GPU 클러스터 및 타일형 디스플레이를 이용한 볼륨 데이터의 고해상도 가시화

이중연
한국과학기술정보연구원 슈퍼컴퓨팅센터
e-mail:jylee@kisti.re.kr

Visualization of Volume Dataset using GPU Cluster and Tiled Display

Joong-Youn Lee
Supercomputing Center,
Korea Institute of Science and Technology Information

요 약

.볼륨 렌더링은 3차원이나 그 이상의 차원의 볼륨 데이터에서 의미있는 정보를 추출해 내어 직관적으로 표출하는 가시화 기법을 말하며 의료영상, 기상학, 유체역학 등 다양한 분야에서 널리 사용되고 있다. 한편, 최근 PC 하드웨어의 급격한 발전으로 과거에는 슈퍼컴퓨터에서나 가능했던 대용량 볼륨 데이터의 가시화가 일반 PC 환경에서도 가능하게 되었다. GPU의 꼭지점 및 픽셀 쉐이더의 수치 계산에 최적화된 벡터 연산으로 빠른 볼륨 가시화를 가능하게 한 것이다. 그러나 GPU의 메모리 용량의 한계로 대용량의 볼륨 데이터를 빠르게 가시화하는 것은 지금까지 어려운 문제로 남아있다. 본 논문에서는 GPU의 텍스쳐 메모리 크기보다 큰 볼륨 데이터를 여러 개의 GPU 메모리에 분산시키고 이를 꼭지점 및 픽셀 쉐이더를 이용하여 빠르게 렌더링하여 타일형 디스플레이에서 고해상도로 가시화하는 시스템을 디자인하고 구현하고자 하였다.

1. 서론

여러 가지 과학적 가시화 기법 중 가장 중요한 위치를 차지하는 볼륨 가시화는 3차원이나 그 이상의 차원의 볼륨 데이터에서 의미있는 정보를 추출해 내어 직관적으로 표출하는 가시화 방법으로 의료영상, 기상학, 유체역학 등 다양한 분야에서 활용되고 있다. 한편, 최근 수년간 PC 그래픽스 하드웨어가 급속도로 발전하면서 전문적인 3D 그래픽스 워크스테이션에서나 가능하였던 고품질, 고성능의 그래픽스 작업이 일반 PC에서도 가능하게 되었다. 특히 고정되어 있던 그래픽스 파이프라인을 사용자가 임의로 수정하여 사용할 수 있도록 한 꼭지점 및 픽셀 쉐이딩(vertex & pixel shading) 기능은 가히 혁명적이라고 할 수 있을 정도로 PC 그래픽스 하드웨어의 성능을 한 단계 끌어올렸으며 CPU와 비교하여 GPU(Graphics Process Unit)라는 새로운 용어를 만들어 내었다[7]. 그러나 GPU를 이용한 렌더링 방법들은 대부분 가시화할 볼륨 데이터를 3차원 텍스쳐로 간주하여 텍스쳐 메모리에 읽어 들이고 렌더링하기 때문에 가시화가 가능한 데이터의 크기가 텍스쳐 메모리의 크기에 제약받는다는 한계가 있다. 현재 최신 PC 그래픽스 하드웨어의 텍스쳐 메모리는 256MB~512MB 수준이기 때문에 볼륨 데이터의 크기 역시 이 정도로 한정되며 여기에 법선 벡터나 팔진트리 데이터 등 고품질/고속으로 영상을 얻기 위한 추가적인 데이터들의 공간을 제외하면 렌더링 가능한 볼륨 데이터의 크기는 더욱 작아지게 된다. 또한 GPU에서 지원하는 렌더링 이미지의 해상도에도 한계가 있기 때문에 이를 초과하는 초고해상도로 이미지를 렌더링하기 위해서는 이미지를 여러 개로 나누어 각 부분별로 따로 렌더링해야 하여 렌더링 속도가 현저히 느려지게 된다. 본 논문에서는 이러한 GPU 기반 볼륨 렌더링의 단점을 극복하기 위해 다수의 GPU를 병렬로 연결한 GPU 클러스터를 이용하여 대용량의 볼륨 데이터를 빠르게 가시화하고자 하였다. 이를 위하여 볼륨 데이터를 미리 나누어 각

렌더링 노드에 분산시켜 병렬로 렌더링하는 Sort-Last 렌더링 기법을 이용하였고, 이렇게 생성된 이미지들을 MPICH를 이용하여 빠르게 컴포지션하여 고해상도로 타일형 디스플레이에서 가시화하였다. 본 논문의 2장에서는 GPU 기반 볼륨 렌더러에 대해 설명하고 3장에서는 GPU 클러스터를 이용한 병렬 볼륨렌더링에 대해 소개한다. 4장에서는 실제 구현 결과를 설명하고 마지막으로 5장에서 결론을 맺는다.

