이화여자대학교 대학원
2008학년도
석사학위 청구논문

CUDA를 이용한 병렬형 충돌검사 및 동역학 시뮬레이션 연구

컴 퓨 터 정 보 통 신 공 학 과

이 연 희

2 0 0 9

CUDA를 이용한 병렬형 충돌검사 및 동역학 시뮬레이션 연구

이 論文을 碩士學位 論文으로 提出함

2009年 1月

梨花女子大學校 大學院

컴퓨터정보통신공학과 李 妍 憙

李 妍 憙의 碩士學位 論文을 認准함

指導敎授 김 영 준 _____

審査委員 이 미 정 _____

박 현 석 _____

김 영 준 _____

梨花女子大學校 大學院

목 차

그 림 목 차

표 목 차

수 식 목 차

코 드 목 차

논 문 개 요

최근 그래픽스 하드웨어를 범용 연산에 활용하는 GPGPU (General Purpose computing on Graphics Processing Units) 분야가 차세대 고속 컴퓨터 연산 기술로 각광을 받게 됨에 따라 그래픽스 하드웨어 제조 회사들이 GPGPU 프로그래밍에 적합한 그래픽스 하드웨어 및 프로그래밍 도구를 개발하게 되었다. 그 중 대표적인 것이 NVIDIA사의 CUDA이다. 이 하드웨어와 도구들은 범용 연산에 적합하지 않은 쉐이딩 언어를 이용해야 한다는 기존 GPU 프로그래밍의 불편함과 어려움을 해소함으로써 고속 연산을 필요로 하는 다양한 분야에서 GPGPU 연구를 활발히 진행할 수 있는 초석을 마련하였다.

실 세계의 물리적 현상들을 컴퓨터 상에 재현하는 기술인 물리 기반 시뮬레이션 기술은 컴퓨터 애니메이션, 게임, 가상 현실, 영상 특수 효과 등 다양한 분야에 활용된다. 이 물리 기반 시뮬레이션에서 쓰이는 중요한 요소 두 가지는 동역학 시뮬레이션과 가상 객체 간 충돌 검사이다.

동역학 시뮬레이션 중 특히 순동역학 시뮬레이션은 가상 현실에서 객체에 외력이 가해졌을 때 객체의 위치, 속도, 가속도를 구하는 기술이다. 흔히 컴퓨터 그래픽스와 같은 응용분야에서 사용되는 동역학 시뮬레이션에서는 움직임을 표현할 대상 객체로 관절체가 주로 이용된다. 관절체는 여러 개의 강체를 관절로 연결하여 나타내는 모델로서, 인간 또는 동물 형태의 캐릭터 및 분자 구조 등을 나타내거나 로봇 시뮬레이션을 구현하기에 적합한 모델이다. 하지만 이 모델은 여러 개의 강체가 관절로 이어져 하

나의 객체를 이루기 때문에 운동 방정식을 계산하기가 매우 복잡하다. 또한 관절체를 이루는 강체의 개수가 많아질수록 계산은 더욱 복잡해진다. 이를 해결하기 위해 동역학 시뮬레이션 연산 단계를 병렬화 하거나 단순화 시켜 계산하는 방법들이 제시된 바 있다.

또한, 가상 객체 간 충돌 검사는 가상의 객체들이 서로 충돌하는 것을 감지하여 객체들이 서로 관통하는 것을 방지하고, 충돌에 의해 발생하는 반발력을 생성함으로써 동역학 시뮬레이션에서 객체들의 움직임에 더욱 사실감을 부여해주는 방법이다. 특히 충돌 검사를 위한 여러 기법과 단계 중에서도 객체들을 구성하는 삼각형 메쉬 충돌 검사는 충돌 검사의 필수적인 단계이자 실시간 충돌 검사 시 주요 병목현상을 일으키는 부분이다.

본 논문에서는 CUDA를 이용해 관절체 동역학 시뮬레이션과 가상 객체들 간 충돌 검사 알고리즘을 각각 병렬적으로 구현하였다. 특히 본 논문에서 구현한 병렬 관절체 동역학 시뮬레이션은 Roy Featherstone의 Divide-and-Conquer 알고리즘 [9, 10]을 기반으로 하였다. 특히 이 알고리즘은 병렬 프로세서의 개수가 n개일 때 이론적으로 O(log(n))의 시간 복잡도를 갖는다고 알려져 있고, 본 논문에서는 이를 CUDA를 이용해 실험적으로 증명해 보았다. 또한 본 논문에서 구현된 가상 객체들 간 병렬 충돌 검사 알고리즘은 삼각형 메쉬 충돌 검사를 모든 쌍의 삼각형 메쉬에 대해 병렬적으로 수행함으로써 전체적인 충돌 검사 알고리즘의 성능을 약 11배까지 향상시킬 수 있었다.

Ⅰ 서 론

A. 연구 배경

1. 그래픽스 하드웨어를 이용한 범용 용도의 연산

실시간 컴퓨터 게임 및 인터랙티브 그래픽스 기술에 대한 수요가 증가하면서 그래피컬 한 영상 처리를 빠르게 수행할 수 있는 그래픽스 하드웨어(GPU)에 대한 요구가 최근 급격하게 증대하였다. 그 결과 오늘날의 GPU는 CPU에 비해 부동 소수점 처리를 비롯한 여러 부분의 연산 능력과 데이터 대역폭 면에서 훨씬 앞선 성능을 발휘하게 되었다. (그림 1-1)은 GPU의 부동 소수점 연산 처리 성능이 CPU에 비해 급격한 속도로 발전한 것을 나타내는 그래프이다.

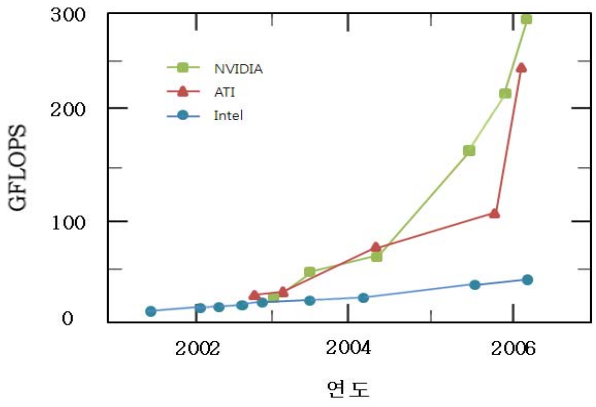


그림 1-1

Intel CPU와 ATI사의 GPU, NVIDIA사의 GPU 간

연도별 신제품의 부동 소수점 연산 성능 비교 [17]

또한 기존 GPU의 용도는 단순한 그래픽스 작업 처리의 가속화였지만 최근에는 그래픽스 영상을 더욱 실제와 가깝게 표현하기 위한 목적으로 프로그래밍이 가능한 GPU가 개발되었으며 이를 이용한 텍스처링 또는 쉐이딩 방법에 대한 연구가 활발히 이루어져왔다 [16, 19, 20, 22]. 이처럼 그래픽 연산 처리를 목적으로 개발된 프로그래밍 가능한 Graphics Hardware 혹은 GPU를 본래의 용도가 아닌 범용 연산의 용도로 활용하는 것을 GPGPU(General Purpose computing on GPU)라고 한다. 많은 프로세서와 넓은 대역폭을 가지고, 빠른 연산 처리에 뛰어나다는 GPU의 장점을 병렬 처리로 연산 횟수를 크게 줄일 수 있는 연산 또는 빠른 데이터 처리를 요구하는 실시간 연산 처리에 적용시킨다면 효율적인 결과를 얻을 수 있다. 예를 들면, 데이터베이스 관리 시스템, 이미지 프로세싱, 물리 기반 모델링, 암호 해독, 인공신경망 연산 등 다양한 분야에 GPU를 응용한 연구들이 있다 [3, 5, 13, 23].

GPU 프로그래밍을 위해 지금까지는 Cg와 같은 고급 쉐이딩 언어가 주로 쓰였다. 그러나 Cg는 그래픽 프로그래밍 작업에 특수화된 언어이므로 범용적인 애플리케이션 개발에는 직관적이지 않고 그래픽스 프로그래밍을 전문적으로 다루어보지 않은 개발자에게 어렵다는 단점이 있다. 하지만 요즘은 이러한 연구 추세에 맞추어 GPU 개발사들이 GPGPU 애플리케이션 개발을 효율적으로 하기 위한 개발 환경 및 아키텍처를 개발하고 있다.

가. CUDA

위의 절에서 언급했던 GPGPU를 위한 개발 환경 및 아키텍처로 대표적인 것이 NVIDIA사의 CUDA이다. CUDA(Compute Unified Device Architecture) 는 NVIDIA

GPU를 위한 통합된 병렬 데이터 연산을 제공하는 소프트웨어 개발 도구이다[15].

CUDA의 가장 큰 장점은 개발자가 보다 쉽고 직관적으로 GPU를 프로그래밍 할 수 있도록 확장된 형태의 C언어를 지원한다는 것이다. 그렇기 때문에 C언어에 익숙한 개발자라면 GPGPU 프로그래밍을 위해 따로 쉐이딩 언어를 익힐 필요가 없다.

CUDA는 병렬 연산을 위한 프로그래밍 단위로 커널(kernel), 그리드(grid), 블록(block) 및 스레드(thread)를 제공한다. 이 중 스레드는 연산이 병렬 처리되는 기본 단위이다. 블록은 서로 의사소통이 가능한 스레드들의 집합이며, 같은 커널을 실행시키는 블록들이 모여서 그리드를 이룬다. 커널은 병렬 처리 해야 할 연산을 담은 함수로서, GPU가 호출하는 기본적인 프로그램을 담고 있는 단위이다.

CUDA의 큰 특징은 공유 메모리(shared memory)의 사용에 있다. 이 공유 메모리는 온칩(On-chip) 메모리로서 로컬 메모리나 글로벌 메모리 같은 장치 메모리(device memory)에 접근하는 것보다 훨씬 빨리 접근할 수 있다. 공유 메모리는 동일 블록내의 스레드들이 공유하며, CUDA는 이 공유 메모리를 이용해 데이터 접근 시간을 단축시킴으로써 기존의 CPU 보다 연산을 더 빨리 처리할 수 있다 (그림 1-2).

일반적으로 GPU는 많은 수의 멀티프로세서를 갖는데, 예를 들면 CUDA를 지원하는 그래픽 카드 중 GeForce 8800GTX는 16개, GeForce GTX 280은 30개의 멀티프로세서를 갖는다. 하나의 멀티프로세서는 물리적으로 한 번에 32개까지의 스레드를 실행할 수 있으며 이 단위를 와프(warp)라고 한다. 한 와프 내의 스레드들은 동일한 명령어를 수행한다. 또한 하나의 멀티프로세서는 GeForce 8800GTX의 경우 동시에 24개, GeForce GTX 280의 경우 동시에 32개까지의 와프를 가질 수 있다. 실질적으로 한 번에는 단 한 개의 와프만이 연산을 수행할 수 있지만 멀티프로세서 상에 올라

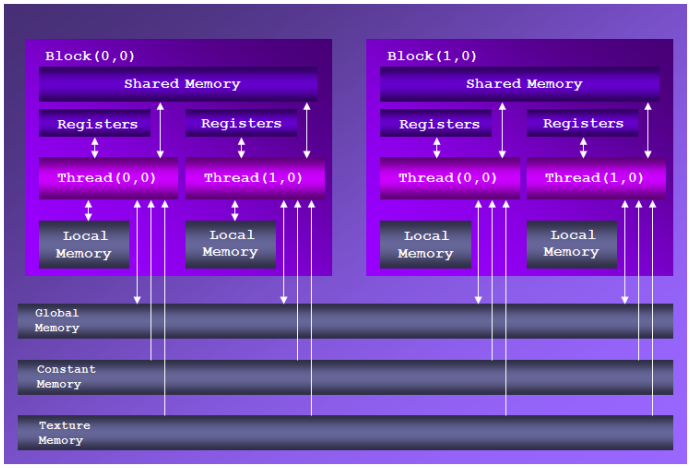


그림 1-2

CUDA의 메모리 구조 [15]

간 와프들은 모두 활성 상태이기 때문에 하나의 와프가 연산을 수행하는 동안 나머지 와프들은 글로벌 메모리로부터 데이터를 기다리는 등의 일을 수행한다. 그리고 하나의 블록은 한 개의 멀티프로세서 상에서만 수행되고 여러 멀티프로세서로 나뉘어서 수행될 수 없다. 멀티프로세서는 최소 1개에서 최대 8개까지의 블록을 실행시킬 수 있는데 하나의 멀티프로세스가 갖게 되는 블록의 개수는 커널의 레지스터와 공유 메모리 사용 빈도에 의해 결정된다.

