

테스트

서지완 협력 작업
cafe.naver.com/hsgraphics
wannaseoji@naver.com

체크할 사항

경계부분 if(x,y,z > 경계 밖이면)이건 어떨건 어떤 특수한 강력한(if z > 10
continue /*z가 100장이나되는데도*/) 처리로 죽지 않게 만들자.

그 처리로 죽지 않게 되었다?

그렇다면 좋은 뉴스임. 단순히 죽지 않게 되어서가 아니고

다음 단계에서... 경계부분 if 가 참이되는 부분을 찾아서 printf할 수도 있음

if(강력한 처리) printf 좌표값 찍고, 파란색으로 점찍고 continue // 버그 잡기
가능

힌트, 조언

초안작성 과정이니까, 양을 많이 쓰자. (저어하는... 삼가하는...) 생각 많이 하지 않는다.

나중에 수정작업을 정말 많이 한다. 초안에서는 쓰는 재료가 많아야 함.

(생각없이 손에서 막... 토하듯이 쓰면 ok)

원래 요구하는 분량이 있다면 1.2배~1.5배 정도 쓰면 좋음. 수정과정에서 분량이 정리됨.

시간에 따라, 초안 작성하는 사람과, 이를 검토하는 사람. 다른 사람이라는 기분.

서론 흐름

큰 틀 : 우리의 연구가 왜 중요한지

의미가 있다고 주장하는 부분이며, 쓸모없지 않음을 방어한다.

//하고자 했던 것과 **PBD**를 선택한 이유

의료 시뮬레이션/**sw** 가 많이 사용됨

의료 시뮬레이션/**sw** -> 수술시 거동 (움직임) -> 속도/정밀도 균형 -> **pbd**

pbd는 ... 한 방법이며, 위치의 변화를 이용해서 움직임을 재 계산하는 방법이다.

pbd를 구현하는데 고려사항은 속도이고,**openmp**를 사용하여 **CPU**를 통한 병렬처리를 수행할 수 있다.

그냥은 안된다. **pbd**가 병렬화가 잘 안됨 -> 메모리 쓰기, 분량은 지루하지 않게 3문장 정도

1장 : 서론

가상 의료 시뮬레이션 소프트웨어는 인체에 대한 3차원 볼륨 데이터를 이용하여 의료인이 가상 환경에서 실제 환경을 모방한 시뮬레이션을 통해 경험위주의 훈련을 할 수 있도록 돕는다. 이를 통해 경험부족으로 발생하는 의료사고 예방과 더불어 환자가 제공받는 의료품질에 대한 개선을 기대할 수 있다.

//수술이란 피부, 기타의 조직을 외과 기구로 찢거나(절단하거나) 도려내거나(절개하거나) 껴매어 붙이거나(봉합하거나) 하여 병을 낫게 하는 구체적인 외과적 치료방법(의료행위)을 말합니다.

의료 시뮬레이션 소프트웨어에서 의료인은 외과 기구를 통한 절개와 봉합등의 외과적 의료행위를 주로 수행한다. 이러한 과정 속에 수술시 대상의 거동(움직임)을 실제와 같이 표현하기 위해서는 물리엔진 사용이 필수적이다.

//3차원 공간(큰 배열)을 다루는 그래픽스에서 병렬화 문제는 중요하다. 특히 의료영상등에 이용되는 배열은 512x512x300의 볼륨데이터를 다루며, 의료영상을 이용한 가상의료 시뮬레이션 프로그램에서의 엔진은 렌더링과 더불어 연산량이 많다.//너무 일반적이다.

물리엔진은 체인메일과 같은 빠른 연산 속도를 위한 엔진과 정밀도를 위한 **FEM** 방법 등의 엔진이 있다.

(원가 체인메일은 어떻고 **FEM**은 어떻다. 들어가면 좋을 것 같은데..)

우리는 속도와 정밀도의 균형을 이루는 **Position Based Dynamics**를 채택하였다. **PBD**는 외력에 의해 변형된 물체의 움직임을 제약조건에 따라 정점에 대한 직접적인 수정을 가하여 움직임을 제어한다. 위치변화에 따라 실시간으로 연산이 많이 되는 우리의 프로그램(가상의료 시뮬레이션 프로그램)은 성능향상이 필수적이다.

기존의 **PBD**엔진은 **openmp**를 사용하여 **CPU**를 통한 병렬처리가 이루어진다. **CPU**는 **GPU**에 비해 코어수가 적어 순차적인 작업에 강하지만 **CPU**보다 많은 코어를 가지고 있기에 병렬적인 작업에는 **GPU**가 유리하다.

본 연구에서는 **GPU**를 제어할 수 있는 **Nvidia**의 **Cuda C/C++**를 이용한 **GPU** 병렬처리를 통해 **PBD**를 병렬 처리하려 한다. //성능을 향상시킬 수 있다.

그 과정에서 **PBD**를 병렬화 하기위해서는 커널런치를 위한 **CPU**에서 **GPU**로의 메모리복사, **GPU**에서 병렬처리를 통해 얻어진 결과값에 대해 **CPU**로의 메모리복사, 스레드간 연산에 대한 독립성확보 등의 어려움을 해결한다.

2장 : 관련연구 - 일반적 시뮬레이션 기법

기존에 어떤 연구들이 있었는가

PBD 자체에 대한 설명이 들어갈 것이며, 잘 요약하면 된다.

PBD 자체를 상대방은 전혀 모른다고 가정하고, 친절하게 코딩할만 하다고
생각할 정도로 세세하게 설명한다.

(그림)-> `solveConstraint` 하는 그림

idea : PBD에 관한 설명 - 어떻게 돌아가는지, 뭐가 좋은지

식에 대한 설명 - `result` 값에 대한 설명

2.2 관련연구 - 기존 PBD에 대한 설명

PBD란 Position Based Dynamics의 약자로, 기존의 힘 기반 엔진의 **explicit integration scheme**으로 인한 오버슈팅 문제를 해결하며, 제어가능성을 점을 직접적으로 조작함으로써 높은 방법이다.

점을 직접적으로 조작한다는 것은, 점의 위치를 **Constraint**를 **Projection**함으로써 제어한다는 것이다.

Constraint란, 외력이 정점에 작용했을때 이동한 정점이 위치해야하는 안정적인 위치이며 $C(P) = 0$ 을 만족하는 방향으로 정점의 위치를 조정하는 것을 말한다.

일반적인 PBD알고리즘은 다음과 같다.

//본 연구는 다음의 엔진 알고리즘 중, **damping**을 하지 않고 진행하였다. ->결론의 한계 보완점으로 가는 것이 맞음

PBD엔진 알고리즘

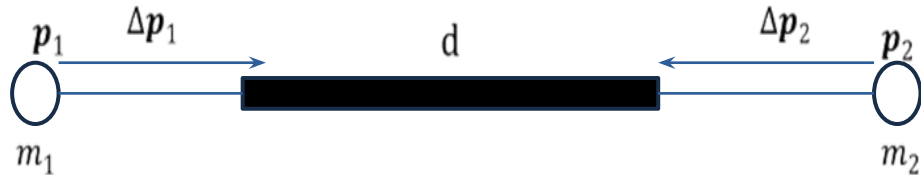
- (1) 정점과 정점이 지니고 있는 속도, 질량, 가속도 값을 초기화 한다. 단 대단히 큰 질량이나 고정된 점에 대해 다루기 쉽게 하기위해 질량의 역수값을 사용한다.
- (2) 중력등의 외력이 작용한다.
- (3) 모든 정점에 대해서 속도를 업데이트 한다.
- (4) 속도 방향으로 정점들의 위치를 수정한다.
- (5) 임의로 정한 몇번의 iteration동안 Constraint projection한다.
- (6) 모든 정점에 대해 수정된 움직임을 반영 시킨다.
- (7) 정점의 속성으로 지닌 속도와 가속도를 업데이트 한다. 이때 속도는 기존 힘 기반 엔진(어떻게 돌아가는지는 모르지만)에서 채택한 방식과 다르게 속도를 정말하게 구하는 layer를 배제하고 정점이 움직이기 전과 움직인 후를 이용하여 구한다