2. GPU 기반의 볼륨 렌더링

2.1. 샘플링

볼륨 데이터에서 적절한 복셀(voxel)을 찾아오는 샘플링 과정은 매우 많은 시간을 필요로 하므로 이를 빠르게 처리하는 것은 주요한 연구 주제 중 하나였다. GPU의 텍스쳐 매핑 기능은 선형보간 또는 삼선형보간을 하드웨어적으로 매우 빠르게 처리할 수 있도록 해주는데, 텍스쳐 매핑을 이용한 볼륨 렌더링은 볼륨 데이터를 텍스쳐로 간주하고 하드웨어 텍스쳐 매핑을 통하여 샘플링을 매우 빠르게 처리하도록 한다[1]. 이렇게 텍스쳐 매핑을 이용하여 샘플링을 하기 위해서는 텍스쳐가 그려질 도형이 필요한데 이를 대리 평면(proxy plane)이라고 한다. 본 논문에서는 간편하게 대리 평면이 영상평면과 평행하도록 하기 위해서 GPU의 꼭지점 쉐이더(vertex shader)를 이용하였는데, 꼭지점 쉐이더에서는 각 꼭지점에 모델뷰 행렬(modelview matrix)과 투영 행렬(projection matrix)를 곱함으로써 각 꼭지점들에 대한 스크린 좌표를 계산하도록 하였다. 여기서, 모델뷰 행렬에 회전 행렬을 곱함으로써 꼭지점이 실질적으로 회전하게 되는데, 대리 평면은 항상 영상평면에 평행을 유지해야 하기 때문에 꼭지점 쉐이더에서 꼭지점에 모델뷰 행렬을 무시하고 투영행렬만 곱하도록 하였고, 대신 각 대리 평면의 꼭지점에 할당되는 텍스쳐 좌표에 모델뷰 행렬을 곱함으로써 볼륨 데이터가 회전하는 것처럼 보이도록 하였다.

2.2. 쉐이딩

샘플링된 복셀들은 전이함수(transfer function)을 통해 색깔 및 투명도가 결정되고 여기에 쉐이딩(shading)을 통해 음영이 입혀지게 된다. 이러한 작업은 프로그래밍이 가능한 쉐이더(programmable shader)를 이용하여 수행된다. 전이함수는 종속 텍스쳐(dependent texture) 기법을 이용하면 쉽게 구현이 가능하다[5]. 본 논문에서는 2차원 텍스쳐를 이용한 2차원 전이함수를 사용하였는데, 복셀값과 복셀의 법선벡터의 크기를 인덱스로 하였다. 법선벡터의 크기가 크면 복셀값이 급격히 변화하므로 물질의 경계부분이라는 뜻이고 그 크기가 작으면 복셀값의 변화가 거의 없는 균일(homogeneous) 영역이라는 뜻이다. 일반적으로 물질의 경계면이 중요하게 인식되므로 이 부분의 투명도를 작게 하고 경계면이 아닌 균일 영역은 상대적으로 덜 중요하므로 투명도를 크게 하였다. 쉐이딩은 퐁의 조명 모델(Phong's illumination model)을 사용하였고, 그래픽스 하드웨어의 픽셀 쉐이딩(pixel shading) 기능을 이용하여 구현하였다. 퐁의 조명 모델을 계산하기 위해서 법선벡터를 복셀값과 함께 3차원 텍스쳐에 저장하였고 이 때문에 텍스쳐의 크기는 본래 볼륨 데이터의 3배 크기가 되었다.

