌍 제거, 꼭지점 공유 삼각형쌍 교차 계산, 일반 삼각형쌍 교차 계산은 완전히 다른 실행 과정(execution flow)을 가지고 있다. 따라서 세 가지 경우를 하나의 쓰레드 코드로 묶어서 처리하기보다는 위와 같이 세 단계로 분리하여 처리한다.

단계3과 4에서 기본적인 계산 단위는 엣지와 삼각형간의 교차점 계산이다. 단계3에서는 공유 꼭지점이 있으므로 2번의 엣지/삼각형 교차가 필요하고 그 중에 하나의 엣지/삼각형 교차점을 찾으면 된다. 단계4에서는 각 삼각형쌍에서 6번의 엣지/삼각형 교차가 필요하고 그 중에 두 개의 엣지/삼각형 교차점을 찾아야 한다. 엣지/삼각형 교차 계산은 다시 엣지/평면 교차점 계산과 이 교차점의 삼각형 내부 검사로 나누어 진다.

이 두 계산에는 두 종류의 판단함수가 공통적으로 사용된다. 먼저 엣지의 한 점 $\mathbf{p}$가 평면을 중심으로 놓여 있는 방향을 결정하는 판단함수

$$f(\mathbf{p}) = (\mathbf{p} - \mathbf{r}) \cdot \mathbf{N}$$

을 사용한다. 여기서 $\mathbf{r}$은 평면 위의 한 점, $\mathbf{N}$은 평면의 법선벡터이다(그림 6 참조). 엣지의 두 점 $\mathbf{p}$와 $\mathbf{q}$에 대하여 $f(\mathbf{p})$와 $f(\mathbf{q})$의 부호가 반대이면 이 엣지는 평면과 교차하므로 교차점 $\mathbf{i}$를 구할 수 있다. 두 번째 판단함수는 평면과의 교차점 $\mathbf{i}$가 삼각형 내부에 들어가는지를 결정하기 위한 함수이다. 삼각형의 j번째 점이 $\mathbf{r}_j$이고, j번째 변의 방향이 $\mathbf{e}_j$일 때, 판단함수는

$$h_j(\mathbf{i}) = (\mathbf{e}_j \times (\mathbf{i} - \mathbf{r}_j)) \cdot \mathbf{N}$$

이다. $j = 0, 1, 2$에 대하여 $h_j(\mathbf{i})$가 모두 양수이면 $\mathbf{i}$는 삼각형의 내부에 있는 점이 된다.

3.2절에서 기술한 부동소숫점 필터링을 적용하기 위하여 판단함수 $f(\mathbf{p})$와 $h_j(\mathbf{i})$에 대한 $|\delta f(\mathbf{p})|$와 $|\delta h_j(\mathbf{i})|$의 상계가 필요하다. 여기서는 입력 삼각형들이 두 다면체의 민코스키합에서 발생하는 합삼각형(sum triangle)들이라고 가정하고 상계를 유도하였다. 다른 종류의 삼각형들에 대해서는 약간의 수정으로 쉽게 유도할 수 있다. 합삼각형의 한 정점 $\mathbf{p}$는 두 다면체에서 나온 정점들을 더한 점이 된다. CLP는 이 정점들을 교란하여 불안전 판단함수를 제거하고, 교란되는 정도는 교란 방향을 고려하지 않을 때 좌표별로 최대 $\pm d_{max}$가 된다고 할 수 있다. 따라서 $|\delta\mathbf{p}|$의 상계는 $2\sqrt{3}d_{max}$가 된다. 이를 이용하여 $|\delta f(\mathbf{p})|$의 상계를 구하면 다음과 같이 유도된다:

$$|\delta f(\mathbf{p})| \leq 2\sqrt{3}d_{max} + ||\mathbf{p} - \mathbf{r}|| \cdot ||\delta N||.$$

