석사학위논문

효율적인 병렬 리샘플링을 이용한 대용량 볼륨 데이터의 실시간 변형체 가시화 방법

2024년

한 성 대 학 교 대 학 원

컴 퓨 터 공 학 과

컴 퓨 터 공 학 전 공

박 채 림

석 사 학 위 논 문
지도교수 계희원

효율적인 병렬 리샘플링을 이용한 대용량 볼륨 데이터의 실시간 변형체 가시화 방법

Real-time Visualization of Large Deformable Volume Data using Enhanced Parallel Resampling

2024년 6월 일

한 성 대 학 교 대 학 원

컴 퓨 터 공 학 과

컴 퓨 터 공 학 전 공

박 채 림

석 사 학 위 논 문
지도교수 계희원

효율적인 병렬 리샘플링을 이용한 대용량 볼륨 데이터의 실시간 변형체 가시화 방법

Real-time Visualization of Large Deformable Volume Data using Enhanced Parallel Resampling

위 논문을 공학 석사학위 논문으로 제출함

2024년 6월 일

한 성 대 학 교 대 학 원

컴 퓨 터 공 학 과

컴 퓨 터 공 학 전 공

박 채 림

박채림의 공학 석사학위 논문을 인준함

2024년 6월 일

심사위원장 김진모 (인)

심 사 위 원 계희원 (인)

심 사 위 원 허준영 (인)

국 문 초 록

효율적인 병렬 리샘플링을 이용한 대용량 볼륨 데이터의 실시간 변형체 가시화 방법

한 성 대 학 교 대 학 원
컴 퓨 터 공 학 과
컴 퓨 터 공 학 전 공
박 채 림

실제 의료 시술에 앞서 가상 의료 시술을 활용한 계획 수립은 의료 품질 개선에 중요한 역할을 한다. 가상 의료 시술이 중요하고 필수적인 기술로 인식되는 배경에는 실제 인체 데이터를 이용한 정밀한 변형 시뮬레이션이 자리하고 있으며, 절개 및 봉합과 같은 복잡한 연산을 수반하는 의료 절차에서 볼륨 기반 변형체 모델링의 적합성이 두드러진다. 과거에는 볼륨 데이터의 방대한 크기로 인해 변형체 계산에 상당한 시간이 소요되었고, 이는 실제 응용의 제약으로 이어졌다. 그러나, 최근 GPU와 여러 알고리즘(체인메일, 매스 스프링)의 발전으로 처리 속도와 효율성이 크게 향상되었다.

본 논문에서는 대규모 볼륨 데이터를 실시간으로 처리하면서, 고품질의 가시화 결과를 얻을 수 있도록 현대의 변형 모델링에서 시간이 많이 소요되는 텍스처 샘플링과 역행렬 계산 과정을 개선하였다. 이후, 블록 기반 가속화 방법을 도입하여 가시화되지 않는 불필요한 영역의 리샘플링을 최소화하였다. 마지막으로, 공유 메모리를 활용하여 추가적인 속도 향상을 이루었다. 본 논

문은 제안 방법들을 통해 화질 저하 없이 처리 속도를 향상시키고, 의료 시뮬레이션 분야에서의 응용 가능성을 확장하였다. 이는 실제 수술에 필요한 빠르고 정밀한 의료 시뮬레이션 시스템 구축으로 이어질 수 있다는 점에서 큰 의의를 갖는다.

【주요어】 의료 시뮬레이션, GPU 병렬 컴퓨팅, 대용량 볼륨 데이터, 공유 메모리, 빈공간 도약

목 차

표 목 차

그 림 목 차

Ⅰ. 서론

1.1 연구 배경 및 목적

의료 데이터의 변형 및 애니메이션은 교육용 가상 수술 시뮬레이션, 의료 계획 수립, 그리고 환자 설명 자료로 유용하다. 데이터 변형 및 가시화 기술은 의료 분야뿐 아니라 전통적인 컴퓨터 그래픽스 연구에서도 중요한 주제로 자리 잡고 있으며, 이에 따라 다양한 관련 연구들이 이루어져 왔다.

데이터 변형 및 가시화 기술 중 표면 기반 방식은 연산이 표면 데이터에 대해서만 적용되므로 연산량이 적고 처리 속도가 빠르다는 장점이 있지만, 절개나 봉합과 같은 위상 변화에는 적용하기 어렵다는 단점이 있다(Nienhuys, H. W., 2001). 의료 시뮬레이션은 절개, 봉합 등의 수술을 포함하기 때문에 위상 변화와 무관하게 적용할 수 있는 볼륨 기반 변형 방식이 적절하다. 그러나 볼륨 데이터의 큰 크기로 인해 변형 계산에 상당한 시간이 소요되므로, 효율적인 처리 방법이 필수적이다.

볼륨 변형 및 가시화는 크게 두 가지 방법으로 구분된다. 첫 번째 방법은 용량이 큰 볼륨 데이터를 변경하지 않고 대신 관찰 경로를 복잡하게 뒤틀어 결과적으로 변형된 영상을 생성하는 것이고, 두 번째 방법은 변형된 볼륨 데이터를 새롭게 생성하고 일반적인 방식으로 가시화하는 것이다.

첫 번째 방법에서는 원본 데이터의 각 광선에 대해 반복적으로 샘플링을 수행한다. 변형된 시야 광선에서 각 샘플링 위치를 계산하기 위해 사용자 변형의 역변환을 계산해야 하는데, 많은 수의 역변환 계산으로 인해 어려움을 겪을 수 있다. 역변환이 존재하지 않는 절개 영역과 같은 예외를 처리하기 어렵고, 뉴턴-랩슨 또는 경사 하강법과 같은 반복적 수치 해법을 사용함으로써 프로세스가 느리고 오차가 발생할 가능성이 있다(Rößler, F., 2008). 두 번째 방법은 원본 볼륨 데이터에서 직접 변형된 볼륨 데이터를 생성한다. 상대적으로 처리 속도가 빠른 포인트 기반의 포워드 매핑을 적용하면 일반적으로 구멍이나 중첩 문제가 발생하여 출력 화질이 저하된다. 따라서 이 문제를 해결

하기 위해 래스터화를 사용하는 사면체 기반의 포워드 매핑을 사용할 수 있다. 이 기술은 계산 시간이 많이 소요되어 이전에는 실현 가능성이 낮았지만, 최근 GPU의 발전으로 가능해졌다. 이 방법은 병렬 리샘플링으로 불리며 GPU 병렬화를 사용하는 대표적인 사면체 기반 포워드 매핑 방식으로 볼륨 변형에 사용된다.

병렬 리샘플링은 세 단계로 수행된다 : (1) 물리 시뮬레이션에서 제어점들의 정확한 좌표 이동을 계산하고, (2) 이 좌표 이동을 바탕으로 내부 공간을 보간하여 새로운 볼륨 데이터를 생성한다. (3) 가시화 단계에서 새로운 볼륨 데이터를 이용해 다양한 볼륨 가시화 방법(Levoy, M., 1988)(Lacroute, P., 1994)을 적용한다. 이 접근 방법은 역변환 없이 새로운 볼륨 데이터를 생성하므로 자연스러운 결과 영상을 생성한다. 본 연구는 미리 계산된 좌표 이동을 기반으로 새로운 볼륨 데이터를 생성하는 과정을 다룬다. 즉, 첫 번째 단계에서 체인메일이나 물리 기반 변형(PBD)과 같은 변형체 연산(체인메일(Gibson, S.F 1997), 매스 스프링(Liu, T., 2013), 위치 기반 동역학(Berndt, I., 2017) 등) 방법이나 미리 정해진 수식에 따른 애니메이션을 통해 각 제어점의 좌표는 계산되어 있다고 가정한다. 이후 생성된 볼륨 데이터를 기존 연구와 동일한 방법으로 가시화한다.

기본적인 병렬 리샘플링 방법(Gascon, J. 2013)(Aguilera, A.R., 2015)은 많은 사면체를 처리하는 데 있어 성능의 한계를 드러내었다. 이에 본 연구는 GPU 기반의 병렬 리샘플링 기법을 바탕으로 성능을 향상시키기 위해 네 가지 방법을 제안한다. 첫 번째는 변형되지 않은 공간의 좌표를 보간하여 출력 복셀에서 텍스처 샘플링을 수행함으로써 필요한 샘플링 횟수를 줄이고 실행 속도를 개선하는 방법이다. 두 번째는 역행렬 계산에서 발생하는 오류를 최소화하기 위해 작은 행렬 요소로 유지하는 방법이다. 세 번째 방법은 불필요한 영역을 효과적으로 제거하고 화질 손상을 방지하기 위해 중첩된 블록 개념을 도입하는 것이다. 마지막으로 GPU의 병렬 처리를 고도화하고 공유 메모리를 활용하여 성능을 더욱 향상시킬 수 있는 방법이다.

본 연구는 볼륨 변형체 가시화 파이프라인의 중간 단계에서 호환성 있게 적용되므로, 다양한 변형 모델을 이용하고 출력 결과를 여러 렌더링 알고리즘

에 적용할 수 있다.

이 논문의 기여는 다음과 같다:
- 대규모 데이터에 대한 효율적인 볼륨 변형 계산 방법 제안
- 고속 변형 가능한 객체 생성 알고리즘을 통해 사용자 지연 시간 개선
- 실시간으로 볼륨 변형 데이터를 생성하는 방법 제안
- 중첩된 블록을 사용하여 화질 손상 없이 불필요한 영역 제거 방법 제안
- 공유 메모리를 활용한 GPU 성능 향상 및 분석을 통한 병렬 리샘플링 개선
- 다양한 변형 및 가시화 방법에 효과적으로 적용 가능한 방법 제안

1.2 논문의 구성

본 논문의 구성은 다음과 같다. 2장에서는 볼륨 변형체, 병렬 리샘플링 그리고 GPU 병렬화에 대해 설명한다. 3장에서는 텍스처 샘플링의 효율성을 높이고 역행렬 계산의 정확성을 개선하는 방법을 중점적으로 다룬다. 4장에서는 블록을 통한 가속화 방법을 제안한다. 5장에서는 공유 메모리를 활용한 처리 속도 향상 방법을 제안한다. 6장에서는 앞선 3장, 4장, 그리고 5장에서 제안한 방법들에 대한 실험 결과를 보인다. 마지막으로 7장에서는 본 논문의 연구를 결론 맺는다.

II. 관련 연구

2.1 볼륨 변형체

물리 기반의 시뮬레이션인 체인메일(Rodríguez, A., 2016)(Bartelheimer, K., 2017), 매스 스프링(Georgii, J., 2005)(Zhang, Y., 2010)(Fortmeier, D., 2013)(Zhang, Z., 2020), 위치 기반 동역학(Bender, J., 2017) 그리고 FEM 모델은 외력 및 내력의 반응을 이용하여 일정 시간이 지난 후의 물체를 구성하는 특징점들의 좌표를 계산한다.

폴리곤 데이터의 경우, 변형에 따라 각 정점의 좌표가 변경되므로 그래픽스 파이프라인을 수정 없이도 변경된 정점을 이용해 가시화를 수행하면 변형된 결과 영상이 자연스럽게 생성된다. 예를 들어, Efficient surgical cutting with PBD(Berndt, I., 2017)에서는 PBD를 활용하여 입자들의 변경된 위치와 방향을 기반으로 마칭 큐브와 유사한 스키닝 기법을 통해 절개된 부분의 매쉬를 자연스럽게 생성한다. 볼륨 데이터의 경우, 512^3에 해당하는 해상도로 인해 데이터의 용량이 커서 각 지점의 좌표를 변형하기 어렵다고 여겨져 왔다. 이 문제에 대한 대안으로, 많은 관련 연구에서는 변형된 데이터를 생성하지 않고, 가시화 단계에서 관찰 광선을 변형하여 변형된 영상을 생성하는 방법을 선택하였다.

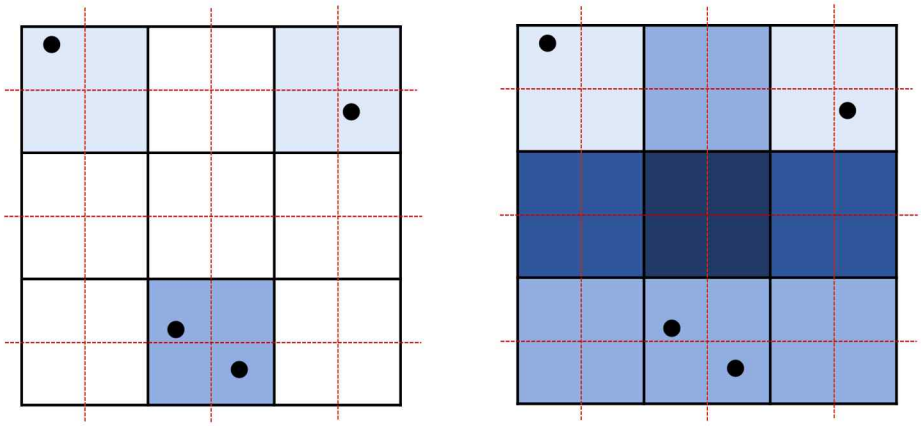
변형체 데이터의 크기가 클 때 발생하는 성능 문제를 해결하기 위해 다양한 방법이 제시되었다. 전체 볼륨 데이터 중 변형이 일부분만 일어난 경우, Adaptive Super Sampling(Kwon, K., 2011)에서는 변형이 심한 부분에 대응되는 화면 영역을 별도의 메모리에 미리 저장하고 렌더링 과정에서 해당 영역에만 수퍼 샘플링을 적용하였다. 전체 영역이 변형되는 경우 Direct volume rendering methods for needle insertion simulation(Fortmeier, D.,

2016)에서는 관심 영역의 크기를 제한하고 중요한 부분만 정밀하게 연산하였다. 최적화를 위해 관심 영역 피라미드 접근 방식을 사용하여, 초기에는 중요한 부분을 위주로 계산하고, 점차 계산 영역의 크기를 확대하여 전체 데이터에 대한 계산을 수행한다. 이는 관심 영역에 더 많은 처리 시간이 소요되며 더 짧은 시간 안에 시각적 타당성을 높이는 데 기여한다.

역변환을 이용한 볼륨 변형체의 계산 방법에서 발생하는 추가적인 문제는, 절개나 병합을 처리하는 연구가 활발히 진행되고 있음에도 불구하고 역변환의 계산이 어렵거나 불가능한 경우 처리가 어렵다는 것이다. 이 문제에 대한 근사적인 해결 방법으로, Discontinuous displacement mapping for volume graphics(Correa, C. D., 2006)에서는 절개된 부분을 표시하는 알파 볼륨을 도입하여 문제를 해결하고자 하였다. 역변환이 존재하는 영역은 법선 벡터와 복합 변형의 계산을 수학적으로 처리하였다. 그러나 본질적인 해결 방법은 폴리곤 모델과 같이 변형된 볼륨 데이터를 직접 생성하는 것이다. 다만, 절개나 병합 작업을 수행할 때 역변환을 계산이 불가능한 경우가 존재한다는 점을 고려해야 한다.

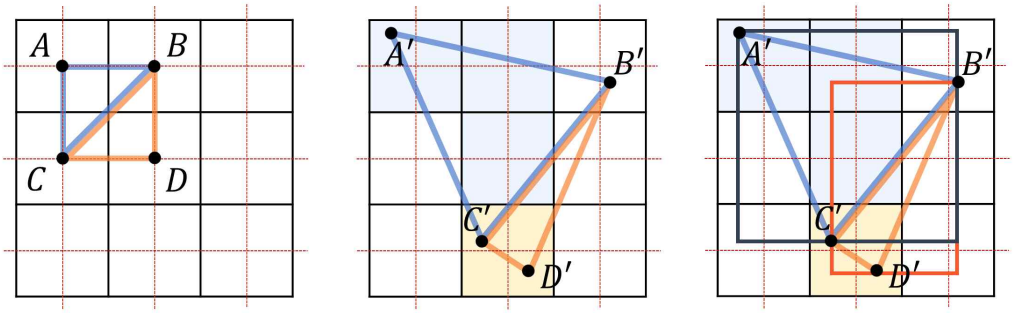
2.2 병렬 리샘플링

병렬 리샘플링은 고속 볼륨 데이터 처리를 위해 여러 데이터 포인트를 동시에 샘플링하는 기법이다. 이 방법 중 하나로 새로운 볼륨 데이터에 순방향 매핑 결과를 저장할 수 있다. 이를 점 기반 방식으로 수행하면, [그림 2-1]의 (a)와 같이 중첩이나 구멍이 발생한다. 커널 필터를 사용할 경우, [그림 2-1]의 (b)와 같이 각 픽셀마다 중첩이 다른 횟수로 발생하여 병렬화가 어렵다는 문제가 있다.



