ISBN 89-5884-672-0 98560

GPU 볼륨 렌더링 프로그램 속도 향상 기법

(Acceleration Technique for Volume Rendering Program Based On GPU)

이 중 연 (Joong Youn Lee)
jylee@kisti.re.kr
Visualization Team, Supercomputing Center

한 국 과 학 기 술 정 보 연 구 원
Korea Institute of Science & Technology Information

<제목 차례>

I. 머리말 ········· 1
가. 볼륨 렌더링이란? ········· 1
나. GPU 소개 ········· 2

II. 기존의 GPU 기반 볼륨 렌더링 연구 ········· 4

III. 기본적인 GPU 기반 볼륨 렌더링 알고리즘 ········· 9
가. 샘플링 ········· 9
나. 쉐이딩 ········· 9
다. 컴포지팅 ········· 10

IV. 속도 향상 기법 ········· 11
가. 이른광선단절법 ········· 11
1) 알파 값 읽기 ········· 12
2) 다중패스 렌더링 ········· 12
나. 빈공간무시법 ········· 13
1) 기본 알고리즘 ········· 13
2) Octree Texture (8진트리 텍스쳐) ········· 14

V. 구현 결과 ········· 21

VI. 결론 ········· 24

VII. 참고문헌 ········· 25

<표 차례>

표 V-1. 각 볼륨 데이터의 크기 ········ 21
표 V-2. ERT 구현 방식과 그래픽스 인터페이스에 따른 성능 비교 ········ 22
표 V-3. 각 볼륨 데이터에 대한 윈도우 크기 및 알고리즘 종류에 따른 렌더링 속도 ······ 23

<그림 차례>

그림 I-1. 볼륨 렌더링 과정 ········· 1
그림 I-2. 최신 GPU 사진들 ········· 2
그림 I-3. GPU 구조 ········· 3
그림 II-1. 어답티브 텍스쳐 맵 ········· 8
그림 II-2. 어답티브 텍스쳐맵을 활용한 볼륨 렌더링 ········· 8
그림 IV-1. 스텐실 버퍼를 이용한 이른 광선 단절법 알고리즘 ········· 13
그림 IV-2. glStencil 함수의 설정 코드 ········· 13
그림 IV-3. 빈 공간 건너뛰기 알고리즘 ········· 14
그림 IV-4. 2D 텍스쳐, 3D 텍스쳐, 8진트리 텍스쳐 ········· 15
그림 IV-5. 8진트리 텍스쳐의 GPU 표현법 ········· 15
그림 IV-6. 8진트리 텍스쳐에서 간접 텍스쳐 매핑 계산 과정 ········· 16
그림 IV-7. 4진트리 메쉬 ········· 18
그림 V-1. 볼륨 데이터의 렌더링 결과 ········· 21

<수식 차례>

수식 II-1. 볼륨 렌더링 원시 적분 식 ···· 6
수식 II-2. 단순화한 볼륨 렌더링 적분 식 ···· 6
수식 III-1. 컴포지팅 연산 ···· 10
수식 IV-1. 앞-뒤 순서를 위한 블렌딩 함수 ···· 11

I. 머리말

가. 볼륨 렌더링이란?

볼륨 렌더링(volume rendering)이란 볼륨 데이터라 불리는 3차원의 불연속적으로 샘플된 데이터를 인간이 쉽게 인지할 수 있는 2D 이미지로 투영해서 가시화하는 기법을 말한다. 물론, 여러 개의 2차원 슬라이스로 변환한 뒤 슬라이스별로 가시화하는 방법도 있겠지만, 이러한 경우에는 각 슬라이스의 상관관계를 파악하기 힘들다는 단점이 존재한다. 볼륨 데이터에는 여러 종류가 있는데, CT나 MRI 촬영을 통해 얻어진 인체 (또는 다른 종류의) 데이터가 대표적이고 실제 측정이나 컴퓨터를 이용한 계산을 통해 얻어진 유체역학, 지진, 기상, 해양 데이터들도 과학 기술 전반에 걸쳐 널리 사용되고 있다. 볼륨 데이터를 가시화하기 위해서는 매우 큰 컴퓨팅 능력이 필요하기 때문에 지금까지는 슈퍼컴퓨터나 고성능 서버 또는 전용 그래픽스 워크스테이션을 이용해서 가시화해왔다. 그러나 2000년대에 들어서면서 NVIDIA GeForce 시리즈나 ATI Radeon 시리즈 등의 일반 PC용 그래픽스 카드들의 성능이 급격히 발전하면서 범용 PC에서 볼륨 렌더링을 하려는 시도가 계속되고 있다. 일반적으로 PC 그래픽스 하드웨어를 이용한 볼륨 렌더링은 3차원 텍스쳐 매핑(3D texture mapping)을 이용하여 많이 구현되는데, 지금까지 보다 빨리 그리고 보다 양질의 영상를 얻기 위한 많은 노력이 있어왔다. 특히, PC용 그래픽스 하드웨어에서 꼭지점 및 프래그먼트 프로그래밍(vertex / fragment programming)이 가능해지면서 과거에는 고가의 그래픽스 워크스테이션 또는 볼륨 렌더링 전용 하드웨어에서나 가능하던 실시간 볼륨 렌더링이 일반적인 PC에서도 가능해지게 되었다.

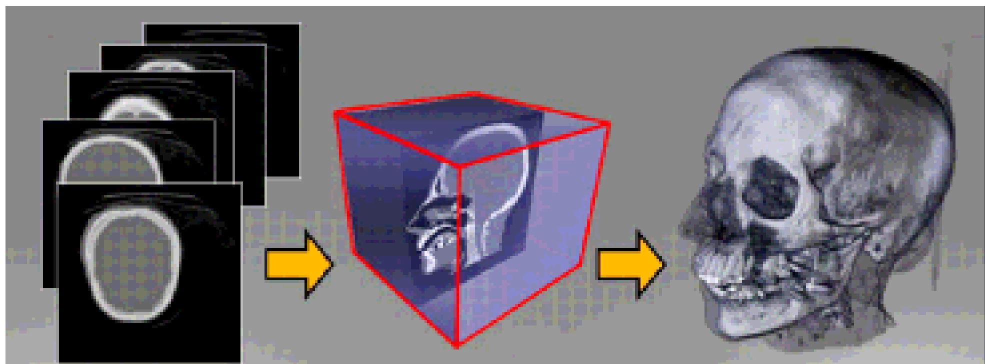


그림 I-1. 볼륨 렌더링 과정

나. GPU 소개

GPU란 Graphic Processing Unit의 약자로서 PC에서 컴퓨터 그래픽을 전문적으로 담당하는 프로세서이다. 과거에는 3차원 그래픽을 처리할 때 모든 연산을 CPU에서 처리했다. 그런데 3차원 그래픽 연산은 대부분 부동소수점 연산이기 때문에 이에 최적화되지 않은 일반 CPU에서 복잡한 장면을 빠르게 처리하기는 거의 불가능했다. 이 때문에 고가의 그래픽스 전용 워크스테이션에는 3차원 그래픽만을 처리하기 위한 전용 프로세서가 채용되기도 했다. 2000년대 들어서 NVIDIA와 ATI 등의 회사에서는 저가의 3차원 그래픽스 프로세서를 개발해서 일반 PC에서도 부담 없이 3차원 그래픽을 처리할 수 있게 되었고, 이를 GPU라는 신용어로 부르게 되었다.

GPU는 부동소수점 연산에 최적화되어 있을 뿐만 아니라 3차원 그래픽스 연산의 기본 단위인 꼭지점이나 픽셀 등을 최대한 빠르게 처리하기 위해 SIMD 형태의 벡터 머신으로 구성되었다. 이는 각 꼭지점들과 픽셀들의 연산이 모두 독립적이면서 동시에 같은 연산들을 처리한다는 점에 착안한 것이다. 2005년 현재 최신의 CPU이 속도는 12GFlops에 불과하지만 최신의 GPU는 50GFlops에 육박할 만큼 빠른 속도를 보이고 있다. 이는 그래픽스 연산에서 크게 중요하지 않은 각종 분기문이나 루프문 등 복잡한 연산들이나 자유로운 메모리 접근, 다양한 자료구조 등을 최대한 배재해서 단순화하고 대신 부동소수점 처리에 최적화했기 때문이다. 이러한 이유로 GPU에는 CPU보다 프로그래밍이 어려운 대신 단순한 프로그램에 대해서는 CPU와 비교할 수 없을 만큼 빠른 처리속도를 보인다.