2. 관절체 동역학 시뮬레이션

컴퓨터 그래픽스, 특히 컴퓨터 애니메이션 분야의 중요한 목표 중의 하나는 물체의 움직임을 실 세계의 그것과 같은 움직임을 만드는 것이다. 특히 물리 기반 시뮬레이션을 사용하여 실제와 같은 움직임을 얻기 위해서는 중력, 점성, 마찰, 충돌 등의 힘

이 강체에 가해졌을 때 그 강체의 물리적인 반응을 동역학적으로 구해야 한다 [18]. 여기서 동역학이란 물리계에서 힘이 물체들의 움직임에 미치는 영향을 구하는 분야이다. 물체들의 움직임을 묘사하는 가장 기초적인 동역학은 뉴턴의 동역학 제 2 법칙 $F = ma$ 이다. 여기서 m은 물체의 질량이며 a는 가속도, F는 물체에 가하는 힘이다. 이 식을 이용해 주어진 힘으로부터 물체의 가속도를 결정할 수 있고, 가속도를 적분함으로써 속도와 위치 또한 구할 수 있다 [7].

여기서 강체란 물리적 현상을 쉽게 기술하기 위하여 도입된 물체로, 이를 구성하는 입자들 사이의 거리가 항상 일정하게 유지되어 외력을 가해도 전체적인 크기나 모양이 변형되지 않는 가상의 물체를 뜻한다. 엄밀한 의미의 강체는 현실에 존재하지 않지만 대부분의 고체는 아주 큰 힘이 가해지지 않는 이상 변형되거나 부서지지 않으므로 강체로 간주한다. 어떤 물체를 강체로 간주하면 물체 각 부분의 상대적 위치가 변하지 않기 때문에 그 물체의 운동을 기술하기가 수월해진다. 모든 운동을 무게 중심의 평행이동과 무게 중심 주위의 각운동량 변화로 나타낼 수 있기 때문이다.

그런데 컴퓨터 애니메이션에서 모든 오브젝트를 하나의 강체만으로 표현할 수는 없다. 인간이나 동물, 기계 등의 물체들을 나타내기에 가장 적합한 모델은 관절체이다. 관절체는 여러 개의 강체를 관절로 연결해 하나의 객체를 이루도록 한 것이기 때문에 부모-자식 관계를 갖는 계층 구조로 이루어져 있다 (그림 1-3). 이런 계층 구조 모델의 애니메이션은 로보틱스 분야에서 직접적으로 유래되었다. 로보틱스에서는 관절에 의해 체인 형태로 연결된 오브젝트들의 집합인 로봇팔의 모델링과 운동에 대한 연구를 진행해왔다. 계층 구조의 관절체는 이를 구성하는 각각의 강체들이 부모 강체와 자식 강체로부터 영향을 주고 받기 때문에 전체 움직임을 구하기 위한 동역학 알고리

즘이 단일한 강체의 움직임을 구하기 위한 그것보다 복잡하다. 관절체 동역학 알고리즘을 보다 간편하게 나타내기 위해 Featherstone 등은 핸들이나 공간 대수와 같은 개념을 도입하기도 한다 [8]. 또한 Featherstone은 관절체를 이루는 계층 구조 트리에서 같은 높이에 있는 노드들에 대한 동역학 시뮬레이션 연산을 병렬적으로 처리할 수 있는 알고리즘을 제안한 바 있다 [9, 10].

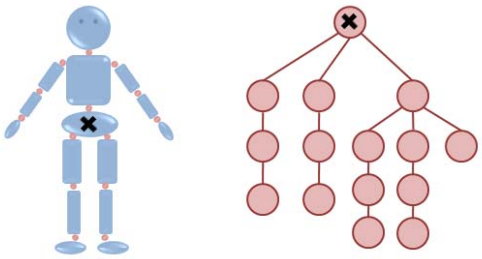


그림 1-3

관절체의 계층 구조 [1]

3. 가상 객체 간 충돌 검사

컴퓨터 그래픽스, 가상현실, 햅틱스, 로보틱스 등의 애플리케이션에서 충돌 검사는 가상의 객체와 그 객체가 일으키는 물리적 사실감을 유지하게 하는 중요한 기술이다. 이러한 충돌검사에서 가장 기본적인 검사는 각각의 오브젝트를 구성하는 메쉬들의 가능한 모든 쌍들 간의 교차 검사를 수행하는 것이다. 가상 오브젝트들의 기본 구성 단위인 메쉬는 주로 삼각형이므로 삼각형 간 교차 검사를 수행하는 알고리즘이 많이 연

구되었다 [14].

하지만 오브젝트들이 수많은 삼각형 메쉬들로 이루어져있기는 경우 이런 오브젝트들에 대해 모든 삼각형 쌍들 간 교차 검사를 수행하는 것은 많은 시간이 소모되어 바람직하지 않다. 따라서 충돌 검사 시에는 일반적으로 BVH(Bounding Volume Hierarchy)를 이용한 알고리즘을 쓴다 [11]. 이 알고리즘들은 객체를 구성하는 기본 도형인 삼각형들을 바운딩 박스에 포함되는 범위에 따라 계층 구조로 구조화한 뒤, 계층적인 충돌 검사로 충돌 가능성이 있는 삼각형 집합만을 추려내는 과정을 거치기 때문에 불필요한 삼각형 레벨에서의 연산을 줄일 수 있다. 이 때 주로 쓰이는 바운딩 박스로는 AABB(Axis Aligned Bounding Box)와 OBB(Oriented Bounding Box) 등이 있다 (그림 1-4).

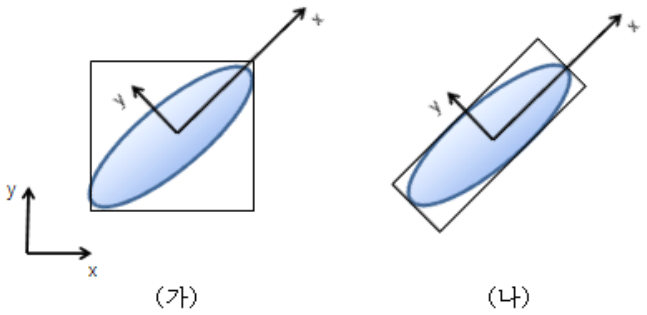


그림 1-4 [4]

(가): AABB (나) OBB

하지만 변형 모델(deformable model) 혹은 파쇄(fracturing) 효과를 동반하는 모델의 충돌 검사를 할 경우엔 위의 BVH 기반의 컬링 과정을 거치더라도 false positive가 많이 생성되어 충돌 검사를 해야 할 삼각형 집합이 많아 진다. 이런 경우엔 삼각형

충돌 검사를 수행하는 부분이 전체 시스템의 연산 시간을 지연시킨다 [24]. 이러한 삼각형 쌍들 간의 충돌 검사의 경우 각 쌍들의 충돌 검사는 병렬적으로 수행이 가능하며 이는 SIMD(Single Instruction Multiple Data) 방식의 병렬 처리에 적합하다. 따라서 병렬 연산으로 많은 삼각형 쌍에 대한 충돌 검사를 동시에 처리한다면 가상 객체 간 충돌 검사 시스템의 성능을 크게 향상시킬 수 있다.

B. 연구의 목적 및 내용

본 논문에서는 관절체 동역학 시뮬레이션과 가상 객체들 간 충돌 검사를 CUDA를 이용해 병렬적으로 구현하였다.

관절체 동역학 시뮬레이션은 Featherstone의 Divide-and-Conquer 방식으로 관절체의 운동을 계산하는 알고리즘을 GPU 상에 구현하였다. 특히 128~2048개의 강체들로 구성된 관절체들에 동역학 시뮬레이션을 수행하여 강체의 개수가 logarithmic하게 증가함에 따라 시뮬레이션의 성능이 그에 선형적으로 증가하는 것을 실험적으로 보였다.

가상 객체들 간 충돌 검사에서는 CPU 기반의 빠른 삼각형 교차 검사 알고리즘을 GPU 기반의 병렬 알고리즘으로 재구성 하였다. 그리고 구현한 알고리즘을 세 개의 테스트 케이스에 대해 수행하여 테스트 케이스의 복잡도에 따른 알고리즘의 수행 속도 차이를 알아보았다. 세 개의 테스트 케이스는 가상의 오브젝트를 구성하는 삼각형 메쉬의 개수에 따라 가장 간단한 테스트 케이스(216×960)에서부터 가장 복잡한 테스트 케이스(7580×26012)까지 나뉜다. 또한 본 연구에서 GPU로 구현한 알고리즘과

기존 CPU 기반 알고리즘 간의 성능 차이에 대해서도 알아보았다.

본 논문의 구성은 다음과 같다. 2장에서는 관절체 동역학 시뮬레이션과 가상 객체 간 충돌 검사에 대한 이전 연구 결과에 대해 기술하였다. 3장에서는 관절체 동역학 시뮬레이션이 기반으로 하는 Divide-and-Conquer 방식의 병렬 동역학 알고리즘에 대해 소개하고, 이 알고리즘을 GPU 상에 구현하기 위한 자료 구조 및 기법에 대해 설명하였다. 4장에서는 기존에 쓰이던 Tomas Möller의 CPU 기반 삼각형 충돌 검사 알고리즘을 기반으로 GPU 기반의 가상 객체 간 병렬 충돌 검사 알고리즘으로 재구성하기 위해 사용한 자료 구조와 구현 기법에 대해 설명하였다. 5장에서는 CUDA 기반 관절체 병렬 동역학 시뮬레이션과 CUDA 기반 병렬 충돌 검사 알고리즘에 대하여 주어진 테스트 케이스들을 수행하고 얻은 수행 결과와 이들의 성능을 비교, 분석하였다. 마지막으로 6장에서는 본 연구에 대한 결론과 향후 연구에 대해 제시한다.

Ⅱ 관련 연구

A. 관절체 동역학 시뮬레이션

서론에서 언급했듯이 복잡한 관절체 동역학 알고리즘을 보다 간편하게 나타내기 위해 Featherstone의 논문 등에서는 핸들이나 공간 대수와 같은 개념을 이용하였다.

여기에서 핸들이란 관절체와 외부 좌표계를 연결하는 인터페이스라고 할 수 있다. 부분 관절체가 관절 혹은 다른 부분 관절체와 이어질 때 이 핸들을 통해서 이어지게 된다. 관절체에서 전역 좌표계로 좌표 이동을 할 때도 핸들을 거친다. (그림 2-1)이 핸들을 추상적으로 나타낸 예이다. 검은색으로 표시된 좌표계는 강체의 지역 좌표계이며, 붉은색으로 표시된 좌표계는 핸들의 좌표계이다. 핸들의 좌표계에 따라 강체 간의 연결이 결정됨을 볼 수 있다.