- (1) **forall** vertices i
- (2) initialize $\mathbf{x}_i = \mathbf{x}_i^0, \mathbf{v}_i = \mathbf{v}_i^0, w_i = 1/m_i$
- (3) **endfor**
- (4) **loop**
- (5) **forall** vertices i **do** $\mathbf{v}_i \leftarrow \mathbf{v}_i + \Delta t w_i \mathbf{f}_{ext}(\mathbf{x}_i)$
- (6) dampVelocities($\mathbf{v}_1, \dots, \mathbf{v}_N$)
- (7) **forall** vertices i **do** $\mathbf{p}_i \leftarrow \mathbf{x}_i + \Delta t \mathbf{v}_i$
- (8) **forall** vertices i **do** generateCollisionConstraints($\mathbf{x}_i \rightarrow \mathbf{p}_i$)
- (9) **loop** solverIterations times
- (10) projectConstraints($C_1, \dots, C_{M+M_{coll}}, \mathbf{p}_1, \dots, \mathbf{p}_N$)
- (11) **endloop**
- (12) **forall** vertices i
- (13) $\mathbf{v}_i \leftarrow (\mathbf{p}_i - \mathbf{x}_i) / \Delta t$
- (14) $\mathbf{x}_i \leftarrow \mathbf{p}_i$
- (15) **endfor**
- (16) velocityUpdate($\mathbf{v}_1, \dots, \mathbf{v}_N$)
- (17) **endloop**

Constraint 중 하나로 Distance Constraint가 있다.
Distance Constraint 의 식은 다음과 같다.(식)

Distance Constraint를 계산하기 위해서는 Collision이 발생하기 이전, 두개의 정점이 이루는 Edge간의 안정적인 거리를 미리 확보해 놓아야한다. ($||p_i - p_j|| - d = 0$ 에서 d 에 대한 이야기)

두개의 P_i 와 P_j 는 서로다른 두개의 정점의 위치이다.
외력이 발생함에 따라 고정되지 않은 두 정점이 외력에 따라 움직이면, 두 정점이 유지하던 거리가 멀어지거나, 줄어든다. PBD는 $C(P)=0$ 을 만족하는 방향으로 움직이도록 하기 때문에, 거리가 멀어지면 서로 잡아당기려고 하고, 거리가 가까워지면 서로 밀어내려고 한다. PBD는 델타P를 Constraint의 gradient 방향으로 제한한다.Constraint의 gradient vector의 방향은 Constraint를 만족하는 가장 빠른 방향이다.(<https://www.cs.cmu.edu/~baraff/sigcourse/slide12.pdf>)본 연구는 다음식을 이용한 projection을 통해 얻은 final correction값을 정점에 반영하여 엔진을 구현하였다.



function $C(\mathbf{p}_1, \mathbf{p}_2) = |\mathbf{p}_1 - \mathbf{p}_2| - d$. The derivative with respect to the points are $\nabla_{\mathbf{p}_1} C(\mathbf{p}_1, \mathbf{p}_2) = \mathbf{n}$ and $\nabla_{\mathbf{p}_2} C(\mathbf{p}_1, \mathbf{p}_2) = -\mathbf{n}$ with $\mathbf{n} = \frac{\mathbf{p}_1 - \mathbf{p}_2}{|\mathbf{p}_1 - \mathbf{p}_2|}$. The scaling factor s is, thus, $s = \frac{|\mathbf{p}_1 - \mathbf{p}_2| - d}{w_1 + w_2}$ and the final corrections

$$\Delta \mathbf{p}_1 = - \frac{w_1}{w_1 + w_2} (|\mathbf{p}_1 - \mathbf{p}_2| - d) \frac{\mathbf{p}_1 - \mathbf{p}_2}{|\mathbf{p}_1 - \mathbf{p}_2|} \quad (10)$$

$$\Delta \mathbf{p}_2 = + \frac{w_2}{w_1 + w_2} (|\mathbf{p}_1 - \mathbf{p}_2| - d) \frac{\mathbf{p}_1 - \mathbf{p}_2}{|\mathbf{p}_1 - \mathbf{p}_2|} \quad (11)$$



따라서, 델타P는 Constraint의 그라디언트벡터에 상수를 곱한 형태로 나타낼 수 있다.

만약 모든 점이 가진 질량이 같다면 (7)번식과 같이 모든점에 대한 scaling factor s 가 동일하고, 만약 각각 질량이 다르다면 wi의 영향을 받게 되는데, wi는 mass의 역수이다.

또한 Pi 와 Pj 가 가진 질량 mi mj에 따라서, mi>mj일 경우 Δpi의 값이 작아져 조금 움직이려고 하고, mi<mj일 경우 델타 pj의 값이 작아져 pj가 조금 움직인다.

(10)전 문단에 대한 설명 : P1에 대한 그라디언트 방향과 P2에 대한 그라디언트 방향이 서로 반대임을 나타냄

따라서 scaling factor s가 구해진다.

s가 구해졌으면 (9)식에 따라 final correction 벡터에 대한 식을 구할 수 있다.

masses later). Given \mathbf{p} we want to find a correction $\Delta\mathbf{p}$ such that $C(\mathbf{p} + \Delta\mathbf{p}) = 0$. This equation can be approximated by

$$C(\mathbf{p} + \Delta\mathbf{p}) \approx C(\mathbf{p}) + \nabla_{\mathbf{p}}C(\mathbf{p}) \cdot \Delta\mathbf{p} = 0. \quad (3)$$

Restricting $\Delta\mathbf{p}$ to be in the direction of $\nabla_{\mathbf{p}}C$ means choosing a scalar λ such that

$$\Delta\mathbf{p} = \lambda \nabla_{\mathbf{p}}C(\mathbf{p}). \quad (4)$$

Substituting Eq. (4) into Eq. (3), solving for λ and substituting it back into Eq. (4) yields the final formula for $\Delta\mathbf{p}$

$$\Delta\mathbf{p} = -\frac{C(\mathbf{p})}{|\nabla_{\mathbf{p}}C(\mathbf{p})|^2} \nabla_{\mathbf{p}}C(\mathbf{p}) \quad (5)$$

which is a regular Newton-Raphson step for the iterative solution of the non-linear equation given by a single constraint. For the correction of an individual point \mathbf{p}_i we have

$$\Delta\mathbf{p}_i = -s \nabla_{\mathbf{p}_i}C(\mathbf{p}_1, \dots, \mathbf{p}_n), \quad (6)$$

where the scaling factor

$$s = \frac{C(\mathbf{p}_1, \dots, \mathbf{p}_n)}{\sum_j |\nabla_{\mathbf{p}_j}C(\mathbf{p}_1, \dots, \mathbf{p}_n)|^2} \quad (7)$$

is the same for all points. If the points have individual masses, we weight the corrections $\Delta\mathbf{p}_i$ by the inverse masses $w_i = 1/m_i$. In this case a point with infinite mass, i.e. $w_i = 0$, does not move for example as expected. Now Eq. (4) is replaced by

$$\Delta\mathbf{p}_i = \lambda w_i \nabla_{\mathbf{p}_i}C(\mathbf{p}) \text{ yielding}$$

$$s = \frac{C(\mathbf{p}_1, \dots, \mathbf{p}_n)}{\sum_j w_j |\nabla_{\mathbf{p}_j}C(\mathbf{p}_1, \dots, \mathbf{p}_n)|^2} \quad (8)$$

for the scaling factor and for the final correction

$$\Delta\mathbf{p}_i = -s w_i \nabla_{\mathbf{p}_i}C(\mathbf{p}_1, \dots, \mathbf{p}_n). \quad (9)$$

To give an example, let us consider the distance constraint function $C(\mathbf{p}_1, \mathbf{p}_2) = |\mathbf{p}_1 - \mathbf{p}_2| - d$. The derivative with respect to the points are $\nabla_{\mathbf{p}_1}C(\mathbf{p}_1, \mathbf{p}_2) = \mathbf{n}$ and $\nabla_{\mathbf{p}_2}C(\mathbf{p}_1, \mathbf{p}_2) = -\mathbf{n}$ with $\mathbf{n} = \frac{\mathbf{p}_1 - \mathbf{p}_2}{|\mathbf{p}_1 - \mathbf{p}_2|}$. The scaling factor s is, thus, $s = \frac{|\mathbf{p}_1 - \mathbf{p}_2| - d}{w_1 + w_2}$ and the final corrections