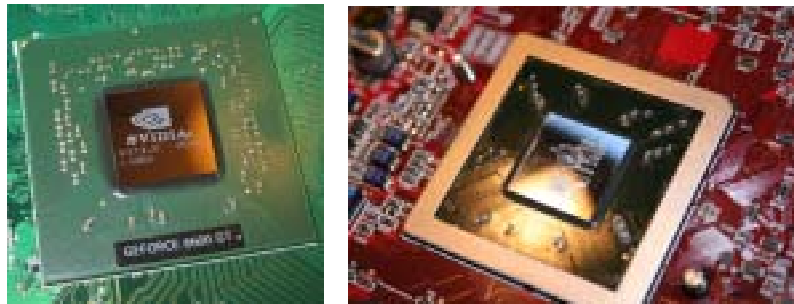


그림 I-2. 최신 GPU 사진들, 왼쪽 : NVIDIA GeForce 6600 GT (NV43), 오른쪽 : ATI Radeon 9800 Pro (R350)

GPU는 크게 꼭지점 프로세서(vertex processor)와 프래그먼트 프로세서(fragment processor)로 나뉜다. 꼭지점 프로세서는 3차원 그래픽에서 각 다각형들의 꼭지점들을 처리하는 프로세서이고 프래그먼트 프로세서는 렌더링 이미지의 픽셀들을 처리하는 프로세서이다. 이러한 이유로 프래그먼트 프로세서는 픽셀 프로세서라고도 불린다. 그림 I-3은 NVIDIA의 Geforce6800 GPU의 구조도이다. 꼭지점 프로세서에서는 CPU로부터 받은 꼭지점들에 대해 모델뷰(modelview)행렬 연산과 투영(projection) 행렬 연산 그리고 이동(translation)이나 회전(rotation), 스케일링(scaling) 등의 연산을 수행한 뒤 프래그먼트 프로세서로 보낸다. 각 꼭지점들은 프래그먼트 프로세서로 보내지기 전에 먼저 Viewing Frustum에 들어가는지 테스트를 하고 다각형으로 구성된 뒤 다른 다각형들에 가려지는 부분은 없는지 깊이 테스트를 한다. 이렇게 해서 최종적으로 살아남은 부분들은 래스터라이저(rasterizer)에 의해 프래그먼트(fragment)로 만들어지고 이 프래그먼트들은 프래그먼트 프로세서로 들어가게 된다. 프래그먼트 프로세서에서는 각 프래그먼트 들에 대해 색깔 및 광원 계산을 한다. 텍스쳐 설정이 되어 있다면 텍스쳐 값을 가져오는 작업을 하는 부분도 이 프로세서에서 처리된다. 이렇게 해서 만들어진 각 프래그먼트들은 최종적으로 다른 프래그먼트 들과 깊이 테스트 및 블렌딩을 한 뒤 프레임버퍼(frame buffer)에 저장된다. 이 과정에서 꼭지점 프로세서나 프래그먼트 프로세서에서 일반적인 OpenGL이나 Direct3D의 그래픽스 연산을 따르지 않고 사용자 임의로 프로그래밍을 할 수 있도록 한 것이 GPU이다.

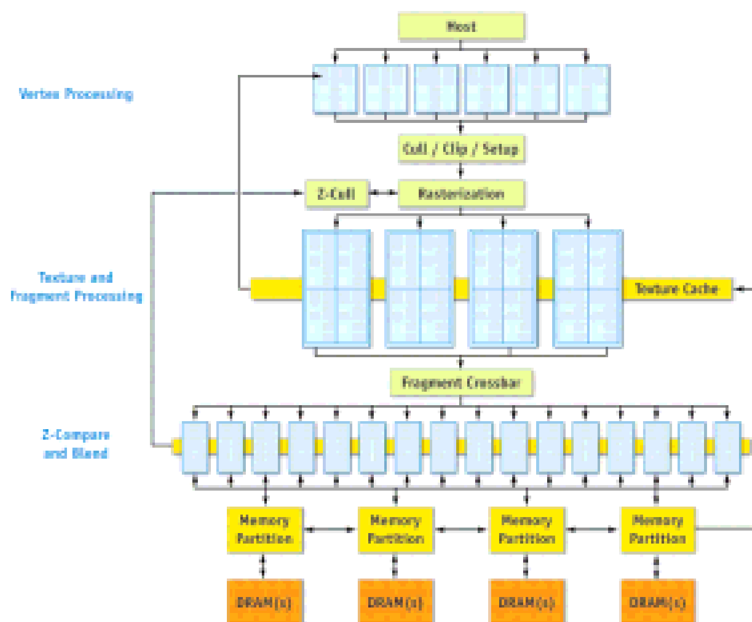


그림 I-3. GPU 구조

II. 기존의 GPU 기반 볼륨 렌더링 연구

볼륨 데이터에 광선을 쏘아 만나는 복셀 값을 가져오는 작업인 샘플링은 볼륨 렌더링에서 가장 많은 시간이 요구되는 작업 중 하나이다. Akeley는 그래픽스 하드웨어의 3차원 텍스쳐 매핑을 이용하여 볼륨 렌더링의 샘플링과 컴포지팅을 매우 빠르게 처리하는 방법을 제안했는데, 이 방법은 아직까지도 대부분의 GPU 기반 볼륨 렌더링의 근간이 되고 있는 방법이다.

Akeley에 의해 3차원 텍스쳐 매핑을 이용한 볼륨 렌더링의 가능성이 최초로 제기된 이후 이를 이용한 볼륨 렌더링에 대한 많은 연구가 계속되어왔다. 볼륨 렌더링의 결과영상의 품질을 향상시키기 위해서는 쉐이딩이 필수적으로 이루어져야 하는데, 일반적으로 쉐이딩에 많이 사용되는 퐁 쉐이딩(Phong shading)의 경우 음영효과를 위해서 각 꼭지점마다 법선벡터를 반드시 필요로 한다. Cullip과 Neumann은 이러한 법선벡터맵(normal map)을 미리 생성하여 퐁 쉐이딩에 사용하는 방법을 처음으로 제안했고, Van Gelder와 Kim은 법선벡터맵과 룩업테이블(lookup table)을 이용하여 볼륨 렌더링을 처음으로 구현했다. Dachille는 이러한 아이디어를 더욱 발전시켜서 큐브맵핑(cube mapping)을 이용하여 룩업테이블을 빠르게 구현했고, 컴포지팅에 일반적으로 사용하는 하드웨어 블렌딩(blending)을 사용하지 않고 CPU를 사용함으로써 이른광선단절법을 구현하여 렌더링 속도를 향상시켰다.

이러한 방법들은 모두 CPU를 이용해서 퐁쉐이딩을 미리 계산하고 이를 룩업테이블로 만들어 사용했기 때문에 적당한 프레임 레이트(frame rate)를 얻을 수 있었지만, 뷰잉(viewing)이나 라이팅(lighting) 파라미터가 변할 경우 룩업테이블을 새로 생성해야 한다는 점에서 분명한 한계가 있었다. Westermann은 이러한 한계를 극복하고자 룩업테이블 방식이 아닌 다른 방식으로 퐁 쉐이딩을 구현했다. 바로 색깔행렬(color matrix)을 이용해서 빠르게 쉐이딩 연산을 한 것이다. 이 렌더링 방법은 퐁 쉐이딩을 빠르게 처리하기는 했으나 볼륨 데이터를 하나의 물질로 간주하여 렌더링했기 때문에 여러 물질을 동시에 가시화하지는 못했다. 이러한 단점을 보완하기 위해 Meissner는 픽셀 텍스쳐(pixel texture)를 사용했다. Meissner의 렌더링 방법은 Westermann의 렌더링 방법과 마찬가지로 색깔행렬을 사용한 것이다. 다만, Wetermann의 방법이 색깔행렬을 사용해서 퐁 쉐이딩을 처리한 것에 비해 Meissner의 방법에서는 퐁 쉐이딩 자체는 따로 계산하고 색깔행렬을 이용해서는 디퓨즈(di

ffuse) 밝기만 계산한 점이 다르다. Westermann와 Meissner의 렌더링 방법은 매우 빠른 시간 안에 퐁의 쉐이딩 모델을 처리하여 광원 파라미터가 변경되는 경우에도 인터랙티브(Interactive)한 프레임 레이트를 보장하였지만, 앰비언트(ambient)와 디퓨즈 라이팅만 처리하고 스펙큘러(specular) 라이팅은 처리하지 못하는 단점이 있었다.