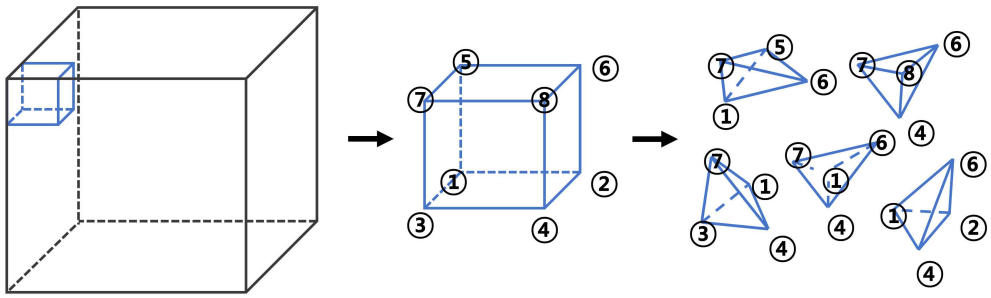
[그림 2-1] 순방향 매핑: (a) 점 기반 방식에서의 구멍과 중첩; (b) 스플래팅 방법에서의 불규칙한 중첩.

이러한 문제를 해결하기 위해, 정점들을 삼각형으로 연결하여 래스터화를 수행하면 중첩 및 구멍 문제를 피할 수 있으며, 병렬 처리도 가능해진다. 변형되지 않은 공간에서 직사각형을 구성하는 네 개의 정점은 변형된 공간에서 인접한 두 개의 삼각형으로 변환된다[그림 2-2]. 변형된 공간에서 픽셀의 중심이 삼각형 내에 있을 때만 해당 픽셀이 최종 이미지에 그려지므로, 각 출력 좌표에서 리샘플링 작업은 한 번씩만 이루어진다.



[그림 2-2] 래스터화 과정: (a) 변형되지 않은 공간에서 격자점을 포함하는 삼각형; (b) 격자점이 삼각형에 포함될 때 변형된 공간으로 변환되어 리샘플링; (c) 각 삼각형의 축에 정렬된 경계 상자(AABB) 내의 격자점에 대한 판정 수행.

3차원 공간에서는 여덟 개의 정점이 육면체 셀을 구성한다. [그림 2-3]에 나타낸 것처럼, 이 셀은 5개의 사면체로 분해된다.



[그림 2-3] 셀 분해 (a) 볼륨 데이터; (b) 셀 그룹;
(c) 각 셀은 5개의 사면체로 분해.

각 사면체는 셀을 구성하는 각 정점의 좌표 변환을 통해 변형된다. 사면체 내에서의 리샘플링 과정은 다음과 같다. 예를 들어, 하나의 셀을 구성하는 데 2^3개의 복셀이 있다고 가정하면, L × M × N 복셀 크기의 볼륨 데이터는 (L − 1) × (M − 1) × (N − 1) 셀로 구성된다. 셀이 B^3 복셀로 구성되어 있다면, 볼륨 데이터는 대략 L/B × M/B × N/B 셀로 구성된다. 셀 크기와 관계없이 각 셀은 다섯 개의 사면체로 분해되며, 각 사면체의 크기는 셀 크기에 비례한다.

출력 복셀이 변형된 사면체 내부에 위치하는지에 대한 여부는 중심 좌표를 사용하여 결정되며, 사면체 내부의 각 복셀 위치에서 리샘플링이 수행된다. 사면체 근처의 영역은 [그림 2-2]의 (c)에 표시된 것처럼 사면체의 AABB에 의해 정의된다. AABB 내의 각 후보 복셀에 대해 중심 좌표가 계산된다. 계산된 값은 3차원 공간에서의 중심 좌표이므로, 4차원 벡터로 표현된다. 벡터의 각 구성 요소가 0과 1 사이일 때, 해당 복셀은 사면체 내부에 있다고 판단된다.

이 접근 방법을 GPU에서 병렬적으로 실행하는 것을 병렬 리샘플링이라고 한다. 대표적인 기존 연구로는 Fast deformation of volume data using tetrahedral mesh rasterization(Gascon, J., 2013)과 A parallel resampling method for interactive deformation of volumetric models(Aguilera, A.R., 2015)가 있다. Gascon의 연구에서는 상대적으로 큰 셀을 사용하여 사면체가 생성되었고, Aguilera 연구에서는 동일한 복셀 크기의 셀이 생성되었다. 이 병렬 리샘플링 방법은 터치 스크린을 사용하여 실시간으로 수행될 수 있으며(Torres, R., 2021), 일반 메쉬에서 볼륨 데이터를 생성할 때 중간 단계에서 사면체 메쉬를 생성함으로써 적용될 수 있다(Chen, J., 2021).

2.3 GPU 병렬화

GPU는 많은 수의 데이터를 병렬적으로 처리할 수 있다. 이러한 처리 과정에서 GPU의 스레드들은 공유 메모리를 활용할 수 있으며, 이를 통해 전역 메모리에 대한 반복적인 접근 대신 공유 메모리에서 데이터를 빠르게 읽어, 처리 속도를 향상시키고 일반 지역 변수 사용 대비 레지스터 사용을 절약한다.

Direct volume rendering methods for needle insertion simulation (Fortmeier, D., 2016)에 따르면, 각 스레드는 인접 데이터를 공유 메모리에 저장함으로써 처리 효율을 극대화하고, 열 기반 레이아웃에서 W 원소를 계산하기 위해 필요한 메모리는 5W이다.

III. 보간을 이용한 효율적인 리샘플링

3.1 좌표 보간을 이용한 효율적인 리샘플링 방법

본 연구는 변형된 볼륨 데이터 생성을 위해 다수의 리샘플링이 수행되는 것에 착안하여, 리샘플링을 효율적으로 수행하는 좌표 보간 방법을 제안한다. 본 연구에서는 병렬 리샘플링의 주요한 두 선행 연구인 Gascon의 방법과 Aguilera의 방법에서 장점을 결합하여 성능을 향상하려 한다. 이번 장에서는 이전 두 연구의 특성을 설명하고 본 연구의 제안 방법을 설명한다. 편의상 GPU의 스레드들의 묶음을 스레드 블록이라고 표현한다.

Gascon의 방법은 각 사면체의 크기가 수십 복셀 이상으로 충분히 크다고 가정하며, 각 스레드는 변형된 볼륨 데이터를 구성하는 각 복셀을 계산한다. 따라서 각 사면체에 하나 또는 여러 개의 스레드 블록들이 대응된다. 하나의 스레드 블록에 속한 스레드들은 사면체 정보(X_0, AABB)를 공유한다. Gascon의 연구에서 사면체의 정보 계산은 CPU에서 수행하고, 사면체의 전체 개수는 5000개 이하로 적은 수이다.

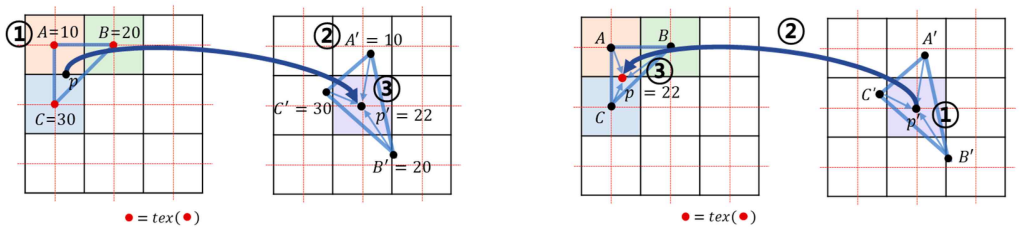
그러나 변형의 부드러운 움직임을 달성하기 위해서는 사면체의 수를 상당히 증가시켜야 한다. 사면체의 개수가 많을 때, Gascon의 방법을 적용하면 문제가 발생한다. 우선, 느린 CPU에서 많은 정보 계산으로 인해 처리 시간이 길어지며, 하나의 스레드 블록에서 수행하는 리샘플링의 횟수가 한두 번 정도로 적으므로 스레드 블록을 구성하는 대부분의 스레드들이 아무 일도 하지 않는 비효율이 발생한다. 본 연구에서는 한 사면체가 복셀 크기만큼 작다고 가정하였으므로, 한 사면체를 스레드 블록으로 구성하고 수백 개의 스레드를 활성화할 이유가 없다.

Aguilera의 방법은 사면체의 정점을 변형된 공간의 좌표와 밀도값으로 정의한다. 변형된 공간의 좌표는 사전 계산된 시뮬레이션 결과를 사용한다. 셀

은 8개의 복셀을 정점으로 하는 육면체이며, 각 복셀 위치에서 텍스처 샘플링을 수행하여 밀도값을 얻는다. 변형된 공간의 격자점을 둘러싼 정점들에 저장된 밀도값을 보간하여 리샘플링 값을 구하고, 이를 변형된 볼륨 데이터에 저장한다. Aguilera의 방법은 Gascon의 방법과 차이가 있다. Gascon의 방법은 커다란 하나의 사면체가 여러 셀을 포함하는 반면, Aguilera의 방법에서는 하나의 셀이 다섯 개의 매우 작은 사면체로 분해된다. 각 스레드는 하나의 셀, 즉 다섯 개의 사면체 처리에 사용된다.

Aguilera의 방법은 각 스레드가 하나의 셀을 처리하고 각 셀은 작은 사면체 5개로 분해되도록 하였다. Gascon의 방법은 각 셀에 대해 샘플링을 8번 수행하는 대신 래스터화 과정에서 텍스처 샘플링하는 방식을 사용하였다. 본 연구에서는 이러한 두 연구의 장점을 결합하여 정밀한 결과를 필요로 하는 분야에 적합한 고속 알고리즘을 구성하였다. 이 방식은 사면체 크기가 작고 각 사면체에 대해 실제로 샘플링을 수행하는 횟수가 적어 효율적이다.

본 연구에서 각 스레드는 셀 단위로 병렬 처리를 수행하며, 하나의 셀은 5개의 사면체로 분해되어 각 사면체에 대해 순차적으로 연산한다. 각 사면체의 AABB에 포함된 각 격자점에서 값을 보간하여 출력값을 계산한다. 본 연구는 각각의 격자점에서 사면체 정점에 대한 보간 가중치인 무게중심 좌표를 계산하려고 한다. 이를 위해서는 변형된 공간에서 사면체 정점들의 좌표를 사용한다.



(a) (b)

[그림 3-1] 텍스처 샘플링 방법 (a) Aguila 방법 (1) 샘플링 (2) 보간 (3) 값 저장; (b) Gascon 방법 (1) 보간 (2) 샘플링 (3) 값 저장.

Aguilera의 방법은 정점에 저장된 밀도값의 가중 평균을 계산한다([그림

3-1]의 (a) 3). 반면, Gascon의 방법은 변형되지 않은 공간의 좌표에 대한 가중 평균을 계산한다([그림3-1]의 (b) 2). 본 연구는 Gascon의 방법을 따라 새로운 좌표를 이용해 텍스처 샘플링하여 밀도값을 구한다([그림3-1]의 (b) 3). 해당 방법은 Aguilera의 방법에 비해 텍스처 샘플링의 횟수를 줄이는 효과가 있다. 변형되지 않은 공간에서 셀의 크기가 1이므로, 평균적으로 각 셀에 대한 텍스처 작성은 한 번만 발생한다. 이는 하나의 셀이 다섯 개의 사면체를 포함하더라도 동일하게 적용된다. 본 연구가 기존 연구에 비해 메모리 참조가 얼마나 효율적인지 구체적으로 보인다. 볼륨 데이터의 크기가 (volx, voly, volz)라고 가정하면, 육면체 셀의 개수만큼 많은 스레드가 런치된다.

$$\text{스레드 개수} = \text{셀 개수} = (volx-1)(voly-1)(volz-1)$$

각 셀에 대해 다섯 개의 사면체가 생성되므로 다음과 같다:

$$\text{사면체 수} = 5(volx-1)(voly-1)(volz-1)$$

출력 복셀 개수는 입력 복셀과 동일한 크기인 $(volx)\times(voly)\times(volz)$개이다. 변형체를 구성하는 사면체들은 서로 겹치지 않고 인접해 있어, 최대 리샘플링 및 쓰기 횟수는 출력 데이터 크기인 $(volx)\times(voly)\times(volz)$만큼 발생한다. 이에 따라, 본 연구에서 하나의 셀에 대한 출력 수는 대략 1번이다. 이는 Aguilera의 방법에서 각 셀당 8번의 텍스처 샘플링이 일어나는 것과 비교할 때, 훨씬 효율적인 결과를 보여준다.

$$\frac{(volx)\times(voly)\times(volz)}{(volx-1)\times(voly-1)\times(volz-1)} \approx 1$$

본 연구는 기존 연구에 비해 약간 더 많은 메모리를 사용한다. 구체적으로 메모리 사용량을 비교하면 다음과 같다. 리샘플링은 변형 후에 수행되기 때문에, 셀의 여덟 정점에 대해 변형 전후 위치를 저장한다. 각 스레드에 필요한

데이터 크기는 8(셀 당 정점) × 2(움직임 전후) × 3(x, y, z) × 4(float의 크기)=192bytes이다. Aguilera의 방법의 경우, 8(셀 당 정점) × (3(x, y, z) × 4(float의 크기) + 2(밀도 값의 크기)) = 112bytes가 사용된다. 제안 방법의 경우, 기존 방법보다 약간 더 많은 메모리를 사용한다. 참고로, Gascon의 방법의 경우, 각 블록에 대해 변형 전의 사면체 좌표를 공유하므로 변형 전의 좌표는 사전 계산된 메모리에서 읽을 수 있다. 이는 메모리를 덜 필요로 하는 것처럼 보이지만, 사면체의 수가 적은 경우에만 사용할 수 있다.

변형체 데이터 생성의 마지막 단계는 샘플링을 수행하여, 각 샘플링 값을 대상 볼륨 데이터에 저장하는 것이다.

$$X_0 = (A\ \ B\ \ C\ \ D) = \begin{pmatrix} A_x & B_x & C_x & D_x \\ A_y & B_y & C_y & D_y \\ A_z & B_z & C_z & D_z \end{pmatrix}$$

$$\overline{X_0} = \begin{pmatrix} X_0 \\ 1^T \end{pmatrix} = \begin{pmatrix} A_x & B_x & C_x & D_x \\ A_y & B_y & C_y & D_y \\ A_z & B_z & C_z & D_z \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

수식1

$$X_1 = (A'\ B'\ C'\ D') = \begin{pmatrix} A'_x & B'_x & C'_x & D'_x \\ A'_y & B'_y & C'_y & D'_y \\ A'_z & B'_z & C'_z & D'_z \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

$$\overline{X_1} = \begin{pmatrix} X_1 \\ 1^T \end{pmatrix} = \begin{pmatrix} A'_x & B'_x & C'_x & D'_x \\ A'_y & B'_y & C'_y & D'_y \\ A'_z & B'_z & C'_z & D'_z \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

$$\begin{pmatrix} x_0 \\ 1 \end{pmatrix} = \overline{X_0} \cdot b,\ b = \overline{X_0}^{-1} \begin{pmatrix} x_0 \\ 1 \end{pmatrix}$$

수식2

$$\begin{pmatrix} x_1 \\ 1 \end{pmatrix} = \overline{X_1} \cdot b,\ b = \overline{X_1}^{-1} \begin{pmatrix} x_1 \\ 1 \end{pmatrix}$$

수식3

$$\begin{pmatrix} x_0 \\ 1 \end{pmatrix} = \overline{X_0} \cdot b = \overline{X_0} \cdot \overline{X_0}^{-1} \begin{pmatrix} x_1 \\ 1 \end{pmatrix}$$ 수식4

수식(1)는 사면체의 네 정점의 좌표를 나타내는 행렬로, X_0은 변형되지 않은 공간에서의 좌표를, X_1은 변형된 공간에서의 좌표를 나타낸다. 이 행렬들은 무게중심 좌표를 기반으로 공간 좌표를 생성하는 변환 작업을 수행한다.

