텍스춰 기반 볼륨 렌더링에서의 스텐실 버퍼를 이용한 픽셀 단위 건너뛰기

이택희[0] 김동호* 이정진 신영길
서울대학교 컴퓨터학부
*숭실대학교 미디어학부
{watersp, finkl, yshin}@cglab.snu.ac.kr *dkim@ssu.ac.kr

Pixel Skipping with Stencil Buffer for Texture Based Volume Rendering

Tek Hee Lee[0] Dongho Kim* Jeong Jin Lee Yeong Gil Shin
Seoul National University *Soongsil University

요약

본 논문에서는 GPU와 스텐실 버퍼(stencil buffer) 및 깊이 버퍼(depth buffer)를 이용하여 가려진 픽셀들을 렌더링 단계 이전에 건너뛰는(skipping) 방법을 제시하고자 한다. 그래픽 카드에 기본적으로 제공되는 기능인 깊이 및 스텐실 버퍼 검사(depth & stencil buffer test)를 이용하여 이진 차폐 맵(binary occlusion map)을 만들고 이를 재사용하여 가려지는 부분의 픽셀들을 효과적으로 건너뛰게 하는 방법이다. 전체 볼륨 데이터는 팔진트리(octree) 구조를 가진 서브볼륨들로 나뉘어 저장되며 시점에 가까운 서브볼륨부터 렌더링에 사용된다. 서브볼륨들을 차례로 렌더링하면서 차폐 맵을 갱신하게 하면, 멀리 있는 서브볼륨들을 렌더링할 때 이미 가려진 픽셀들을 렌더링에서 제외할 수 있다.

1 서론

최근 들어 하드웨어 텍스춰 매핑을 이용한 볼륨 렌더링이 널리 쓰이고 있다. 이는 범용 그래픽 하드웨어가 볼륨 렌더링에서 쓰이는 여러 가지 연산들을 빠르게 처리해 줄 수 있는 기능을 가지며, 또한 빠른 속도로 발전하고 있기 때문이다.

텍스춰 기반의 볼륨 렌더링에서는 일반적으로 3차원 텍스춰를 사용하며, 시선에 수직인 직사각형들을 proxy geometry로 이용한다. 이 직사각형들에 텍스춰 매핑을 적용하고 연속적으로 중첩하여 그리면서 볼륨 렌더링을 수행하게 된다.[1],[2]

또한 음영을 넣기 위해서 해당 복셀의 법선 벡터 값을 텍스춰의 RGB 채널에 넣어주게 된다. 입력된 법선 벡터는 fragment shader 내에서 적당한 음영 처리 연산을 거치게 되며, 최종 결과에 반영된다.[3]

이와 같은 일반적인 하드웨어 기반의 볼륨 렌더링은 몇가지 한계점을 지니고 있다.

첫째, RGBA 채널을 모두 사용하기 때문에 원 볼륨 데이터의 4배에 해당하는 메모리가 소요된다. 이는 제한된 텍스춰 메모리를 효과적으로 사용할 수 없는 단점을 가지고 있다.

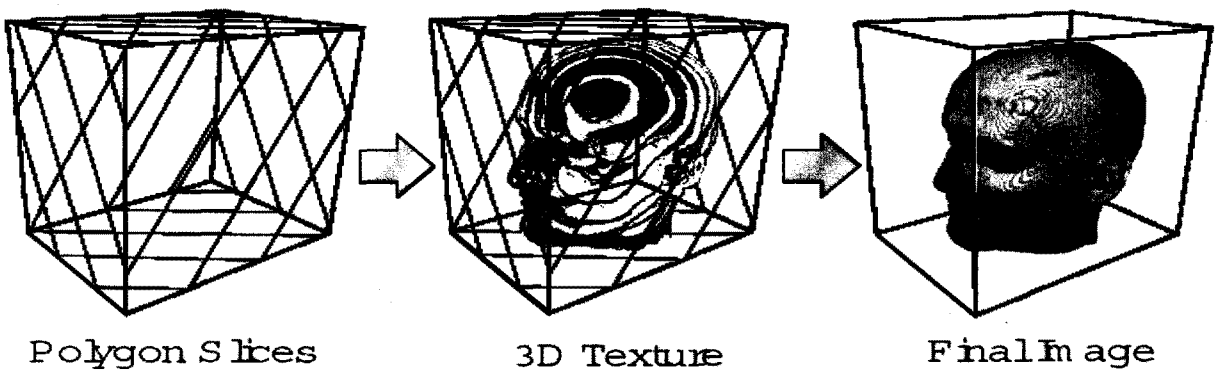

그림 1. 텍스춰 매핑 하드웨어를 이용한 볼륨렌더링

둘째, back-to-front 순서로 볼륨 렌더링을 하게 되면 early ray termination과 같은 알고리즘을 적용할 수 없게 되어 보이지 않는 부분까지 모두 계산하여야 하는 단점이 있다.