초창기의 그래픽스 하드웨어를 이용한 볼륨 렌더링은 모두 SGI 그래픽스 워크스테이션을 기반으로 구현되었다. 볼륨 렌더링을 위해 필수적으로 필요한 3차원 텍스쳐 매핑 기능 및 대용량의 텍스쳐 메모리, 픽셀 텍스쳐 등의 OpenGL 확장(extension) 지원 등이 고가의 SGI 그래픽스 워크스테이션에서만 가능했기 때문이다. 그러나 NVIDIA나 ATI와 같은 PC용 그래픽스 하드웨어 제조사들이 프로그래밍이 가능한 그래픽스 하드웨어를 개발하면서 PC에서도 볼륨 렌더링을 처리하려는 시도가 계속되어졌다.

Rezk-Salama는 NVIDIA GeForce 256 하드웨어의 레지스터 컴바이너 (register combiner)를 이용해서 멀티패스 방식으로 볼륨 렌더링을 구현했다. 그런데, 이 그래픽스 하드웨어에서 3차원 텍스쳐 매핑을 지원하지 않는 관계로 여러개의 2차원 텍스쳐 매핑을 이용해야 했다. 볼륨 데이터 샘플링에 3차원 텍스쳐 매핑이 이용되는 가장 큰 이유는 하드웨어적으로 3선형보간(tri-linear interpolation)을 지원하기 때문이다. 2차원 텍스쳐 매핑은 3선형보간을 지원하지 못하기 때문에 Rezk-Salama는 2번의 텍스쳐맵핑으로 2번의 선형보간을 하고 레지스터 콤바이너의 멀티 텍스쳐링(multi-texturing)으로 앞의 보간한 값들을 또 선형보간하여 3선형보간을 구현했다.

이 렌더링 방법은 샘플링과 컴포지팅의 경우에는 PC 수준의 그래픽스 하드웨어를 이용하여 빠르게 처리했지만 퐁 쉐이딩을 빠르게 처리하는 방법은 제시하지 못했다. Meissner는 이러한 단점을 극복하여 PC 그래픽스 하드웨어에서도 퐁 쉐이딩을 빠르게 처리할 수 있는 방법을 제안했다. Meissner는 NVIDIA에서 GeForce3를 발표하며 새롭게 추가한 텍스쳐 쉐이더 기능을 이용하여 퐁 쉐이딩을 구현했다. Meissner의 볼륨 렌더링 방법은 Rezk-Salama의 방법과 마찬가지로 2차원 텍스쳐 매핑을 이용한 방법이다. 그러나 여기에 텍스쳐 쉐이더(texture shader) 기능을 이용해서 퐁 쉐이딩을 구현했다. NVIDIA의 텍스쳐 쉐이더는 말 그대로 텍스쳐들을 이용해서 쉐이더 기능을 구현한 것으로 ATI의 프래그먼트 쉐이더와 마찬가지로 프래그먼트 프로그래밍 기능을 제공한다. 이 텍스쳐 쉐이더에서는 벡터 또는 반사 벡터를 이용한 큐브매핑을 지원하는데, Meissner는 이 기능을 이용해서 퐁 쉐이딩을 구현했다.

지금까지 살펴본 텍스쳐 하드웨어를 이용한 볼륨 렌더링 기법들은 모두 본질적으로 텍스쳐 맵핑 하드웨어의 기능인 3선형보간과 영상 누적기능을 이용하여 볼륨 데이터의 샘플링, 콤포지팅을 가속함으로써 속도 향상을 도모했다. 그러나 대부분의 기법들은 이러한 하드웨어를 이용함으로써 어느 정도의 속도 향상은 얻을 수 있었으나 하드웨어 자체의 제약으로 전통적인 볼륨 렌더링 알고리즘에서 속도 향상을 위해 제시된 여러 가지 기법들 - 이른광선단절법(Early Ray Termination) 및 빈공간무시법(Empty Space Skipping)을 구현하기에는 많은 어려움이 있었다. Kruger는 이러한 점을 극복하고자 최신의 그래픽스 하드웨어를 이용해서 GPU 광선쏘기법(raycasting)을 제안했다. Kruger의 GPU 광선쏘기법은 이전의 방법들과 마찬가지로 3차원 텍스쳐에 볼륨 데이터를 저장하고 하드웨어 텍스쳐 매핑을 이용해서 샘플링을 가속했으나, 프록시 도형을 전혀 사용하지 않고 GPU에 CPU의 알고리즘을 적용시킴으로써 샘플링과 컴포지팅 방법을 기존의 GPU 기반방법과는 전혀 다르게 했다.

Engel은 미리 적분한 볼륨 렌더링 방식으로 샘플링 회수가 나이퀘스트(Nyquest) 이론에 미치지 못해도 노이즈없는 영상이 렌더링될 수 있는 방법을 제안했다. 볼륨 렌더링식은 본래 수식 II-1과 같은 복잡한 적분식으로 되어있다.

$$I = \int_0^D \mathrm{color}\,(\mathbf{x}(\lambda))\exp(-\int_0^\lambda \mathrm{extinction}\,(\mathbf{x}(\lambda'))d\lambda')d\lambda$$

수식 II-1. 볼륨 렌더링 원시 적분 식

그러나 이 적분식은 계산 시간이 매우 오래 걸리므로 간략화한 아래의 계산식을 이용한다.

$$I \approx \sum_{i=0}^{n} \widetilde{C}_i \prod_{j=0}^{i-1}(1-\alpha_j)$$
$$\widetilde{C}_i \approx \tilde{c}(s(\mathbf{x}(id)))d$$
$$\alpha_j \approx 1-\exp(-\tau(s(\mathbf{x}(jd)))d)$$

수식 II-2. 단순화한 볼륨 렌더링 적분 식

Engel은 전이함수(transfer function)을 생성할 때 가장 뒤의 간략화된 연산을 사용하지 않고 앞부분의 실제 볼륨 렌더링 적분을 이용하는 방법을 제안했다. 실제 적분 연산을 미리 계산해 놓은 뒤 이를 텍스쳐로 저장해 놓고 필요할 때 마다 참조해서 사용하는 형식으로 속도 향상을 꾀한 것이다.

볼륨 데이터의 크기가 텍스쳐 메모리 크기보다 클 경우에는 보통 볼륨 데이터를 여러개의 작은 서브볼륨으로 나누어서 각각의 서브 볼륨을 따로 메모리에 올린 뒤에 렌더링한다. 그러나 이러한 방법은 렌더링을 여러 번 나누어서 해야 하기 때문에 렌더링 속도가 떨어지는 단점이 존재한다. 이러한 단점을 극복하기 위해서 많은 연구가 있어왔는데, Meissner는 OpenGL 표준 압축기법인 S3TC를 이용하여 손실압축을 했다. LaMar와 Boada는 벡터양자화(vector quantization)를 이용하여 볼륨 데이터를 압축했고, Weiler와 Guethe는 웨이블릿(wavelet)을 이용해서 볼륨 데이터를 압축했다. 이러한 압축 기법들은 모두 손실압축 기법들로 대용량 데이터를 텍스쳐 메모리에 올릴 수는 있게 했지만 대신 결과 영상의 품질을 저하시키게 했다. Kraus는 어답티브 텍스쳐맵(adaptive texture map)을 이용해서 큰 손실없이 볼륨 데이터를 압축했다. 어답티브 텍스쳐맵이란 텍스쳐 데이터 중 비어있는 부분을 없애고 그림이 있는 부분만을 모아서 묶는 기법이다. 그림 II-1의 왼쪽 윗부분 그림과 같이 텍스쳐맵을 일정한 크기로 자른 뒤 데이터가 없는 부분은 지우고 있는 부분만 남긴다. 그 뒤에 왼쪽 아래부분 그림과 같이 데이터가 많은 부분은 최대한 세밀하게 샘플링하여 크기를 그대로 유지하고 데이터가 적을수록 듬성듬성 샘플링하여 크기를 줄인다. 이렇게 해서 그림 II-1의 오른쪽 아래 그림과 같이 압축한 텍스쳐맵을 생성하는 것이다. 이때, 압축한 텍스쳐에는 일정크기로 자른 텍스쳐 데이터들을 크기가 큰 순서대로 넣고 원래 텍스쳐로 복원했을 때의 텍스쳐 좌표를 인덱스 데이터에 기록해 놓는다. 이러한 모든 작업들은 프레그먼트 쉐이더의 프로그래밍 기능으로 구현하였고, Kraus는 이를 이용하여 볼륨 데이터를 압축했다.