샘플링 과정은 변형된 좌표 x_1에서 얻은 변형 전의 좌표 x_0에서 수행된다. x_0의 중심 좌표와 x_1의 중심 좌표가 동일하다는 성질을 이용하여, x_1에 해당하는 x_0를 추정한다. 한 사면체의 변환은 아핀 변환으로 가정되므로, x_0의 중심 좌표와 x_1의 중심 좌표는 수식(2)와 수식(3)과 같다.

무게중심 좌표(b)를 얻기 위해, 복셀의 좌표 x_1을 사차원 동차 좌표 형태로 정의하고(수식(3)), 이를 사면체의 네 정점 좌표로 구성된 행렬의 역과 곱한다. 수식(4)에 나타난 대로, x_0은 무게중심 좌표 b와 변형되지 않은 좌표의 사면체 네 정점을 사용하는 열 벡터 행렬인 X_0을 곱하여 얻는다. 이를 알고리즘으로 나타내면 다음과 같다.

알고리즘 1. 무게중심 좌표 계산

Aguilera의 방법

```
1: struct vertex
2: float x,y,z;
3: short value;
4: Procedure SampleTetrahedron(vertex A, B, C, D, Tex3D outGrid)
5:    aabb boundingBox = outGrid.computeAABB(A, B, C, D);
6:     foreach (voxel in boundingBox)
7:        float4 baryCoords = computeBaryCoords(voxel.center, A, B, C,
D);
8:        if (centerLiesInsideTetrahedron(baryCoords))
9:          short newValue = interpolateValue(baryCoords, A, B, C, D);
10:         setValue(voxel, newValue);
11:       end if
12:    end foreach
13: end procedure
```

제안 방법

```
1: struct vertex
2: float x,y,z;
3: float tx,tx,tz;
4: Procedure SampleTetrahedron(Mat4 X̄0, vertex A, B, C, D, Tex3D
outGrid, Tex3D inVolume)
5:    aabb boundingBox = outGrid.computeAABB(A, B, C, D);
6:    foreach (voxel IN boundingBox)
7:       float4 baryCoords = computeBaryCoords(voxel.center, A, B, C,
D); /* 수식 3 */
8:       if (centerLiesInsideTetrahedron(baryCoords))
9:          float4 inpos = X̄0 * baryCoords;
10:         float4 newValue = tex3D(inVolume, inpos.xyz);
11:       setValue(voxel, newValue);
12:    end if
13:  end foreach
14: end procedure
```

3.2 무게중심 보간을 이용한 사면체 처리

수식(4)에 기술된 대로, 각 사면체에 대한 역행렬 계산을 수행한다. 본 연구는 Aguilera의 연구에서 활용된 6,500만 개의 사면체에 비해 훨씬 많은 78,600만 개의 사면체를 다루고자 하므로, 보다 효율적인 계산 방식의 필요성이 제기된다. 해당 절에서는 효율적인 역행렬 계산의 중요성을 설명하고, 사면체의 수가 증가함에 따라 발생하는 수치적 불안전성에 대해 논의한다.

3.2.1 무게중심 보간과 역행렬 계산

본 연구에서는 각 출력 격자점의 밀도값을 계산하기 위해 무게중심 좌표를 활용하여 변형 전 좌표를 계산하고 텍스쳐 샘플링을 수행한다. 알고리즘 1의 무게중심 좌표 계산에 제시된 바와 같이, 무게중심 좌표는 수식(3)에 따라 역행렬을 사용하여 계산된다. 사면체의 수가 증가하고 각 사면체의 부피가 감소함에 따라, 역행렬 계산은 수치적으로 불안정해진다. 예를 들어, 사면체를 구성하는 네 점의 좌표가 A(255.9, 256.7, 133.1), B(256.7, 255.9, 133.4), C(256.7, 256.7, 132.3), D(255.9, 255.9, 132.3)인 경우, 이들 점을 열 벡터로 사용하여 행렬 X_1의 역행렬을 구해야 한다. 이 과정은 각 출력 격자점에 대한 정확한 밀도값을 얻는 데 중요하다.

$$\overline{X_1} = \begin{bmatrix} 255.9 & 256.7 & 256.7 & 255.9 \\ 256.7 & 255.8 & 256.7 & 255.9 \\ 133.1 & 133.4 & 132.3 & 132.3 \\ 1 & 1 & 1 & 1 \end{bmatrix} \quad \text{수식5}$$

그런데 $\overline{X_1}$의 역행렬을 단정도 부동 소수점(float)으로 계산할 경우, 오차가 크게 발생한다. 부록 A에서는 역행렬을 구하는 첫 단계인 행렬식을 구하

는 과정을 보여준다. 정확한 행렬식 값이 1.216임에도 불구하고, 단정도 부동 소수점으로 계산했을 때 2.010187로 계산되는 오차가 확인된다. 이러한 오차는 역행렬 계산 중 a·b·c−d·e·f 형태의 계산이 반복되기 때문에 발생한다. 좌표값을 곱한 결과인 a·b·c와 d·e·f는 각각 큰 값(〉10^6)을 나타내지만, 이들의 차이인 a·b·c − d·e·f의 결과는 상대적으로 작은 값(〈10)이므로 단정도 부동 소수점에서 쉽게 오류가 발생한다. 본 연구에서는 각 셀의 크기가 작기 때문에, 사면체를 구성하는 인접 정점 좌표값이 비슷하다. 셀의 수가 증가함에 따라 이 오차는 더욱 두드러질 수 있다.

$$\overline{X_1}^{-1} = \begin{bmatrix} -0.723684 & 0.526316 & 0.723684 & -0.526316 \\ 0.723684 & -0.526316 & 0.526316 & -0.723684 \\ 0.526316 & 0.526316 & -0.526316 & -0.526316 \\ -69.631579 & -69.631579 & -250.243421 & 390.506579 \end{bmatrix} \quad \text{수식6}$$

(배정밀도)

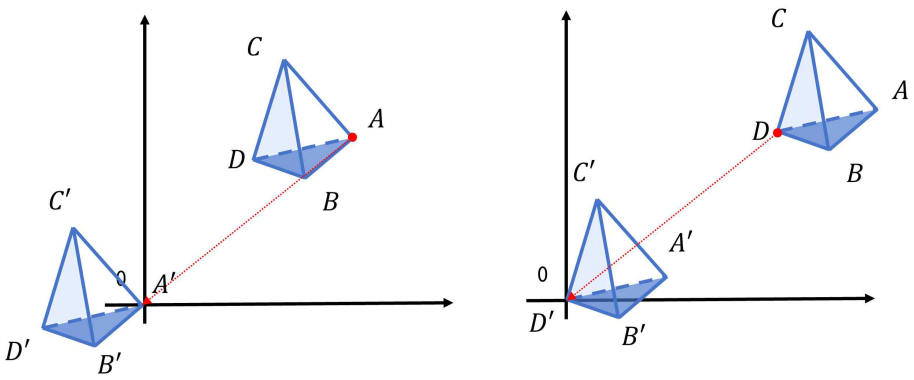
$$\overline{X_1}^{-1} = \begin{bmatrix} -0.437228 & 0.318691 & 0.437228 & -0.318691 \\ 0.437228 & 0.318691 & 0.318691 & -0.437228 \\ 0.322577 & 0.318691 & -0.322577 & -0.320634 \\ -42.284821 & -42.284821 & -151.230408 & 236.297531 \end{bmatrix} \quad \text{수식7}$$

(단정밀도)

이 문제를 해결하는 기본적인 방법은 배정도 부동 소수점을 사용하는 것이다. 그러나 대부분의 일반적인 GPU는 배정도 연산 하드웨어가 적어 배정도 부동 소수점 연산은 단정도 연산에 비해 훨씬 느리다. 이러한 문제를 극복하기 위해, 다음 절에서는 무게중심 좌표의 기준점을 사용하는 방법을 설명한다.

3.2.2 기준점을 이용한 무게중심 좌표 계산

무게중심 좌표(b)를 얻기 위해서만 역행렬 계산을 사용하며, 이는 각 점이 사면체 내부에 위치하는지를 판단하는 데 사용된다. 사면체의 모든 점을 이동시키더라도 무게중심 좌표는 변하지 않는 성질을 갖는다. 좌표값들을 가능한 한 작게 유지하기 위해, 각 점을 원점에 가깝도록 평행이동하는 전략을 사용한다[그림 3-2].



[그림 3-2] 정점 평행이동 (a) 각 점을 원점으로 평행이동;
(b) 마지막 정점을 원점으로 평행이동.

편의를 위해 사면체의 네 정점 중 마지막 정점 D를 원점으로 평행이동하였다[그림 3-2](b). 사면체의 크기가 매우 작기 때문에, 사면체의 AABB 내의 모든 점의 좌표는 원점에 매우 가까이 위치한다. 이러한 조정으로 a·b·c와 d·e·f의 각 값이 작아지게 되며, 이는 단정도 부동 소수점를 사용하여도 오차가 충분히 작게 유지할 수 있도록 한다.

한편, 수식(2)는 다음과 같이 나타낼 수 있다:

$$\begin{pmatrix} A_x & B_x & C_x & D_x \\ A_y & B_y & C_y & D_y \\ A_z & B_z & C_z & D_z \\ 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} b_A \\ b_B \\ b_C \\ b_D \end{pmatrix} = \begin{pmatrix} X_x \\ X_y \\ X_z \\ 1 \end{pmatrix}$$ 수식8

모든 점 A, B, C, D, 그리고 X가 -D만큼 평행이동하면, 다음과 같이 표현될 수 있다:

$$\begin{pmatrix} A_x - D_x & B_x - D_x & C_x - D_x & D_x - D_x \\ A_y - D_y & B_y - D_y & C_y - D_y & D_y - D_y \\ A_z - D_z & B_z - D_z & C_z - D_z & D_z - D_z \\ 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} b_A \\ b_B \\ b_C \\ b_D \end{pmatrix}$$

$$= \begin{pmatrix} A_x - D_x & B_x - D_x & C_x - D_x & 0 \\ A_y - D_y & B_y - D_y & C_y - D_y & 0 \\ A_z - D_z & B_z - D_z & C_z - D_z & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} b_A \\ b_B \\ b_C \\ b_D \end{pmatrix} = \begin{pmatrix} X_x - D_x \\ X_y - D_y \\ X_z - D_z \\ 1 \end{pmatrix}$$ 수식9

따라서, 3 × 3 행렬만 관찰하면 다음과 같은 결과를 얻을 수 있다:

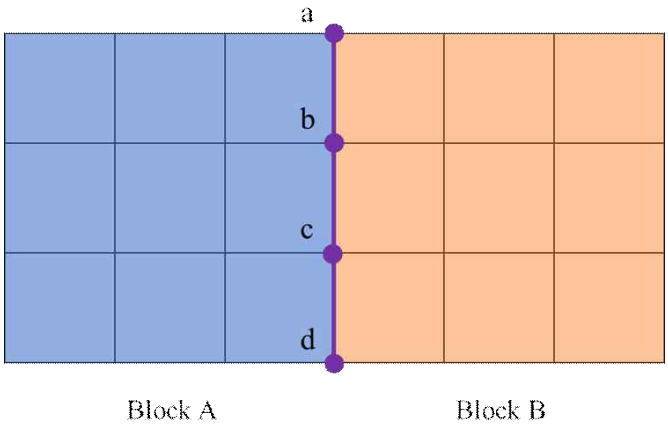
$$\begin{pmatrix} A_x - D_x & B_x - D_x & C_x - D_x \\ A_y - D_y & B_y - D_y & C_y - D_y \\ A_z - D_z & B_z - D_z & C_z - D_z \end{pmatrix} \begin{pmatrix} b_A \\ b_B \\ b_C \end{pmatrix} = \begin{pmatrix} X_x - D_x \\ X_y - D_y \\ X_z - D_z \end{pmatrix}$$ 수식10

여기서, $b^3(b_A, b_B, b_C)$는 4 × 4 행렬 대신 3 × 3 행렬의 역행렬을 계산함으로써 얻을 수 있다(수식(10)). 또한, b_D값은 $b_A+b_B+b_C+b_D=1$을 사용하여 계산한다. 역행렬 계산 과정에서 a·b·c − d·e·f 대신 a·b − c·d 형태가 사용되므로, 오류 없이 단정도 부동 소수점을 사용할 수 있다. 기준점을 이용한 무게중심 좌표의 계산은 새로운 제안은 아니지만, 배정도 부동 소수점 대신 단정도 부동 소수점을 사용할 수 있다는 점이 중요하다.

IV. 중첩된 블록을 이용한 빈공간 제거

4.1 블록 기반 가속화

이번 장에서는 3장의 내용을 기반으로 볼륨 데이터를 더욱 고속으로 생성하는 방법을 제안한다. 일반적으로 모든 부분을 리샘플링하면 오류 없는 변형된 볼륨을 얻을 수 있다. 그러나, 이번 장에서는 블록 개념을 정교하게 도입하여 데이터의 일부만 리샘플링하여 화질 저하 없이 결과 영상을 효율적으로 생성하는 방법을 제안한다.

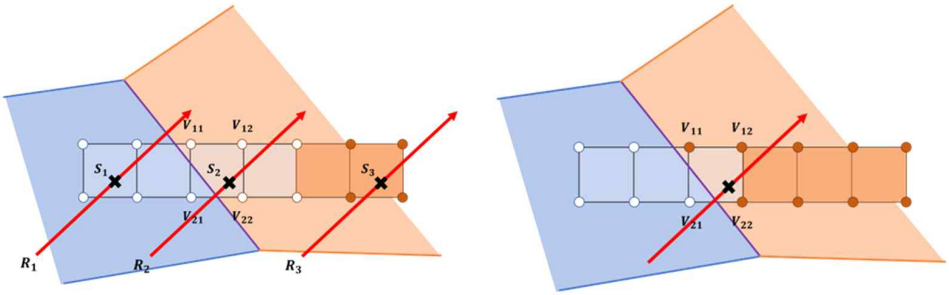


[그림 4-1] 블록 정의.

블록은 일반적으로 볼륨 데이터를 특정 크기의 육면체로 나누어 경계면을 공유하는 형태로 정의된다(Levoy, M., 1990). [그림 4-1]에 따르면, 각 격자점은 복셀을 나타내며, 한 블록의 크기는 4×4이다. 복셀 a, b, c, d는 두 블록 A, B 동시에 속한다.

변형 전의 각 블록은 사용자 입력에 따라 변형되어 변형된 공간에 저장되며, 이를 사용하여 볼륨 렌더링을 수행한다. 이때 최종 출력 화면에 영향을

주지 않는 블록은 생략하여도 무방하다. 따라서, 어떤 블록 전체가 투명하다고 가정할 때, 해당 블록 영역의 변형된 볼륨 데이터는 리샘플링과 저장 과정을 생략함으로써 효율을 향상시킬 수 있다.



(a) TT (b) TN

[그림 4-2] 블록 기반 가속화 방법.

이전 장에서 설명한 방법과 새롭게 제안하는 블록 기반 방법을 구체적 예를 들어 설명한다. [그림 4-2]에서 파란색은 투명한 영역, 주황색은 불투명한 영역을 나타내며, 이는 두 블록이 변형된 공간에서 차지하는 영역을 의미한다. v_{11}, v_{12}, v_{21}, v_{22}를 포함한 각 격자점의 작은 원은 변형된 공간에서의 복셀을 나타낸다. 투명하다고 판단된 복셀에는 색을 칠하지 않고, 불투명하다고 판단된 복셀에만 색을 칠하였다.