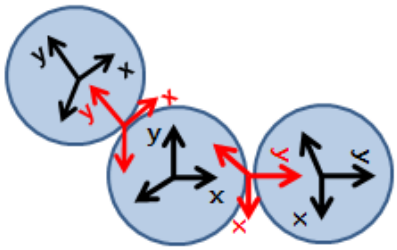
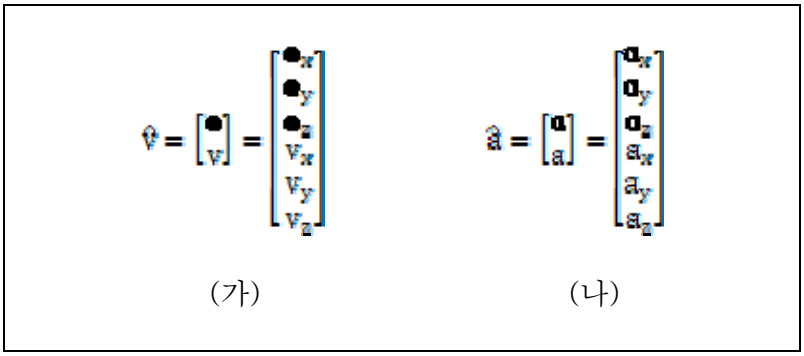


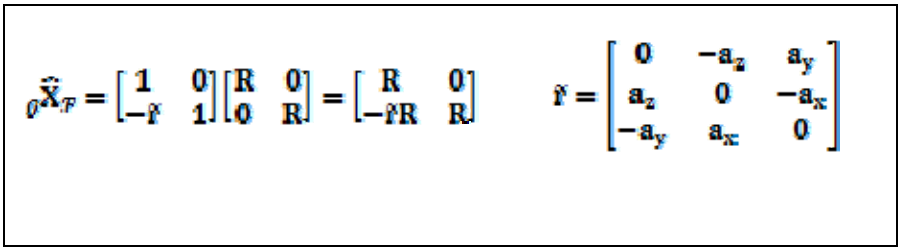
그림 2-1

관절체의 강체들을 연결하는 핸들



(가) (나)

수식 2-1



수식 2-2

공간 대수에서 공간 벡터는 강체의 속도나 가속도, 관성 등을 6차원 벡터나 텐서로 간결하게 표현하는 기수법이다. 예를 들면 (수식 2-1)에서 (가)는 속도 공간 벡터를 나타낸 것이고, (나)는 가속도 공간 벡터를 나타낸 것이다. 속도 공간 벡터의 첫 번째 요소 ω는 3차원의 각속도 벡터이며, 두 번째 요소 v는 3차원의 선속도 벡터이다. 두 요소가 합쳐져서 6차원의 속도 공간 벡터를 이룬다. 가속도 공간 벡터의 구성 방식도 속도 공간 벡터와 동일하다. 다만 가속도 공간 벡터의 첫 번째 요소는 각가속도 벡터이고, 두 번째 요소는 선가속도 벡터인 것이 차이이다. 그리고 공간 벡터의 좌표계 변환을 위한 공간 행렬이 (수식 2-2)와 같은 공간 변환 행렬이다. 여기서 ${}^{B}\hat{X}_{A}$는 F 좌

표계에서 $\mathcal{G}$ 좌표계로의 변환을 뜻하는 6×6 공간 행렬이며, $\mathbf{R}$은 $\mathcal{F}$ 좌표계의 방위에서 $\mathcal{G}$ 좌표계의 방위로 변환하는 3×3 회전 행렬, $\mathbf{r}$은 $\mathcal{F}$ 좌표계의 중심에서 $\mathcal{G}$ 좌표계 중심까지의 3차원 거리 벡터를 의미한다.

이외에도 최근까지 관절체 동역학 시뮬레이션을 위한 많은 연구들이 진행되어 왔으며 특히 로보틱스 분야에서 활발히 연구가 진행되어 왔는데 그 중 대표적인 것이 순동역학이다. 순동역학은 주어진 외력과 능동적인 관절의 힘으로부터 관절체를 구성하는 강체 각각의 가속도와 움직임을 계산하는 방식이다 [21]. 지금까지도 이 순동역학에 대한 연구는 활발히 이루어지고 있는데 그 중 가장 일반적인 것이 재귀적인 공식 계산을 이용한 순동역학 방식이다 [2, 8, 12]. 하지만 이와 같은 순동역학 알고리즘은 이에 필요한 수식이 복잡하여 관절체를 이루는 강체의 개수를 n개라 했을 때 알고리즘 연산에 소요되는 비용이 O(n) 정도로 크다. 따라서 많은 강체들로 이루어진 관절체나 많은 수의 관절체 집합에 대한 시뮬레이션을 구현할 때는 이러한 방식을 이용하기가 어렵다. 이 점을 보완하기 위해 제안된 방법 중 하나로 관절체의 자유도와 능동적인 관절의 힘 및 외력이 작용하는 위치를 이용해 자동적으로 동역학 시뮬레이션의 연산 단계를 단순화 시켜 계산하는 알고리즘이 있다 [21]. 이 알고리즘은 위의 일반적인 알고리즘들 보다 빠른 성능 향상을 보이지만 빠른 시뮬레이션 수행을 위해 연산 단계를 단순화 시킬수록 정확도가 떨어진다는 단점을 갖는다.

순동역학 시뮬레이션을 빠르게 처리하기 위한 또 다른 방법으로 시뮬레이션에 필요한 운동 방정식을 Divide-and-Conquer 방식을 이용해 병렬적으로 계산하는 알고리즘[9, 10]이 있다. 이 알고리즘은 프로세서의 개수가 n개라고 할 때 이론적으로 O(log(n))의 시간 복잡도를 갖게 되므로 효율성이 프로세서의 개수에 크게 의존하며,

프로세스가 한 개일 때는 오히려 일반적인 동역학 시뮬레이션 알고리즘 보다 성능이 떨어진다는 한계가 있다. 또한 이 알고리즘을 직접 병렬적으로 구현하기 위해서는 많은 프로세서를 갖는 고가의 컴퓨터 등 구현에 필요한 환경을 갖추는 것이 용이하지 않아 지금까지 그 효율성이 실질적으로 증명된 바가 없다. 본 논문에서는 GPU의 내재된 많은 프로세서를 이용하여 이 알고리즘의 성능을 실험적으로 증명하였다.

B. 삼각형 프리미티브 단계에서의 객체 간 충돌 검사

가상 객체 간의 충돌 검사 시에는 가상 객체를 이루는 단위인 프리미티브들 간의 충돌 검사가 필요하다. 컴퓨터 그래픽스에서 프리미티브가 되는 단위는 삼각형으로 Möller의 논문에서는 CPU 기반의 효율적인 삼각형 충돌 검사 알고리즘을 제시한다 [14]. 이 알고리즘은 현재 알려진 삼각형 간 충돌 검사 알고리즘 중 가장 효율적이기 때문에 많은 충돌 검사 알고리즘들이 삼각형 간 충돌 검사 단계에서 이를 이용한다. 하지만 검사해야 할 삼각형의 개수가 아주 많은 경우에는 이 알고리즘 역시 모든 삼각형 쌍에 대하여 연산하는데 상당 시간이 소요된다.

Zhang의 논문에서는 AABB 바운딩 박스 단위의 충돌 검사 시 GPU를 이용해 병렬적으로 수행함으로써 연산에 소요되는 시간을 단축했다 [24]. 하지만 이 AABB 바운딩 박스를 이용한 컬링의 과정 후에도 충돌 가능성이 있는 삼각형 쌍이 많아 남아있는 경우, 삼각형 교차 검사 수행으로 인한 병목 현상이 발생한다. 또한 이 논문에서 이용한 GPU는 GPGPU에 최적화된 GPU가 아니기 때문에 범용적인 계산에 적합하지 않은 쉐이딩 언어로 구현되었다.

Choi의 논문은 가상 모델 간 충돌 검사 시 삼각형 간 충돌 검사 단계에서 GPGPU를 활용한다 [6]. 이를 CPU로 구현했을 때와는 달리 삼각형 쌍의 수가 많아져도 연산 시간이 크게 지연되지 않지만 구현에 이용한 GPU의 한계로 인해 1000개 이상의 삼각형을 갖는 복잡도가 높은 모델에는 적용하기 어렵다는 문제점이 있다. 그리고 이 논문 역시 GPGPU 프로그래밍을 위해 쉐이딩 언어를 사용하였다.

Ⅲ GPU 기반 병렬 관절체 동역학 시뮬레이션

A. 관절체 동역학 병렬 연산 알고리즘

본 연구에서 구현한 병렬 관절체 동역학 시뮬레이션은 Featherstone의 Divide-and-Conquer 관절체 동역학 알고리즘을 기반으로 한다[9, 10]. Divide-and-Conquer 관절체 동역학 알고리즘은 관절체를 이루는 강체들의 운동을 조합하여 전체 관절체의 운동을 구할 수 있다는 것에 착안한 알고리즘이다. 관절체의 운동은 (수식 3-1)에 기반하여 구한다.

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix} = \begin{bmatrix} \emptyset_1 & \emptyset_{12} & \cdots & \emptyset_{1m} \\ \emptyset_{21} & \emptyset_2 & \cdots & \emptyset_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ \emptyset_{m1} & \emptyset_{m2} & \cdots & \emptyset_m \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_m \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

a_i: i번째 핸들의 공간 가속도
f_i: i번째 핸들에 작용하는 외력
b_i: i번째 핸들에 작용하는 외력이 0일 때의 가속도(바이어스 가속도)
$\emptyset_i$: i번째 핸들의 공간 역관성 행렬
$\emptyset_{ij}$: i번째 핸들과 j번째 핸들 간의 상호 결합 공간 역관성 행렬

수식 3-1

이 알고리즘의 구현을 위해서는 먼저 시뮬레이션 하고자 하는 관절체를 어셈블리 트

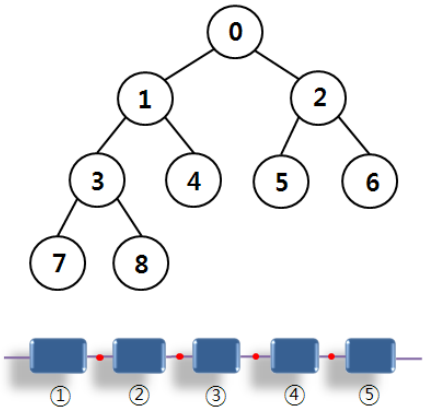


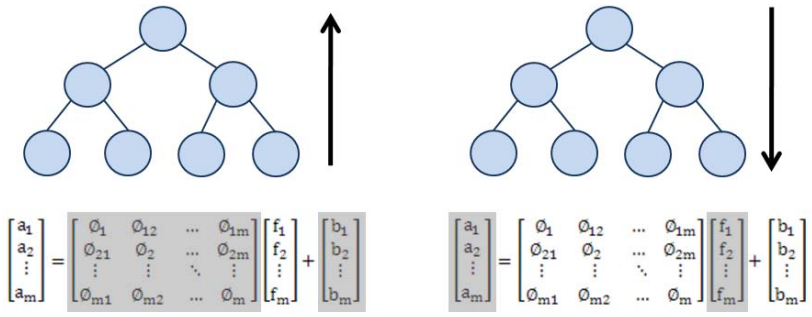
그림 3-1

관절체의 어셈블리 트리 예

리로 구성하는 단계가 필요하다. (그림 3-1)은 5개의 강체로 이루어진 관절체를 어셈블리 트리로 구성한 예이다. 강체 ①~⑤는 각각 트리의 종단 노드 7, 8, 4, 5, 6과 1:1 대응을 이룬다. 트리의 중간 노드 1, 2, 3은 각자가 갖는 왼쪽 자식 노드와 오른쪽 자식 노드가 가리키는 강체 혹은 부분 관절체를 합쳐서 만든 더 큰 부분 관절체이다. 그리하여 0번의 루트 노드는 전체 관절체를 가리키게 된다. 어셈블리 트리를 구성하는 단계는 시뮬레이션 시 최초의 한 번만 이루어진다.