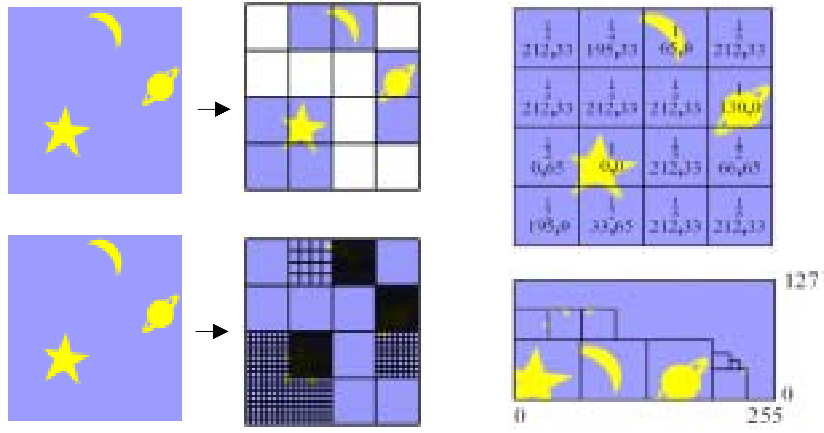
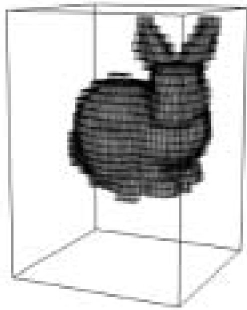
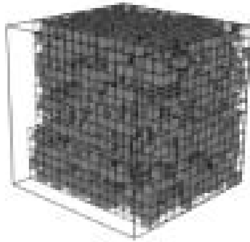


그림 II-1. 어답티브 텍스쳐 맵

Non-empty cells of the **32**^{3} index data grid



Data blocks packed into a **256**^{3} texture



Result Image

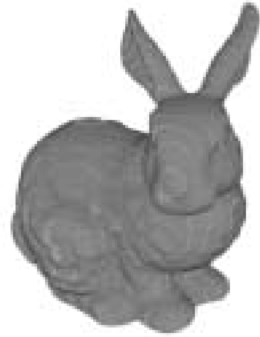


그림 II-2. 어답티브 텍스쳐맵을 활용한 볼륨 렌더링

III. 기본적인 GPU 기반 볼륨 렌더링 알고리즘

가. 샘플링

볼륨 렌더링은 크게 3가지 과정으로 이루어지는데 샘플링(sampling), 쉐이딩(shading) 및 컴포지팅(compositing)이 그것이다. 볼륨 데이터에서 적절한 복셀(voxel)을 찾아오는 샘플링 과정은 매우 많은 시간을 필요로 하므로 이를 빠르게 처리하는 것은 주요한 연구 주제 중 하나였다. 그래픽스 하드웨어의 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. 이렇게 텍스쳐 매핑을 이용하여 샘플링을 하기 위해서는 텍스쳐가 입혀질 도형이 필요한데 이를 대리 평면(proxy plane)이라고 한다. 보통 2차원 텍스쳐에 볼륨 데이터를 저장할 경우에는 대리 평면을 텍스쳐에 평행하도록 배치하고, 3차원 텍스쳐에 볼륨 데이터를 저장할 경우에는 영상평면(image plane)에 평행하도록 배치한다. 간편하게 대리 평면이 영상평면과 평행하도록 하기 위해서 그래픽스 하드웨어의 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 한다. 여기서, 모델뷰 행렬에 회전 행렬을 곱함으로해서 꼭지점이 실질적으로 회전하게 되는데, 대리 평면은 항상 영상평면에 평행으로 회전하면 안되기 때문에 꼭지점 쉐이더에서 꼭지점에 모델뷰 행렬을 무시하고 투영행렬만 곱해지도록 하였고, 대신 각 대리 평면의 꼭지점에 할당되는 텍스쳐 좌표에 모델뷰 행렬을 곱함으로써 볼륨 데이터가 회전하는 것처럼 보이도록 하였다.

나. 쉐이딩

샘플링된 복셀들은 전이함수(transfer function)을 통해 색깔 및 투명도가 결정되고 여기에 쉐이딩을 통해 음영이 입혀지게 된다. 이러한 작업은 프로그래밍이 가능한 쉐이더(programmable shader)를 이용하여 수행된다. 전이함수는 종속 텍스쳐 (dependent texture) 기법을 이용하여 구현될 수 있는데, 종

속 텍스쳐란 한번 텍스쳐 매핑한 값을 텍스쳐 좌표로 활용하여 다시 텍스쳐 매핑을 하는 기법을 말한다. 본 논문에서는 2차원 텍스쳐를 이용한 2차원 전이함수를 사용하였는데, 복셀값과 복셀의 그라디언트(gradient) 크기를 인덱스로 하였다. 복셀의 그라디언트 크기가 크면 복셀값이 급격히 변화하므로 물질의 경계부분이라는 뜻이고 그라디언트 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.

다. 컴포지팅

컴포지팅은 그래픽스 하드웨어의 알파 블렌딩(alpha blending) 기능을 이용하여 수행한다. 이때 블렌딩 순서가 매우 중요한데 앞에서부터 뒤로 블렌딩하는 방법(front-to-back order)과 뒤에서부터 앞으로 블렌딩하는 방법(back-to-front order)이 있다. 본 논문에서는 뒤에서부터 앞으로 블렌딩하도록하였으며 이에 따라 대리 도형을 그리는 순서 역시 시점에서 가장 먼 도형부터 가까운 순서대로 그리도록 하였다. 뒤에서 앞으로 블렌딩할 때에 사용되는 알파 블렌딩 연산은 아래의 식과 같은데, 이 식에서 C_i는 i번째 복셀에서의 색깔을나타내고 A_i는 i번째 복셀에서의 투명도를 나타낸다

$$C'_i = C_i + (1 - A_i) C'_{i+1}$$

수식 III-1. 컴포지팅 연산

IV. 속도 향상 기법

본 논문에서 구현한 볼륨 렌더링 알고리즘에서는 모든 샘플링되는 복셀에 대해 프래그먼트 프로그램에서 퐁 쉐이딩을 했다. 이때, 높은 수준의 렌더링 이미지를 얻기 위해서 특별히 반사 광원(specular light)을 포함한 완전한 퐁 쉐이딩을 사용하여 매우 복잡한 연산을 수행하도록 했다. 이러한 이유로 전체 그래픽스 파이프라인 중 프래그먼트 프로그램에서 병목 현상을 보여 렌더링 성능이 저하됨을 알 수 있었다. 이러한 속도 저하 현상을 해결하기 위해 본 논문에서는 대표적인 볼륨 렌더링 가속 기법인 이른광선단절법과 빈 공간무시법을 GPU에서 구현했다. 본 논문에서 구현한 향상된 알고리즘들은 공통적으로 프래그먼트 프로그램에서 해당 복셀을 쉐이딩하기 이전에 미리 그 프로그먼트가 최종 이미지에 영향을 미치는 지를 파악하여 영향을 미치지 않는 프래그먼트들은 퐁 쉐이딩 연산을 수행하지 않는 방법으로 렌더링 속도의 향상을 이끌어냈다.

가. 이른광선단절법

이른광선단절법(Early Ray Termination: 이상 ERT)은 대표적인 볼륨 렌더링 방법인 광선추적법(Ray-casting)에서의 속도 향상 기법 중 하나로, 이미지 평면의 각 픽셀에서 광선을 쏘고 그 광선을 계속 추적하면서 만나는 복셀들을 그 픽셀에 누적시키는 과정에서, 픽셀이 불투명해지면 광선을 일찍 단절시키고 더 이상 추적을 하지 않도록 하는 렌더링 속도 향상 기법이다. GPU 볼륨 렌더러에 이 기법을 적용시키기 위해 대리 도형을 앞-뒤 순서(Front-to-Back order)로 그리고, 매 대리 도형을 그릴 때 마다 각 픽셀이 불투명해질 경우 스텐실 버퍼를 1로 설정하여 더 이상 그 픽셀에 그릴 수 없도록 하면 ERT의 구현이 가능하다. 앞-뒤 순서로 도형을 그리기 위해서는 블렌딩 함수를 뒤-앞 순서와는 다르게 설정해야 하는데, 이는 수식 IV-1과 같다.

$$C_{dst} = (1-\alpha_{dst})C_{src} + \alpha_{dst}C_{dst}$$
$$\alpha_{dst} = (1-\alpha_{dst)}\alpha_{src} + \alpha_{dst}$$

수식 IV-1. 앞-뒤 순서를 위한 블렌딩 함수

불투명한 픽셀의 스텐실 버퍼를 1로 설정하는 것은 두 가지 방법을 사용해서 구현이 가능하다.