원 볼륨 데이터에서는 병렬 리샘플링을 수행하여 복셀값을 채운다. 투명한 블록에서 생성된 복셀은 모두 투명하며, 불투명한 블록에서 생성된 복셀은 투명할 수도 있고 불투명할 수도 있다.

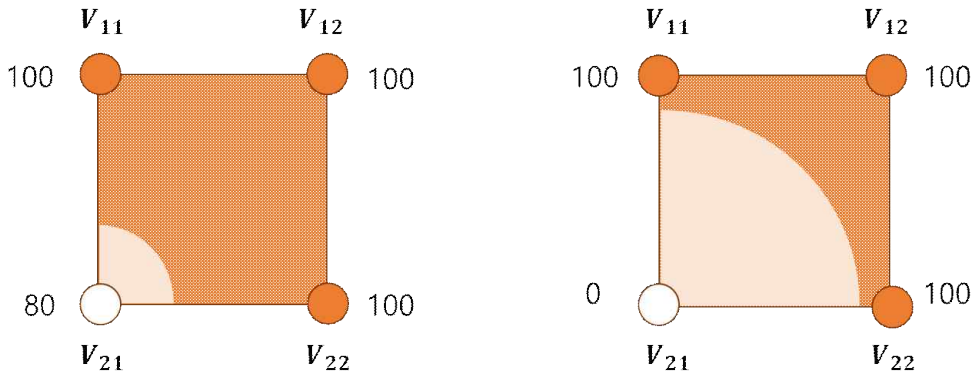
가시화 단계에서는 변형된 공간의 볼륨 데이터에 광선 추적법 알고리즘을 적용하여 영상을 생성한다. 사용자가 지정한 관찰 방향으로 광선이 발사되며, 하나의 픽셀에 대응하는 광선 하나를 빨간색 화살표로 표시하였다. 화살표 위에서 일정 간격마다 해당 위치에서 주변 복셀값을 보간하여 샘플링 값을 구한다. 그중 한 위치를 x로 표시하였다.

기존 방법의 경우, [그림 4-2]의 (a), (b) 상관없이 모든 복셀을 리샘플링 하므로, 모든 복셀에 값이 존재한다. 따라서 x로 표시된 지점의 주변값을 보간하더라도 값의 왜곡이 발생하지 않는다. 반면, 제안한 블록 가속화 방법을 이용하여 계산하는 경우 왜곡이 발생할 수 있다. 이 상황을 자세히 설명하면 다음과 같다. 단, 설명의 편의를 위해 가시화 단계에서 값이 86 이하인 샘플 값을 모두 투명하게 처리한다고 가정한다.

[그림 4-2]의 (a)에서 R_1에 속한 S_1은 투명한 블록 내부에 위치하므로 가시화 시 투명하게 처리되어야 한다. 병렬 리샘플링으로 변형된 볼륨을 생성할 때, 투명한 블록의 위치에는 보간 계산이 생략되므로 초깃값이 그대로 저장된다. 가시화 단계에서 이렇게 변형된 볼륨에서 S_1 보간을 수행하면 초깃값 0이 얻어진다. 이 값은 원본의 볼륨 데이터와는 비교했을 때 왜곡된 값이지만, 보간된 0은 투명하게 처리되므로 최종적으로 출력 화면에는 아무런 영향을 미치지 않는다.

R_3에 속한 S_3는 불투명한 블록에 위치하므로 S_3 주변 복셀들은 모두 리샘플링되어 정확한 값이 존재한다. 이 복셀값을 보간하여 얻은 샘플링 결과는 기존 방법과 동일하게 나타나므로, 최종적으로 출력 화면에 영향을 주지 않는다. 즉, S_1의 경우는 값이 변경되지만 변경 전후 색상이 모두 투명하므로 문제가 되지 않고, S_3는 값의 왜곡이 없어 문제가 되지 않는다.

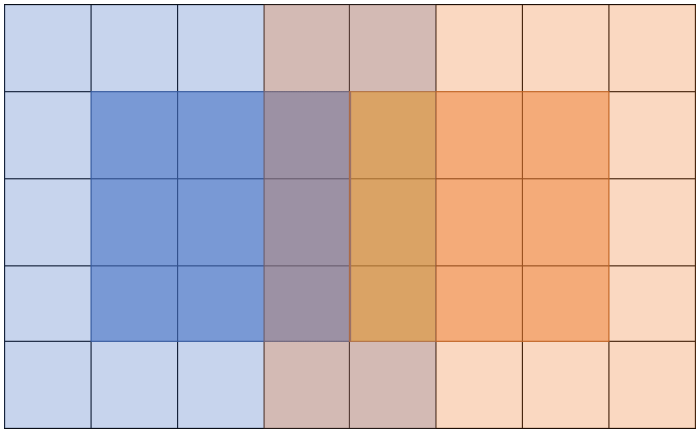
이제 블록의 경계에 위치한 R_2에 속한 S_2에 대해 설명한다. 블록 기반 가속화 방법을 적용하여 [그림 4-2]의 (a)를 처리할 경우, 복셀 v_{11}, v_{12}, v_{22}는 불투명한 블록에 속하므로 리샘플링 되어 올바른 값 80이 저장된다. 한편, 복셀 v_{21}은 투명한 블록에 속해 있으므로 초깃값 0이 유지된다. 따라서 광선의 샘플링 지점 S_2에서 세 개의 올바른 값과 하나의 초깃값을 이용한 보간 계산 결과값이 60으로 왜곡된다. 그러나 이 값은 불투명도 및 색상 전이 함수를 통해 올바른 값 80과 함께 투명하게 처리되어 화질에 영향을 주지 않는다. 이렇게 [그림 4-2]의 (a)에서는 출력 결과 영상에 오류가 발생하지 않는다.



[그림 4-3] 왜곡.

그러나 [그림 4-2]의 (b)에서는 출력 이미지에 결함이 발생하는 상황을 보여준다. 이 문제의 원인과 결과를 분석하고, 문제를 해결하기 위한 새로운 방안을 제시한다. [그림 4-2]의 (b) 일부를 확대하여 [그림 4-3]으로 나타냈다. 예를 들어, 임곗값이 86인 상황에서 v_{11}, v_{12}, v_{22}의 값이 100이고, v_{21}은 80으로 임곗값 이하인 상황을 가정한다. 기존의 방법에서는 모든 복셀에 대해 리샘플링을 수행하므로, 86 이상의 값이 있는 영역은 [그림 4-3]의 (a)에 색칠된 부분으로 명확하게 나타난다. 그러나 블록 기반 가속화를 사용하는 경우 v_{21}이 0으로 저장되어 해당 셀 내에서의 보간 결과 80의 값을 얻게 되며, 임곗값 86의 구간을 색칠할 경우 그림 (b)와 같이 화질 저하 및 파임이 발생한다.

이러한 문제는 투명한 블럭과 불투명한 블록이 인접하고, 블록 경계 근처에서 투명 블록 소속 복셀은 투명하고, 불투명 블록 소속 복셀은 불투명할 때 종종 발생한다. 투명한 블록 소속의 복셀이 0으로 초기화되면서 불투명한 복셀값과 보간되어 값을 변경되고 이로 인해 문제가 발생한다. 이 파임 현상은 샘플의 색상 및 투명도를 왜곡하고, 법선 벡터 추정 시 오류를 유발하므로 출력 영상에 노이즈 패턴을 발생시킨다.



[그림 4-4] 블록 마진.

이 문제를 해결하기 위해 본 연구에서는 블록의 기능을 확장하는 새로운 방법을 제안한다. 이 방법은 하나의 블록이 더 넓은 범위를 고려할 수 있도록 블록의 영역을 확장한다. 원래 블록 범위가 아닌 확장된 블록 범위를 '마진'으로 정의하고, 블록 간의 간격은 유지하면서 일부 경계 영역이 중첩되도록 한다. 확장된 블록의 두께를 m으로 설정하며, 기존 블록 크기가 b^3이었다면 새로운 블록 크기는 $(b+2m)^3$이 된다.

[표 4-1] 마진 적용 전후 블록 상태 요약

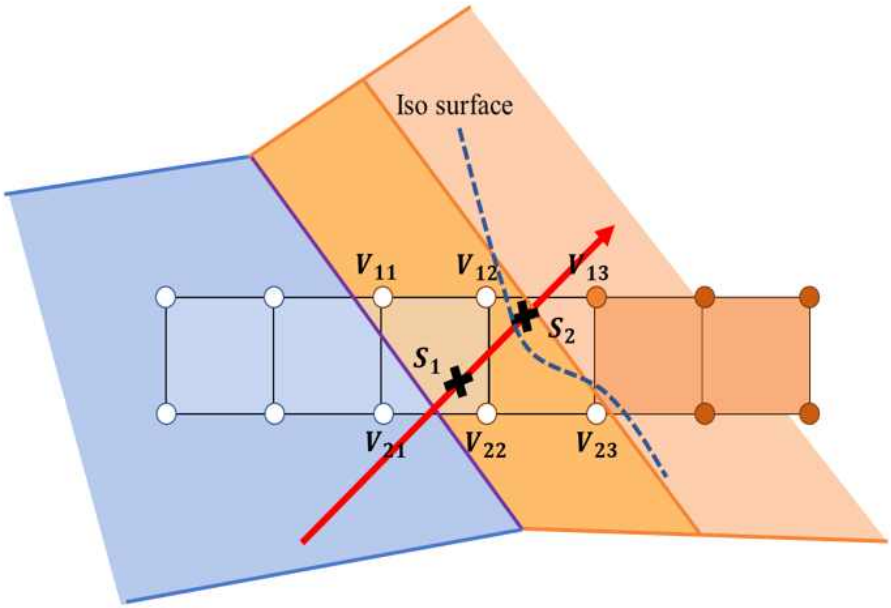
적용 전 \ 적용 후	투명(T)	불투명(N)
투명(T)	블록 건너뜀	리샘플링 수행
불투명(N)		기존 방법과 동일

마진을 적용하기 전과 후의 블록 상태는 (T,T), (T,N), (N,N)으로 나타낼 수 있다. 여기서 'T'는 투명한 블록을, 'N'은 불투명한 블록을 나타낸다. 마진은 보다 넓은 영역을 검사하므로, N으로 분류된 블록이 마진 적용 후 T로

변환될 가능성은 없으므로 (N,T) 상태는 존재하지 않는다.

마진을 추가했을 때 N이 되는 경우, 즉 (T, N)과 (N, N)은 해당 블록이 N으로 판정되어 블록 전체를 리샘플링하기 때문에 블록 내부에서 화질 문제가 발생하지 않는다. 한편, (T,T)의 경우에는 마진을 적용해도 블록이 전체적으로 투명하다고 판단되어 리샘플링을 수행하지 않는다. 이 경우, 제안한 마진 블록 기반 가속화 방법을 적용하면 화질 저하가 없음을 보장할 수 있다.

화질 저하는 주로 블록의 경계에서 발생한다. 투명한 블록과 인접한 블록이 모두 투명할 경우, 두 블록의 경계에서의 리샘플링 값도 투명하므로 리샘플링을 생략해도 화질에 영향이 없다. 이에 따라, 투명한 블록과 인접 블록이 불투명한 경우에만 주의 깊게 검토한다.



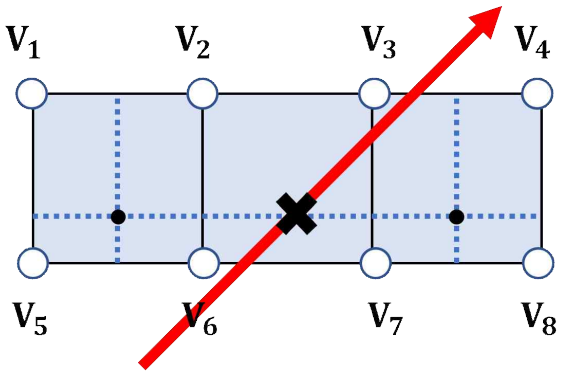
[그림 4-5] 마진 블록 기반 가속화 방법.

[그림 4-5]에서 왼쪽 블록은 투명하다고 가정하므로, 마진 영역의 복셀까지 투명한 것이 보장된다. 따라서 투명한 블록과 마진 영역에 포함된 복셀 (v_{12}, v_{13}, v_{22}, v_{23})은 모두 투명하며, S1의 보간 결과도 투명하다. 제안 방법의

경우, v_{21}에 초깃값을 저장하며 S_1의 보간 값도 그에 따라 왜곡된다. 기존 방법의 경우, 리샘플링을 수행하여 값이 계산되고 그 값으로부터 보간하여 얻은 결과값은 투명하게 판정된다. 즉, 보간 결과가 올바른 값에 비해 왜곡되어도 투명성은 유지된다.

이제 샘플 지점 S_2에 대해 고려해 보면, S_2는 불투명한 복셀의 영향을 받아 관찰 가능한 샘플이다. [그림 4-5]의 S_2에서 보간에 참여하는 복셀(v_{12}, v_{13}, v_{22}, v_{23}) 모두는 불투명한 블록에 속하므로, 블록 내의 모든 복셀값들은 정확하게 계산되어 올바르게 존재한다. 따라서 S_2에서 이루어진 보간은 값의 변경이나 왜곡 없이, 가속화를 적용하지 않은 보간 결과와 동일하게 유지된다.

정리하면, 마진을 도입한 제안 방법에서 불투명으로 판정된 블록은 화질 저하 없이 처리되며 속도 향상도 없다. 투명으로 판정된 블록은 범위를 약간 벗어난 경계 부분 복셀까지 투명성이 보장되므로, 보간 결과도 투명하다. 이 경우, 리샘플링을 생략하여 보간 값의 왜곡이 발생해도 결과의 투명성이 유지되므로, 제안 방법을 통해 화질 손실 없는 속도 향상을 얻을 수 있다. 본 연구는 투명한 블록이 많을수록 리샘플링 시간이 크게 줄어들어 효과적이다.



[그림 4-6] 샘플링 포인트 보간.

마진의 크기는 출력 영상의 화질과 전체적인 성능에 중요한 영향을 미치

므로, 적합한 마진 값을 결정하는 것은 중요하다. [그림 4-5]의 S_1 위치에서 단순히 보간 작업만 수행하는 경우, 인접한 복셀값들을 사용하므로 마진을 1로 설정하여도 충분하다. 그러나 조명 연산을 포함하는 경우, 법선 벡터 계산을 위해 추가적인 복셀 데이터가 필요하므로 더 넓은 마진 설정이 요구된다.

예를 들어, 법선 벡터를 계산할 때, 인접 지점들에서의 보간이 추가적으로 요구된다. 이 과정에 참여하는 복셀(v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8)들을 모두 고려해야 한다[그림 4-6]. 이 경우 샘플 지점에서 최대 2 거리 떨어진 복셀의 값을 읽어야 하므로, 오류를 방지하기 위해 마진을 2로 설정하는 것이 적합하다. 만약 마진을 1로 설정하면, 최악의 경우 v_1이나 v_4 값이 초깃값으로 남아 있을 수 있다. 이로 인해 투명도 및 색상 계산에 참여하는 복셀 v_2, v_3, v_6, v_7은 오류 없이 처리되며 물체의 형태와 색상이 정확하게 출력된다. 그러나 법선 벡터 계산에서 오류가 발생하여 잘못된 조명 효과로 인한 얼룩이 생길 수 있다. 더 나아가 삼차원 공간에서 데이터 분포가 가장 나쁜 경우를 가정할 때, 마진을 2^3으로 설정하면 오류 없이 법선 벡터 계산이 가능하다. 그러나 실용적으로는 실험 및 테스트를 통해 가시적으로 오류가 없는 값으로 결정하면 된다. 마진 크기가 너무 큰 블록은 먼 거리에 있는 불투명한 점의 영향을 받아 불투명 판정 블록 비율이 증가하고, 리샘플링 시간도 증가하는 문제를 야기한다.

안전한 마진 범위는 복셀에서 보간할 때 필요한 안전 거리로, 변형된 공간에서 측정된다. 본 연구에서는 마진 적용을 전처리 과정에서 변형 전 공간에 적용하여 변형 정도를 사전에 예측하고 적당한 마진 값을 결정한다. 특히 볼륨 데이터의 축소 변환에서 그 차이가 두드러지므로, 변형 전 공간에서 마진을 충분히 크게 설정해야 한다.