마찬가지로 $|\delta h_j(\mathbf{i})|$의 상계도 구할 수 있다:

$$\begin{aligned} |\delta h_j(\mathbf{i})| \quad \leq \quad & \|\mathbf{N} \times \mathbf{e}_j\| \cdot (\|\delta\mathbf{i}\| + \sqrt{3}d_{max}) + \\ & \|(\mathbf{i} - \mathbf{r}_j) \times \mathbf{N}\| \cdot (2\sqrt{3}d_{max}) + \\ & \|\mathbf{e}_j \times (\mathbf{i} - \mathbf{r}_j)\| \cdot \|\delta\mathbf{N}\|. \end{aligned}$$

여기서 삼각형의 법선벡터 $\mathbf{N} = \mathbf{v} \times \mathbf{w} / \|\mathbf{v} \times \mathbf{w}\|$이라면, $\|\delta\mathbf{N}\|$의 상계는 다음과 같이 유도된다:

$$\|\delta\mathbf{N}\| \leq \frac{4\sqrt{3}d_{max}(\|\mathbf{w}\| + \|\mathbf{v}\|)}{\|\mathbf{v} \times \mathbf{w}\|}. \tag{3}$$

교차점 $\mathbf{i} = \mathbf{p} + t \cdot \mathbf{u}$, $\mathbf{u} = \mathbf{q} - \mathbf{p}$이므로, $\|\delta\mathbf{i}\|$의 상계는 다음과 같이 구해진다:

$$\begin{aligned} \|\delta\mathbf{i}\| \quad \leq \quad & \sqrt{3}d_{max}(1 + 2t) + \|\mathbf{u}\| \cdot \\ & \frac{2\sqrt{3}(1 + t)d_{max} + \|\mathbf{r}_0 - \mathbf{p}\| + t\|\mathbf{u}\|\|\delta\mathbf{N}\|}{|\mathbf{u} \cdot \mathbf{N}|}. \end{aligned}$$

$\|\delta \mathbf{N}\|$의 상계는 계산 시간을 단축하기 위해 각 삼각형의 법선 벡터 N과 함께 미리 계산해 두고 사용한다.

단계3과 4에서 부동소숫점 필터링을 통과하지 못한 판단 함수가 발생하면 그 삼각형쌍은 불안전한 쌍으로 분류된다. GPU 교차 계산이 모두 끝나면 불안전한 쌍들은 CPU 쓰레드로 보내져 CLP에 의해 처리된다.

5. 결과

제안된 삼각형 교차 알고리즘은 기존에 개발된 CLP를 이용한 민코스키합 알고리즘에 적용하였다. 민코스키합 알고리즘은 두 다면체 간의 콘볼루션을 계산하여 합삼각형(sum triangle)들을 구하고, 이 합삼각형들간의 교차를 계산한다. 다음 교차선에 의하여 합삼각형을 분할하고, 최외곽 삼각형부터 분할된 삼각형들의 연결관계를 따라 너비우선탐색(breadth-first search)을 수행하여 민코스키합의 최외곽 경계면을 찾아낸다[11]. 여기서 합삼각형의 교차 계산에 새로 제안된 GPU 교차 알고리즘을 적용하였다. GPU 알고리즘에서 얻어진 교차 계산 결과는 기존 알고리즘의 교차 계산 결과와 비교하여 정확성을 검증하였다. 실험에 사용된 환경은 인텔 i7 3.07GHz CPU와 Nvidia GTX580을 장착한 하드웨어에서 수행되었고, GPU 프로그램은 Nvidia CUDA를 사용하여 구현하였다. GPU에서의 모든 계산은 단밀도 부동소숫점을 사용하였고, CPU쓰레드에서 불안전 삼각형쌍들의 교차계산에서만 배밀도 부동소숫점을 사용하였다.