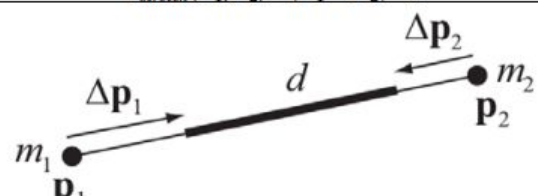
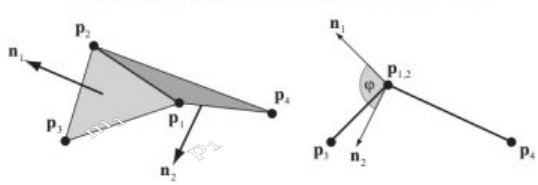
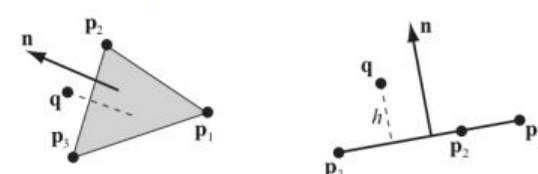
$$\Delta\mathbf{p}_1 = -\frac{w_1}{w_1 + w_2} (|\mathbf{p}_1 - \mathbf{p}_2| - d) \frac{\mathbf{p}_1 - \mathbf{p}_2}{|\mathbf{p}_1 - \mathbf{p}_2|} \quad (10)$$

$$\Delta\mathbf{p}_2 = +\frac{w_2}{w_1 + w_2} (|\mathbf{p}_1 - \mathbf{p}_2| - d) \frac{\mathbf{p}_1 - \mathbf{p}_2}{|\mathbf{p}_1 - \mathbf{p}_2|} \quad (11)$$

(그림판)

- solve constraint에 대한 그림

TABLE 1. COMMON CONSTRAINT TYPES

<p>a. Projection of the stretching constraint: $c_{stretch}(\mathbf{P}_1, \mathbf{P}_2) = \mathbf{P}_1 - \mathbf{P}_2 - d$</p> 	<p>b. The bending constraint: $c_{bend}(\mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3, \mathbf{P}_4) = \arccos(\mathbf{n}_1 \cdot \mathbf{n}_2) - \varphi_0$</p> 
<p>c. The collision constraint: $c_{coll}(\mathbf{q}, \mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3) = (\mathbf{q} - \mathbf{P}_1) \cdot \mathbf{n} - h$</p> 	

- (1) **forall** vertices i
- (2) initialize $\mathbf{x}_i = \mathbf{x}_i^0, \mathbf{v}_i = \mathbf{v}_i^0, w_i = 1/m_i$
- (3) **endfor**
- (4) **loop**
- (5) **forall** vertices i do $\mathbf{v}_i \leftarrow \mathbf{v}_i + \Delta t w_i \mathbf{f}_{ext}(\mathbf{x}_i)$
- (6) dampVelocities($\mathbf{v}_1, \dots, \mathbf{v}_N$)
- (7) **forall** vertices i do $\mathbf{p}_i \leftarrow \mathbf{x}_i + \Delta t \mathbf{v}_i$
- (8) **forall** vertices i do generateCollisionConstraints(\mathbf{x}_i)
- (9) **loop** solverIterations times
- (10) projectConstraints($C_1, \dots, C_{M+M_{coll}}, \mathbf{p}_1, \dots, \mathbf{p}_N$)
- (11) **endloop**
- (12) **forall** vertices i
- (13) $\mathbf{v}_i \leftarrow (\mathbf{p}_i - \mathbf{x}_i) / \Delta t$
- (14) $\mathbf{x}_i \leftarrow \mathbf{p}_i$
- (15) **endfor**
- (16) velocityUpdate($\mathbf{v}_1, \dots, \mathbf{v}_N$)
- (17) **endloop**

3장 : 본론

우리가 무엇을 했나.

병렬화가 중요 문제이며,

읽으면 다 알 정도로 **ConstraintGroups**에 대한 설명을 해야한다.

(그림) -> **edge** 독립성을 확보한다는 것의 의미를 색을 다르게 칠해서 나타낸다던지 집합으로 표시한다던지

idea :

(병렬화 했다) 병렬화 어려워

왜 어렵지?

병렬화 어려운 이유 설명 - 독립성

병렬화 수행 방법

문제 정의: 병렬화의 필요성과 어려움

데이터의 크기가 증가하면서, 연산 시간도 따라 증가한다.

엔진에서 사용되는 시뮬레이션 단위를 **simCell** 6면체로 정의한다. **Simcell**은 인접한 정점 8개로 이루어진다. 전체 볼륨데이터는 $(x-1)*(y-1)*(z-1)$ 개의 **simCell**로 구성된다. **simCell**은 육면체이므로 구조와 연산이 간단한 5개의 사면체들로 분해된다. 사면체의 각 변을 **Edge**라고 한다.

///엣지가 뭔데? 설명 엣지란 두 정점을 이은 구조체이며, 한면에 가르치르지 않고, 6면체를 4면체 5개를 이루도록 만들었을때, 만들어지는 선분이다//

본 연구에서 **Edge**를 결정하는 방법을
예를 들면, **xyz**축 각각 **2**개의 정점이
존재하는 가장 간단한 상황에서 **Edge**의
수는 (그림)과 같이 **18**개이며, 정점의
수를 각 축에 대하여 하나씩
확장할때마다 엣지는 (표)와 같이 크게
증가한다.

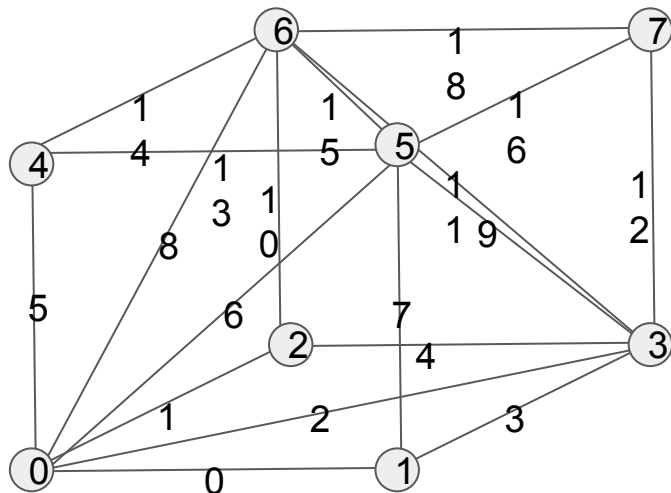
엣지를 만족하려면,

정점이 중복되면 안되며 (ex) (0,0) (1,1)

정점 순서를 고려하지 않는다.

(ex) (0,1) == (1,0)

또한, 그림에서 (1,7)과 같이
tetrahedron를 이루지 않으면 안된다.