또한 블록의 크기를 적절하게 설정하는 것이 중요하다. 예를 들어, 어떤 블록의 대부분이 투명하지만 내부의 작은 영역이 불투명한 경우, 전체 블록에 대해 계산을 수행해야 한다. 불투명한 큰 블록을 여러 개의 작은 블록으로 나누면, 그중 다수가 투명하다고 판단되어 생략될 수 있다. 따라서, 블록의 크기

가 너무 큰 경우 생략되는 블록의 비율이 감소하여 비효율적이다. 반면, 블록 크기가 너무 작은 경우에는 하드웨어 구현에서 비효율이 발생할 수 있다(제 5장 참고). 이러한 점을 고려하여 적절한 블록 크기를 설정하는 것이 필요하다.

V. 공유 메모리를 활용한 GPU 효율 향상

이번 장에서는 알고리즘 수준에서의 가속화보다 개발 단계에서의 가속화에 중점을 둔다. 이는 그래픽스 하드웨어를 보다 효율적으로 활용하여 동일한 코드를 더 빠르게 실행시키는 것으로, 모든 하드웨어 구성 요소가 중단 없이 지속적으로 작업을 수행하도록 하는 것이다.

5.1 gpu의 병렬 처리 개요

CUDA는 NVIDIA의 병렬 컴퓨팅 플랫폼으로, GPU를 사용하여 병렬 처리를 효율적으로 수행할 수 있게 하며, 다수의 병렬 프로세서를 통해 신속한 병렬 처리를 가능하게 한다. CUDA에서 스레드 블록은 스레드들의 묶음으로, 하나의 스레드 블록은 최대 1024개의 스레드를 관리할 수 있다. 32개의 스레드는 하나의 와프(Warp)로 묶여 동일한 코드를 실행하게 되며, 이는 SIMT(Single Instruction Multiple Threads) 방식으로 작동한다. 따라서 스레드 블록에서 사용하는 스레드의 수를 32나 64의 배수로 설정하면 나머지 없이 효율적으로 운영할 수 있다. 각 스레드의 작업을 수행하는 하드웨어의 최소 단위를 SP(streaming processor)라고 하며, 하나의 SM(streaming multiprocessor)는 여러 개의 SP로 구성된다. 한 개 또는 여러 와프는 하나의 SM에 할당된다.

온칩 메모리는 프로세서 내부에 직접 통합되어 있어, 데이터 처리 속도를 최적화하는 데 기여하는 고속의 메모리이다. 레지스터는 온칩 프로세서 내에 위치하며, 각 스레드별로 독립된 메모리 공간을 제공한다. 각 스레드는 다른 스레드의 레지스터에 접근할 수 없다. 레지스터는 접근 속도가 매우 빠르다는 장점이 있지만, 메모리 용량이 제한적이어서 과도하게 사용할 경우 병렬 처리 효율성이 저하될 수 있다.

공유 메모리도 온칩에 위치해 있으며, 레지스터와 유사하게 매우 빠른 1-2 사이클의 접근 속도를 가진다. 같은 스레드 블록에 속하는 여러 스레드가 이를 공유하여 읽고 쓸 수 있다. 그러나 공유 메모리의 용량은 오프칩 메모리에 비해 매우 제한적으로, 한 SM 당 64KB로 설정되어 있다. 또한, 하나의 SM이 여러 스레드 블록의 자원을 나누어 사용하므로 각 스레드 블록당 실제로 사용할 수 있는 공유 메모리 양은 더욱 적다.

변형체 계산은 물리 시뮬레이션 단계에서 수행되며, 그 결과는 큰 용량을 차지하기 때문에 오프칩 메모리인 전역 메모리에 저장된다. 리샘플링 단계에서는 이러한 결과를 여러 번 읽어서 변형체 데이터를 생성한다. 전역 메모리에서 데이터를 읽는 과정은 매우 느린데, 이러한 변형체 정보 접근을 수백만 번 반복하게 되므로 더 효율적인 접근 방법이 필요하다.

5.2 공유 메모리 사용

전역 메모리에서 여러 번 데이터를 읽는 과정은 매우 느리기 때문에, 각 스레드가 전역 메모리에 저장된 정보를 레지스터로 복사하여 여러 번 사용하는 방법을 사용한다. 전역 메모리에는 물리 시뮬레이션을 통해 변형된 제어점의 좌표가 저장되어 있으며, 이는 스레드의 레지스터로 복사되어야 한다. 이때, 각 스레드가 서로의 레지스터 데이터를 공유할 수 없고, 각 스레드는 하나의 셀을 담당하기 때문에 인접한 두 셀의 경계에 위치한 제어점들은 여러 스레드의 레지스터에 중복 저장되어야 한다. 삼차원으로 볼 때, 하나의 복셀이 8개의 스레드에 의해 공유되므로, 중복 저장으로 인한 메모리 사용량이 8배 증가한다.

이 문제를 해결하기 위해 본 연구에서는 공유 메모리를 활용한다. 공유 메모리를 활용하면 한 벌의 데이터만 저장하면 되고, 이는 같은 스레드 블록 내의 여러 스레드가 접근할 수 있다. 그러나 스레드 블록의 경계에서는 여전히

데이터의 중복 저장이 필요다. 중복 정도는 w^3 / $(w-2)^3$으로 계산되며, 블록 크기가 8인 경우 중복도는 2보다 작다.

공유 메모리의 활용은 레지스터 사용량을 줄일 수 있다. 레지스터의 크기가 32k×4bytes로 한정되어 있고 이를 여러 와프가 공유하므로, 공유 메모리를 활용함으로써 각 와프가 사용하는 레지스터 수가 줄어들고 활성 와프의 수와 시스템의 효율이 증가한다.

본 연구의 공유 메모리 활용 방법은 다음과 같다. 공유 메모리에 저장된 데이터에 접근할 때 현재 스레드 ID를 사용한다. 삼차원 공간에서, 하나의 블록에 속한 스레드 개수가 n^3이라 가정하면 각 스레드는 육면체 셀 공간 하나를 담당하여 연산하므로, 셀을 구성하는 8개의 정점 복셀 정보가 필요하다. 스레드 ID가 (x, y, z)인 경우, 공유 메모리의 sm[x][y][z] 위치뿐만 아니라 sm[x][y][z]부터 sm[x+1][y+1][z+1]까지의 범위 내 모든 공유 메모리에 접근한다. 스레드 값의 범위는 (0,0,0)~(n−1, n−1, n−1)이므로 필요한 sm 인덱스의 범위는 (0, 0, 0)~(n, n, n)이다. 따라서 sm의 크기는 sm[n+1][n+1][n+1]이다.

즉, 스레드 개수는 n^3개이며, 필요한 공유 메모리의 크기는 $(n+1)^3$이다. 이는 공유 메모리의 초기화 작업이 스레드별 $(n+1)^3/n^3$ 번 작업을 수행함을 의미한다. 예를 들어, 스레드 개수가 4^3일 때 필요한 공유 메모리의 크기는 $(4+1)^3$이며, 이를 4^3으로 나눈 값은 약 1.9이다. 이는 각 스레드가 공유 메모리에 값을 저장하기 위해 2번 이내의 작업을 수행하면 충분함을 의미한다.

이때, 스레드의 개수가 2^3와 같이 너무 적은 경우 이 횟수는 $(2+1)^3/2^3=3.3$으로 증가하지만 대부분의 경우 스레드 개수는 2^3보다 많기 때문에 이러한 경우는 드물다. 따라서 스레드가 명령어 수행의 동기화를 감안하면, 하나의 스레드가 2번 작업을 통해 공유 메모리에 값을 저장할 수 있다면, 최적의 방법이라고 할 수 있다.

본 연구의 제안 방법은 다음과 같다:

thread1Did = threadId(x,y,z) $= z \times n \times n + y \times n + x$

sm1dId = sm[x][y][z] = $z \times (n+1) \times (n+1) + y \times (n+1) + x$

thread1Did는 스레드를 1차원으로 해석한 번호이며, sm1dId는 공유 메모리를 1차원으로 해석한 위치를 나타낸다. 각 스레드는 공유 메모리의 올바른 위치를 계산해야 한다. 예를 들어 n이 4인 경우, sm1did의 범위는 [0,63]이고 thread1Did의 범위는 [0,124]이다. n이 4 이상인 경우, 제안된 방법을 통해 각 스레드는 두 번의 반복 연산으로 공유 메모리를 채울 수 있다. 이는 이전에 설명된 8번의 반복 작업에 비해 중복 작업 없이 메모리 활용 효율성을 높인다. 구체적인 알고리즘 설명은 다음과 같다.

알고리즘 2. 공유 메모리 활용

공유 메모리 활용 방법

```
: /* Kernel launch */
cudaMemset(DeformedVolume, 0, VOLX*VOLY*VOLZ*sizeof(short))
Procedure resampleKernel <<< <block, thread >>>

1:  struct vertex
2:  vec3 op /*original position*/
3:  vec3 tp  /* pos_transfomed*/
4:  global__ void resampleKernel
5:  int tx = threadIdx.x,  ty = threadIdx.y,  tz = threadIdx.z
6:  int myId = tz * n * n+ ty * n+ tx
7:  __shared__ sm buffer[n + 1][n + 1][n + 1]
8: vertex v =  X0, X1
9:  if myId < (n+1)*(n+1)*(n+1) then buffer[n+1][n+1][n+1] = v
10: SampleTetrahedron(buffer[tz][ty][tx], buffer[tz + 1][ty][tx],
buffer[tz + 1][ty][tx + 1], buffer[tz + 1][ty + 1][tx])
11: SampleTetrahedron(buffer[tz + 1][ty + 1][tx], buffer[tz + 1][ty +
1][tx + 1], buffer[tz + 1][ty][tx + 1], buffer[tz][ty + 1][tx + 1])
12: SampleTetrahedron(buffer[tz + 1][ty][tx + 1], buffer[tz][ty][tx +
1], buffer[tz][ty][tx], buffer[tz][ty + 1][tx + 1])
```

```
13: SampleTetrahedron(buffer[tz][ty][tx], buffer[tz][ty + 1][tx],
buffer[tz + 1][ty + 1][tx], buffer[tz][ty + 1][tx + 1])
14: SampleTetrahedron(buffer[tz + 1][ty + 1][tx], buffer[tz + 1][ty][tx
+ 1], buffer[tz][ty + 1][tx + 1], buffer[tz][ty][tx])

Procedure SampleTetrahedron(vertex A, vertex B, vertex C, vertex D)
1:    aabb boundingBox = computeAABB(A.tp, B.tp, C.tp, D.tp)
2:    foreach (voxel IN boundingBox of target volume)
3:     float4 baryCoords = computeBaryCoords(voxel.center, A, B, C,
D)
4:     if (centerLiesInsideTetrahedron(baryCoords))
5:       float4 inpos = A.op* baryCoords[0] +B.op* baryCoords[1]
+C.op*baryCoords[2]+ D.op*baryCoords[3]
6:       float4 newValue = tex3D(inVolume, inpos.xyz)
7:     setValue(voxel, newValue);
8:     end if
9:    end foreach
10: end Procedure
```

변형 후 볼륨 데이터를 저장할 3차원 배열을 생성하고 초깃값으로 설정한다. 커널 런치를 위해 볼륨 데이터를 블록 크기로 분할하여 하나의 스레드 블록이 하나의 블록을 처리하도록 한다. 커널 런치 후, 함수 내에서 각 스레드 번호를 1차원으로 계산하고, 이를 기반으로 공유 메모리 내의 위치를 계산한다.

위에서 언급한 바와 같이 공유 메모리 버퍼는 sm[n+1][n+1][n+1] 크기로 설정되며, 각 원소는 변형 전과 변형 후의 위치값을 모두 저장한다. 변형 전 위치를 저장하는 것은 샘플링이 변형체 계산 후에 이루어지기 때문이다. 변형 후의 위치값은 물리 엔진에서 계산된 좌표 이동값을 볼륨 데이터 형식으로 저장하거나, 지정된 움직임(평행이동 및 회전)이 있다면, 해당 움직임 값을 수식을 통해 계산하여 사용한다.

각 스레드는 공유 메모리의 원소에 값을 입력하며, 공유 메모리의 크기가 스레드 수의 배수로 나누어 떨어지지 않으므로, 스레드의 작업은 공유 메모리 범위 내에서 한정된다.

각 스레드는 하나의 셀을 처리하고, 하나의 셀은 5개의 사면체로 분해되어 각 사면체에 대한 연산을 순차적으로 수행한다. AABB 내부의 격자점들을 후보로 선정하고, 각 격자점마다 사면체 정점에 대한 무게중심 좌표를 계산하여 사면체 포함 여부를 판정한다.

복셀의 좌표(x_1)를 사면체 정점 좌표들(X_1)로 이루어진 행렬의 역행렬과 곱하여 무게중심 좌표를 얻는다. 무게중심 좌표는 아핀 변환 전후의 값이 동일하므로, 무게중심 좌표와 변환 전 사면체 정점 좌표들(X_0)을 가중 평균하여 변환 전 좌표(x_0)를 구하고, 해당 위치에서 텍스처 샘플링을 수행한다. 그 결과를 출력 볼륨 데이터에 저장한다.

5.3 블록 크기 결정 방법

이번 절에서는 블록 크기 설정이 전체적인 시스템 성능에 미치는 영향을 설명한다. 일반적으로 블록 크기를 작게 설정하는 경우, 투명하다고 판정되는 블록의 수가 증가하고 전체 블록 중 계산을 생략하는 블록의 비율이 상승하는 이득이 있다. 그러나 이는 전체 블록의 개수 증가로 인한 관리 비용 증가를 초래한다. 블록 크기를 크게 설정하는 경우, 이와 반대의 효과가 있어 블록 크기를 적절하게 설정하는 것은 중요하다.

GPU 구현을 고려할 때, 블록 크기에 따라 각 블록의 레지스터와 공유 메모리의 사용량이 달라진다. 3.1절에서 언급된 바와 같이 블록에 포함된 좌표 데이터는 공유 메모리에 저장되기 때문에, 레지스터 및 공유 메모리의 한계 내에서 블록의 크기를 결정해야 한다.

예를 들어, 블록 한 변의 길이 n을 4로 설정하면, 한 스레드 블록은 64개의 스레드로 구성된다. 한 스레드 블록이 사용하는 공유 메모리는 $(4+1)^3 \times 2(X_0, X_1) \times 3(x, y, z) \times 4(bytes) = 3{,}000$ bytes가 필요하다. 64KB의 하드웨어 공유 메모리를 기준으로 이론적으로 21개의 스레드 블록을 사용

할 수 있지만, SM(Streaming Multiprocessor)에 할당 가능한 최대 블록 수는 16개로 제한되므로 실제로 활성화할 수 있는 스레드 수는 64×16=1,024개이다.

만약 n을 2로 줄이면, 각 스레드 블록은 8개의 스레드로 구성되며, 더 적은 공유 메모리를 사용하여, 8×16=128개의 스레드만 운영할 수 있다. 공유 메모리 사용량은 줄어들지만, 할당 가능한 블록 수는 여전히 16개로 제한되어 있기 때문이다. 많은 스레드를 운영하기 위해서는 스레드 블록 하나의 스레드 개수가 충분히 많아야 한다.

반면, n을 8로 설정하면, 한 스레드 블록은 512개의 스레드로 구성되며, 사용되는 공유 메모리는$(8+1)^3 \times 32(bytes) = 17,496$ bytes이다. 경우 활성 블록 수는 3개로 제한되며, 총 활성화할 수 있는 스레드 수는 512×3=1,536개로, n이 4일 때의 1,024개에 비해 더 많다.

n을 16으로 설정하면, 한 스레드 블록은 4,096개의 스레드로 구성되며, 사용되는 공유 메모리는 $(16+1)^3 \times 32(bytes) = 117,912$ bytes로, 64KB의 공유 메모리 한계를 초과하게 되어 커널 런치 오류가 발생한다. 따라서 병렬성을 효율적으로 강화하기 위해서는 충분히 큰 블록 크기를 설정하여 공유 메모리를 상대적으로 절약하고 동시에 사용 가능한 블록 수를 고려해야 한다.