1) 알파 값 읽기

첫 번째 방법은 프레임버퍼의 알파 값을 메인메모리로 읽어서 CPU에서 불투명도를 체크한 뒤 직접 스텐실 버퍼에 쓰는 것이다. 즉, 대리 도형을 앞에서 뒤의 순서로 그리면서 매 대리 도형을 그릴 때 마다 `glReadPixels()` 함수를 이용해 프레임 버퍼를 메인 메모리로 읽은 후 알파 값을 비교해서 1과 가까우면 `glDrawPixels()` 함수로 스텐실 버퍼의 해당 픽셀을 1로 설정하면 된다. 이 방법은 알고리즘은 간단하지만 그래픽스 메모리 내용을 메인메모리로 다시 읽어 들여야 하기 때문에 속도 저하가 일어날 수 있으며 실제로 구현한 결과도 뒤에 설명할 다중패스 렌더링 방식에 비해 성능이 낮았다.

2) 다중패스 렌더링

두 번째 방법은 알파 텍스쳐와 다중 패스 렌더링을 통해서 그래픽스 메모리의 내용을 메인 메모리로 읽어 들이지 않고 스텐실 버퍼를 설정하는 방법이다. 우선, 매번 대리 도형을 그릴 때마다 렌더링된 결과 중 알파 값을 저장해서 알파 텍스쳐를 생성해야 하는데, 이는 `glCopyTexImage()` 함수 또는 OpenGL의 PBO(Pixel Buffer Object) 확장으로 쉽게 구현이 가능하다. 이 텍스쳐를 다시 프레임 버퍼에 그리면서 프래그먼트 프로그램에서 알파 값을 비교해서 1과 가까우면 깊이 값을 0으로, 아니면 1로 설정한다. 이렇게 깊이 버퍼를 설정한 뒤 다시 대리 도형을 그리면 깊이 테스트를 통과하지 못한 픽셀들 - 알파 값이 1과 가까운 - 에 대해서만 스텐실 버퍼 값을 1로 설정할 수 있다. 두 번째 방법의 전체적인 알고리즘은 그림 IV-1과 같고, 스텐실 함수 설정 방법은 그림 IV-2와 같다. 이 방법은 앞에 설명한 `glReadPixels()` 함수로 알파 값을 읽어들이는 방법에 비해 실제 구현 시에 성능이 높았음을 알 수 있었다. 따라서 본 논문에서는 두 번째 방법을 이용해서 볼륨 렌더링을 수행했다.

```
1st pass :
   Generate alpha texture from frame buffer
   Draw Proxy Slice with alpha texture
   Check alpha value in fragment program
     if (alpha value > threshold) depth = 0
     else  depth = 1
2nd pass :
   Draw proxy slice
   Do stencil test
     if (fail stencil test) skip fragment program for Phong shading
     else if (fail depth test)
        stencil = 1
        skip fragment program for Phong shading
     else  do fragment program for Phong shading
```

그림 IV-1. 스텐실 버퍼를 이용한 이른 광선 단절법 알고리즘

```
glStencilFunc(GL_NOTEQUAL, 1, 1);
glStencilOp(GL_KEEP, GL_REPLACE, GL_KEEP);
```

그림 IV-2. glStencil 함수의 설정 코드

나. 빈공간무시법

1) 기본 알고리즘

이른광선단절법과 함께 전통적인 볼륨 렌더링에서 많이 사용되는 가속 기법으로 빈공간무시법(Empty Space Skipping : 이상 ESS)이 있다. 이 방법을 구현하기 위해 다음과 같은 방법을 사용했다. 전이함수가 적용되지 않는 복셀값을 가지는 복셀에 대해서는 빈 공간으로 취급해서 쉐이딩을 수행하지 않고 건너뛰도록 해서 렌더링 속도를 향상시키고자 했고, 이를 이중 패스 렌더링과 이른 깊이 테스트를 이용해서 구현했다. 이른 깊이 테스트는 프래그먼트 프로그램을 수행하기 전에 미리 깊이 테스트를 해서 테스트를 통과하는 프래그먼트에 대해서는 프래그먼트 프로그램을 실행하고, 통과하지 못하는 프로그먼트들은 모두 통과해버리는 기법을 말한다. 이러한 이른 깊이 테스트를 이용해서 빈 복셀에 대해서는 퐁 쉐이딩을 수행하지 않도록 했는데, 이중 패스 렌더링 시 똑같은 대리 도형을 같은 위치에 두 번 그리도록 하고, 처음 그릴 때는 복잡한 퐁 쉐이딩을 적용시키지 않고 단순히 복셀 체크만 수행해서 그 복셀이 비었는지 여부만을 판단한다. 복셀이 비었을 경우에는 깊이 버

퍼를 0으로 설정하여 두 번째 렌더링 시 깊이 테스트에서 건너뛰도록 하고 복셀이 비어있지 않았을 경우에는 깊이 버퍼를 1로 설정하여 두 번째 렌더링 시 복잡한 퐁 쉐이딩을 수행하도록 했다. 전체적인 알고리즘은 그림 IV-3과 같다.

```
1st pass :
   Draw proxy slice
   Check voxel value in fragment program
   if (voxel value >= threshold) depth  = 1
   else  depth = 0
2nd pass :
   Draw proxy slice
   Apply Z test
   if (depth == 0) skip fragment program for Phong shading
   else do fragment program for Phong shading
```

그림 IV-3. 빈 공간 건너뛰기 알고리즘

2) Octree Texture (8진트리 텍스쳐)

일반적으로 영화나 게임 등의 분야에서는 어떤 3차원 폴리곤 모델을 보다 현실감있게 보이기 위해 2차원 텍스쳐 이미지를 이용한 매핑하는 기법을 많이 사용한다. 그러나 이 방법은 모델이 매우 정교하고 복잡한 경우에는 매핑을 하기가 쉽지 않다는 문제가 있다. 반면에 3차원 텍스쳐는 이러한 복잡한 매핑을 극복하고 폴리곤의 World Coordinate 좌표의 각 x, y, z 요소를 [0, 1]로 스케일링하기만 하면 바로 Texture Coordinate 좌표가 되어 간단하게 매핑을 할 수 있도록 한다. 그러나 3차원 텍스쳐 매핑의 경우엔 2차원 텍스쳐에 비해 상대적으로 메모리를 많이 필요로 하는 단점이 있다. Octree Texture(8진트리 텍스쳐)는 이러한 2차원 및 3차원 텍스쳐의 단점을 극복하여 간단한 매핑과 적은 메모리 점유의 특징을 모두 가지고 있는 텍스쳐 매핑 기법이다. Octree Texture는 그림 IV-4와 같이 3차원 텍스쳐를 8진트리로 나누고 실제 폴리곤 매쉬에 매핑되는 텍셀(texel)이 위치한 노드만을 살린 텍스쳐를 말한다. 그러나 이 방법은 폴리곤 매쉬가 변경이 되면 Octree Texture 역시 새로 만들어야 하는 단점이 존재한다.

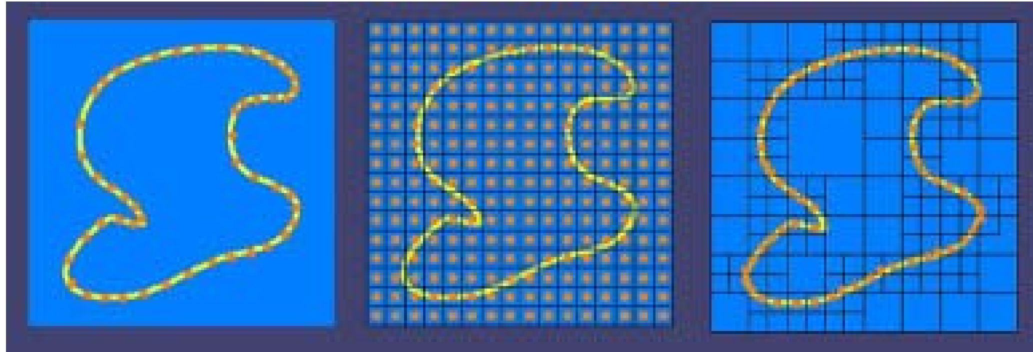


그림 IV-4. 2D 텍스쳐, 3D 텍스쳐, 8진트리 텍스쳐

8진트리 텍스쳐 매핑은 GPU의 꼭지점 프로그램 및 프래그먼트 프로그램을 이용해서도 구현이 가능하다. 8진트리 텍스쳐 매핑은 필수적으로 8진트리를 탐색하는 루틴이 들어가야 한다. 일반적으로 CPU에서는 이를 Linked List나 긴 행렬의 인덱싱으로 쉽게 구현이 가능하지만 GPU에서는 자료구조의 여러 제약조건으로 이를 구현하기가 쉽지 않다. 따라서 Lefebvre, Hornus, Neyret 등은 GPU Gems2의 "Octree Textures on the GPU" 챕터에서 이를 간접 텍스쳐 매핑(indirect texture mapping)을 이용해서 구현했다. 그림 IV-5는 설명의 편의를 위해 Quadtree(4진트리)로 설명한 그림인데, 4진트리(혹은 8진트리)의 루트 노드부터 깊이 우선 순서(depth first order)로 각 노드들을 저장한다. 단말 노드를 제외한 모든 노드들은 RGB에 자식 노드의 index를 저장하고, 단말 노드는 실제 데이터를 저장하도록 한다. 그림 IV-5에서 C가 단말 노드인데, 이를 참조하기 위해서는 우선 A를 방문해서(텍스쳐 매핑해서) B로의 인덱스를 찾고, 다시 B를 방문해서 C로의 인덱스를 찾은 후 최종적으로 C의 값을 가져오게 된다.