A simCell has 18 Edges

~~{(0,0)}{(1,0)}{(2,0)}{(3,0)}{(4,0)}{(5,0)}{(6,0)}{(7,0)}~~

(0,1)~~{(1,1)}{(2,1)}{(3,1)}{(4,1)}{(5,1)}{(6,1)}{(7,1)}~~

(0,2)~~{(1,2)}{(2,2)}{(3,2)}{(4,2)}{(5,2)}{(6,2)}{(7,2)}~~

(0,3)~~{(1,3)}{(2,3)}{(3,3)}{(4,3)}{(5,3)}{(6,3)}{(7,3)}~~

(0,4)~~{(1,4)}{(2,4)}{(3,4)}{(4,4)}{(5,4)}{(6,4)}{(7,4)}~~

(0,5)~~{(1,5)}{(2,5)}{(3,5)}{(4,5)}{(5,5)}{(6,5)}{(7,5)}~~

(0,6)~~{(1,6)}{(2,6)}{(3,6)}{(4,6)}{(5,6)}{(6,6)}{(7,6)}~~

~~{(0,7)}{(1,7)}{(2,7)}~~**(3,7)**~~{(4,7)}{(5,7)}{(6,7)}{(7,7)}~~

edge 2x2x2에서 18개의 이유

One cube has 18 edges

 ~~$(0,0)(1,0)(2,0)(3,0)(4,0)(5,0)(6,0)(7,0)$~~

(0,1)~~(1,1)(2,1)(3,1)(4,1)(5,1)(6,1)(7,1)~~

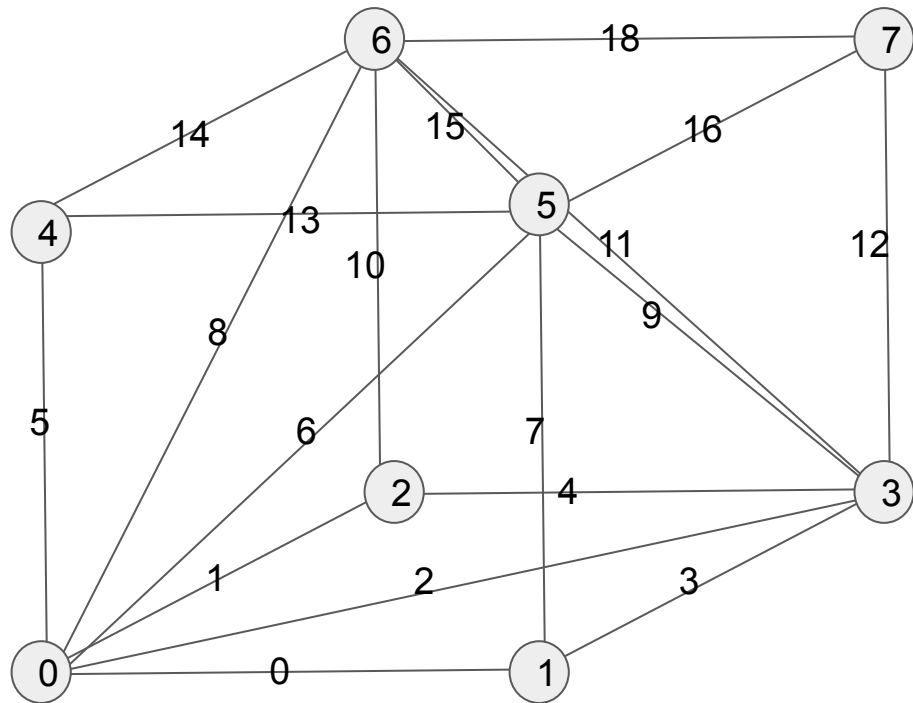
(0,2) ~~(1,2)(2,2)(3,2)(4,2)(5,2)(6,2)(7,2)~~

(0,3)(1,3)(2,3)~~(3,3)(4,3)(5,3)(6,3)(7,3)~~

~~(0,4)(1,4)(2,4)(3,4)(4,4)(5,4)(6,4)(7,4)~~

~~(0,5)(1,5)(2,5)(3,5)(4,5)(5,5)(6,5)(7,5)~~

~~(0,6)~~~~(1,6)~~~~(2,6)~~~~(3,6)~~~~(4,6)~~~~(5,6)~~~~(6,6)~~~~(7,6)~~

~~(0,7)~~~~(1,7)~~~~(2,7)~~**(3,7)**~~(4,7)~~**(5,7)****(6,7)**~~(7,7)~~

(표)와 같이 크게 증가한다.

Distance Constraint를 사용한 projection의 연산단위는 Edge이기 때문에 Edge가 증가함에 따라 계산량이 늘어나며, 계산량이 늘어남에 따라 시스템의 속도가 저하되는 문제점이 생긴다. 연산을 실시간으로 해야 하는 우리 프로그램에서는 속도 향상이 중요하다. 따라서 CPU를 통해 적은수의 코어로 병렬화 처리를 하는 것이 아닌 GPU를 통한 병렬적인 함수호출을 통해, 연산속도를 증가시킬 필요성이 있다.

현대 cpu나 gpu에서는 병렬 연산을 지원하므로, 해당 알고리즘을 병렬화 하고자 한다.

병렬화를 수행하기 위해서는 여러개의 스레드를 생성하여 각 스레드가 동시에 일을 수행한다. 이때, 스레드 간에 데이터 값을 교환하는 연관성이 있다면, 상태를 유지하기 위해 많은 연산이 소모된다. 따라서, 일반적인 병렬화는 각각 스레드에 서로 연관되지 않은 일을 하도록 수정하는 방법이 필요하다. 병렬화과정 전 병렬화를 위한 전처리로 엣지에 대한 독립성을 확보해야한다. 하지만 이 과정은 코딩하기 까다롭고 그것이 실제로 독립적인지 판별해야 한다. // 단점은 5장 6장에서.... 기 때문에 시간이 오래걸린다.

X x Y x Z	number of edges
2 x 2 x 2	18
3 x 3 x 3	90
4 x 4 x 4	252
5 x 5 x 5	540

연관성?-독립성이란? 독립적인 연산

우리 프로젝트에서 연관성이 어디에서 발생하는가. (문제정의. 해결은 다음 페이지)

PBD에서 대부분의 연산시간은 제약조건에 따라 움직임을 수정하는 **Constraint Projection** 에서 발생한다. 이 과정에서 발생하는 시간을 병렬화로 가속화하려 한다.

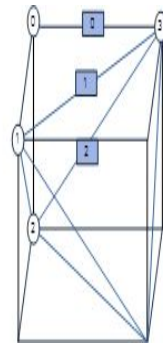
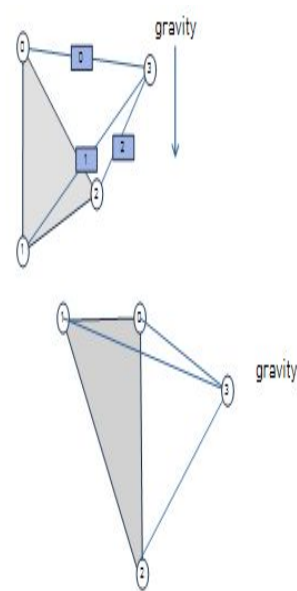
우리 프로젝트에서 병렬화를 진행하기 이전, 앞서 어려움으로 언급했던 스레드간 연산에 대한 독립성을 확보해야하는 부분은 **Constraint** 를 **projection**하는 부분이다. **distance constraint**를 제약조건으로 적용한 우리의 프로젝트의 **projection** 부분에서 **GPU**병렬화를 진행하기 위해 **Edge**에 대한 독립성을 확보할 필요가 있다.

연관되어있다는 것은 엣지들끼리 정점을 공유하면서 병렬화에 문제를 초래하는 상황이며, **simCell**의 정점 하나에 적어도 **3**개 이상의 엣지가 연관되어있다.

엣지 연관성에 대한 예

예를 들어 사면체의 이동을 가정하자. 삼각형면 하나는 벽에 고정되어 있고, 점 하나만 움직인다. 그런 상황에서 점하나에 연결된 **3개의 Edge**가 병렬처리되어 한번에 움직이게 되면, 정점 입장에서 어느값을 먼저 반영해야하는지 모르며, 값을 한번에 반영하지 못하게 원자성을 부여한다면, 병렬처리를 하는 의미가 모호해진다. 엣지 **1, 2, 3** 이 움직임에 따라 정점 **3**이 반영해야하는 움직임을 개발자가 직접 정해주어야 하는 문제점이 생기며 이는 비효율적이다.

독립적인 연산을 보장한다는 것은 방금 설명한 연관되어 있지 않은 엣지들을 찾아 분리하여 병렬적으로 가능하게 함을 의미한다.



<벽에 붙지 않은 3개의 선이 중력을 받아 아래로 떨어지는 상황> -
연관성에 대한 설명

제안 방법 설명

따라서 연구에서 제안하는 방법은 다음과 같다.

병렬처리할수있는 엣지들을 서로다른 그룹으로 따로 묶는과정 특정 정점을 공유하는 엣지들은 서로 다른 그룹에 속해야한다.

1.처리해야하는 제약조건 (constraint)의 수 만큼 반복한다(각각의 **제약조건에 대하여 다음을 처리한다**). 반복횟수는 몇가지 종류의 제약조건을 사용할지에 따라 다르며, distance constraint를 채택한 본 연구에서 **Constraint**의 수는 엣지의 수와 같게된다.

2.각각의 엣지에 포함된 두개의 정점에 대해서 엣지끼리\ 비교한다.(각 그룹 에속한 정점들과 비교한다.)

3-1. 만약 Group에 포함된 정점이 아니라면, 같은 그룹에 포함시킨다.

3-2.Group에 해당 정점이 포함되어있다면, 새로운 그룹을 만든다.

이런과정을 통해 만들어진 엣지 배열은 한 그룹내에 존재하는 모든 엣지는 서로 독립적이므로, 해당 그룹 내부에 대해서 병렬처리가 가능하다.

실험을 통해서 추후 사이즈 몇까지 커지면, cpu<gpu성능인지 판별할것(버그가 고쳐진다면)

제안 방법이 독립적인 이유

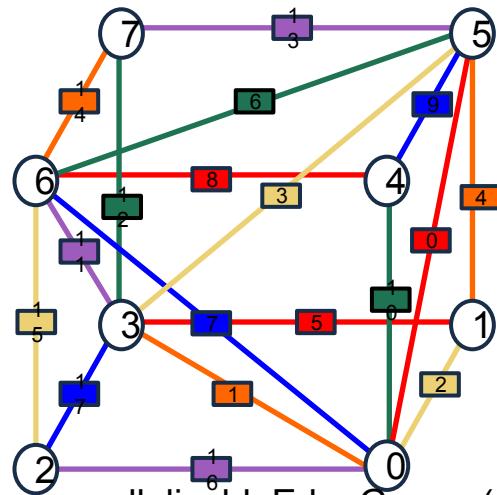
이렇게 **edge**그룹을 만들게 되면, 같은 정점을 공유하는 엣지가 같은 그룹에 속하지 못하도록 막으며, 만일 존재한다면 새로운 엣지그룹을 생성하기 하여 엣지를 저장하므로, 엣지그룹내에서는 같은 정점을 공유하는 엣지가 존재하지 않게된다.