트리가 완성된 후에는 크게 네 가지 단계로 이루어진 루프를 반복하며 동역학 시뮬레이션을 수행한다. 첫 번째 단계에서는 트리의 종단 노드에서부터 루트 노드까지 운행하며 관절의 위치로부터 관절체의 위치를 계산하고, 두 번째 단계에서는 트리의 루트 노드에서부터 종단 노드까지 운행하며 관절의 속도로부터 관절체의 속도를 계산한다. 이 두 단계를 일컬어 '준비 단계'라 하며 이 준비 단계에서 각 관절체의 핸들 좌

표계 간 또는 핸들과 전역 좌표계 간의 공간 변환 행렬도 구한다. 세 번째 단계는 알고리즘의 '주요 단계'로 (그림 3-2)의 (가)에서처럼 트리의 종단 노드에서부터 루트 노드까지 운행하며 각 부모 노드는 왼쪽 자식 노드와 오른쪽 자식 노드의 역관성 및 바이어스 가속도로부터 자신의 역관성과 바이어스 가속도를 계산한다. 마지막 네 번째 단계는 '치환 단계'로서 (그림 3-2)의 (나)에서처럼 루트 노드에서부터 종단 노드까지 운행하는 동안 각 자식 노드는 부모 노드로부터 받은 외력과 가속도, 바이어스 가속도로부터 자신에게 작용하는 외력과 가속도를 구한다.



(가) (나)

그림 3-2

Divide-and-Conquer 알고리즘의 연산 과정

(가): 역관성과 바이어스 가속도 계산 순서 (나): 외력과 가속도 계산 순서

B. CUDA를 이용한 관절체 동역학 시뮬레이션 구현

본 연구에서는 병렬 알고리즘 기반의 Divide-and-Conquer 관절체 동역학 알고리즘을 CUDA의 프로그래밍 환경과 GPU 하드웨어 환경에 맞도록 구현하였다. 이번 절에서는 그 과정에서 필요한 자료구조와 구현 단계, 그리고 알고리즘을 수행하는 커널에 할당하는 그리드와 블록의 크기에 대해서 설명한다.

1. 자료구조

일반적으로 트리를 구현할 때 자식노드와 부모노드는 서로 포인터를 이용해 가리키도록 하지만 데이터를 CPU에서 GPU로 복사할 때 이 포인터 주소 값이 손실되기 때문에 본 논문에서는 관절체를 나타내는 어셈블리 트리를 배열로 표현한다. 어셈블리 트리는 종단 노드의 개수가 관절체를 이루는 강체의 개수와 같은 이진 트리이고, 이진 트리의 전체 노드 개수는 $((\text{종단노드 개수}) \times 2 - 1)$개 이므로 강체의 개수를 미리 알면 트리를 구성하는데 필요한 전체 노드의 개수는 트리 배열을 할당하기 전에 미리 계산할 수 있다. (코드 3-1)에 나타낸 트리 배열은 'cuKinematicTree'라는 구조체의 배열이며 이 구조체는 왼쪽 자식 노드의 인덱스와 오른쪽 자식 노드의 인덱스, 부모 노드의 인덱스를 원소로 가진다. 이외에도 해당 노드가 가리키는 관절체가 갖는 핸들 개수, 관절 인덱스, 핸들 인덱스, 공간 역관성 행렬, 바이어스 가속도 등 Divide-and-Conquer 알고리즘 연산 시 각 트리 노드 단계마다 필요한 정보들 역시 원소로 갖는다.

```
  typedef struct cuKinematicTree {
       int leftChild;                                          // 왼쪽 자식 노드의 인덱스
       int rightChild;                                         // 오른쪽 자식 노드의 인덱
스
       int parent;                                             // 부모 노드의 인덱스
       int nHandles;                                           // 포함하는 핸들의 개수
       int handleIdx[MAX_HANDLE_NUM];                          // 핸들 인덱스
       int jointIdx;                                           // 관절 인덱스
       matrix66 phi[MAX_HANDLE_NUM][MAX_HANDLE_NUM];  // 공간 역관성
행렬
       vector6 b[MAX_HANDLE_NUM];                              // 바이어스 가속도
                        ...
    }
```

코드 3-1

관절체 구성에 필요한 핸들 정보는 ‘cuKinematicTreeHandle’이라는 구조체의 배열에 저장한다. 이 핸들 배열에는 각 강체가 갖는 핸들이 순차적으로 저장되어 있으며, 어셈블리 트리 노드들은 각자가 가리키는 부분 관절체의 핸들에 대해 인덱스 정보만을 갖는다.

(코드 3-2)는‘cuKinematicTreeHandle’ 구조체의 코드이다. 이 구조체는 각 핸들로부터 강체, 부모 노드의 핸들, 비 연결 핸들, 전역 좌표계 등으로의 좌표 변환을 나타내는 공간 변환 행렬들을 원소로 갖는다. 이 중 비 연결 핸들이란 해당 핸들이 다른 관절체와의 연결을 담당하는 인터페이스 역할을 할 때 그 핸들이 속한 부분 관절체가

갖는 나머지 핸들들을 의미한다. 따라서 비 연결 핸들로의 공간 변환 행렬은 ((해당 관절체의 핸들 개수) – 1)의 크기를 갖는 배열이 된다. 하지만 구조체 정의 단계에서는 시뮬레이션에 사용될 관절체가 갖는 핸들의 개수를 알 수가 없으므로 정적으로 정의해놓은 최대 핸들 개수를 이용해 배열의 크기를 정의한다. 이 구조체는 위의 공간 변환 행렬들 외에도 각 핸들에 작용하는 속도와 가속도, 외력에 대한 정보를 저장한다.

```
typedef struct cuKinematicTreeHandle {
    spatialTransform transformToObject;   // 강체로의 공간 변환
    spatialTransform transformToParent;   // 부모 노드의 주요 핸들로의 공간 변환
    spatialTransform transformToSecondary[MAX_HANDLE_NUM-1];
                                          // 비 연결 핸들로의 공간 변환
    spatialTransform transformToWorld;    // 전역 좌표계로의 공간 변환
    vector6 v;                            // 속도
    vector6 a;                            // 가속도
    vector6 f;                            // 외력
}
```

코드 3-2

관절체를 이루는 관절은 ‘cuKinematicTreeRevoluteJoint’라는 구조체 배열로 저장한다. 핸들 정보와 마찬가지로 어셈블리 트리 노드들은 각자가 갖는 관절에 대한 정보를 인덱스로 저장하고 필요할 때마다 인덱스를 이용해 관절 배열에 접근한다. ‘cuKinematicTreeRevoluteJoint’ 구조체는 관절의 위치, 관절의 속도, 관절의 가속도,

관절의 공간 변환 행렬, 관절이 갖는 능동적인 힘 등을 원소로 갖는다 (코드 3-3).

```
typedef struct cuKinematicTreeRevoluteJoint {
    float q;                    // 관절의 위치
    float qd;                   // 관절 속도
    float qdd;                  // 관절 가속도
    spatialTransform X;         // 관절의 공간 변환
    vector6 Q;                  // 관절의 능동적인 힘
      ...                                    }
```

코드 3-3

'cuRigidBody' 구조체 배열은 관절체를 이루는 개별적인 강체들에 대한 정보를 저장하며 강체 개수만큼의 크기를 갖는다. 이 구조체의 원소는 강체 각각의 질량, 관성, 질량 중심, 강체에 작용하는 외력, 강체 핸들 사이의 거리 등의 정보이다 (코드 3-4).

```
typedef struct cuRigidBody {
    float mass;                 // 질량
    vector3 centerOfMass;       // 질량 중심
    vector6 fk;                 // 외력
    float handleOffset;         // 핸들 사이 거리      }
```

코드 3-4

Divide-and-Conquer 관절체 동역학 알고리즘은 동역학 연산이 강체 단위 또는 어셈블리 트리의 높이 단위로 병렬 처리되기 때문에 각 높이의 노드 개수를 따로 기억해두어야 한다. 어셈블리 트리가 완전 이진 트리가 되는 경우엔 높이 정보만으로도 노드 개수를 계산할 수 있지만 가지가 있는 관절체처럼 불규칙적인 형태의 관절체를 나타낸 어셈블리 트리의 경우엔 완전 이진 트리가 되기 어려우므로 높이 정보만으로 각 높이의 노드 개수를 계산할 수 없기때문이다. 단, 알고리즘에서 종단 노드들은 높이가 다르더라도 항상 함께 병렬 처리되기 때문에 각 높이의 노드 개수는 루트 노드 개수나 중간 노드 개수만을 포함한다. 각 높이 당 노드 개수는 (트리 높이+1) 크기를 갖는 nNodesOfEachLevel 배열에 0 번째 높이부터 차례로 저장되며 이 배열의 마지막에는 종단 노드의 개수, 즉 강체의 개수를 저장하여 트리의 병렬적 운행에 용이하도록 하였다 (그림 3-3).

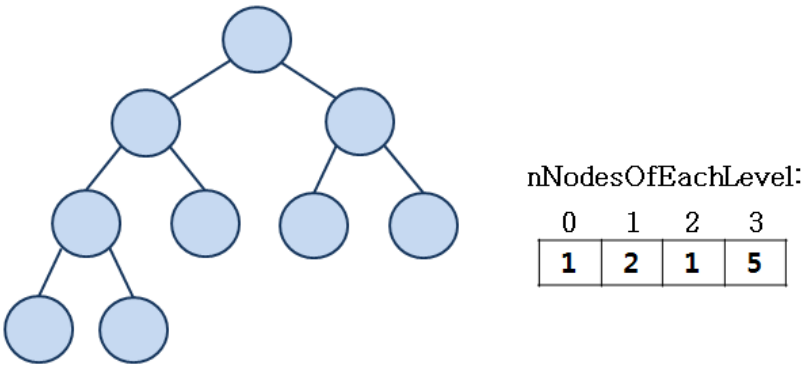


그림 3-3

2. 커널 그리드와 블록의 크기

커널 함수를 호출 시 블록에 들어갈 스레드의 개수는 커널 함수에서 병렬 연산될 어셈블리 트리의 노드들이 트리의 어느 높이에 위치하였는가에 따라 결정된다. 만약 해당 노드들이 n번째 높이의 노드들이라면 총 2^n개의 스레드가 할당된다. 그리드에 들어갈 블록의 개수는 결정된 스레드의 개수에 따라 달라진다. 결정된 총 스레드의 수가 CUDA에서 물리적으로 병렬 처리 되는 스레드들의 단위인 와프(warp)의 크기 32개보다 적다면 그리드 내 블록의 개수는 1개로 하고, 그 블록에 모든 스레드들을 할당한다. 반대로 총 스레드의 수가 와프의 크기보다 크다면 블록 내 스레드의 개수는 와프 크기 32개만큼 할당하고 그리드 내 블록의 개수는 ((총 스레드 수)/(와프 크기)) 만큼 할당한다. 예를 들면 트리의 맨 마지막 높이에 위치한 종단 노드들에 대해 커널 함수를 실행할 때, 마지막 높이를 L, WARP_SIZE를 32라고 한다면 커널 함수에 할당할 그리드의 크기 및 블록의 크기를 결정하는 코드는 (코드 3-5)와 같다.

```
if(2^L < WARP_SIZE) {
    dim3   grid(1, 1, 1);
    dim3   block(2^L, 1, 1);      }
else {
    dim3   grid(2^L/WARP_SIZE, 1, 1);
    dim3   block(WARP_SIZE, 1, 1);      }
```

코드 3-5

3. 동역학 시뮬레이션 커널 구현

커널 함수를 어셈블리 트리의 높이 단위로 실행시키기 위해서는 할당된 스레드들이 어셈블리 트리를 나타내는 cuKinematicTree 구조체 배열에서 각자 연산을 담당한 노드의 위치를 식별할 수 있도록 해주는 2개의 변수를 추가로 두어야 한다.

그 중 첫 번째 변수는 연산할 트리 높이에 위치한 노드들 중 가장 처음 위치하는 노드의 인덱스이다. 커널 함수에 할당된 스레드들은 자신들의 인덱스로 상대적인 위치를 알 수 있을 뿐이기 때문에 cuKinematicTree 구조체 배열에서의 절대적인 위치를 알기 위해서는 첫 번째 스레드가 연산을 담당할 노드의 위치를 알아야 한다.