본 연구에서는 볼륨 블록이 하나의 스레드 블록에 대응하도록 설정하여, 볼륨 블록 크기를 충분히 크게 조정하여 블록 간 인접 비율을 낮춤으로써 공유 메모리 사용을 최적화한다. 그러나 공유 메모리의 크기 제한, 레지스터 사용량의 한계, 그리고 하나의 스레드 블록에서 활용 가능한 스레드 수에 의해 지나치게 큰 블록은 사용이 제한된다. 이러한 조건들을 종합적으로 고려하여 본 연구에서는 n의 적절한 크기를 8로 설정하였다. 이 크기는 GPU의 하드웨어 자원을 최대한 활용하면서도 효율적인 메모리 관리를 가능하게 하는 최적의 조건을 제공한다.

VI. 실험 및 결과

6.1 실험 설정

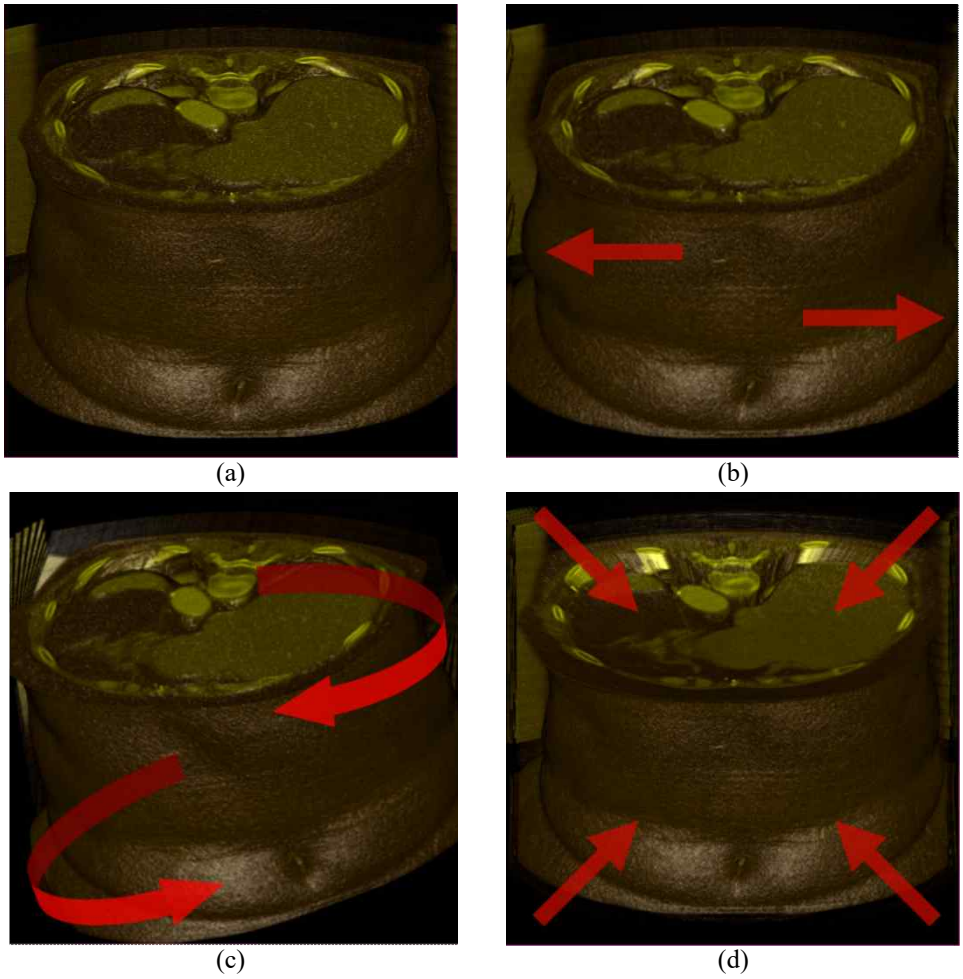
프로그램 개발은 visual studio와 CUDA를 이용한 병렬 프로그램을 이용하여, c++로 제작하였다. 렌더링은 기존의 방법을 수정 없이 사용하였으며, 레이 캐스팅(Levoy, M., 1988)을 사용하고 CUDA (Kim, J., 2019)를 이용하여 병렬 처리하였다.

실험은 GeForce GTX 1650 Mobile이 장착된 노트북과 GeForce RTX 2080, GeForce rtx 3800이 장착된 데스크탑 총 세 가지 컴퓨터에서 수행하였다. 실험에 사용된 볼륨 데이터는 익명화된 의료 이미지 CT 데이터였으며, 세부 사항은 [표 6-1]에 기재하였다.

[표 6-1] 의료 영상 CT 데이터 크기 및 용량

	데이터 크기	용량
복부	$512 \times 512 \times 300$	150MB
폐	$512 \times 512 \times 316$	158MB
대장	$512 \times 512 \times 141$	70.5MB
하체	$512 \times 512 \times 600$	300MB

본 연구에서 변형 결과가 존재한다고 가정하고 병렬 리샘플링을 통해 볼륨 데이터를 생성하는 데 중점을 두었다. 따라서 별도의 물리 기반 변형을 수행하지 않고 간단한 사전 계산된 공식으로 데이터를 변형하였다. 실험을 위해 생성한 세 가지 변형은 다음과 같다.



[그림 6-1] 데이터 변형 (a) 변형 전; (b) 웨이브; (c) 트위스트; (d) 버블.

웨이브 :

$$p' = p + \sin(p \cdot z + t) \cdot x \qquad \text{수식11}$$

트위스트 :

$$p' = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} (p - c) + c \qquad \text{수식12}$$

$$\theta = \sin t \cdot (p - c) \cdot z$$

버블 : $$p' = (1 + \frac{\sin t}{|p-c|}) \cdot (p-c) + c$$ 수식13

웨이브 변형([그림 6-1]의 (b))은 x축 방향으로 횡단 이동한다. 트위스트 변형([그림 6-1]의 (c))은 z축을 중심으로 비틀기와 회전을 수행한다. 마지막으로 버블 변형([그림 6-1]의 (d))은 구형으로 표현되며 지역적인 팽창 및 수축이 발생한다. 변형되지 않은 위치를 p라고 가정하고, 볼륨 데이터의 중심점을 c, 현재 시간을 t, 변형된 위치를 p'라고 할 때, 각 변형은 다음과 같이 표현된다. 또한, x, y, z는 각 축 방향의 단위 벡터이다.

다음 절에서는 제안 방법의 효과를 다양한 테스트 변형을 통해 보인다. 실험 결과를 효과적으로 나타내기 위해, 실험 순서는 다음과 같이 수행하였다. 먼저, 6.2절에서는 3.2절에서 설명된 효율적인 무게중심 보간 방법의 결과를 제시하고, 이어서 3.1절에서 제안된 샘플링 방법의 효과를 보인다. 이후 6.3절에서는 중첩 블록 방법과 공유 메모리 활용의 향상 정도를 보인다.

6.2 보간을 이용한 효율적인 리샘플링 방법 실험

위 실험은 GeForce GTX 1650이 장착된 노트북과 GeForce RTX 2080이 장착된 데스크탑 컴퓨터에서 수행되었다.

6.2.1. 효율적인 무게중심 보간 실험

3.2절에서 설명된 무게중심 보간 방법의 효율성을 보이기 위해, 역행렬 계산에 대한 다양한 방법을 분석하였다. 복부 데이터를 500프레임을 실행하는데 소요된 평균 시간을 측정하였다. [표 6-2]에서는 리샘플링 시간만 표시되며, 렌더링 시간은 별도로 측정하였다. 데스크탑에서의 평균 렌더링 시간은 1ms 미만으로 측정되었고, 노트북에서는 대략 14ms로 측정되었다. 이는 실시

간 시각화를 가능하게 할 정도로 충분히 빠른 속도이다.

[표 6-2] 역행렬 계산 방법에 따른 리샘플링 측정 결과

	데스크탑				노트북			
변형	4d double (a)	3d double (b)	3d float (c)	(a)/(c)	4d double (a)	3d double (b)	3d float (c)	(a)/(c)
웨이브	348.14	150.55	22.55	15.43x	892.85	360.26	60.40	14.78x
트위스트	376.11	166.57	26.56	14.16x	989.42	415.30	98.55	10.03x
버블	385.36	168.49	17.44	22.09x	940.06	402.53	58.39	16.09x

다양한 역행렬 계산 방법에 따른 소요 시간을 측정한 결과, [표 6-2]에 나타난 것처럼 실험(a)의 실행 속도가 가장 느리다. 이는 4×4 행렬의 역을 계산하는 데 상당한 계산이 필요하기 때문이다. 제안 방법을 이용하여 정점을 원점으로 평행이동하면 행렬 크기를 4×4에서 3×3으로 줄일 수 있다. 실험(b)에서는 배정도 부동 소수점을 사용하여 3×3 행렬의 역을 계산하므로 계산량이 크게 줄었다. 그 결과, 실험(a)와 비교하여 속도가 두 배 이상 향상되었다. 3.2절에서 설명한 바와 같이, 단정도 부동 소수점을 사용한 실험(c)의 경우, 실험(b)와 비교하여 속도가 4배에서 6배 향상되었다. 제안된 방법(c)과 기존 방법(a)의 실행 시간을 비교하면, 성능이 10배에서 22배 향상되었다.

제안 방법으로 역행렬 계산 속도를 향상하였으나, 데스크탑과 노트북에서 개선 정도에 다른 패턴이 관찰되었다. 데스크탑에서는 14배에서 22배, 노트북에서는 10배에서 16배 향상되었다. 일반적으로 노트북의 그래픽 메모리는 에너지와 열 문제로 인해 데스크탑에 비해 성능이 낮다. 따라서 노트북에서는 메모리 읽기/쓰기 병목 현상이 더 심각하다. 실험 (a)의 경우, 대부분의 시간이 산술 연산으로 구성된 역행렬 계산에 소요된다. 또한 실험(c)의 경우, 역행렬 계산이 최적화되어 줄어들면서 리샘플링과 같은 메모리 연산이 중요해진

다. 따라서 실험(c)의 성능 향상이 노트북에서 다소 감소하였다. 따라서 고속 메모리를 갖춘 하드웨어에서 제안된 방법의 효과가 더 클 것으로 예상된다.

웨이브의 경우, 계산이 더 간단하기 때문에 실험(a)에서 트위스트나 버블보다 약간 빠르다. 웨이브에 복잡한 명령어를 추가하면 웨이브의 속도가 트위스트와 버블과 비슷해진다. 총 실행 시간에는 사전에 정의된 변형의 계산 시간, 역행렬 계산 시간, 텍스처 샘플링을 사용한 데이터 생성 시간이 포함된다.

제안 방법으로 생성된 출력 영상에 오류가 없는지 확인하기 위해 기존 방법과 제안된 방법을 사용하여 생성된 출력 영상을 비교하였다. 제안 방법과 기존 방법을 사용한 출력 영상의 경우, 차이 값이 정확히 0이었다. 따라서, 사면체를 원점으로 이동시켜 좌표값을 작게 유지하면, 작은 셀과 단정도 부동소수점을 사용하더라도 역행렬을 충분한 정밀도로 계산할 수 있다.

6.2.2. 좌표 보간을 이용한 효율적 샘플링 실험

3.1절에서 제안된 샘플링 방법을 적용하여 성능 향상을 분석한다. [표 6-2]의 실험(c)에서 가장 높은 성능을 보인 3×3 단정도 부동 소수점 행렬 계산 방법을 [표 6-3]에 일반적으로 적용했다. 해당 실험 또한 500프레임에 대한 평균 시간을 측정하였다. Aguila 방법(a)은 한 셀에 대해 샘플링된 여덟 개의 값에서 가중 평균을 얻었지만, 제안 방법(b)은 원본 볼륨의 좌표를 가중 평균하여 출력 복셀 당 한 번만 샘플링한다. 제안 방법을 실행했을 때, 변형에 따라 속도가 10-20% 개선되었다. 성능 향상이 현저하지는 않았지만, 이미 최적화된 작업에 추가적인 성능 개선을 제공했다는 점에서 의미가 있다.

[표 6-3] 보간 방법에 따른 리샘플링 측정 결과

	데스크탑			노트북		
변형	Aguilera 방법 (a)	제안 방법 (b)	(a)/(b)	Aguilera 방법 (a)	제안 방법 (b)	(a)/(b)
웨이브	22.55	19.75	1.14x	60.40	51.27	1.17x
트위스트	26.56	23.82	1.11x	98.55	90.78	1.09x
버블	17.44	14.08	1.23x	58.39	47.18	1.23x

위 실험에서 성능 향상 정도는 변형 유형에 따라 달랐다. 버블의 경우, 데이터의 일부만 변형되어 데이터 재사용성이 높았다. 반면, 트위스트의 경우 변형이 복잡하여, 트위스트를 사용하여 여러 스레드가 변형된 점들을 저장할 때, 회전으로 인해 저장해야 할 메모리 주소들이 연속적이지 않아 메모리 접근의 지역성과 효율성이 감소했다. 정리하면, 제안 방법은 지역 범위 변형에서 더 효과적이었다. 의료 시뮬레이션 또한 인체 데이터의 일부를 당기거나 절개하는 것과 같이 지역적으로 변형이 일어나므로 제안 방법은 일반적인 의료 수술 시뮬레이션에 적합하다.

본 연구에서 실행 시간은 각 셀에 대해 동일한 작업을 수행했기 때문에, 뼈와 연조직 배열과 같은 밀도 값의 분포와는 독립적이나, 실행 시간은 변형 패턴과 볼륨 데이터의 크기와 관련이 있다. 각각의 변형을 다양한 데이터에 적용했을 때, 실행 시간은 데이터 크기와 비례 관계를 보였다. [표 6-4]는 다양한 데이터에 웨이브 변형을 적용한 결과를 제시한다. 폐, 대장, 다리 데이터에 대한 렌더링 결과는 [그림 6-2]에서 보인다.

[그림 6-2] 렌더링 결과 (a) 폐; (b) 대장; (c) 다리.

[표 6-4] 의료 데이터의 보간을 통한 리샘플링 측정 결과 (웨이브 변형)

데이터	Aguilera 방법	제안 방법
복부(300)	60.40	51.27
폐(316)	63.53	54.87
대장(141)	29.37	24.46
다리(600)	149.37	127.19

6.3 중첩된 블록을 이용한 가속화 방법 실험

6.3절 실험은 GeForce rtx 3800을 장착한 데스크탑에서 수행되었다. 본 연구에서는 렌더링 속도는 1ms 이내로 이미 충분히 가속화되어, 이에 대한 분석은 수행하지 않았다. 대신 본 연구의 주요 목적인 리샘플링 가속화에 실험의 초점을 맞추었다. 다양한 변형 조건과 서로 다른 크기와 분포의 데이터에 대해 병렬 리샘플링 속도를 비교하기 위해 VR과 ISO 두 가지 다른 렌더링 방법을 활용하여 성능을 비교하고자 하였다.

[표 6-5] 병렬 리샘플링 측정 결과(VR)

	기존 방법	블록 가속화 방법 적용				
		블록만 적용 (A)	마진(B)	SM 활용		생략 비율
				SM(C)	마진(D)	
복부	21.15ms	5.03ms (4.20x)	5.69ms (3.71x)	4.14ms (5.10x)	4.96ms (4.26x)	84.18%
머리	38.21ms	20.40ms (1.87x)	22.21ms (1.72x)	16.34ms (2.33x)	17.63ms (2.16x)	49.06%
하체	123.95ms	20.70ms (5.98x)	26.15ms (4.73x)	15.34ms (8.08x)	20.13ms (6.15x)	83.86%

VR 렌더링의 경우, [표 6-4]의 복부 데이터를 기존 방법으로 병렬 리샘플링하면 21.15ms가 소요된다. 본 논문에서 제안한 블록 가속화 방법을 적용하면, 4.20배 가속화된 5.03ms가 소요된다. 머리 데이터의 경우, 기존 38.21ms에서 1.87배 가속화된 20.40ms 소요된다. 하체 데이터의 경우, 기존 123.95ms에서 5.98배 가속화된 20.70ms 소요된다. 블록 가속화 방법은 리샘플링 블록의 최댓값이 전이 함수의 임곗값보다 작은 경우 블록을 생략하므로, 임곗값이 증가함에 따라 생략 비율이 증가한다. 머리 데이터의 경우, 근육과 뼈를 관찰하므로 임곗값이 낮아 생략 비율이 49.06% 정도이며, 기존 방법에 비해 약 1.87배 가속화된 것을 확인할 수 있다.