2.3. 속도 향상 기법

본 논문에서는 모든 복셀에 대해 픽셀 쉐이더에서 퐁 쉐이딩을 적용하였는데, 높은 수준의 렌더링 이미지를 얻기 위하여 특별히 반사 광원(specular light)을 포함한 완전한 퐁 쉐이딩을 사용하여 매우 복잡한 연산을 수행하도록 하였다. 이러한 이유로 전체 그래픽스 파이프라인 중 픽셀 쉐이더에서 병목현상을 보여 전체 성능이 저하됨을 알 수 있었다. 이러한 단점을 극복하기 위해서 쉐이딩이 필요 없는 복셀에 대해서는 쉐이딩을 수행하지 않고 건너뛰도록 하여 전체 속도를 향상 시키고자 하였고, 이를 이중 패스 렌더링과 이른 깊이 테스트를 사용하여 구현하였다. 이른 깊이 테스트는 픽셀 쉐이더를 수행하기 전에 미리 깊이 테스트를 하여 테스트를 통과하는 프래그먼트(fragment)에 대해서만 픽셀 쉐이더를 적용하고 통과하지 못한 프래그먼트들은 모두 픽셀 쉐이더 작업을 생략하도록 하는 기법이다. 이러한 이른 깊이 테스트를 이용하여 빈 복셀에 대해서는 퐁 쉐이딩을 수행하지 않도록 하였는데, 이중 패스 렌더링 시 똑같은 대리 도형을 같은 위치에 두 번 그리도록 하고, 처음 그릴 때는 복잡한 퐁 쉐이딩을 적용시키지 않고 단순히 복셀 체크만 수행하여 그 복셀이 비었는지 여부만을 판단한다. 복셀이 비었을 경우에는 깊이 버퍼를 0으로 설정하여 두 번째 렌더링 시 깊이 테스트에서 건너뛰도록 하고 복셀이 비어 있지 않았을 경우에는 깊이 버퍼를 1로 설정하여 두 번째 렌더링 시 복잡한 퐁 쉐이딩을 수행하도

록 하였다. 전체적인 알고리즘은 다음과 같다.

```
Firstpass:
    Draw proxy slice
    Check voxel value in pixel shader
    if (voxel value >= threshold) depth buffer = 1
    elseif (voxel value < threshold) depth buffer = 0
Secondpass:
    Draw proxy slice
    Apply Z test
    if (depth buffer == 0) skip pixel shader
    else do pixel shader
```

그림 1. 이른 깊이 테스트를 이용한 빈공간 건너뛰기 기법 알고리즘

3. GPU 클러스터를 이용한 병렬 렌더링

3.1. 렌더링

GPU를 이용한 볼륨 렌더링은 기존 CPU를 이용한 렌더링 방식에 비해 속도가 매우 빠르다는 장점이 있으나 GPU 메모리의 크기에 한계가 있고 메모리나 하드디스크 등의 다른 저장 장치와의 통신 속도가 매우 느리기 때문에 렌더링 가능한 데이터의 크기에 한계가 있는 단점이 있다. 또한, GPU와 다른 저장 장치와의 통신 속도 중 특별히 쓰기 속도가 매우 느린데, 이러한 특징으로 인해 데이터를 분산시켜 렌더링할 경우 렌더링 과정에서 최대한 각 노드 상호간에 통신을 최소화해야 한다. 이러한 점을 극복하기 위하여 본 논문에서는 렌더링될 데이터를 그래픽스 메모리의 크기에 맞도록 나누어 미리 부분 볼륨 데이터를 생성해 놓고 각 렌더링 노드의 그래픽스 메모리에 분산시켜 놓았다가 렌더링 시점에서는 각 렌더링 노드들 간에 상호 통신이 전혀 없도록 하는 Sort-Last 기법을 통하여 렌더링되도록 하였다. Sort-Last 기법에서는 렌더링될 데이터가 최종 렌더링 이미지의 어느 영역에 위치하던지 상관하지 않고 일단 렌더링한 뒤 최종 컴포지팅 단계에서 그 위치를 맞추도록 한다. 이 렌더링 기법에서는 디스플레이 작업이 수행되기 까지는 각 렌더링 노드들이 서로 독립적으로 동작하기 때문에 기존의 볼륨 렌더링 방식과 마찬가지 방식으로 각 렌더링 노드에서 할당된 데이터를 렌더링하면 된다. 이때, 생성된 렌더링 이미지는 최종 이미지가 아닌 중간 단계의 이미지이므로, 최종 컴포지팅 단계에서 사용할 수 있도록 각 픽셀의 투명도(alpha) 및 깊이(depth) 정보를 버리지 않고 함께 출력하여야 한다.