두 번째 변수는 할당된 스레드 중에서 연산에 참여할 스레드의 개수를 나타낸다. 앞서 설명한 바와 같이 커널 함수가 n번째 높이에 위치한 노드들에 대해 병렬 연산을 수행할 때는 2^n개의 스레드가 할당된다. 만약 시뮬레이션 하는 관절체가 강체의 개수가 2^n개인 직렬 관절체라면 이를 나타내는 어셈블리 트리는 각 높이의 노드 개수가 항상 2^n개인 완전 이진 트리이므로 커널 함수에 할당된 모든 스레드가 연산을 수행하게 된다. 하지만 시뮬레이션 하는 관절체를 완전 이진 트리로 나타낼 수 없는 경우, 트리 각 높이의 노드 개수가 항상 2^n개가 될 수는 없기 때문에 커널 함수에 할당된 스레드 중에서 불필요한 스레드가 발생한다. 때문에 이런 불필요한 스레드들을 식별하는 변수가 필요하다. 예를 들어 (그림 3-4)에서 트리의 두 번째 높이에 위치한 중간노드 3, 4에 대해 커널 연산을 수행하기 위해 총 2^2개의 스레드가 할당된다. 따라서 각 스레드의 인덱스는 0~3이 되는데, 이 인덱스로 cuKinematicTree 구조체 배열 6번 인덱스와 7번 인덱스에 저장된 중간 노드 3, 4에 접근하기 위해서 각 스레드 인덱스에 중간 노드 3의 배열 인덱스인 6을 더해주어야 한다. 하지만 두 번째 높이에 위

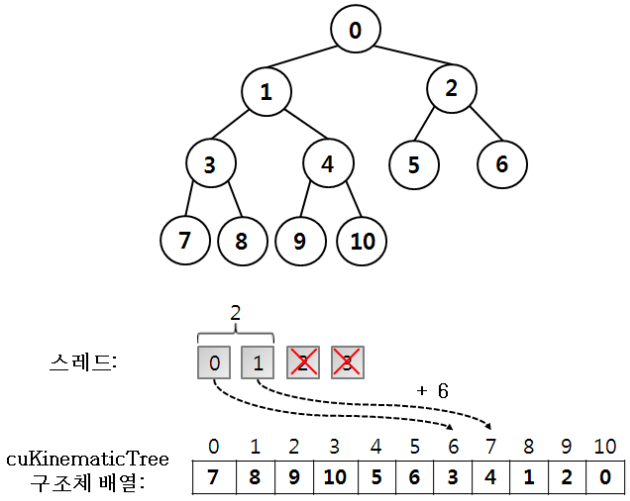


그림 3-4

치한 중간 노드의 개수는 2개이므로 커널에 할당된 4개의 스레드 중 2개의 스레드만 연산을 수행해야 한다. 이를 위해 실제로 연산에 관여할 스레드의 개수는 2개라는 것을 변수에 저장해두고 인덱스가 2보다 작은 스레드들만 연산을 수행하도록 한다.

위의 내용을 바탕으로 어셈블리 트리를 종단 노드에서 루트 노드까지 운행하며 높이 단위로 병렬 처리하는 코드와 루트 노드에서 종단 노드까지 운행하며 높이 단위로 병렬 처리하는 코드를 각각 (코드 3-6)과 (코드 3-7)에 나타내었다. 여기서 level은 어셈블리 트리의 높이, 그리고 nNodes는 어셈블리 트리의 전체 노드 수, WARP_SIZE는 32를 뜻한다. ab는 cuKinematicTree 구조체 배열이고, start 변수는 트리의 각 높이마다 가장 처음 저장돼있는 노드의 인덱스이며, limit는 커널 호출 시 실질적으로 연산을 수행할 스레드의 개수를 의미한다. 그리고 두 코드 모두 2^{level}은 32보다 크다고 가정한다.

```
int start = 0;
int limit = nNodesOfEachLevel[level];
int n = 2^level
dim3 grid(n/WARP_SIZE, 1, 1);
dim3 block(WARP_SIZE, 1, 1);
for(int i=level; i>=0; ) {
    Kernel<<< grid, block >>>( ab, start, limit, …);
    start + = limit;
    limit = nNodesOfEachLevel[--i];
    n = 2^i
    if(n>WARP_SIZE)   grid.x = n/WARP_SIZE;
    else {
        grid.x=1;
        block.x=n;
    }
}
```

코드 3-6

종단 노드에서 루트 노드까지의 커널 함수 실행

```
int start = nNodes-1;
int limit = nNodesOfEachLevel[0];
int n = 2^0

dim3 grid(1, 1, 1);
dim3 block(n, 1, 1);

for(int i=0; i<level; i++) {
    Kernel<<< grid, block >>>( ab, start, limit, …);
    limit = nNodesOfEachLevel[++i]
    start -= limit;
    n = 2^i
    if(n>WARP_SIZE) {
        grid.x=n/WARP_SIZE;
        block.x=WARP_SIZE;
    }
    else   block.x=n;
}
```

코드 3-7

루트 노드에서 종단 노드까지의 커널 함수 실행

여기까지는 트리의 높이 단위 운행에 관한 코드이다. 다시 말하면 트리의 높이 단위 운행은 CPU에 의해 계산이 되고, 노드마다 이루어지는 동역학 시뮬레이션 연산은

커널 함수 내부에서 이루어진다.

```
__global__ void Kernel(cuKinematicTree* ab, cuKinematicTreeHandle* h,
                       cuKinematicTreeRevoluteJoint* j, int start, int limit) {
    int Bx = blockDim.x;
    int By = blockDim.y;
    int i = threadIdx.x+ (blockIdx.x*Bx)+ (threadIdx.y*2*Bx)+ (blockIdx.y*2*Bx*By);
    int nodeIdx = start+ i;
    if(nodeIdx < limit) {
        int childA = ab[nodeIdx].leftChild;
        int childB = ab[nodeIdx].rightChild;
        int PHIdxA = ab[childA].principalHandleIdx;
        int PHIdxB = ab[childB].principalHandleIdx;
        int PHGlobalIdxA = ab[childA].handleIdx[PHIdxA];
        int PHGlobalIdxB = ab[childB].handleIdx[PHIdxB];
        int jointIdx = ab[nodeIdx].jointIdx;

        h[PHGlobalIdxB].f = W*Φ^A_21*f^A_1 - W*Φ^B_11*f^B_2 + γ;
        h[PHGlobalIdxA].f = -j[jointIdx].X * h_[PHGlobalIdxB].f;
        j[jointIdx].qdd = (S^T*V*S)^-1*(Q - S^T*V*(Φ^A_21*f^A_1 - Φ^B_11*f^B_2 + β));
    }
}
```

코드 3-8

치환 단계 커널 함수 내부

커널 함수 내부에서의 동역학 시뮬레이션 연산은 (코드 3-8)과 같이 이루어진다. (코드 3-8)은 치환 단계의 커널 함수를 간략히 나타낸 것이다. 치환 단계는 루트 노드에서 시작해 종단 노드에 이를 때까지 높이 단위로 실행되기 때문에 이 커널 함수는 (코드 3-7)의 코드에 의하여 호출된다. 커널 함수가 호출되면 그 내부에서 스레드들은 먼저 각자의 인덱스를 계산하고 이를 매개변수 start와 더해 cuKinematicTree 배열에서 자신이 맡을 노드의 인덱스 nodeIdx를 구한다. 그리고 nodeIdx의 값이 매개변수 limit의 값보다 작은 경우에만 동역학 시뮬레이션 연산 코드를 수행하며, 크거나 같은 경우에는 수행하지 않는다. nodeIdx가 유효한 값이라면 cuKinematicTree 구조체 배열 ab의 nodeIdx 위치에 접근해 왼쪽 자식 노드와 오른쪽 자식 노드, 이들의 연결 핸들, 관절 인덱스 등의 정보 역시 읽어온다. 그리고 ab의 nodeIdx 위치에 저장된 외력, 역관성 행렬 등의 값을 이용해 왼쪽 자식 노드의 연결 핸들에 작용하는 외력과 오른쪽 자식 노드의 연결 핸들에 작용하는 외력을 구한다. 또한, nodeIdx 위치의 노드가 포함하는 관절의 가속도까지 구하고 나면 치환 단계의 커널 함수가 종료된다. 이 커널 함수는 한 번 실행 시, 관절체 어셈블리 트리의 동일한 한 높이에 위치한 노드들의 관절 가속도와 그 자식 노드들의 핸들에 가해지는 외력이 동시에 계산된다.

4. 데이터 전송의 최소화

CPU 프로그래밍과는 달리 GPU를 이용한 프로그래밍은 CPU와 GPU 간 데이터 전송이라는 비용이 추가적으로 든다. GPU에서 데이터 연산을 처리하기 위해 CPU가 데이터를 넘겨주어야 하고, GPU에서 연산이 다 끝난 후엔 처리된 데이터를 다시 CPU로 넘겨주어야 하기 때문이다. 특히 후자의 경우에 전송 시간이 더 오래 걸린다. 따라

서 최소한 적은 양의 데이터를 전송하도록 해야 전체 시뮬레이션의 수행 시간을 줄일 수가 있다.

본 구현에서 CPU로부터 GPU로 데이터를 넘겨주는 과정은 시뮬레이션 최초에 한 번 발생한다. 이 때는 알고리즘에 필요한 모든 데이터를 모두 GPU에 넘겨주게 된다. 하지만 시뮬레이션 루프가 한 차례 수행되고 나면 필요한 것은 오직 시뮬레이션 디스플레이를 위한 강체 각각의 지역 좌표계와 가상 공간의 전역 좌표계 간 공간 변환 행렬뿐이다. 이 공간 변환 행렬을 구하기 위해서는 강체의 지역 좌표계와 강체 핸들의 지역 좌표계 간 공간 변환 행렬과 강체 핸들의 지역 좌표계와 전역 좌표계 간 공간 변환 행렬, 이렇게 두 가지의 공간 변환 행렬이 필요하다. 하지만 이 정보는 모두 cuKinematicTreeHandle 구조체 배열에 저장이 돼있고, 이 구조체 배열에는 위의 공간 변환 행렬 이외에 시뮬레이션 디스플레이에 불필요한 다른 공간 변환 행렬들이 많이 저장돼있다. 또한 시뮬레이션 디스플레이를 위한 공간 변환 행렬은 강체의 개수만큼 필요할 뿐이지만 cuKinematicTreeHandle 구조체 배열의 크기는 각각의 강체가 가진 핸들 수를 모두 합친 크기와 같기 때문에 필요한 정보의 크기보다 훨씬 크다. 이런 불필요한 데이터 전송을 피하기 위해 objectToWorld 라는 공간 변환 행렬 타입의 배열을 선언하고 그 크기를 강체의 개수만큼 정의하였다. 그리고 cuKinematicTreeHandle 구조체 배열을 모두 CPU로 전송하는 대신 GPU 상에서 강체 각각의 지역 좌표계로부터 전역 좌표계까지의 공간 변환 행렬을 병렬적으로 계산하는 단계를 추가하였다. 추가된 이 단계에서는 그 결과를 objectToWorld 배열에 저장하고, 최종적으로 이 배열만이 CPU로 전송된다.

Ⅳ. GPU 기반 가상 객체들 간 병렬 충돌 검사

A. CPU기반 삼각형 충돌 검사 알고리즘

본 연구는 Tomas Möller가 제안한 CPU기반의 효율적 삼각형 충돌검사 알고리즘 [2]에 기반하여 삼각형 충돌 검사 함수를 구현하였다.

각각 정점 v_0, v_1, v_2의 집합과 정점 u_0, u_1, u_2의 집합을 꼭지점으로 하는 삼각형 T_1과 T_2가 있다고 할 때, 이 두 삼각형에 대해 충돌 검사를 한다고 가정하면 우선 삼각형 T_2를 포함하는 평면의 방정식 P_2을 (수식 4-1)로 구한다.

$$P_2 : N_2 \cdot \chi + d_2$$
$$N_2 = (u_2 - u_0) \times (u_2 - u_0)$$
$$d_2 = -N_2 \cdot u_0$$

수식 4-1

그리고 삼각형 T_1의 꼭지점들로부터 평면 P_2까지의 거리 d_{vi}를 (수식 4-2)로 구할 수 있다. 만약 $d_{vi} \neq 0,\ i = 0,1,2$ 이고 모두 같은 부호를 갖는다면 삼각형 T_1은 평면 P_2의 half space에 포함된다는 것을 의미하므로 충돌 검사 결과를 거짓으로 보고하고 종료한다. 같은 과정을 정점 u_0, u_1, u_2와 삼각형 T_1에 대해서도 반복한 후, 이 검사를

$$d_{v_i} = N_2 \cdot v_i + d_2, \qquad i = 0, 1, 2$$

수식 4-2

통과한 삼각형 쌍에 대해서는 다음과 같은 검사를 실시한다.