같은 정점을 공유하는 엣지가 없다는 것은 연산에 있어서 서로 영향을 미치지 않음을 나타낸다. 정점을 공유하지 않는 엣지들은 서로 다중 쓰레드환경에서 독립적인 연산이 가능하다. 이는 곧 엣지독립성이 확보된 상태이다.

정점개수가 $2 \times 2 \times 2$ 인 정육면체의 예에서 볼 수 있듯, 다음과 같이 엣지 독립성이 확보되었다.(그림설명과 실제 값)

제안한 방법으로 만들어진 배열의 모양은 다음과 같이 2차원 배열의 형태를 띤다. 하나의 행에 독립적으로 처리 가능한 엣지들이 모여있는 배열을 **parallelizableEdgeGroups(PEG)**로 정의한다. 병렬처리가 가능한 단위인 **PEG**의 한 행이 하나의 **Group**이 된다. 병렬처리 하는 과정에서 다음 그림과 같이 **PEG[n]**번째의 원소가 병렬적으로 처리될 것이다.

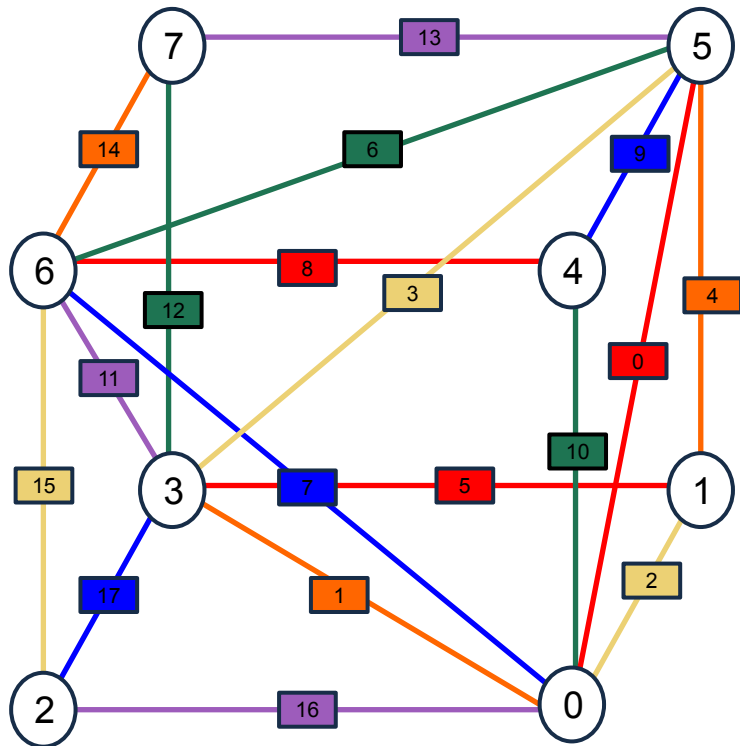
우리의 코드에서는 엣지그룹의 원소수가 많을수록 한번에 처리할 수 있는 엣지 양이 늘어나므로 병렬화의 효율이 올라간다.



parallelizableEdgeGroups(표) $2 \times 2 \times 2$

index	엣지번호		
0	0	5	8
1	1	4	14
2	2	3	15
3	6	10	12
4	7	9	17
5	11	13	16

그림판 (엣지)



m_edges[12]		
값(V):		
이름	값	형식
m_edges	{ size=18 }	std::vector<unsigned int, allocator>
[capacity]	19	unsigned int
[allocator]	allocator	std::allocator<unsigned int>
[0]	{ m_vert=0x009f23b8 {0, 5} st=1.000000000 rl=1.41421354 }	Edge
[1]	{ m_vert=0x009f23c8 {0, 3} st=1.000000000 rl=1.41421354 }	Edge
식(E):		
값(V):		
이름	값	형식
[0]	{ size=3 }	std::vector<unsigned int, allocator>
[capacity]	3	unsigned int
[allocator]	allocator	std::allocator<unsigned int>
[0]	0	unsigned int
[1]	5	unsigned int
[2]	8	unsigned int
[Raw 뷰]	{ _Mypair=allocator }	std::vector<unsigned int, allocator>
[1]	{ size=3 }	std::vector<unsigned int, allocator>
[capacity]	3	unsigned int
[allocator]	allocator	std::allocator<unsigned int>
[0]	1	unsigned int
[1]	4	unsigned int
[2]	14	unsigned int
[Raw 뷰]	{ _Mypair=allocator }	std::vector<unsigned int, allocator>
[2]	{ size=3 }	std::vector<unsigned int, allocator>
[capacity]	3	unsigned int
[allocator]	allocator	std::allocator<unsigned int>
[0]	2	unsigned int
[1]	3	unsigned int
[2]	15	unsigned int
[Raw 뷰]	{ _Mypair=allocator }	std::vector<unsigned int, allocator>
[3]	{ size=3 }	std::vector<unsigned int, allocator>
[capacity]	3	unsigned int
[allocator]	allocator	std::allocator<unsigned int>
[0]	6	unsigned int
[1]	10	unsigned int
[2]	12	unsigned int
[Raw 뷰]	{ _Mypair=allocator }	std::vector<unsigned int, allocator>
[4]	{ size=3 }	std::vector<unsigned int, allocator>
[capacity]	3	unsigned int
[allocator]	allocator	std::allocator<unsigned int>
[0]	7	unsigned int
[1]	9	unsigned int
[2]	17	unsigned int
[Raw 뷰]	{ _Mypair=allocator }	std::vector<unsigned int, allocator>
[5]	{ size=3 }	std::vector<unsigned int, allocator>
[capacity]	3	unsigned int

Edge 독립성을 확보한 배열

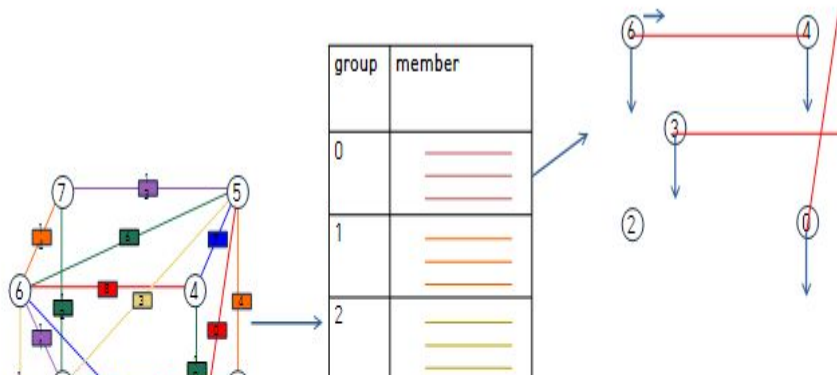
ups에

index	엣지번호		
0	0	5	8
1	1	4	14
2	2	3	15
3	6	10	12
4	7	9	17
5	11	13	16

제안한 방법으로 만들어진 배열의 모양은 다음과 같이 2차원 배열의 형태를 띤다. 하나의 행에 독립적으로 처리 가능한 엣지들이 모여있는 배열을 **parallelizableEdgeGroups**로 정의한다.

parallelizableEdgeGroups의 한 행을 **Group**으로 정의한다. 병렬처리 하는 과정에서 다음 그림과 같이 **parallelizableEdgeGroups[n]**번째가 병렬적으로 처리될 것이다.

(//앞쪽으로 이동하였음)



initParallelizableEdgeGroups (전 initConstraintGroups)메서드

본 연구에서는 엣지독립성 확보를 위한 메서드를 다음과 같이 구현하였다.

initParralizableEdgeGroups 설명 :

1. 두개의 정점 정보를 담고 있는 엣지 구조체가 **input**으로 들어온다.
2. 새로운 엣지그룹을 만들 필요가 있는지 검사한다. 새로운 엣지 그룹을 만드는 경우는, 제일 처음 엣지가 들어오거나, 기존 엣지 그룹에 대해 정점이 연관되어 있는지 판단하여 기존 엣지그룹에 해당 엣지의 정점이 들어있는지 판단하여 들어있다면, 새로운 엣지그룹을 생성해야한다.
3. 새로운 그룹을 만들었다면, 새로만든 그룹에 **input**으로 들어온 엣지의 정점번호를 추가한다.
4. 만약 엣지에 포함된 정점이 기존 엣지그룹에 들어있는 정점과 일치하지 않는다면, 엣지그룹을 새로 생성하지 않고 기존엣지그룹에 추가한다.
5. 1,2,3,4과정을 모든 엣지에 대하여 진행한다.