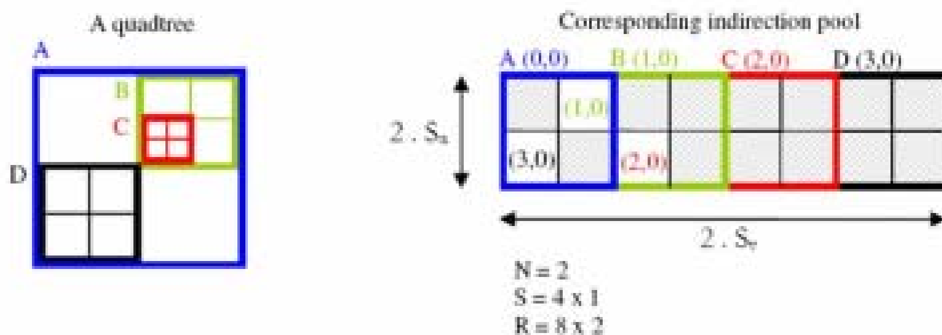


그림 IV-5. 8진트리 텍스쳐의 GPU 표현법

이러한 간접 텍스쳐 매핑 연산은 그림 IV-6과 같이 계산된다. 간접 풀(indirection pool)로 부터 텍스쳐 매핑하는 좌표는 $P = \frac{I_D + frac(M \cdot N^D)}{S}$ 로 계산된다. 여기서 ID는 해당 노드의 인덱스이고, M은 텍스쳐 좌표, S는 간접 풀의 각 노드의 개수이다.

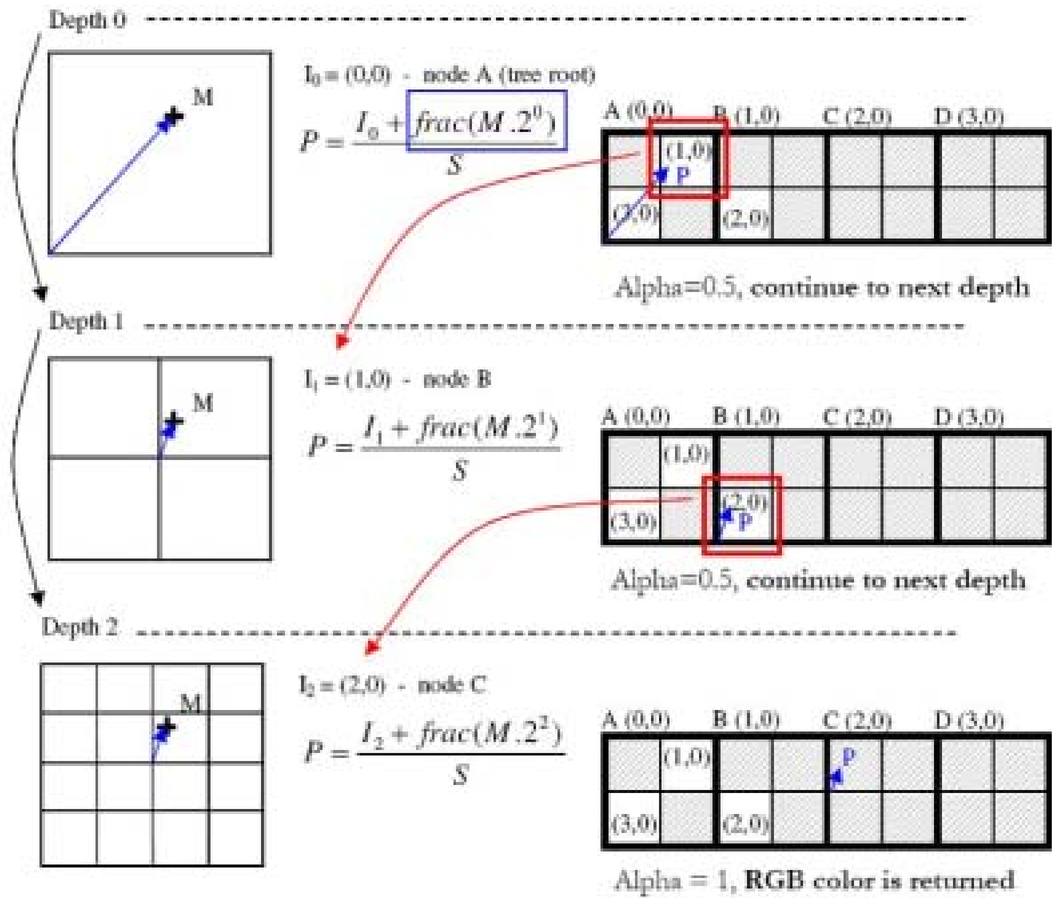


그림 IV-6. 8진트리 텍스쳐에서 간접 텍스쳐 매핑 계산 과정

전통적으로 CPU에서 빈공간무시법을 구현하기 위해 8진트리를 많이 이용해 왔는데, 이는 이 자료구조를 통해 빠르게 빈공간을 찾을 수 있고 빠르게 다음 빈 공간 (또는 비어있지 않은 공간)을 찾을 수 있기 때문이다. GPU를 이용한 볼륨 렌더링에서도 비슷한 원리로 8진트리를 사용해서 빈공간무시법을 구현할 수 있을 것이다. 다만, GPU는 매우 빠른 속도로 삼선형보간을 할 수 있기 때문에 이를 생략해도 속도향상의 정도가 적고, 퐁 쉐이딩만을 건너뛰는 부분에서만 효과만 있어 CPU의 경우보다는 전체적인 속도향상 정도가

덜할 것이다.

Lefebvre등의 GPU에서의 8진트리 텍스쳐 매핑기법은 위의 8진트리 텍스쳐 매핑을 GPU에서 구현한 첫 번째 방법으로 그 의의가 있다고 하겠다. 또한 3차원 텍스쳐의 크기를 상당부분 줄일 수 있는 부가적인 효과도 있다. 그러나 이를 그대로 GPU 볼륨 렌더링의 빈 공간 건너뛰기에 적용시키기에는 여러 가지 문제가 발생한다. 우선, Lefebvre등의 방법은 텍스쳐의 각 텍셀들이 모두 실제 복셀값이 아니고 상당수가 자식 노드의 인덱스이기 때문에 하드웨어 삼선형보간을 그대로 사용할 수 없다. 따라서 반드시 Nearest 필터를 사용하고 삼선형보간이 필요한 부분은 프래그먼트 프로그램에서 직접 구현을 해야 한다. 두 번째 문제는 단말노드에서 복셀값과 그라디언트(gradient)를 동시에 저장할 수가 없다. 텍스쳐의 텍셀을 RGBA로 했을 경우, 일반적인 볼륨 렌더링에서는 RGB에 그라디언트를, A에는 복셀 값을 저장한다. 그러나 이 경우엔 A에 단말노드임을 표시하는 값이 저장되어야 하므로 RGB 중 하나에 복셀 값을 저장해야 한다. 즉, 그라디언트를 저장할 공간이 부족한 것이다. 물론 이러한 문제는 그라디언트를 패킹(packing)해서 16 또는 32bit 정도로 줄이거나 벡터양자화(vector quantization)를 사용해서 해결할 수도 있으나 추가적인 연산 또는 텍스쳐 매핑이 필요하다는 문제가 발생한다. 세번째로, 8진트리의 탐색시 간접 텍스쳐 매핑을 이용했는데, 텍스쳐의 크기가 커지면 커질 수록 8진트리의 깊이가 깊어지면서 텍스쳐 매핑에 걸리는 시간이 점점 길어진다는 문제가 있다. 즉, 기존의 방법은 단 한번의 텍스쳐 매핑만으로 텍셀(복셀)값을 가져올 수 있지만 Lefebvre등의 방법을 이용하면 볼륨 데이터의 크기에 따라 6에서 10번의 텍스쳐 매핑을 해야만 하나의 텍셀을 가져올 수 있다. 물론 빈 공간의 경우는 단말노드까지 내려가지 않기 때문에 1~(n-1)번 (n = log2(width))만 매핑하면 된다. 따라서 이 방법을 이용하면 필연적으로 하나의 복셀을 샘플링하는데 걸리는 시간이 기본적인 방법을 사용할 때보다 기하급수적으로 늘어나게 되어 프래그먼트 프로그램에서 퐁 쉐이딩을 건너뛴다 하더라도 렌더링 속도가 오히려 늘어나게 된다. 이러한 문제점을 해결하기 위해서는 CPU에서의 8진트리 탐색과 마찬가지로 GPU에서도 인접한 복셀들간의 연관성(coherence)을 살려서 8진트리의 부모노드 전체를 건너뛰어 모든 복셀을 다 샘플링할 필요가 없도록 해야 한다. 그러나 GPU의 구조 상 각 복셀들을 처리하는 프래그먼트가 모두 독립적으로 동작하고 서로 통신할 수 없기 때문에 직관적으로 이를 구현하는 것은 매우 어렵다. 이러한 문제를 해결하기 위해 다음과 같이 기본적인 GPU 볼륨 렌더링