3.2. 타일형 디스플레이에 컴포지팅

일반적으로 GPU는 한번에 렌더링할 수 있는 이미지의 해상도에 한계가 있다. 또한, 이 한계가 충분히 크다 할지라도, 매우 큰 해상도로 렌더링할 경우에는 그 속도가 많이 늦어지게 된다. 본 논문에서는 4320×2100에 달하는 초고해상도를 지원하는 타일형 디스플레이에 전체 스크린으로 볼륨 데이터를 가시화하고자 하기 때문에 이를 최대한 빠르게 처리하도록 해야 한다. 이러한 점을 극복하기 위해서 GPU에 비해 상대적으로 계산에 여유가 있는 CPU를 이용해 렌더링될 영역을 미리 계산하고 이 영역만 렌더링한 뒤 이 이미지를 잘라서 디스플레이할 컴포지팅 노드로 전송하도록 하였다. 물론, 이러한 계산은 OpenGL 좌표계 변환 연산이므로 GPU를 이용하는 것이 CPU를 이용하는 것에 비해 빠르지만 GPU는 계속해서 렌더링 작업을 하느라 로드가 많이 걸리는 반면, CPU는 거의 로드가 없으므로 이 작업은 CPU로 분산시키고자 하였다. 만약 특정 렌더링 노드가 담당하는 부분 볼륨 데이터가 타일형 디스플레이의 전체 스크린에 할당되도록 크게 디스플레이되게 되면 소용이 없지만, 보통은 전체 타일형 디스플레이 중 일부 영역에만 디스플레이되기 때문에 최대 1/(전체 컴포지팅 노드 수) 만큼의 렌더링 속도의 향상이 있을 수 있다[2]. 이 렌더링 속도 향상 기법은 각 렌더링 노드가 담당한 부분 볼륨 데이터를 직육면체로 간주하고 꼭지점 8개에의 OpenGL 좌표계 변환을 통해 스크린 좌표계(screen coordinates)로 변환하여 그 볼륨 데이터가 위치할 스크린을 미리 알아내는 방법을 이용하였다. 그림 2에서와 같이 각 렌더링 노드는 렌더링할 때, 볼륨 데이터를 스크린 좌표계 상의 해당 위치로 이동시키지 않고 그냥 초기 위치에서 렌더링한다. 그리고 CPU가 계산한 이동할 타일형 디스플레이 상의 스크린 좌표계 값에 따라 렌더링된 영상을 나누고 이를 해당되는 타일형 디스플레이를 담당하는 컴포지팅 노드로 전송하도록 한다. 이러한 작업은 모두 MPICH를 이용하여 구현하였다.

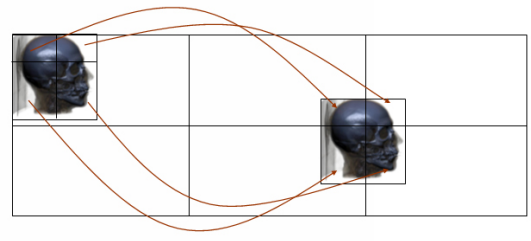

그림 2, 타일형 디스플레이상에의 렌더링

4. 구현 결과

본 논문에서 실험한 볼륨 데이터는 미국 NLM (National Library of Medicine)에서 제작한 Visible Human 데이터중 768개의 슬라이스로 총 384MB의 용량이고, 이를 6개의 64MB 크기로 나뉘어서 렌더링하였다. 구현에 사용된 그래픽스 하드웨어는 NVIDIA의 Quadro FX3000G로 그래픽스 메모리는 256MB이지만 퐁 쉐이딩을 위한 법선 벡터 데이터가 실제 볼륨 데이터의 3배 크기를 필요로 하기 때문에 가용한 메모리 크기 256MB의 1/4인 64MB 크기만큼만 렌더링에 사용하였다. 각 렌더링 노드(GPU)에서는 미리 할당된 데이터를 일반적인 GPU 기반 볼륨 렌더러와 마찬가지 방식으로 렌더링하였다. 이렇게 각 렌더링 노드에서 데이터를 렌더링하면 glReadPixels 함수를 이용하여 RGBA 및 깊이 값을 읽어 들이고 이를 MPICH를 이용하여 디스플레이할 타일을 담당하는 컴포지팅 노드로 전송하였다. 이때, 데이터를 보내는 노드와 받아들이는 노드의 설정을 잘 해야 하는데, 이를 잘못 설정하면 교착상태에 빠지게 되기 때문이다. 본 논문에서는 전체 노드들을 대상으로 미리 이진 트리를 생성하여 각 노드별로 데이터를 교환하도록 하여 교착상태에 빠지지 않도록 하였다. 다만, 컴포지팅에 참여하는 노드들의 수가 2^n의 형태가 아닌 관계로 최적으로 컴포지팅을 수행하지는 못하였다. 이렇게 전송받은 이미지들은 glDrawPixels 함수를 통해 블렌딩하였으며 항상 일정한 순서를 유지하도록 깊이 버퍼값을 비교하며 블렌딩하였다. 총 6개의 노드를 이용하여 컴포지팅하였으므로 3번의 통신 및 블렌딩 작업만으로 전체 컴포지팅 작업이 이루어졌으며 이는 전체 렌더링 시간에 별다른 영향을 미치지 않았다. 이렇게 렌더링된 결과는 4,200X2,100의 해상도를 지원하는 타일형 디스플레이에 가시화하도록 하였고, 렌더링 속도는 평균 초당 1프레임 정도였다.