시스템구성

이제 우리 시스템에서 정점과 엣지를 어떻게 정의하는지 설명한다.

정점 초기화 과정

우리는 정점을 위치, 속도, 가속도, 질량, 질량에 대한 역수 값을 가지고 있는 구조체로 구현하였다. 정점을 초기화 하기위해서는 $x*y*z$ 만큼 반복하면서, 정점인덱스에 접근하여 위치의 초기값으로 좌표를 대입한다. 속도에 대한 초기값은 0이며, 가속도에 대한 초기값은 y 방향으로 -9.81 이다.

고정된 점을 표현하기 위한 방법으로는 질량의 역수값을 0으로 초기화하고, 가속도 또한 0으로 설정 해준다.

질량의 역수가 0이라는 것은 곧 infinite mass를 나타내기 때문에 고정된 점을 나타낼 수 있게 된다.

엣지 초기화 과정

엣지는 두개의 정점과 경도 두 정점사이의 안정한 거리를 나타내는 **restLength** 값을 저장하고 있는 구조체로 구현하였다. 엣지를 초기화 하는 과정은 다음과 같다.

공통엣지를 활용하기 위해서 인접한 셀을 만들 때 정점을 대칭이동하여 만든 배열이 필요하다.

1. **simCell**을 구성하는 모든 사면체에 대하여 반복하면서
2. 사면체에 대한 **edge**를 결정하기 위해 6번 반복한다.
3. 대칭이동하여 만들어낸 정점 배열의 원소를 두개씩 가져와 **edge**로 구성한다.
4. 2번째 엣지를 만들기 전부터 3번의 과정중에서 정점순서가 다르지만 동일한 엣지가 존재하는지 검사해주어야 한다.

4장 :쿠다세부 : 메모리 관리

우리의 프로그램에서는 Cuda C/C++을 사용하여, PEG에 대해 Costraint Projection과정중 Group의 원소를 병렬화 하는 작업을 수행하였다.

//메모리 커널런치 메모리할당 해제 구조 (code없이 말과 그림으로 설명)

서론에서 언급된 메모리관리에 대하여, 다음과 같은 처리가 필요하다.

먼저, 데이터(엣지, 정점, Group)에 대해 한번만 초기화 해주면 되는 gpu 메모리와, cpu메모리와 정보를 교환하고 계속 크기를 바꾸어 할당해줘야하는 gpu메모리 부분으로 나눈다.

엔진의 정점과 엣지(는 한번만하면)를 초기화하는 과정은 projection iteration이 돌기 전 CPU내에서 한번만 수행하면 된다. 또한 gpu내부에서도 정점, 엣지에 대한 memory allocation은 한번만 수행하여 cpu메모리에 저장된 초기값을 받아 할당을 진행하고 그 후에는 CPU와 GPU사이에서의 복사만 이루어진다.

//한번만 초기화해주면 되는 메모리로는 엣지와 정점 배열에 대한 메모리가 있다.

흐름도 , 어디까지 CPU 어디까지 GPU 블록 다이어그램으로 나타낼수 있나?

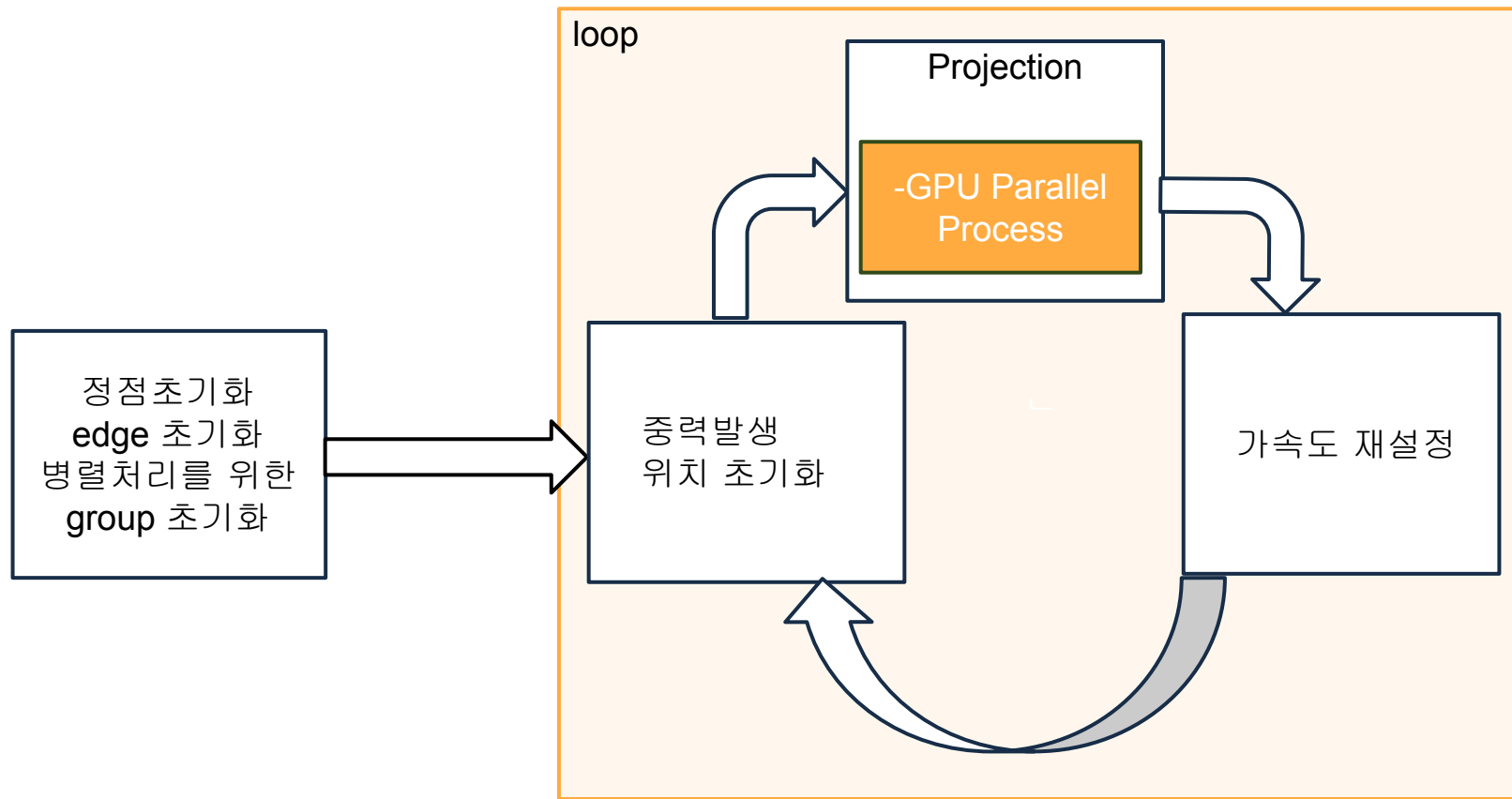
반면, PEG는, 서로다른 크기의 Group을 가지고 있다.

매번 group 변경시마다 CPU에서 GPU로 Memory할당과 해제를 반복하면 비효율적이다. 본 연구는 CPU에서 2차원 배열인 PEG의 행의 크기 만큼 돌면서 가장 큰 Group의 크기만큼만 동적할당하고 GPU공간으로는 복사만 효율적으로 수행하였다. //이 과정은 Projection마다 한번씩 메모리 할당을 해주어야한다. 절개시 동적으로 Group의 크기가 변할수 있기 때문이다. (현재 가진 프로그램만 생각한다면, 한번만 수행해도 괜찮음을 확인함.)->최소 cpu에서도 파라미터로 보낼 parallelizableEdgeGroups들의 n번째 Group에 대한 memory allocation 를 진행해주어야 하며 => 검증결과 cpu에서는 그룹의 최대크기만큼 한번만 잡아놓고 복사, 파라미터로 넘겨받은 Group에 대해서, GPU내에서 Group의 크기만큼 복사가 이루어진다. 이 과정은 Projection마다 PEG의 행의 개수만큼 진행된다.