알고리즘을 수정했다

- 수정된 알고리즘

이 알고리즘에서는 일반적인 GPU 볼륨 렌더링과는 달리 대리 도형(proxy slice)를 단 하나의 사각형 폴리곤만을 사용하지 않고, 그림 IV-7과 같이 하나의 슬라이스를 4진트리로 나누고 루트 노드에서부터 단말 노드까지 개별적인 사각형 폴리곤을 만들고 차례로 렌더링하도록 한다. 즉, 그림에서와 같이 가장 먼저 전체 슬라이스를 둘러싸는 사각형(빨간색)부터 그리는데, 이 사각형은 4진트리의 루트 노드를 의미한다. 그 뒤에 4등분된 조그만 사각형을 그리고 (녹색) 그 조그만 사각형을 또 4등분하여 그린다(파란색). 이런식으로 계속 4등분하며 그린다. 이 사격형 폴리곤들을 4진트리 메쉬(Quadtree Mesh)라고 하겠다. 이 알고리즘은 전체적으로 2패스로 나뉘어지는데, 첫 번째 패스에서는 4진트리 메쉬들을 폴리곤 모드로 그리지 않고 꼭지점 모드로 꼭지점만을 그리고, 8진트리 텍스쳐 매핑을 통해 해당 노드와 자식 노드들이 비었는지 아닌지를 판단하도록 한다. 두 번째 패스에서 다시 4진트리 메쉬를 그리는데, 이때는 폴리곤 모드로 렌더링해서 폴리곤 내부의 프래그먼트들이 생성되도록 한다. 만약 해당 노드가 비었다면 이후에 발생하는 모든 꼭지점과 프래그먼트들이 생략되도록 각 프래그먼트들의 깊이 값을 0으로 설정하고, 단말 노드라면 실제 퐁 쉐이딩을 하는 등 실제 렌더링 수행을 한다. 마지막으로 볼륨 데이터의 8진트리는 미리 생성되어있다고 가정한다. 가시화 대상에 따라 다르겠지만 전이함수를 미리 적용시키고 투명도(opacity)가 0인 부분은 무조건 빈 공간으로 간주하면 8진트리 자체의 크기를 최대한 줄일 수 있다. 세부적인 알고리즘을 설명하면 다음과 같다.

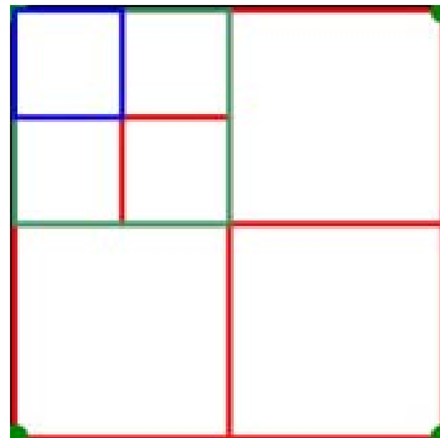


그림 IV-7. 4진트리 메쉬

첫번째 패스:

1. 4진트리 메쉬를 꼭지점만 그린다.

2. 프래그먼트 프로그램 :

   a. 만약 루트 노드라면 정상적으로 8진트리 텍스쳐 매핑

   b. 루트 노드가 아니라면 이전 패스에서 부터 받은 FBO를 읽어 자식 노드의 인덱스를 받아 8진트리 텍스쳐 매핑

   c. 8진트리 텍스쳐 매핑을 한 뒤, 네 꼭지점에서의 텍스쳐 매핑 결과를 비교한다. 이때, GPU의 프래그먼트 프로그램에서는 다른 프래그먼트에서의 계산 값을 가져오는 연산이 없기 때문에 한 프래그먼트 프로세스에서 다른 꼭지점의 텍스쳐 매핑도 함께 해서 비교해야 한다. (한 프로세스에서 8진트리 텍스쳐 매핑을 4번해야 한다)

   d. 텍스쳐 매핑 결과 값(자식 노드의 인덱스)은 color.rgb에 저장한다.

   e. 만약 빈 노드이면 color.a = 0, 아니면 color.a = 1로 한다.

   f. 만약 단말 노드이면 color.a = 0.5로 한다.

3. 프래그먼트 프로그램에서 계산된 결과를 Frame Buffer Object(FBO)로 생성해서 두번째 패스로 넘긴다.

두번째 패스:

1. 4진트리 메쉬를 꼭지점 뿐만 아니라 모두 그린다.

2. 꼭지점 프로그램 :

   a. 첫 번째 패스에서 생성된 FBO를 받아서 꼭지점 텍스쳐 매핑을 한다.

   b. 해당 텍스쳐 매핑 값 중 alpha값이 0이면 이후 노드는 모두 비었으므로 생략하도록 color.a = 0으로 한다. 아니면 color.a = 1이다.

3. 프래그먼트 프로그램 :

   a. 꼭지점 프로세서로 부터 받은 color.a값을 비교, 0이면 이후의 모든 꼭지점과 프래그먼트를 건너뛰도록 depth = 0으로 한다.

   b. color.a가 0.5이면 (단말노드이면) 완전한 퐁 쉐이딩을 한다.

   c. color.a가 1이면 color.rgb에 자식 노드의 인덱스 값을 저장한 뒤 다시 첫번째 패스를 수행한다.

이 알고리즘을 모든 단말노드에 도달할 때까지 반복해야 한다.

이 알고리즘은 아주 이상적인 형태는 아니고, 최대한 GPU에서 구현 가능하도록 수정한 것이다. 만약 GPU의 꼭지점 텍스쳐 매핑이 3차원 텍스쳐에 대해서도 충분한 성능을 낼 수 있고, 각 꼭지점 연산 또는 프래그먼트 연산에 대한 gather 연산을 지원한다면 보다 효율적인 알고리즘이 나올 것이다. 그러나 GPU의 이러한 제약 때문에 조금은 복잡한 알고리즘이 나왔다.

V. 구현 결과

본 기술문서에서는 듀얼 인텔 제온 3.06GHz, 4GB 메모리, 256MB 메모리를 가진 AGP 8X 방식의 GeForce 6800GT 그래픽스 카드를 장착한 시스템을 이용해서 볼륨 렌더러를 구현했으며 쉐이더 모델 3.0을 사용했다. 실험에 사용한 데이터는 표 V-1과 같다.

볼륨 데이터	볼륨 크기	텍스쳐 크기
Engine	256×256×110	256×256×128
CT Head	256×256×225	256×256×256

표 V-1. 각 볼륨 데이터의 크기

각 볼륨 데이터에 대해 서로 다른 2가지 종류의 전이 함수를 적용했다. 첫 번째 전이 함수는 가시화하고자 하는 복셀들 중, 값이 작은 복셀들은 투명하게 하고 큰 복셀들은 불투명하게 했고, 두 번째 전이 함수는 가시화하고자 하는 복셀들을 모두 불투명하게 했다. 가시화 결과는 그림 V-1과 같다.

그림 V-1. 볼륨 데이터의 렌더링 결과 (좌측 상단부터 bighead1, bighead2, engine1, engine2)

ERT를 구현할 때는 두 가지 방법을 이용했는데, 알파 텍스쳐와 다중 패스 렌더링을 이용한 방식이 프레임버퍼에서 알파 값을 읽어서 CPU에서 비교하는 방식에 비해 성능이 월등히 뛰어났다. 이는, 그래픽스 메모리와 메인 메모리 간의 인터페이스에서 병목현상이 일어나기 때문이다. 실험 장비에서 사용한 그래픽스 카드가 AGP 8배속이기 때문에 이러한 현상이 유난히 더 나타났지만, PCI Express 16배속 방식의 동일한 그래픽스 카드를 장착한 환경에서도 정도의 차이가 있을 뿐 병목 현상은 여전히 나타나는 것으로 확인되었다. 표 V-2에서 ERT의 두 가지 구현 방식에 따른 속도 차이와 그래픽스 카드 인터페이스에 따른 속도 차이를 비교했다. 데이터는 bighead를 사용했고, 첫 번째 전이함수를 사용했으며, 256X256 해상도로 렌더링했다.