실험은 5개의 다면체쌍에 대한 민코스크합 계산(그림7, 8)을 가지고 하였고, 제안된 GPU 알고리즘은 콘볼루션에서 구해진 합삼각형들의 교차를 계산에 적용하였다. 그 결과는 표1에 정리되어 있다. 결과를 보면 평균적으로 하나의 합삼각형에 대하여 2.5번에서 14.9번의 교차가 발생하므로 실험 사례들의 교차율이 매우 높은 것을 알 수 있다. 각 단계별 계산시간은 표2에 정리되어 있다. k-d 트리 구성시간을 보면 knot⊕helix가 dragon⊕helix에 비해 삼각형수가 적은데도 불구하고 더 많은 시간이 걸리는 것을 볼 수 있는데, 이 것은 삼각형들의 밀집도가 더 높아서 분할면에서 복사되는 삼각형들이 많이 발생하였기 때문이다. 불안전한 삼각형쌍들을 CPU에서 처리하는 시간을 보면 GPU에서 처리되는 시간에 비해 상당히 긴 것을 알 수 있다. knot⊕torus의 예를 보면 전체 삼각형쌍 중에 2.3%에 해당하는 불안전한 삼각형쌍들을 CPU에서 처리하는데 걸리는 시간이 GPU 교차계산의 5배에 이르는 것을 알 수 있다. 이러한 차이가 생기는 가장 큰 요인은 GPU의 성능이 압도적으로 높기 때문이기도 하지만, 배밀도 부동소숫점 연산으로 인한 CPU 연산시간 증가와 CLP를 적용하는데 따른 추가부담이 큰 영향을 미친 것으로 생각된다. 전체적인 성능향상은 CPU의 단일 코어를 사용하여 실행한 기존 알고리즘 대비 최대 90.6배에 이른다. 불안전 삼각형쌍들이 상대적으로 많았던 knot⊕torus에서는 가장 낮은 16.7배의 성능향상만을 얻을 수 있었지만, 평균 성능향상은 48배정도가 된다. 하드웨어의 성능 차이가 7-10배 정도인 점을 감안하면 GPU에서 사용한 단일정밀도 연산과 부동소숫점 필터링이 성능향상에 큰 기여를 한 것으로 생각된다. CLP로 인한 입력 데이터의 변형은 표1의 마지막 열에 정리되어 있는데, 최대 6.9×10^{-10}의 변형이 생기는 것을 볼 수 있다.

6. 결론 및 향후 연구

우리는 대량의 삼각형 집합 안에서의 교차 계산을 효율적이며 강인하게 처리할 수 있는 GPU 알고리즘을 제안하였다. 이 알고리즘은 k-d 트리의 구성, 삼각형쌍 생성, 정확한 교차 계산을 모두 GPU에서 처리하였다. k-d 트리는 분할 과정에 복사된 삼각형들이 많이 생성된다. 이러한 삼각형들로 인해 만들어지는 중복 삼각형쌍들을 효율적으로 제거하기 위하여 분할 인덱스를 도입하였다. 수치적 강인성을 높이기 위하여 부동소숫점 필터링을 통해 불안전한 삼각형쌍들을 분리하고 CLP를 이용하여 CPU쓰레드에서 처리하도록 하였다. 제안된 알고리즘은 기존의 민코스키합 알고리즘의 합삼각형 교차계산에 적용하여 효율성과 강인성을 입증하였다.