두 평면 P_1과 P_2가 충돌하는 직선을 L이라고 할 때 직선 L과 각각의 삼각형 T_1, T_2가 충돌하는 선분을 구한다. 이 두 선분이 겹쳐지면 삼각형 T_1과 T_2는 충돌하는 것이고, 겹치지 않는다면 충돌하지 않는 것이다.

(그림 4-1)[8]에서 삼각형 T_1과 직선 L이 교차하는 선분의 한 쪽 끝점을 구하는 식은 (수식 4-3)과 같다.

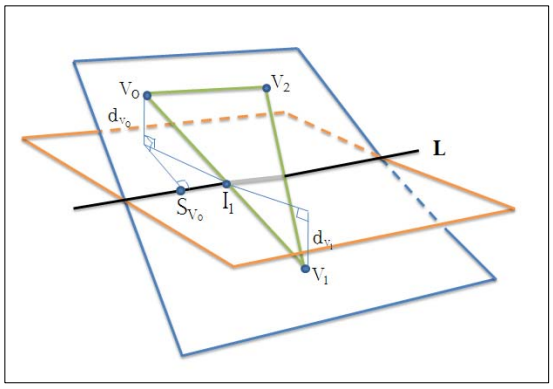


그림 4-1

(수식 4-3)에서 A는 D 위의 어떤 점이다. 위와 비슷한 방법으로 다른 끝점도 구하고, 삼각형 T_2와 직선 L과의 교차점들도 모두 구하면 두 선분의 교차 여부를 알 수 있다.

$$I_1 = S_{v_0} + (S_{v_1} - S_{v_0}) \frac{d_{v_2}}{d_{v_0} - d_{v_1}}$$

$$S_{v_1} = D \cdot (v_1 - A)$$

$$D = N_1 \times N_2$$

수식 4-3

B. CUDA를 이용한 병렬 충돌 검사

CUDA를 이용해 삼각형 충돌 검사를 구현하기 위해서는 위의 효율적 삼각형 충돌 검사 알고리즘을 병렬 처리 해야 한다. 이번 절에서는 병렬 연산을 위한 자료 구조와 커널 함수에 할당할 그리드와 블록의 크기 및 알고리즘에 대해 설명한다.

1. 자료 구조

CUDA를 이용한 구현을 위해서 충돌 검사 알고리즘에 필요한 모든 데이터를 배열화 해야 한다. 우선 두 객체들을 이루는 삼각형들의 정점 정보를 저장하기 위한 배열은 모두 6개가 있다. v0, v1, v2, u0, u1, u2가 그 배열들이다. v0, v1, v2는 객체 O_1을 이루는 삼각형들의 정점 정보를 갖는데, 각각 삼각형의 정점 하나씩을 담당한다. O_1을 이루는 삼각형의 개수가 n_1이라고 했을 때 이 세 배열의 크기는 $(3 \times n_1)$이 된다. 정점 하나의 정보를 저장하기 위해서 3차원의 벡터 값이 필요하기 때문이다. 이와 마찬

가지로 u0, u1, u2는 객체 O_2를 이루는 삼각형들의 정점 정보를 저장하며, O_2를 이루는 삼각형의 개수를 n_2라고 했을 때 이 세 배열의 크기는 $(3 \times n_2)$가 된다.

커널 함수에서 충돌 검사 연산 알고리즘을 마친 후 그 결과값은 $(n_1 \times n_2)$ 행렬에 bool 타입으로 저장된다. 이 결과 행렬이 최종적으로 GPU에서 CPU로 전송되는 데이터이기 때문에 그 크기를 줄이기 위해서 작은 데이터 타입인 bool 타입을 선택하였다.

2. 커널 그리드와 블록의 크기

CUDA프로그래밍을 위해서는 충돌연산에 사용할 스레드의 개수와 스레드 블록의 크기를 정해 주어야 한다. 충돌 검사를 할 두 개의 객체 O_1과 O_2를 이루는 삼각형의 개수를 각각 n_1과 n_2라고 했을 때, 이 알고리즘을 수행할 그리드 내의 총 스레드 개수는 $(n_1 \times n_2)$개가 된다. 각 스레드는 각각 한 쌍의 삼각형에 대해 충돌 검사를 수행한다. 임의의 스레드가 맡는 한 쌍의 삼각형 조합은 그리드 내의 그 스레드 위치에 따라 결정된다. 그리드는 그 구성이 추상적으로 $(n_1 \times n_2)$개의 2차원 행렬 모양으로 스레드가 세로로 n_1개, 가로로 n_2개 정렬 되어 있는 것과 같다. 예를 들어 세로로 i번째, 가로로 j번째에 위치한 스레드가 충돌 검사를 할 삼각형 쌍은 객체 O_1의 i 번째 삼각형과 객체 O_2의 j 번째 삼각형의 조합이 된다.

스레드 블록의 크기는 한 블록 내의 스레드 개수를 의미하는데, 이에 따라 한 그리드 내의 블록 개수도 정해 진다. 스레드 블록의 크기는 2차원 또는 3차원으로 정의할 수 있는데, 이 알고리즘을 위한 스레드 블록의 크기는 2차원으로 정의하였다. 스레드 블록의 세로 크기 b_{hig}와 가로 크기 b_{wid}는 삼각형 개수 n_1과 n_2에 따라 달리 정의되며,

따라서 그리드 내의 블록 개수는 $\left(\frac{n_1}{b_{hig}}\right) \times \left(\frac{n_2}{b_{wid}}\right)$ 가 된다.

3. 병렬 충돌 검사 알고리즘

병렬처리의 호스트에 해당하는 CPU는 먼저 각 객체들을 이루는 삼각형들에 대한 정점의 좌표를 읽어 들여서 호스트 변수에 저장한다. 그리고 커널 함수 호출 전, GPU상에 그 호스트 변수들과 같은 크기의 장치 변수 저장 공간을 할당한 뒤 그 값을 넘겨준다. 또한 충돌 검사 결과를 저장할 $(n_1 \times n_2)$ 크기의 bool타입 행렬 또한 장치 변수로서 GPU메모리에 할당한다. 이 장치 변수들은 커널 함수 호출 시 매개 변수로 전달한다. 커널 함수를 호출하는 코드의 예제는 다음의 (코드 4-1)과 같다. 여기서, b_wid와 b_hig는 각각 블록의 x축 방향 크기와 y축 방향 크기이고, T1과 T2는 각각 첫 번째 객체를 구성하는 삼각형의 개수와 두 번째 객체를 구성하는 삼각형의 개수이다. 또한, d_v0, d_v1, d_v2, d_u0, d_u1, d_u2은 각각 삼각형들의 정점 좌표 값을 갖는 변수들이고, d_odata는 결과값 저장을 위해 할당해둔 변수이다.

```
dim3    threads(b_wid, b_hig );
dim3    grid( T1/b_wid, T2/b_hig );
kernel<<< grid, threads >>>(d_odata, d_v0, d_v1, d_v2, d_u0, d_u1, d_u2);
```

코드 4-1

GPU상의 커널 함수는 삼각형 집합의 충돌 검사를 담당한다. 이 때 삼각형 충돌 검사

알고리즘은 앞서 소개한 Tomas Möller의 알고리즘에 기반을 둔다. CPU의 호스트 프로그램이 커널을 호출하면 이 함수는 우선 매개 변수로 넘겨받은 삼각형들의 정점에 대한 정보를 공유 메모리에 저장하며, 이들은 충돌 검사 과정에서 반복적으로 사용되는 값이며 같은 블록 내의 스레드들이 공통으로 사용하는 값이기도 하므로 공유 메모리에 복사해 둔다.

커널 함수의 삼각형 충돌 검사 알고리즘이 Tomas Möller의 알고리즘에 기반을 두고 있긴 하지만 이를 병렬 처리 방식으로 재 구성 하였기 때문에 CPU에서 시행할 때와는 달리 충돌 검사를 삼각형 조합의 개수 $(n_1 \times n_2)$번 만큼 반복하지 않는다. 대신 스레드의 위치를 가리키는 스레드 인덱스와 블록의 위치를 가리키는 블록 인덱스를 이용해 $(n_1 \times n_2)$ 개의 스레드가 각각 담당해야 할 한 쌍의 삼각형을 지정해준다. 이러한 삼각형 쌍들의 인덱스는 코드 (4-2)와 같이 계산한다.

```
int tri1 = b_hig * blockIdx.y +  threadIdx.y;
int tri2 = b_wid * blockIdx.x +  threadIdx.x;
```

코드 4-2

여기서 blockIdx는 CUDA가 제공하는 변수로서 이 변수를 사용하는 스레드가 속한 블록의 인덱스를 값으로 가지며, 각 충돌검사 스레드는 첫 번째 객체의 tri1 번째 삼각형과 두 번째 객체의 tri2 번째 삼각형 간의 충돌 검사를 맡는다.

각 스레드는 자신이 담당한 삼각형 쌍에 대한 충돌검사가 끝나면, 결과를 저장하기

위해 할당해 둔 장치 행렬 변수에 충돌 검사 결과를 입력한다. 이 때 각 스레드는 그리드에서의 자신의 위치와 동일한 값을 인덱스로 갖는 행렬 상의 저장 공간에 결과를 입력한다. 모든 스레드들이 연산을 완료하면 시스템은 커널 함수를 종료하고 GPU 저장 공간 내의 결과 행렬 변수에 저장된 충돌 검사 값을 CPU로 넘겨준다.

V. 구현 및 결과

A. GPU 기반 관절체 동역학 시뮬레이션 구현 및 결과

본 연구에서 제안한 관절체 병렬 동역학 시뮬레이션은 C와 CUDA를 이용해 구현되었으며, 시뮬레이션 결과를 시각적으로 확인하기 위해 사용한 그래픽스 라이브러리는 OpenGL이다. 구현에 사용한 CPU는 Intel Core2 Duo E6550이며, GPU는 NVIDIA GeForce8800 GTX를 사용하였다.

1. 벤치마킹 시나리오

본 연구에서는 관절체를 구성하는 강체의 개수가 다른 다섯 개의 관절체를 테스트 모델로 준비하였으며 각 관절체가 갖는 강체의 개수는 (표 5-1)에 명세하였다. 이들 관절체에서 임의의 강체를 선택해 임의의 방향으로 힘을 가하는 시뮬레이션을 수행하고 시뮬레이션 수행 시간을 측정하였다. 각 시뮬레이션에서 관절체가 갖는 관절들의 위치는 무작위로 주었다.

테스트 모델	관절체 1	관절체 2	관절체 3	관절체 4	관절체 5
강체 개수	128	256	512	1024	2048

표 5-1

2. 구현 결과

(그림 5-1)은 관절체 1부터 관절체 5까지 각각의 시뮬레이션 디스플레이 결과를 시간의 흐름에 따라 나타낸 것이다. 그림에서 연두색으로 표시된 부분이 힘을 받는 강체가 위치한 곳이다. 시뮬레이션이 루프를 한 번 거치는데 소요 된 평균 수행 시간은 각각 관절체 1의 경우 34.74ms이며 이 중 순수하게 연산에 든 시간은 34.72ms, GPU로부터 CPU로 결과값을 전송하는데 든 시간은 0.02ms 이다. 관절체 2는 총 40.28ms의 수행 시간이 소요되며 이 중 GPU 연산 시간은 40.23ms, 결과값 전송 시간은 0.05ms이다. 관절체 3은 총 수행 시간으로 44.84ms가 소요되고 GPU 연산 시간은 48.80ms, 결과값 전송 시간은 0.04ms이다. 관절체 4의 경우 총 수행 시간 52.15ms 중 GPU 연산 시간에 52.08ms, 결과값 전송 시간에 0.07ms가 소요된다. 마지막으로 관절체 5의 총 수행 시간은 57.94ms 이며 GPU 연산 시간에 57.82ms, 결과값 전송 시간에 0.12ms가 소요된다. 위에 언급한 측정 시간은 모두 총 10번의 수행에 대한 평균 시간이다.