//gpu 내에서도 기존 그룹에 대한 값을 유지하지 않고 파라미터로 넘어온 parallelizableEdgeGroups의 인자의 크기에 맞춰 새로 memory allocation과 복사를 진행해주어야 한다.

연산이 끝나면, gpu내부에 존재하는 정점에 값을 반영한다. 사용하는 물리 시스템의 종류에 따라 속도의 update가 cpu에서 일어날 수도 있다. 그렇다면 이 경우 gpu에 있는 값들을 cpu로 복사해주어야한다.

있지만 지금은 텍스트에 집중



4장 : 쿠다세부 : thread 생성과 kernel launch

kernel launch를 위한 thread생성은 다음과 같은 과정으로 이루어진다.

cpu로부터 받아온 엣지 그룹의 인자의 개수가 최대 생성가능한 스레드 수 보다 작은경우, 스레드를 엣지그룹의 인자의 개수만큼 생성하여 커널런치 해주면 된다.

//kernel<<< >>>에 들어가는 파라미터는 `<numBlocks, threadsPerBlock>`이다

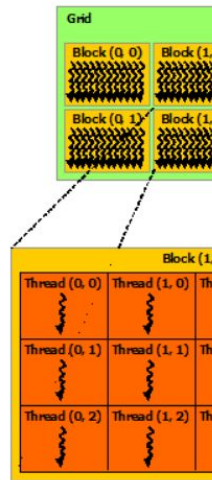
<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html> 출처

각 edge constraint에 대해 thread가 생성되어 병렬 처리된다. 만약, 어떤 그룹에 속한 constraint의 수가 최대 생성가능한 스레드 수 보다 큰 경우, 다음과 같은 처리가 필요하다.(최대로 생성 가능한 스레드 수가512라고 가정하였을 경우)

블록의 수는 다음과 같이 정해진다. $\text{numBlocks} = (\text{numOfConstraints} + 511) / 512;$

블록당 생성될 스레드 수는 다음과 같다. $\text{threadsPerBlock} = 512;$

커널함수의 파라미터는 gpu메모리에 복사된 Group과, gpu메모리에 복사된 Edge 와 gpu메모리에 복사된 정점이다.



4장 : 쿠다세부 : kernel 호출 스택

커널함수가 불리면 스레드의 인덱스에 따라 해야 할 일을 따로 지정해주어야 한다. 인자로 전달된 **Group**의 인덱스를 정해주고(자세하게) 외력에 의해 움직인 정점에 대해서 **distance constraint**를 적용하여 (왜?? 무엇을 위해서?=>). 외력에 의해 변형된 좌표를 수정하기 위한 **constraint projection** 해준다

projection에서 파라미터로 전달되는 것은 **PEG**의 **group**번호에 대한 배열이며, 이는 각 스레드가 처리해야 할 엣지번호를 의미한다. 그리고... 엣지구조체, 정점 구조체이다.

Solve_DistanceConstraint함수 설명

Solve_DistanceConstraint함수는 앞서 언급한 **Distance Constraint**의 식을 코드로 나타낸다. **input**은 두 정점 구조체의 정보이며, **correction**값을 구해 기존 정점정보에 이동량을 반영한다.

Solve_DistanceConstraint 메서드 설명

본 연구에서 점의 위치를 직접적으로 수정하기 위해 사용한 Distance Constraint는 다음과 같이 구현된다.

1. 두 점의 `inversemass` 값을 이용해 `weight sum`을 구한다.
2. 이때 `weight sum` 이 0일경우(`infinite mass`이므로) `false` 를 반환하여 연산을 시행하지 않는다.
3. 두 점을 이용하여 방향벡터를 구한다.
4. 방향벡터의 `norm`을 구한다
5. 방향벡터를 `norm`으로 나누어 길이가 1인 방향벡터로 만든다.
6. 구해진 방향벡터의 크기에 두 점의 안정한 거리를 뺀 값을 `weight sum`으로 나눈 값에 경도(`stiffness`)와 길이가 1이 된 방향벡터를 곱한다.
7. 결과로 나온 값에 두 점점 각각의 `inverse Mass`값을 가중치로 하여 `final correction` 값을 구한다.
8. `true`를 반환하여 기존 좌표값을 갱신한다.

이렇게 이동량을 반영한 정점은 다시 GPU메모리로 반영시키고, GPU메모리에서 CPU로 복사하는 과정이 필요하다.

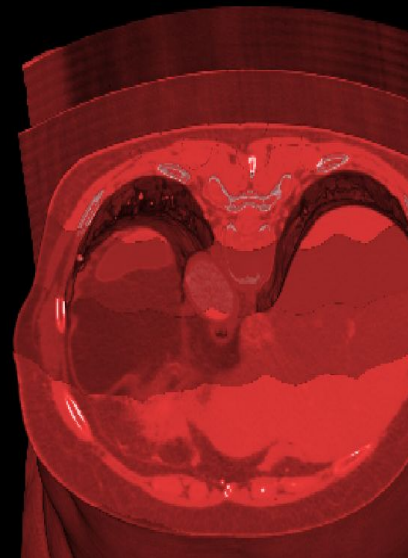
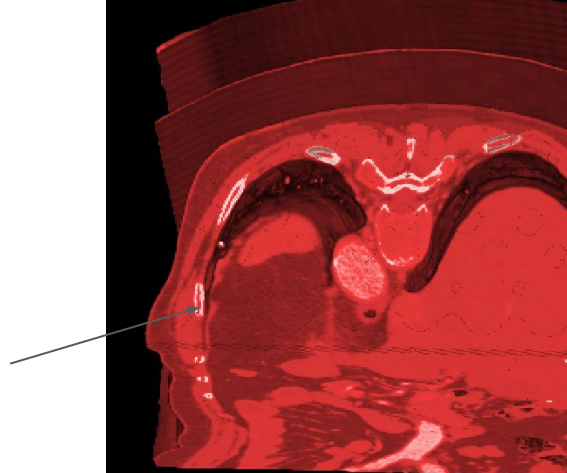
CPU로 연산된 정점이 돌아오게 되면 한번의 iteration이 끝났다. 이 과정을 모든 엣지가 안정한 상태를 유지할 때까지 반복한다. **안정한 상태라는 것은 $C(P)=0$ 에 가까운 상태이다.**

5장 : 실험

렌더링 결과 //refactoring 먼저 -> 리팩토링 완료

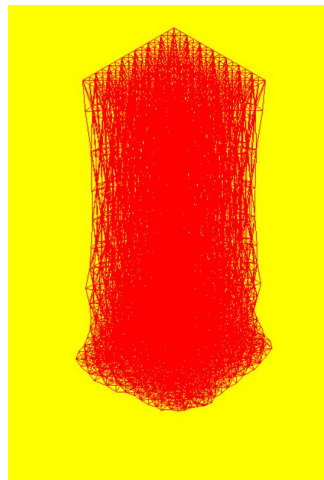
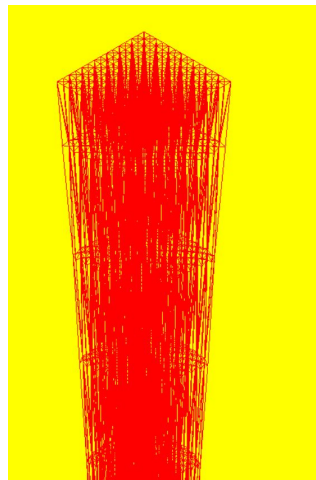
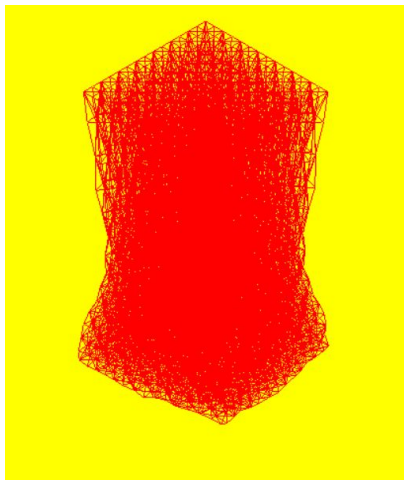
openmp와의 비교 및 역전이 일어나는 구간(parameter study)