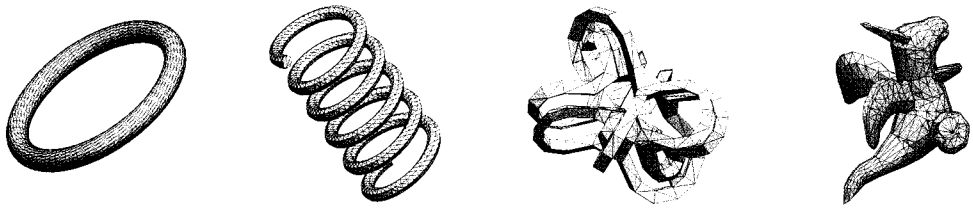
향후 연구로 교차 계산 결과로부터 삼각형 집합의 공간 배치 구조(3D arrangement)를 구성하는 전 과정을 GPU로 구현하려고 한다. 공간 배치 구조는 이론적으로 n개의 삼각형에 대하여 최대 $O(n^3)$의 복잡도를 가진다. 따라서 상당히 계산 시간이 많이 걸리지만 공간 배치 구조가 고속으로 정확하게 계산될 수 있다면 많은 응용 분야에 이용될 수 있다. 특히, 3차원 민코스키합 계산부터 다면체의 불 연산, 스윕 볼륨 계산, 오프셋 곡면 등의 가속이 대표적으로 중요한 응용 분야이다. 우리는 기존에 민코스키합 계산을 위해 개발한 공간 배치 구조 알고리즘을 GPU에 맞게 변형하는 방향으로 접근하려고 한다.

감사의 글

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

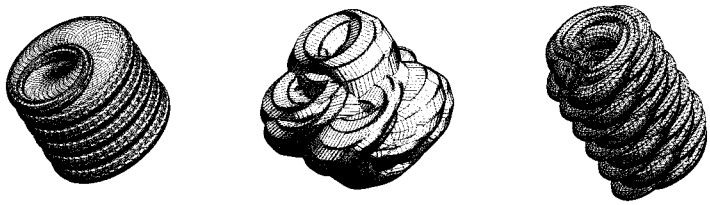
참고 문헌

[1] S. Gottschalk, M. C. Lin, and D. Manocha, "Obbtree: A hierarchical structure for rapid interference detection," in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '96. ACM, Jun 1996, pp. 171–180.

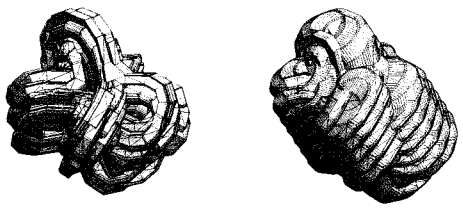


torus helix knot dragon

그림 7: 민코스키합 계산에 사용된 다면체들



torus ⊕ helix knot ⊕ torus dragon ⊕ helix



dragon ⊕ knot knot ⊕ helix

그림 8: 민코스키합 계산 결과

[2] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast bvh construction on gpus," *Eurographics 2009*, vol. 28, no. 2, pp. 1–10, Dec 2009.

[3] C. Lauterbach, Q. Mo, and D. Manocha, "gproximity: Hierarchical gpu-based operations for collision and distance queries," *Computer Graphics Forum*, Jan 2010.

[4] H. Samet, *The Design and Analysis of Spatial Data Structures*. Addision-Wesley, 1989.

[5] I. Wald and V. Havran, "On building fast kd-trees for ray tracing, and on doing that in $o(n \log n)$," in *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, Jan 2006, pp. 61–69.

[6] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time kd-tree construction on graphics hardware," *ACM Transactions on Graphics*, vol. 27, no. 5, pp. 126:1—126:11, Dec 2008.

[7] W. Jung, H. Shin, and B. K. Choi, "Self-intersection removal in triangular mesh offsetting," *Computer-Aided Design and Applications*, vol. 1, pp. 477–484, Jan 2004.

[8] P. Volino and N. M. Thalmann, "Efficient self-collision detection on smoothly discretized surface animations using geometrical shape regularity," *Computer Graphics Forum*, vol. 13, no. 3, pp. 155–166, August 1994.

[9] M. Campen and L. Kobbelt, "Exact and robust (self) intersections for polygonal meshes," *Computer Graphics Forum*, vol. 29, no. 2, pp. 397–406, Jan 2010.

[10] M. Campen and L. Kobbelt, "Polygonal boundary evaluation of minkowski sums and swept volums," *Computer Graphics Forum*, vol. 29, no. 5, pp. 1613–1622, 2010.