생략 비율은 관찰하고자 하는 임곗값에 따라 달라지며, 이는 성능과 강한 비례성을 보인다. 복부 데이터의 경우, 골격과 일부 혈관만 관찰하므로 임곗값이 높아 생략되는 비율도 84.18%로 매우 높으며, 기존 방법에 비해 약 4.20배 가속화된 것을 확인할 수 있다. 하체 데이터의 경우, 생략되는 블록의 비율이 83.86%로 머리 데이터와 유사하지만 약 5.98배 가속화된 것을 확인할 수 있다. 이는 제안한 블록 가속화 방법이 큰 데이터에서 더욱 효과적인

성능 향상을 가져올 수 있음을 나타낸다. 이 내용은 뒤에서 더 자세히 다룬다 [표 6-7].

[표 6-6] 병렬 리샘플링 측정 결과(ISO)

	기존 방법	블록 가속화 방법 적용				
		블록만 적용 (A)	마진(B)	SM 활용		생략 비율
				SM(C)	마진(D)	
복부	21.15ms	5.05ms (4.18x)	5.74ms (3.68x)	4.16ms (5.08x)	4.86ms (4.35x)	84.34%
머리	38.21ms	7.87ms (4.85x)	10.53ms (3.62x)	6.61ms (5.78x)	7.62ms (5.01x)	84.34%
하체	123.95ms	20.27ms (6.11x)	26.35ms (4.70x)	15.35ms (8.07x)	19.86ms (6.24x)	83.95%

ISO 렌더링의 경우, 특정 값을 가시화하므로 해당 블럭의 최솟값이 임곗값보다 크거나 최댓값이 임곗값보다 작을 경우, 불필요한 리샘플링을 방지하기 위해 즉시 반환 처리한다. 이 접근 방식은 리샘플링 횟수를 감소시키고 시간을 단축시키는 효과를 가져온다. 특히, 복부 및 하체 데이터와는 달리 임곗값이 낮은 머리 데이터의 경우, ISO 렌더링을 적용함으로써 높은 임곗값까지 추가적으로 생략되어 VR 렌더링 시 49.06%였던 생략 비율이 84.34%로 상승하며, 더 높은 성능 향상을 보인다.

특별한 고려 없이 블록을 단순히 적용할 경우, 약간의 노이즈가 발생할 수 있다. 이 문제를 해결하기 위해 제안된 마진 블록 기반 가속화 방법은 블록에 마진 값을 2로 설정함으로써 노이즈를 상당히 감소시키는 동시에 가속화 효과를 유지할 수 있었다. [표 6-6]의 (C)와 (D)에서 보듯이, 마진을 적용한 결

과 속도 저하는 미미하였으며, 동시에 이미지의 품질도 개선되었다. 따라서 이 기법은 성능 저하를 최소화하면서 이미지 품질을 개선할 수 있는 유용한 접근법으로 평가된다.

이제부터는 제안한 공유 메모리 활용 방법을 사용하여 얻은 성능 향상에 대해 설명한다. 제 5장에서 제안된 공유 메모리 활용 기법을 적용했을 때, 각 와프가 사용하는 레지스터의 수가 감소하여 활성 와프의 수 증가하고 시스템의 전반적인 효율성이 향상되었다. 자주 사용되는 데이터에 대한 접근성이 향상됨에 따라 생략 비율과는 독립적으로 각각의 데이터 세트에 대해 VR 및 ISO 렌더링 기법을 적용할 경우 일관된 추가적인 성능 향상이 관찰되었다. [표 6-5]와 [표 6-6]에 따르면, 블록을 생략하는 경우, 공유 메모리의 성능 향상은 (C)/(A) 비율로, 각각 1.21배, 1.24배. 1.34배로 나타났다. 추가적으로, SM의 효과를 더욱 정밀하게 분석하기 위하여 블록을 건너뛰지 않고, 공유 메모리 기능만을 활용하여 시간을 측정한 결과, 복부, 머리, 하체 데이터 세트에 대해 각각 17.15ms, 30.42ms, 101.69ms의 처리 시간이 기록되었다. 이는 공유 메모리를 활용할 때 각각 1.23배, 1.25배, 1.24배의 성능 향상을 보였다. 이러한 결과는 데이터의 종류 및 분포, 생략 비율의 유무와 관계없이 공유 메모리의 활용이 1.2배에서 1.3배 사이의 일관된 추가 성능 향상을 제공함을 시사한다.

SM 기법이 일관적으로 약 1.2배의 성능 향상을 보이는 이유는 생략 판단 과정이 효율적으로 수행되어 소요되는 시간이 거의 0에 가깝기 때문이다. 각 블록의 성능 향상이 일정하며, 전체 시스템의 성능은 처리되어야 하는 블록들의 성능 향상을 단순히 합산한 결과이다. 추가적으로 생략에 소요되는 시간은 거의 없으므로, 전체적인 성능에 크게 영향을 미치지 않는다. 생략에 소요되는 시간이 길 경우, 데이터의 분포에 따라 성능 향상의 정도에 큰 차이가 발생할 수 있다. 그러나 높은 생략 비율과 낮은 생략 비율을 가진 경우에도 관찰된 성능 향상은 일정하게 나타난다. 이는 SM 기법이 다양한 조건에서도 일관된 성능 향상을 제공함을 시사한다.

[표 6-7] 데이터 크기에 따른 리샘플링 측정 결과

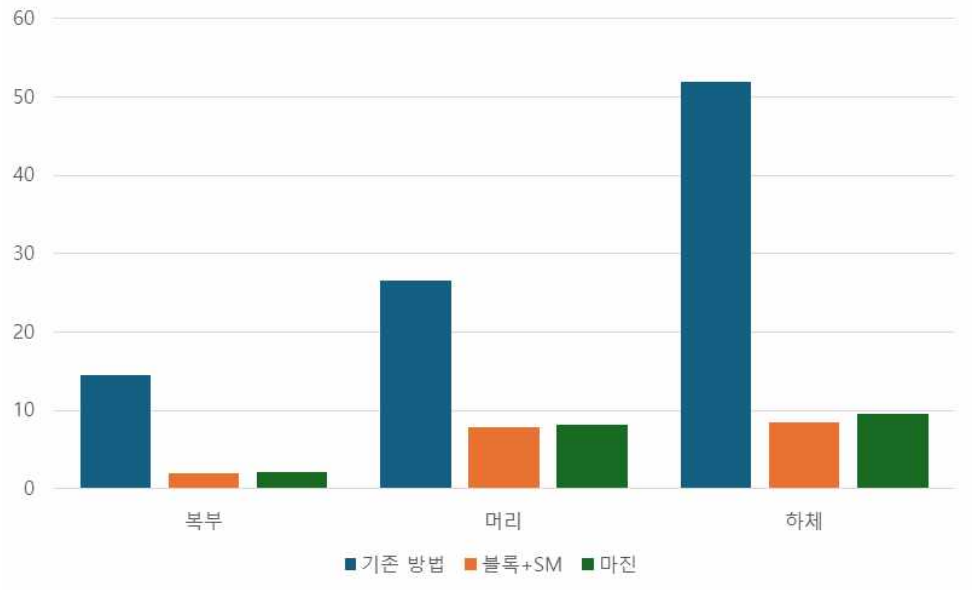
	기존 방법	VR			ISO		
		블록만 적용 (A)	SM 활용		블록만 적용 (A)	SM 활용	
			SM (C)	마진 (D)		SM (C)	마진 (D)
복부 (300)	21.15ms	5.03ms (4.20x)	4.14ms (5.10x)	4.96ms (0.83x)	5.05ms (4.18x)	4.16ms (5.08x)	4.8ms (0.85x)
복부 (1200)	114.41ms	22.01ms (5.19x)	16.71ms (6.84x)	22.05ms (0.75x)	21.96ms (5.20x)	16.46ms (6.95x)	22.04ms (0.74x)

[표 6-7]은 데이터 크기에 따른 성능 향상을 분석하기 위해 설계되었다. 복부(300) 데이터를 네 번 복사하여 생성된 복부(1200) 데이터를 사용하여, 데이터 분포와 같은 다른 변수들을 통제하고 데이터 크기의 영향만을 분석하였다. 복부(1200) 데이터에서 병렬 리샘플링 시간은 기존 114.41ms에서 22.01ms로 감소하여 약 5.19배 향상을 보였다. 이는 복부(300)의 4.20배 향상률과 비교할 때, 더 높은 성능 향상을 나타낸다.

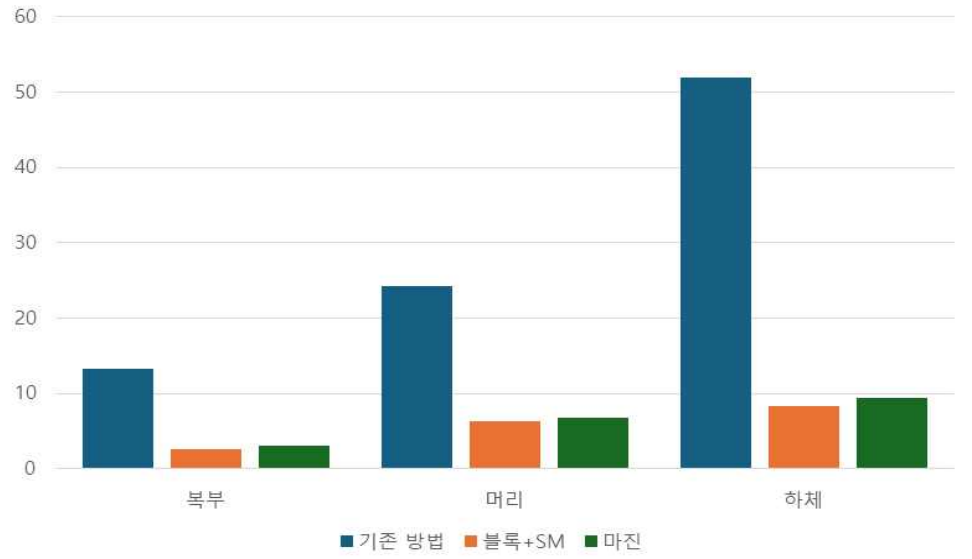
시간 측정에는 병렬화 작업 외에도 커널 런치 비용과 파라미터 복사와 같은 순차적 작업에도 소요되며, 이러한 고정 비용은 데이터 크기가 작은 경우 속도 향상의 속도 향상의 한계를 초래한다. 결과적으로 제안된 블록 기반 가속화 방법은 큰 데이터에서 더욱 효과적인 성능 향상을 제공한다. 실험 결과는 미리 정의된 공식에 따라 변형되었으며, 추가적으로 다양한 물리 엔진을 사용한 결과는 [표 6-8]에 보인다.

구체적으로, [표 6-8]은 체인메일과 PBD를 사용한 다양한 물리 시뮬레이션의 병렬 리샘플링 속도를 측정하고 비교하였다. 제안된 방법이 다양한 물리 엔진과 결합하여 효과적으로 작동함을 보여주기 위해 다음 실험을 수행하였다.

[표 6-8] 다양한 물리 시뮬레이션의 병렬 리샘플링 속도



(a) 체인메일



(b) PBD

본 연구에서는 다양한 물리 엔진에 따른 움직임 양상의 변화가 전체 시스템의 성능 향상에 어떤 영향을 미치는지를 분석하였다. 추가적으로, 해당 과

정에서 다른 모듈과의 호환성이 높음을 확인할 수 있었다.

[표 6-8]의 (a) 복부 데이터의 경우, 제안된 공유 메모리 활용 방법과 블록 기반 가속 방법(블록+SM)을 적용할 때, 병렬 리샘플링 시간이 기존 14.54ms에서 1.91ms로 대폭 감소하여 약 7.6배의 성능 향상을 보인다. 머리 데이터의 경우, 기존 26.54ms에서 제안한 방법으로 7.89ms로 감소하여 약 3.36배의 성능 향상을 보인다. 하체 데이터의 경우 51.94ms에서 9.52ms로 감소하여 대략 5.45배의 성능 향상을 보인다.

그러나 블록+SM 방법은 일부 출력 영상의 오류를 유발할 수 있으므로, 마진 크기를 2로 설정하여 이 문제를 해결하였다. 이에 따라 복부 데이터에서 병렬 리샘플링 시간이 블록+SM 방법 대비 0.25ms 증가하지만, 병렬 리샘플링 시간은 여전히 2.16ms에 불과하다. 다른 데이터에 대해서도 대략 1ms 내외의 리샘플링 시간이 증가한다. 제안된 방법을 통해 화질 손실 없이 3.26배에서 6.73배의 성능 향상을 달성하였다.

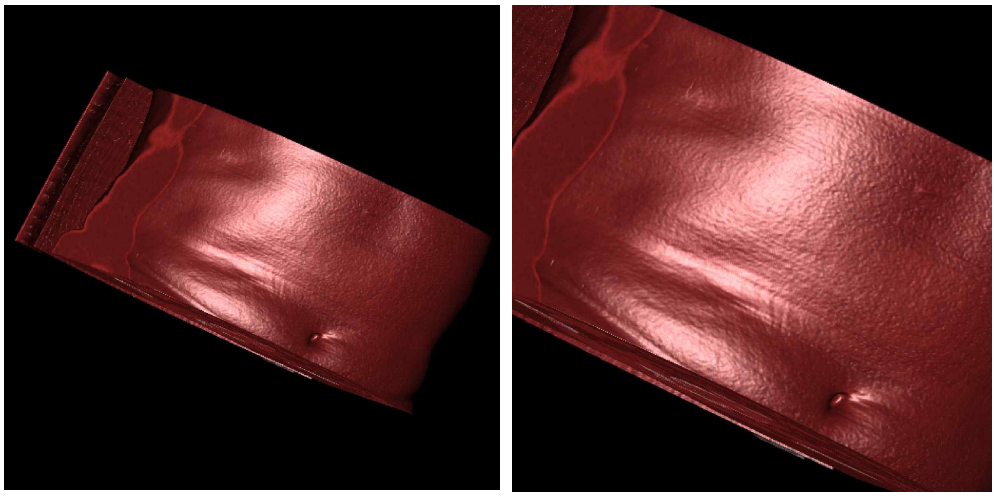
PBD를 적용한 결과도 유사한 성능 향상을 보였다. [표 6-8]의 (b) 복부 데이터의 경우, 병렬 리샘플링 시간이 기존 13.27ms에서 2.59ms로 감소하여 약 5.12배의 성능 향상을 보인다. 머리와 하체 데이터는 각각 3.84배와 6.26배의 향상을 보인다. 이러한 결과는 전반적으로 3.84배에서 6.26배 사이의 성능 향상을 나타내며, 리샘플링 시간의 단축이 전체 시스템의 성능을 크게 개선됨을 보인다.