렌더링을 큰범위로 보이도록 해서 실험 다시 해야할 것

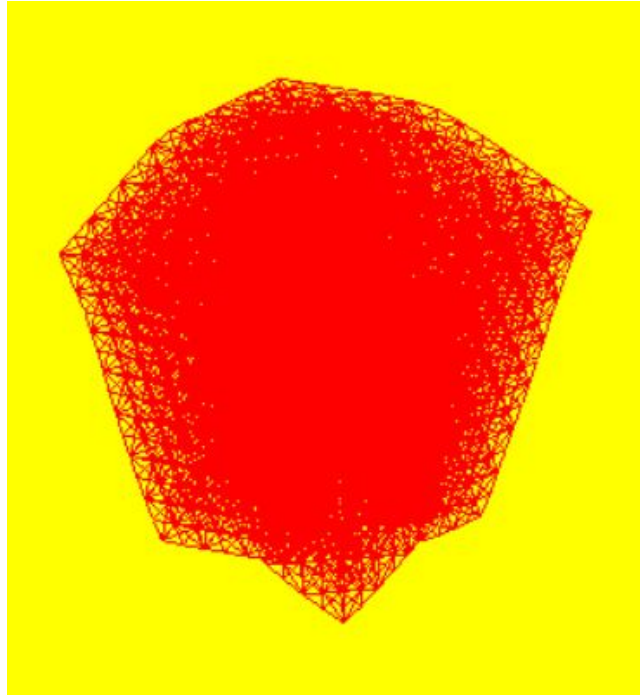
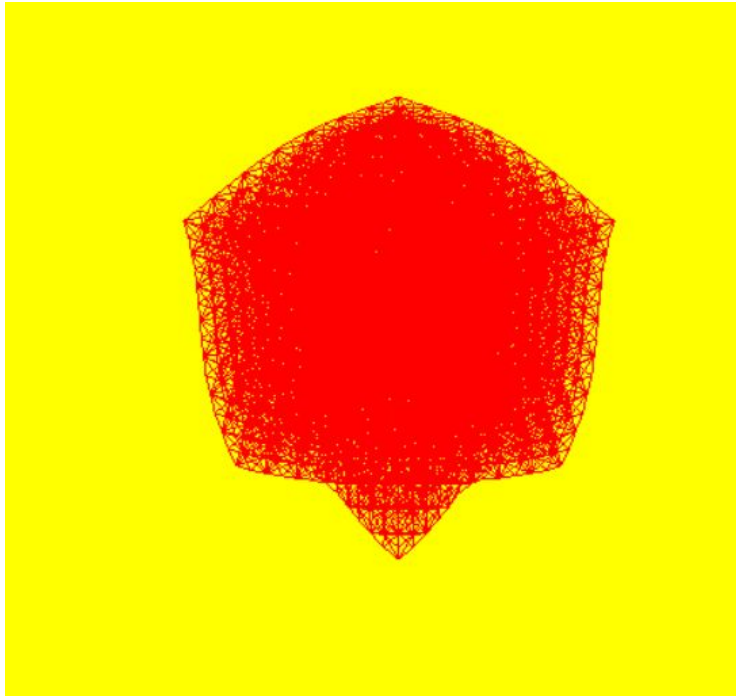


실험 1

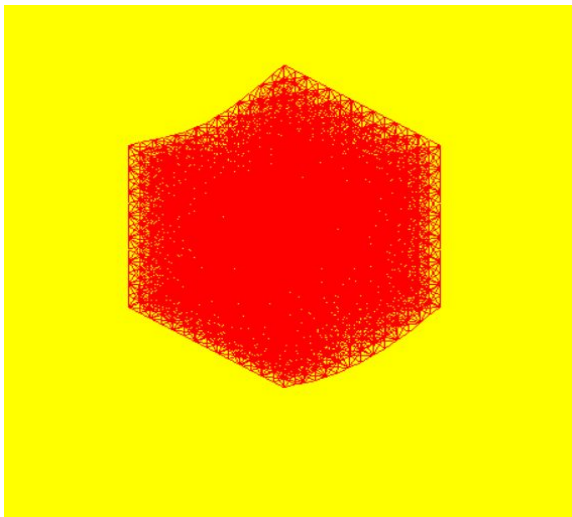
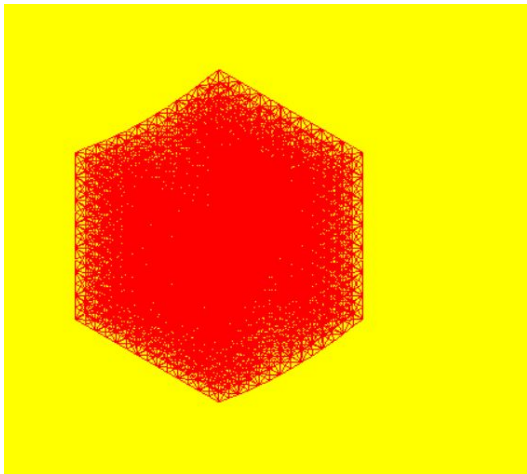
z 축에 대해서 z 의 최댓값에 대한 평면을
고정시키고, **stiffness** 를 0.01로
주었을때에 대한 실험



실험2

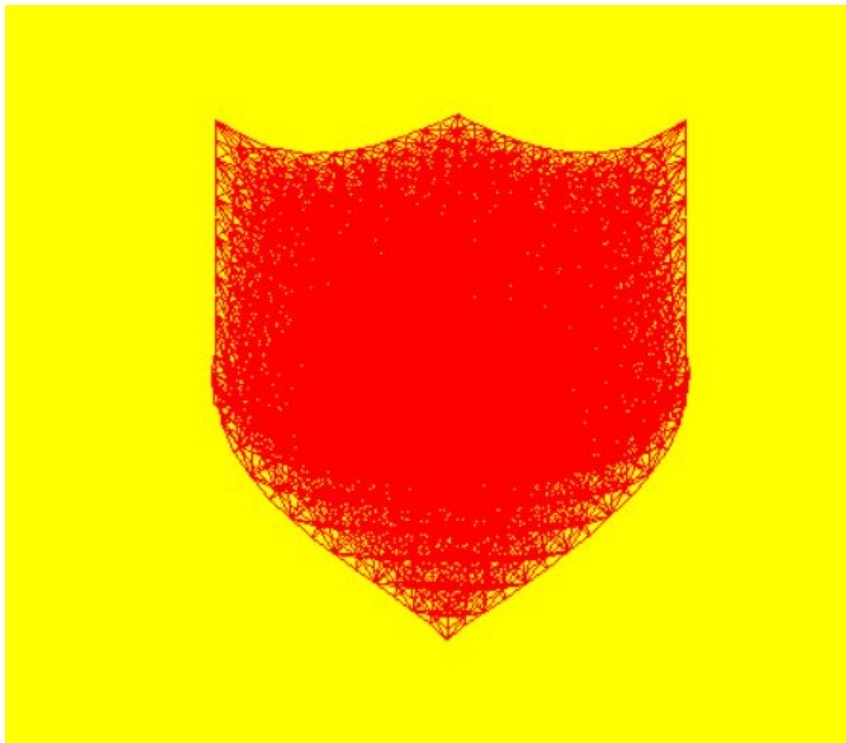


실험 3



```
if (vertices[index].pz ==  
7.0 &&  
vertices[index].px==7.0)
```

실험 4



```
if ((vertices[index].px == 0.0 ||  
vertices[index].pz ==  
0.0)&&(vertices[index].px == 14.0 ||  
vertices[index].pz == 14.0))
```

실험 5

openmp 공부 후, iteration 1회당, 속도 비교..?

6장 :결론

보완점, limitation

보완점 init시에도 병렬화가 필요하다. edge 만들기등

연산의 모든 과정이 gpu안에서 돌고, (projection iteration의 수 까지 고려해서) 필요한 시점에 cpu로 한번만 넘겨준다면 아름다운 결과가 있을것이다.

limitation cuda내부에서 메모리문제를 잘 해결하지 못한 것 같다.

//또한 속도,가속도에 대한 업데이트가 cpu내에서 이루어지므로 iteration의 한 시행마다 cpu에서 gpu로 값을 복사해야하는 오버헤드가 있다. -> 6장 연구의 한계, 향후 연구

참고문헌

[Müller 2007] M. Müller, B. Heidelberger, M. Hennix and J. Ratcliff, "Position based dynamics," *Journal of Visual Communication and Image Representation*, Vol 18, No 2, pp. 109-118, 2007.

10.24

문서관점 - 워드로 평범한 형식으로 10pt 복붙으로 분량이 어느정도 나오는지
13page정도 나오면 괜찮음.

실험관점 - 캡스톤에 붙여서 볼륨데이터가 출력출력해야함 rendering 바꾼
코드에 붙이기 위해서 어떻게 붙일지를 정확히 알아와라.