ERT 구현 방식	알파 값 읽기	다중 패스 렌더링
AGP 8X	14.91	16.52
PCI Express 16X	17.24	18.57

표 V-2. ERT 구현 방식과 그래픽스 인터페이스에 따른 성능 비교

ESS의 경우에는 기본적인 이른 깊이 테스트를 이용한 방법만 사용했다. 8진 트리 텍스쳐를 이용하는 경우는 아래의 이유로 아직 구현이 완료되지 못했다.

1. OpenGL Extension Specification에서의 FBO Extension에 따라 구현했음에도 프래그먼트 프로그램에서 생성한 FBO가 다음 패스의 꼭지점 프로그램에서 읽혀지지 않았다. 물론 프레임버퍼에서부터 복사해서 텍스쳐를 생성해도 되지만 속도 저하가 많았다.

2. 이 알고리즘은 프래그먼트 프로그램 및 꼭지점 프로그램에서 복잡한 연산을 해야 하는데, 버그를 잡기 위한 디버거가 충분치 못하다. 프래그먼트 프로그램의 경우엔 픽셀에 값을 기록한 뒤 다시 읽어 들여서 출력해보는 형식으로라도 디버깅이 가능하지만 꼭지점 프로그램의 경우엔 계산 값이 자동으로 보간(interpolation)되어 버려서 잘못된 값이 출력되어 디버깅에 어려움이 많았다.

표 V-3은 각 볼륨 데이터와 전이 함수에 대해 속도 향상 기법들을 적용할 때와 하지 않았을 때의 성능을 보여준다. 각 경우에 대해 256X256과 512X512의 두 가지 해상도로 렌더링했고, 첫 번째 숫자는 프레임 레이트, 괄호안의 두 번째 숫자는 기본 렌더링 방식 대비 속도 향상의 정도를 나타낸다. ERT의 경우 본 논문에서 구현한 방식이 다중 패스 렌더링을 필요로 하기 때문에 실행에 추가적인 부하가 걸리게 된다. 이러한 이유로 매 대리 도형마다 ERT를 적용할 경우, 퐁 쉐이딩을 생략하여 속도를 향상시키는 정도보다 부하로 인한 속도 저하 부분이 더욱 커서 오히려 속도가 더 떨어지게 된다. 데이터의 대부분이 투명한 engine1의 경우에는 그 정도가 더욱 심하다. 이러한 점을 극복하기 위해서 본 논문에서는 ERT 계산을 매 대리 도형마다 실행시키지 않고 16개의 대리 도형을 그릴 때 마다 한번 씩만 실행하도록 했다. 표 V-3을 보면 ESS에 비해 ERT로 인한 속도 향상은 그리 많지 않음을 알 수 있다. 특히, engine1의 경우 투명한 부분이 대부분이어서 거의 속도 향상이 없었다. 표면을 불투명하게 처리하는 두 번째 전이 함수를 사용할 때가 첫 번째 전이 함수를 사용할 때 보다 ERT에서 성능 향상이 더욱 높았고, 렌더링 해상도가 512X512인 경우가 256X256인 경우보다 성능 향상의 정도가 컸음을 알 수 있다.

데이터 종류	bighead1		bighead2	
렌더링 해상도	256×256	512×512	256×256	512×512
기본	13.25	4.48	13.25	4.48
ERT	16.52 (1.25)	5.75 (1.28)	20.99 (1.58)	7.45 (1.66)
ESS	25.14 (1.90)	10.04 (2.24)	25.13 (1.90)	10.04 (2.24)
ERT + ESS	34.50 (2.60)	15.61 (3.48)	41.63 (3.14)	20.40 (4.55)
데이터 종류	**engine1**		**engine2**	
렌더링 해상도	256×256	512×512	256×256	512×512
기본	14.64	4.52	14.64	4.52
ERT	14.85 (1.01)	4.60 (1.02)	18.26 (1.25)	5.70 (1.26)
ESS	37.59 (2.57)	15.80 (3.50)	37.61 (2.57)	15.81 (3.50)
ERT + ESS	34.89 (2.38)	14.54 (3.22)	40.70 (2.78)	18.48 (4.08)

표 V-3. 각 볼륨 데이터에 대한 윈도우 크기 및 알고리즘 종류에 따른 렌더링 속도 (frames/sec)

VI. 결론

본 기술문서에서는 GPU 환경에서의 볼륨 렌더링 방법에 대해 설명하고, 이 환경에서 렌더링을 빠르게 처리하기 위해 전통적인 볼륨 렌더링 가속 기법인 이른광선단절법과 빈공간무시법를 구현했다. 이른광선단절법은 스텐실 테스트를 이용해서 구현했는데, GPU에서 스텐실 버퍼 및 프레임 버퍼의 값을 직접 접근하는 기능을 제공하지 않기 때문에 다중 패스 렌더링 등의 복잡한 방법을 통해서만 구현이 가능했다. 향후 GPU에서 이러한 기능을 지원한다면 보다 빠르고 효율적으로 이 기능의 구현이 가능할 것이다. 빈공간무시법의 경우 이른 깊이 테스트를 이용해서 구현했고, 이른광선단절법보다 속도 향상에 도움이 되었다. 다만, 본 논문에서 구현한 빈공간무시법은 각 픽셀 단위로만 빈 픽셀인지 아닌지를 판단해서 속도 향상에 한계가 있었다. 8진트리 텍스쳐를 이용하는 방법에 대해서도 연구했지만 실제 구현에는 실패했다. 향후에 8진 트리나 BSP 트리 등의 자료구조를 이용해서 보다 넓은 영역을 한꺼번에 빈 공간인지를 판단하는 방식으로 더욱 빠르게 렌더링하는 기법을 구현할 계획이다.

VII. 참고문헌

[1] K. Akeley, Reality Engine Graphics. ACM Computer Graphics, Proc. SIGGRAPH '93, pages 109-116, 1993

[2] T.J. Cullip and U. Neumann, Accelerating Volume Reconstruction with 3D Texture Hardware. Technical Report TR93-027, University of North Carolina, Chapel Hill N.C., 1993.

[3] A. Van Gelder and K. Kim, Direct Volume Rendering with Shading via Three-Dimensional Textures. ACM Symposium on Volume Visualization '96, pages 23-30, 1996

[4] F. Dachille et al, High-Quality Volume Rendering using Texture Mapping Hardware. Proc. of Eurographics/SIGGRAPH Graphics Hardware Workshop 1998, 1998

[5] R. Westermann and T. Ertl, Efficiently using Graphics Hardware in Volume Rendering Applications, ACM Computer Graphics, Proc. SIGGRAPH '98, pages 169-178, 1998

[6] C. Rezk-Salama et al, Interactive Volume Rendering on Standard PC Graphics Hardware using Multi-Textures and Multi-Stage Rasterization, Proc. Eurographics/SIGGRAPH Graphics Hardware Workshop 2000, 2000

[7] M. Meissner, U. Hoffmann, and W. Strasser, Enabling Classification and Shading for 3D Texture Mapping based Volume Rendering, Proc. IEEE Visualization 99, pages 207-214, 1999

[8] J. Kruger and R. Westermann, Acceleration Techniques for GPU-based Volume Rendering, Proc. IEEE Visualization 2003, 2003

[9] K. Engel, M. Kraus and T. Ertl, High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading, Proc. Graphics Hardware, 2001

[10] M. Kraus, T. Ertl, Adaptive Texture Maps, Proc. of Eurographics/SIGGRPAH Workshop on Graphics Hardware 2002, pages 1-10, 2002

[11] E. LaMar, B. Hamann, K. Joy, Multiresolution Techniques for Interacti

ve Hardware Texturing based Volume Visualization, Proc. of IEEE Visualization 99, pages 355-361, 1999

[12] I. Boada, I. Navazo, R. Scopigno, Multiresolution Volume Visualization with a Texture-Based Octree, The Visual Computer 17, 3, pages 185-197, 2001

[13] M. Weiler et al, Level-Of-Detail Volume Rendering via 3D Textures, IEEE Volume Visualization and Graphics Symposium 2000, 2000

[14] S. Guthe et al, Interactive Rendering of Large Volume Data Sets, IEEE Visualization 2002, 2002