5. 결론

본 논문에서는 GPU를 이용한 볼륨 렌더링 기법을 활용해서 GPU 클러스터에서 병렬 렌더링을 수행했고, 한정된 크기의 데이터만 렌더링이 가능한 GPU들을 이용하여 대용량 데이터를 비교적 빠르게, 그리고 고해상도로 렌더링하는 것이 가능했다. 그러나 외부 장치와의 속도가 매우 느린 GPU의 특성상 Sort-Last 방식의 볼륨 렌더링만 가능하였다. 렌더링 속도는 초당 1프레임 정도로 그다지 빠르지 않았

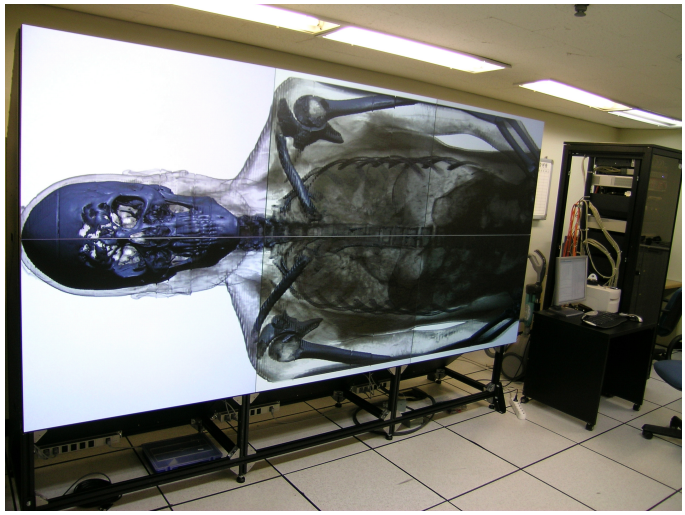

그림 3. 렌더링 영상

는데, 너무 고해상도로 렌더링을 하다 보니 픽셀 쉐이더에서 병목현상이 발생했기 때문이다. 이러한 점을 극복하기 위해서는 하나의 스크린을 여러 개의 GPU로 렌더링하는 방식이 필요하며 PCI-Express 방식의 그래픽스 하드웨어에서는 GPU와 메인 메모리 등의 외부 장치와의 속도가 개선되었으므로 이를 이용하면 해결책을 찾을 수 있을 것으로 보인다.

참고문헌

[1] Akeley, "RealityEngine Graphics", Proceeding on SIGGRAPH 93 Conference, 1993.

[2] Humphreys, Buck, Eldridge, Hanrahan, "Distributed Rendering for Scalable Displays", Proceeding on Super Computing, 2000

[3] Kniss et al., "Interactive Texture-Based Volume Rendering for Large Data Sets", Proceeding on IEEE Computer Graphics and Application, July 2001

[4] Wylie, Pavlakos, Lewis, Moreland, "Scalable Rendering on PC Clusters", Proceeding on IEEE Computer Graphics and Application, July 2001

[5] Engel, Kraus, Ertl, "High-Quality Pre-Integrated Volume Rendering using Hardware-Accelerated Pixel Shading", Proceeding on Eurographics/SIGGRAPH Workshop on Graphics Hardware, 2001.

[6] OpenGL ARB, "OpenGL Programing Guide 4Th Edition", Addison-Wesley, 2003

[7] NVIDIA, "NVIDIA GPU Programming Guide version 2.2.0", 2005