그림 (5-2)는 관절체의 강체 개수 변화에 따른 시뮬레이션의 수행 속도를 그래프로 나타낸 것이다. 그래프의 가로축은 관절체의 강체 개수이고, 세로축은 시뮬레이션 평균 수행 속도이며 단위는 밀리세컨드(millisecond)이다. 가로축이 2배씩 대수(對數)적으로 증가하는데 비해 수행시간은 선형적으로 증가함을 알 수 있다. 이를 통해 Featherstone의 Divide-and-Conquer 알고리즘이 n개의 프로세서에 대해 O(log(n))의 시간 복잡도를 갖는 것을 확인할 수 있다.

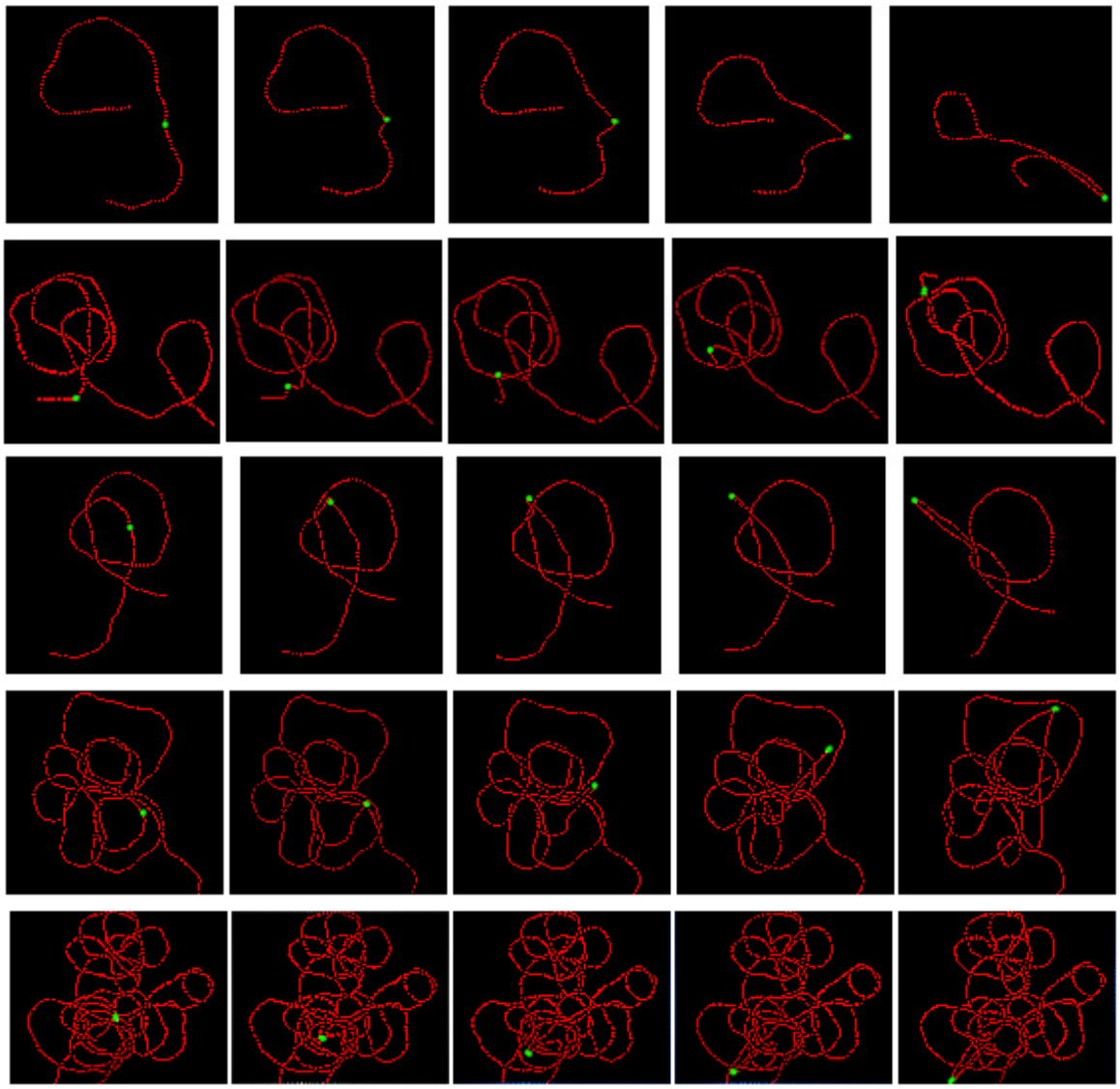


그림 5-1

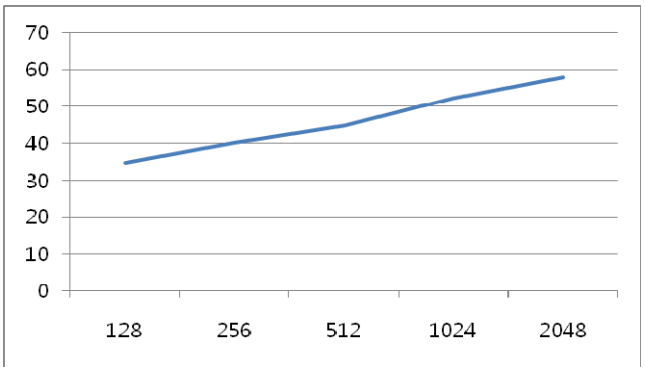


그림 5-2

GPU 기반 병렬 관절체 동역학 시뮬레이션의 수행 속도

가로축은 관절체의 강체 개수, 세로축은 시뮬레이션 수행 시간(millisecond)이다

B. GPU 기반 가상 객체들 간 충돌 검사 구현 및 결과

본 연구의 구현 환경은 위의 관절체 병렬 동역학 시뮬레이션의 구현 환경과 동일하다. 이 시스템은 CPU로 동일한 알고리즘을 실행하였을 때에 비해 수십 배 이상의 성능 향상을 가져왔으며 단일 부동소수점 연산(single floating point operation)만을 이용했을 경우, GPU의 image space상의 충돌 검사 방법들과는 달리, 충돌검사의 결과가 CPU방식과 정확히 일치한다.

1. 벤치마킹 시나리오

본 연구에서는 삼각형의 개수에 따른 성능 변화를 알아보고자 5개의 테스트 모델(sphere, Cylinder, Azucar, Cup, Speen)을 준비하였다. 각 모델이 갖는 삼각형의 개

수는 (표 5-2)에 나와있다. 이 다섯 개의 모델로 세 가지 시나리오를 만들었다. 첫 번째 시나리오는 Cylinder 모델과 Sphere 모델의 충돌 검사로 가능한 삼각형 쌍의 개수는 20만 개가 되어 세 가지 시나리오 중 복잡도가 가장 낮다. 두 번째는 Azucar모델과 Cup모델 간의 충돌 검사인데 이 경우는 가능한 삼각형 쌍의 개수가 4천만 개이고, 마지막 세 번째 시나리오는 Cup 모델과 Spoon 모델 간의 충돌 검사로서 가능한 삼각형 쌍의 개수가 2억 개가 되어 가장 높은 복잡도를 갖는다.

	모델1	모델2
시나리오1	960 (Sphere)	216 (Cylinder)
시나리오2	7580 (Cup)	5250 (Azucar)
시나리오3	7580 (Cup)	26012 (Spoon)

표 5-2

2. 구현 결과

(그림 5-3)은 첫 번째 시나리오의 결과를 시각적으로 나타낸 것이다. 가장 왼쪽 그림은 두 개의 테스트 모델을 스무스 쉐이딩으로 나타낸 것이며, 가운데 그림은 와이어로 나타낸 것이다. 그리고 가장 오른쪽 그림은 두 모델의 삼각형들 중 충돌을 일으키는 삼각형들만을 나타낸 것이다. 왼쪽 그림에서 빨간 부분은 충돌하는 부분을 의미한다. 가운데 그림에서 흰색이나 파란색의 삼각형은 어떤 삼각형과도 충돌하지 않는 삼각형을 나타내며, 빨간색이나 초록색의 삼각형은 하나 이상의 삼각형과 충돌하는 삼각형이다. 이 중 GPU를 사용한 충돌검사에만 소요된 시간은 평균 1.40ms이며 CPU에서 GPU로 매개 변수 값을 전송하는 데 드는 시간은 평균 0.16ms, GPU에서

CPU로 결과값을 전송하는 데 드는 시간은 평균 0.78ms이다.

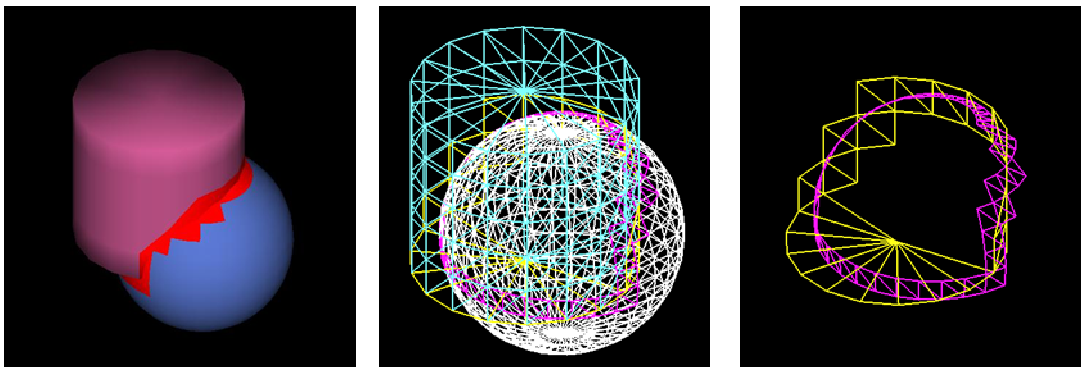


그림 5-3

(그림 5-4)는 두 번째 시나리오의 결과이다. 이 시나리오에서 CPU로부터 GPU로 매개 변수 값을 전송하는 시간은 0.47ms, GPU로부터 CPU로 결과값을 전송하는 시간은 34.57ms, 그리고 GPU로 충돌 검사를 하는 순수 연산 시간은 148.86ms이다.

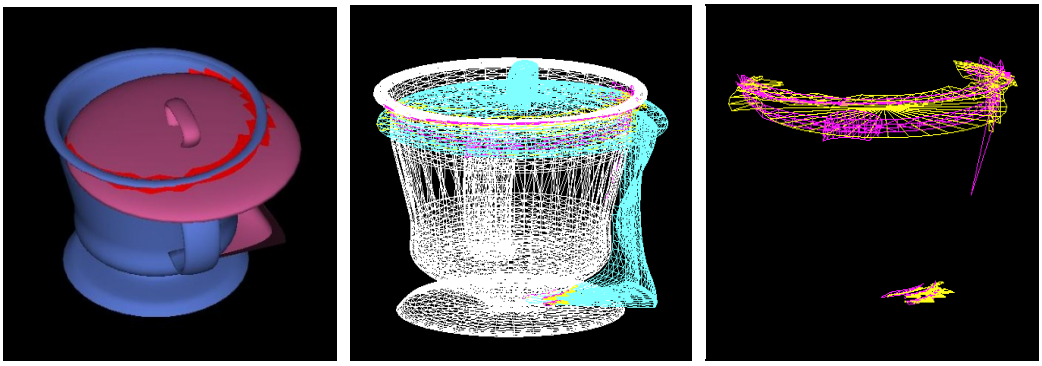


그림 5-4

(그림 5-5)는 세 번째 시나리오의 결과이다. 이 시나리오에서 CPU에서 GPU로의 매개변수 값 전송 시간은 1.03ms, GPU에서 CPU로의 결과값 전송 시간은 171.86ms, 그리고 충돌검사에만 소모된 시간은 909.24ms이다. 삼각형의 개수가 많아질수록 데이터의 양이 많아지므로 CPU와 GPU간 데이터 전송 시간이 길어지고, 병렬 처리를 하는 멀티프로세서의 개수에도 한계가 있으므로 스레드의 개수가 늘어남에 따라 기다리게 되는 스레드들이 발생함으로 인해 GPU 연산 시간도 길어지게 되는 것이다.