실험 결과 분석을 통해 본 연구에서는 리샘플링 횟수를 단축함으로써 전체 시스템의 동작 시간을 크게 감소시킬 수 있었다. 본 연구에서 개발한 체인메일 알고리즘의 연산 시간은 복부 데이터에서 42.8ms, 머리 데이터에서 90.1ms, 하체 데이터에서 258.7ms로 측정되었다. 체인메일의 해상도는 볼륨 데이터의 해상도와 동일하며 GPU로 구현되었다. 복부 데이터에 적용했을 때, 물리 엔진을 포함한 시스템의 동작 시간이 기존 57.34ms에서 44.96ms 감소하여 약 1.27배의 성능 향상을 보였다. 머리, 하체 데이터의 경우, 전체 시스

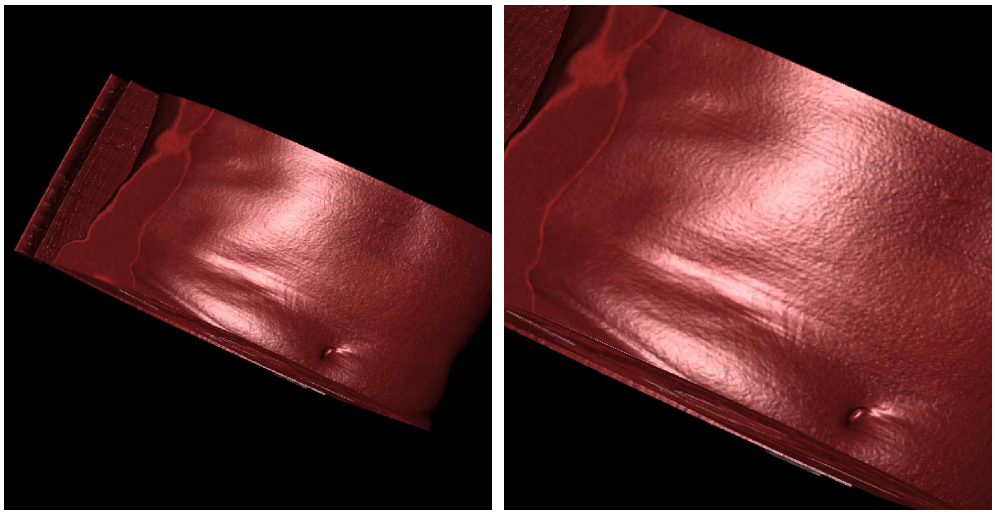
템은 약 20% 이상의 성능 향상을 보인다. 현재 병목 지점인 물리 엔진의 성능이 추가적으로 개선될 경우, 성능 향상의 폭은 더 커질 것으로 예상된다.

PBD의 연산 시간은 복부 데이터에서 4.75ms, 머리 데이터에서 4.43ms, 하체 데이터에서 5.68ms로 측정되었다. 단, 본 연구에서 사용된 PBD의 시뮬레이션 해상도는 128^3으로 체인메일 512^3보다 작다는 점을 고려해야 한다. 물리 엔진의 처리 속도가 빠른 경우, 병목 현상은 주로 병렬 리샘플링에서 발생하게 되며, 본 연구에서는 병렬 리샘플링 시간을 단축시키므로, 성능 향상을 이루었다.

이제부터는 블록 기반 가속 방법을 적용했을 때 발생할 수 있는 파임 현상을 다룬다. 블록 기반 가속 방법은 블록의 최댓값과 최솟값을 기준으로 데이터가 특정 범위에 포함되지 않는 경우 생략 처리한다. 그러나, 인접 블록의 보간 과정에서 왜곡된 값을 활용하게 되고, 결과적으로 데이터의 노말값이 왜곡되는 문제가 발생한다. 이러한 왜곡은 [그림 6-3]에서 보이듯, 볼륨 데이터에 파임 현상과 같은 노이즈를 유발한다.



[그림 6-3] 마진 적용 전.



[그림 6-4] 마진 적용 후.

4장에서 언급한 바와 같이, 판단 범위를 넓히기 위해 인접한 값들을 포함하여 범위를 판단하고 생략할 수 있도록 마진을 설정하였다. 이러한 설정의 속도 차이를 비교하였다[표 6-5]. 마진을 설정하였을 때와 하지 않았을 때의 속도 차이는 미미하였다. 또한, 마진이 설정되지 않은 경우 발생하였던 파임 현상은 [그림 6-4]에서 보듯이 해소되었다. 이러한 결과는 마진 설정을 통해 인접 값의 영향을 고려함으로써 파임 현상을 효과적으로 해결할 수 있음을 보인다.

VII. 결론

본 논문에서는 볼륨 기반 변형체 생성 기법인 병렬 리샘플링을 바탕으로 효율적으로 가속화하기 위한 방법을 제안하였다.

현대의 변형 모델링에서 정교한 움직임을 구현하기 위해 셀의 크기가 작아짐에 따라 셀의 수가 크게 증가한다. 따라서 기존 방법에서 시간이 많이 소요되는 텍스처 샘플링과 역행렬 계산 과정을 개선하는 방법을 제안하였다. 이후, 가시화되지 않는 불필요한 영역의 리샘플링을 최소화하기 위해 블록 기반 가속화 기법을 제안하여 데이터의 일부분만을 선택적으로 리샘플링할 수 있도록 하였다. 또한, 블록 활용 시 발생할 수 있는 화질 손상 문제를 마진 개념을 도입하여 해결하였다. 마지막으로 GPU의 병렬 처리를 고도화하여 SM을 활용하여 추가적인 성능 향상을 이루었다.

3장에서는 사면체의 꼭짓점 위치를 활용한 역행렬 계산에서 발생하는 오류를 최소화하기 위해, 작은 행렬 요소를 유지하는 방법을 도입하였다. 본 연구에서 제안된 기법은 각 사면체의 기준점을 원점으로 설정하여, 모든 꼭짓점을 원점으로 평행이동시키는 방법을 통해 4×4 행렬 대신 3×3 행렬을 사용함으로써, 필요한 연산량을 줄여 성능을 크게 향상시켰다. 또한, 텍스처 샘플링의 효율성을 높이기 위해 변형되지 않은 공간의 좌표를 보간하여 출력 복셀에서 텍스처 샘플링을 수행함으로써, 필요한 샘플링 횟수를 줄이고 실행 속도를 개선하였다. 이 방법은 특히 셀 수가 증가할수록 더 큰 성능 향상을 가져왔다. 세 가지 다른 변형 패턴을 적용한 실험을 통해, 변형에 따른 성능 향상을 확인하였으며, 변형 패턴이 상대적으로 간단할 때 성능 향상 정도가 더 높았다. 따라서 제안된 접근 방식은 인체의 일부에서 변형이 발생하는 의료 시뮬레이션에 적합하다. 데스크탑과 노트북과 같은 저성능 컴퓨터에서는 초당 10프레임, 일반 데스크탑에서는 초당 40프레임 이상으로 실시간 속도를 유지할 수 있었다.

4장에서는 블록의 투명성/불투명성을 판단하여 가시화되는 불투명한 블록만을 리샘플링하고, 최종 출력 화면에 영향을 주지 않는 투명한 블록은 생략함으로써 처리 속도를 크게 향상시켰다. 이 방법을 통해 일반적으로 임곗값이 높을수록 가시화되는 영역이 감소하여, 생략 비율이 증가하고 성능이 개선되는 결과를 얻었다. 특히, 볼륨 렌더링에서는 절반 이상의 데이터가 생략됨에 따라 1152장의 큰 데이터의 경우에도 20ms 내에서 실시간 처리가 가능하였다. ISO 렌더링의 경우, 생략되는 블록이 더 많아 성능 향상이 5.08배에서 8.07배로 더욱 높았다. 본 연구에서 제안한 마진은 블록 경계 주변에서 발생할 수 있는 파임 및 화질 저하 문제를 해결한다. 마진은 더 넓은 범위를 리샘플링 하도록 하여 속도 차이는 거의 나지 않으면서 화질 저하를 방지한다.

5장에서는 셀의 경계에 속한 제어점들을 여러 스레드의 레지스터에 중복 저장하기 위해 공유 메모리를 활용함으로써 레지스터의 사용량을 줄이고, SM을 활용하여 각 와프가 사용하는 레지스터의 수를 줄여 활성 와프 수와 시스템의 효율을 증가시켰다. 블록과 마진을 도입하여 충분히 향상된 성능에 SM을 활용하여 약 1.2배의 추가적인 성능 향상을 이끌어냈다. 이러한 방법을 통해 기존 방법에 비해 2.33배에서 8.08배까지 성능이 향상되었다.

본 연구에서는 제안 방법을 통해 화질 저하 없이 속도 향상을 이루었으며, 임상에서 사용되는 크기의 볼륨 데이터에 대해 실시간 변형 가능한 볼륨 시각화를 가능하게 하였다. 볼륨 변환에 사전 정의된 공식을 적용하는 접근은 이 연구의 한계점이지만, 제안된 방법은 다른 변환 방법과의 결합 가능성 덕분에 높은 확장성을 가진다. 또한, 본 연구 결과는 실제 수술에 필요한 절개나 봉합 등의 시뮬레이션을 포함한 시스템 구축에도 활용될 수 있으며, 의료 시뮬레이션 분야에서의 응용 가능성을 크게 확장하고 있다. 향후 연구로는 절개나 봉합과 같이 실제 수술에 필요한 시뮬레이션을 포함한 시스템을 구축하는 것이다.

참 고 문 헌

1. 국외문헌

Nienhuys, H. W., & Frank van der Stappen, A. (2001, October). A surgery simulation supporting cuts and finite element deformation. In International conference on medical image computing and computer-assisted intervention (pp. 145-152).

Rößler, F.; Wolff, T.; Ertl, T. Direct GPU-based Volume Deformation. In Proceedings of the CURAC, Leipzig, Germany, 24-26 September 2008; pp. 65-68.

Levoy, M. (1988). Display of surfaces from volume data. IEEE Computer graphics and Applications, 8(3), 29-37.

Lacroute, P., & Levoy, M. (1994, July). Fast volume rendering using a shear-warp factorization of the viewing transformation. In Proceedings of the 21st annual conference on Computer graphics and interactive techniques (pp. 451-458).

Gibson, S.F. 3D Chainmail: A Fast Algorithm for Deforming Volumetric Objects. In Proceedings of the ACM Siggraph Symp Interact 3D Graph Games 1997, Providence, RI, USA, 27-30 April 1997; p. 149-ff.

Liu, T.; Bargteil, A.W.; O'Brien, J.F.; Kavan, L. Fast simulation of mass-spring systems. ACM Trans. Graph (TOG) 2013, 32, 1-7.

Berndt, I.; Torchelsen, R.; Maciel, A. Efficient surgical cutting with position-based dynamics. IEEE Comput. Graph. Appl. 2017, 37, 24-31.

Gascon, J., Espadero, J. M., Perez, A. G., Torres, R., & Otaduy, M. A. (2013, July). Fast deformation of volume data using tetrahedral

mesh rasterization. In Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation (pp. 181-185).

Aguilera, A. R., Salas, A. L., Perandrés, D. M., & Otaduy, M. A. (2015). A parallel resampling method for interactive deformation of volumetric models. Computers & Graphics, 53, 147-155.

Rodríguez, A., León, A., & Arroyo, G. (2016). Parallel deformation of heterogeneous ChainMail models: Application to interactive deformation of large medical volumes. Computers in Biology and Medicine, 79, 222-232.

Bartelheimer, K., Teske, H., Bendl, R., & Giske, K. (2017). Tissue-specific transformation model for CT-images. Current Directions in Biomedical Engineering, 3(2), 525-528.

Georgii, J., & Westermann, R. (2005). Mass-spring systems on the GPU. Simulation modelling practice and theory, 13(8), 693-702.

Zhang, Y., Zhao, J., Yuan, Z., Ding, Y., Long, C., & Xiong, L. (2010, April). CUDA based GPU programming to simulate 3D tissue deformation. In 2010 International Conference on Biomedical Engineering and Computer Science (pp. 1-5). IEEE.

Fortmeier, D., Mastmeyer, A., & Handels, H. (2013). Image-based palpation simulation with soft tissue deformations using chainmail on the GPU. In Bildverarbeitung für die Medizin 2013 (pp. 140-145). Springer, Berlin, Heidelberg.

Zhang, Z. (2020, November). Soft-Body Simulation With CUDA Based on Mass-Spring Model and Verlet Integration Scheme. In ASME International Mechanical Engineering Congress and Exposition (Vol. 84546, p. V07AT07A025). American Society of Mechanical Engineers.

Bender, J., Müller, M., & Macklin, M. (2017). A survey on position

based dynamics, 2017. Proceedings of the European Association for Computer Graphics: Tutorials, 1-31.

Berndt, I., Torchelsen, R., & Maciel, A. (2017). Efficient surgical cutting with position-based dynamics. IEEE computer graphics and applications, 37(3), 24-31.

Kwon, K., Chae, S., & Shin, B. S. (2011). Anti-aliasing on deformed area using adaptive super sampling during volume ray-casting. Biomedical Engineering Letters, 1(3), 168.

Fortmeier, D. (2016). Direct volume rendering methods for needle insertion simulation (Doctoral dissertation, Zentrale Hochschulbibliothek Lübeck).

Correa, C. D., Silver, D., & Chen, M. (2006). Discontinuous displacement mapping for volume graphics. In VG@ SIGGRAPH (pp. 9-16).

Torres, R.; Rodríguez, A.; Otaduy, M. Hands-On Deformation of Volumetric Anatomical Images on a Touch screen. Appl. Sci. 2021, 11, 9502.

Chen, J.; Tai, K.W.; Chen, W.C.; Ouhyoung, M. Robust Voxelization and Visualization by Improved Tetrahedral Mesh Generation. arXiv 2021, arXiv:2106.01326.

Levoy, M. (1990). Efficient ray tracing of volume data. ACM Transactions on Graphics (TOG), 9(3), 245-261.

Levoy, M. Display of surfaces from volume data. IEEE Comput. Graph. Appl. 1988, 8, 29 - 37.

Kim, J.; Ha, T.; Kye, H. Real-Time Computed Tomography Volume Visualization with Ambient Occlusion of Hand-Drawn Transfer Function Using Local Vicinity Statistic. Healthc. Inform. Res. 2019, 25, 297 - 304.

부 록

부록 A

```
template <typename T>
T Determinant ( ) {
  T n11 = 255.9, n12 = 256.7, n13 = 256.7, n14 = 255.9;
  T n21 = 256.7, n22 = 255.9, n23 = 256.7, n24 = 255.9;
  T n31 = 133.1, n32 = 133.4, n33 = 132.3, n34 = 132.3;
  T n41 = 1, n42 = 1, n43 = 1, n44 = 1;
  T t11 = n23*n34*n42 - n24*n33*n42 + n24*n32*n43 - n22*n34*n43 -
n23*n32*n44 + n22*n33*n44;
  T t12 = n14*n33*n42 - n13*n34*n42 - n14*n32*n43 + n12*n34*n43 +
n13*n32*n44 - n12*n33*n44;
  T t13 = n13*n24*n42 - n14*n23*n42 + n14*n22*n43 - n12*n24*n43 -
n13*n22*n44 + n12*n23*n44;
  T t14 = n14*n23*n32 - n13*n24*n32 - n14*n22*n33 + n12*n24*n33 +
n13*n22*n34 - n12*n23*n34;
  T det = n11*t11 + n21*t12 + n31*t13 + n41*t14;
  return det;
}
```

ABSTRACT

Real-time Visualization of Large Deformable Volume Data using Enhanced Parallel Resampling

Park, ChaiLim

Major in Computer Engineering

Dept. of Computer Engineering

The Graduate School

Hansung University

Virtual medical procedure planning prior to actual medical procedures plays an important role in improving healthcare quality. The recognition of virtual medical procedures as an important and essential technology is driven by the accurate deformation simulation using real human body data, and the suitability of volume-based deformable body modelling for medical procedures involving complex operations such as incisions and suturing. In the past, due to the vast size of volume data, deformation calculations took a significant amount of time, which limited real-world applications. However, recent advances in GPUs and several algorithms(chain-mail, mass-spring) have significantly improved the processing speed and efficiency.

In this paper, we propose four methods that can process large volumes of data in real-time while providing high-quality visualisation results. We improve the time-consuming process of texture sampling and inverse matrix computation in modern deformation modelling. Then, we introduce a block-based acceleration technique to minimise resampling of unnecessary, non-visible regions. Finally, additional speedup was achieved by utilising shared memory. The proposed methodology improves

processing speed without compromising image quality and extends its applicability in medical simulation. This is of great significance as it can lead to the construction of fast and precise medical simulation systems required for real-world surgery.

【Keyword】 massive computing for volume deformation, parallel resampling, GPU parallel computing, Empty-space skipping