[11] V. Milenkovic, E. Sacks, and M.-H. Kyung, "Robust minkowski sums of polyhedra via controlled linear perturbation," in *Proceedings of Symposium on Solid and Physical Modeling 2010*, Jan 2010, pp. 23–30.

	sum polygons	polygon pairs	Safe pairs		Unsafe pairs		Perturbation δ
			distant	intersection	distant	intersection	
torus ⊕ helix	54,482	207,142	138,489	67,598	481	574	1.3×10^{-12}
knot ⊕ torus	38,327	193,249	116,393	72,365	2,745	1,746	6.9×10^{-10}
dragon ⊕ helix	136,163	829,670	624,642	203,332	1,104	592	9.6×10^{-14}
dragon ⊕ knot	75,777	1,670,012	1,384,978	283,745	954	335	7.1×10^{-13}
knot ⊕ helix	129,432	1,516,188	549,606	965,983	162	437	1.7×10^{-12}

표 1: 입력 삼각형 집합과 결과.

	kd-tree	polygon pairs	intersection (GPU)	intersection (unsafe pairs)	total time
torus ⊕ helix	9	8.6	10	15.7	43
knot ⊕ torus	10	10.4	10.5	52.6	83.5
dragon ⊕ helix	22	28.2	27.8	21.3	99.2
dragon ⊕ knot	20.9	55.3	47.6	13.5	137
knot ⊕ helix	28.4	72	54.3	7.95	163

표 2: 삼각형 교차계산 성능 결과(ms).

[12] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with cuda," in *GPU Gems 3*, H. Nguyen, Ed. Addison Wesley, August 2007, ch. 39, pp. 851–876.

[13] O. Tropp, A. Tal, and I. Shimshoni, "A fast triangle to triangle intersection test for collision detection," *Computer Animation and Virtual Worlds*, vol. 17, pp. 527–535, 2006.

[14] S. Abrams and P. K. Allen, "Computing swept volumes," *Journal of Visualization and Computer Animation*, vol. 11, pp. 69–82, Jan 2000.

[15] A. Requicha and H. Voelcker, "Boolean operations in solid modeling: Boundary evaluation and merging algorithms," in *Proceedings of the IEEE*, Jan 1985, pp. 30–44.

[16] X. Zhang, Y. J. Kim, and D. Manocha, "Reliable sweeps," in *2009 SIAM/ACM Joint Conference on Geometric and Solid Modeling*. ACM, Aug 2009, pp. 373–378.

[17] Y. J. Kim, G. Varadhan, M. C. Lin, and D. Manocha, "Fast swept volume approximation of complex polyhedral models," *Computer-Aided Design*, vol. 36, pp. 1013–1027, Jun 2004.

	our GPU algorithm(s)	full CPU algorithm(s)	speedup
torus ⊕ helix	0.043	1.59	36.9
knot ⊕ torus	0.084	1.4	16.7
dragon ⊕ helix	0.099	5.93	59.9
dragon ⊕ knot	0.137	5.11	37.3
knot ⊕ helix	0.163	14.78	90.6

표 3: 기존 알고리즘 대비 제안된 GPU 알고리즘의 삼각형 교차 계산에서의 성능 향상비.

〈저자소개〉

경민호

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

곽종근

- 1993년 포항공과대학교 전자계산학과 학사 졸업
- 1996년 포항공과대학교 전자계산학과 석사 졸업
- 2011년 아주대학교 산업공학과 박사 졸업
- 2011년~현재 주식회사 유디엠텍 대표이사
- 관심분야 : 기하 알고리즘, Virtual Factory

최정주

- 1990년 한국과학기술원 전산학과 학사
- 1992년 포항공과대학교 컴퓨터공학과 석사
- 1997년 포항공과대학교 컴퓨터공학과 박사
- 2002년~현재 아주대학교 미디어학부 부교수
- 관심분야 : 컴퓨터그래픽스, 비사실적 애니메이션 및 렌더링, 영상처리, 그래픽시스템