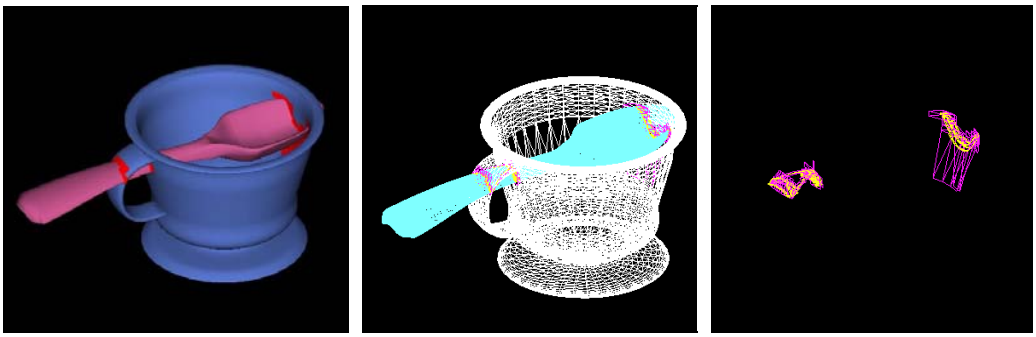


그림 5-5

3. CPU 기반 알고리즘과의 성능 비교

(표 5-3)은 앞 절의 시나리오 1, 2, 3을 CPU버전과 CUDA버전으로 실행하였을 때 각각의 계산 수행 시간을 비교한 것이다. 단위는 millisecond(ms)이며, 각각 10번씩 실행시켜 얻은 수행 시간의 평균값이다. CUDA 버전의 경우엔 순수 GPU 연산 시간뿐 아니라, CPU와 GPU 간 데이터 전송 시간도 총 수행 시간에 포함시켰다. 이 표를 보면 CUDA를 이용한 병렬 알고리즘이 CPU기반 순차 알고리즘에 비해 성능이 월등히 좋다는 것을 알 수 있다. 시나리오 1의 경우 데이터 전송 시간을 합한 수행 소요

	시나리오1	시나리오2	시나리오3
CPU	11.84	2099.5	10377.62
GPU(CUDA)	2.34	183.9	1084.22

표 5-3

시간은 CPU보다 약 5배 빠르다. 또한 시나리오2의 경우엔 GPU버전이 CPU버전 보다 약 11배 빠르고, 시나리오3의 경우엔 GPU버전이 CPU버전 보다 약 10배 빠른 것을 확인할 수 있다. 삼각형의 개수가 늘어남에 따라 병렬 알고리즘의 연산 처리 시간도 늘어나긴 하지만, 본 실험에 사용한 GPU의 경우 128개의 프로세서를 이용해 충돌 검사를 병렬적으로 수행하기 때문에 같은 복잡도의 모델에 대해 순차적으로 충돌 검사를 수행하는 CPU보다 훨씬 빠른 성능을 보여준다.

VI. 결론 및 향후 연구

본 논문에서는 GPGPU 연산에 최적화된 프로그래밍 개발 환경인 CUDA를 이용하여 다수의 강체를 갖는 관절체에 대한 병렬 동역학 시뮬레이션을 구현하였다. 이 구현에서 기반으로 한 Featherstone의 Divide-and-Conquer 알고리즘은 프로세서의 개수가 n개일 때 이론적으로 O(log(n))의 시간 복잡도를 갖지만 이전에 이를 증명해 보인 적이 없었다. 본 연구에서는 GPU의 많은 프로세서를 이용해 이 알고리즘을 병렬적으로 구현함으로써 그 효율성을 실험적으로 보였다.

또한 본 논문에서는 CUDA를 이용한 GPGPU 방식으로 가상 모델 간 충돌 검사를 병렬적으로 구현하였다. 이는 각각의 모델이 갖는 두 개의 삼각형 집합으로부터 가능한 모든 삼각형 쌍에 대해 충돌 검사를 수행하는 경우, CPU에 비해 훨씬 효율적이다. 이 연구를 통해 많은 데이터에 대해 SIMD 방식으로 처리할 수 있는 알고리즘의 경우 GPU로 구현한다면 연산 속도를 줄여 전체적인 시스템 성능을 높일 수 있다는 것을 확인할 수 있었다. 이 CUDA 기반 병렬 충돌 검사 알고리즘을 BVH를 이용한 충돌 검사 등의 다른 여러 가지 충돌 검사 알고리즘에서 삼각형 메쉬 교차 검사 단계에 병목 현상이 발생하는 경우에 적용시키면 그러한 병목 현상을 해소할 수 있을 것이라 본다.

향후 과제로는 첫째, 병렬 동역학 시뮬레이션에 가지가 있는 관절체와 같이 불규칙한 형태의 관절체를 쉽게 구성할 수 있는 인터페이스를 제작하여 많은 강체를 가진 불규칙한 형태의 관절체를 적용시키는 것이다. 불규칙한 형태의 관절체는 직렬 관절

체와는 달리 프로그램 차원에서 자동으로 구성해주지 못하기 때문에 사용자가 프로그램 코드 차원에서 수동으로 구성해주어야 한다. 불규칙한 형태의 관절체를 쉽게 구성할 수 있는 인터페이스를 제작한다면 본 연구를 분자 운동 시뮬레이션, 캐릭터 애니메이션, 머리카락 시뮬레이션, 햅틱스와 같은 다양한 애플리케이션에 적용시킬 수 있을 것이다. 둘째는 CUDA 기반 병렬 동역학 시뮬레이션에서 충돌 검사 부분에 CUDA 기반 병렬 충돌 검사 알고리즘을 적용시키는 것으로 이를 통해 사실감 있고 효율적인 동역학 시뮬레이션을 구현할 수 있을 것이다.

참고 문헌

[1] 김용준, 3D 게임 프로그래밍, 한빛미디어, 2003.

[2] D. Bae, E. Haug, A Recursive Formulation for Constrained Mechanical Systems Dynamics. Part 1: Open-loop systems, Mechanical Structures and Machines, Vol. 15(3), 359-382, 1987.

[3] N. Bandi, C. Sun, D. Agrawal, A. Elabbadi, Hardware acceleration in commercial databases: A case study of spatial operations, In Proceedings of the Thirtieth International Conference on Very Large Data Bases, 1021-1032, 2004

[4] C. Basdogan, College of Engineering Koc University: http://network.ku.edu.tr/~cbasdogan/Tutorials/haptic_tutorial.html

[5] C. Bohn, Kohonen Feature Mapping Through Graphics Hardware, In Proceedings of 3rd International Conference on Computational Intelligence and Neurosciences, 1998.

[6] Y. J. Choi, Y. J. Kim, M. H. Kim, Self-CD: Interactive Self-Collision Detection for Deformable Body Simulation Using GPUs, Asian Simulation Conference, Vol. 3398, 2005.

[7] D. H. Eberly, Game Physics, Morgan Kaufmann, 2003

[8] R. Featherstone, Robot Dynamics Algorithms, Kluwer, 1987.

[9] R. Featherstone, A Divide-and-Conquer Articulated Body Algorithm for

Parallel O(log(n)) Calculation of Rigid Body Dynamics. Part 1: Basic Algorithm, International Journal of Robotics Research, 18(9), 867-875, 1999.

[10] R. Featherstone, A Divide-and-Conquer Articulated Body Algorithm for Parallel O(log(n)) Calculation of Rigid Body Dynamics. Part 2: Trees, Loops, and Accuracy, International Journal of Robotics Research, 18(9), 876-892, 1999.

[11] H. J. Haverkort, Results on Geometric Networks and Data Structures, Ph.D. thesis, Utrecht University, 2004.

[12] J. Hollerbach, A Recursive Lagrangian Formulation of Manipulator Dynamics and a Comparative Study of Dynamics Formulation Complexity, IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-10, No.11, 1980.

[13] G. Kedem, Y. Ishihara, Brute Force Attack on UNIX Passwords with SIMD Computer, In Proceedings of the 8th USENIX Security Symposium, 1999.

[14] T. Möller, A Fast Triangle-Triangle Intersection Test, Journal of Graphics Tools, 2(2), 25-30, 1997.

[15] NVIDIA, NVIDIA CUDA Programming Guide 2.0, 2008

[16] M. Olano, A. Lastra, A Shading Language on Graphics Hardware: The PixelFlow Shading System, In Proceedings of SIGGRAPH, 159-168, 1998.

[17] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, T. Purcell, A Survey of General-Purpose Computation on Graphics Hardware, Computer Graphics Forum, 26(1), 80-113, 2007.

[18] R. Parent, Computer Animation Algorithms and Techniques, Morgan Kaufmann, 203-231, 2002.

[19] M.S. Peercy, M. Olano, J. Airey, P. J. Ungar, Interactive Multi-Pass Programmable Shading, In Proceedings of SIGGRAPH, 425-432, 2000.

[20] K. Proudfoot, W.R. Mark, S. Tzvetkov, P. Hanrahan, A Real-Time Procedural Shading System for Programmable Graphics Hardware, In Proceedings of SIGGRAPH, 159-170, 2001.

[21] S. Redon, N. Gallopo, M. C. Lin, Adaptive Dynamics of Articulated Bodies, In ACM Transactions on Graphics (SIGGRAPH), 24(3), 2005.

[22] J. Rhoades, G. Turk, A. Bell, A. State, U. Neumann, A. Varshney, Real-Time Procedural Textures, In Proceedings of Symposium on Interactive 3D Graphics, 95-100, 1992.

[23] R. Strzodka, M. Droske, M. Rumpf, Fast image registration in DX9 graphics hardware, Journal of Medical Informatics and Technologies, 43-49, 2003

[24] X. Zhang, Y. J. Kim, Interactive collision detection for deformable models using Streaming AABBs, IEEE Transactions on Visualization and Computer Graphics, 13(2), 2007.

ABSTRACT

This dissertation addresses how articulated body dynamics and primitive-level collision detection can be implemented in a massively parallel fashion using NVIDIA's CUDA. Physics-based simulation can realize the real-world phenomena on a computer. The technology has been applied to many fields in computer graphics including computer animations, games, virtual reality, movie special effect, and so on. The two key components of physics-based simulation include collision response based on dynamics simulation and collision detection between virtual objects.

The dynamics simulation, especially forward dynamics simulation, computes the positions, velocities, and accelerations of objects when external forces are applied to them. Articulated bodies are a prevalent form in graphical and robot dynamics simulation since they can represent human and animal characters, industrial robots, molecule models, etc. An articulated body consists of several rigid bodies linked by joints. However, it is quite complex and expensive to solve the underlying motion equation for an articulated body. In order to tackle this problem, parallel or simplification algorithms for articulated body dynamics have been proposed.

On the other hand, collision detection between virtual objects can prevent

objects from penetrating each other. Collision detection should also precede collision response, and this way dynamics simulation can be made realistic. Triangle-level intersection tests are a necessary step in any collision detection methods and often become a computational bottleneck in a collision detection problem.

We have implemented parallel articulated body dynamics simulation based on Roy Featherstone's Divide-and-Conquer algorithm [9, 10] using CUDA. This algorithm is known to have an O(log(n)) time complexity when the number of available parallel processors is n. We have empirically shown in the dissertation that this is the case using CUDA. We also have implemented all pairwise triangle-level intersection tests in a massively parallel fashion using CUDA and significantly improve the run-time performance by more than a factor of 11.