본 논문에서는 첫 번째 문제를 실시간으로 법선 벡터를 계산 함으로써 해결하고 두 번째 문제는 서브볼륨 단위의 렌더링과 가려진 픽셀의 건너뛰기를 수행함으로써 해결하고자 한다. 두 번째 문제의 경우 최종 렌더링 결과에 영향을 미치지 않는 픽셀들을 렌더링에서 제외시킴으로써 계산시간에서의 성능 향상을 실현할 수 있게 된다.

이를 위해서는 시점에 가까이 있는 서브볼륨부터 렌더링해야 되기 때문에 서브볼륨들은 front-to-back 순서로 사용되게 되고 각 서브볼륨이 렌더링될 때, 하드웨어에서 가속해 주는 스텐실 버퍼를 이용한 차폐 맵이 갱신된다. 이 차폐 맵은 이후의 서브볼륨들이 렌더링될 때 픽셀들의 가시성 검사에 사용된다.

앞으로의 논문 구성은 다음과 같다.

2장에서는 하드웨어 기반의 볼륨 렌더링에 대해 소개한다. 3장에서는 본 논문에서 제안하는 여러 기법들을 자세히 설명하며, 4장에서는 실제로 구현한 방법을 소개한다. 5장에서는 실험 결과를 제시하고, 마지막으로 6장에서는 결론과 함께 문제점을 토의하고 앞으로의 연구 방향을 제시한다.

2 텍스춰 기반의 볼륨 렌더링

3차원 텍스춰 매핑 하드웨어를 이용한 볼륨 렌더링의 기본적인 아이디어는 그림 1과 같이 볼륨 데이터를 텍스춰 메모리로 가져온 후, 볼륨 텍스춰 도메인상에서 시선에 수직한 직사각형들로 구성되는 proxy geometry를 만든다. Proxy geometry를 구성하는 다각형들에 볼륨 데이터를 이용한 텍스처 매핑이 이루어지고 이들이 블랜딩되면서 최종 이미지가 얻어지는 것이다. 이때 저장되어 있는 볼륨 데이터 중에서 원하는 영역에 속하는 값들만을 가시화하고 싶다면 복셀의 밀도(intensity) 값에 따른 분류(classify)를 수행하여야 한다. 일반적으로, 불투명도 변환함수(opacity transfer function)를 사용하여 밀도 값을 불투명도인 알파 값으로 변환하는데, 변환함수를 참조 테이블 형식으로 만들어 하드웨어에서 사용되는 1차원 텍스춰로 보내면 다각형 상의 각 픽셀은 해당 밀도 값으로 이를 참조하여 알파 값을 얻을 수 있다.

보다 사실적인 영상을 위해서는 음영 처리가 필요하다.[4] 이를 위해서는 각 픽셀마다 음영 처리를 위한 연산이 이루어져야 하는데 현재의 그래픽 하드웨어는 픽셀 처리 단계에서도 사용자의 정의에 따른 연산이 가능하도록 하고 있다. 그림 2는 현재 상용화되고 있는 ATI Radeon 9700이나 nVIDIA geForce FX와 같은 그래픽 하드웨어의 GPU(graphics processing unit)에서 이루어지고 있는 프로그램 가능한 파이프라인을 보여주고 있다. 응용프로그램에서 proxy geometry를 정의해서 GPU에 보내면 첫 단계에서는 다각형의 각 정점을 시각 변환시키는 등의 버텍스 처리(vertex processing)가 이루어지고 다음 raterization 단계에서 다각형을 픽셀 단위로 쪼갠다. 마지막 픽셀 처리단계에서는 각 픽셀마다 블랜딩 또는 깊이 테스트 등의 연산이 이루어진다. 최근 출시된 GPU들은 버텍스 처리과정과 픽셀 처리과정을 프

로그램 가능하도록 지원하고 있다. 이러한 프로그램은

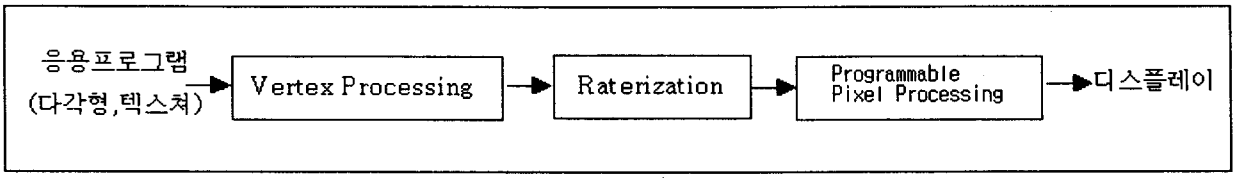

그림 2. 프로그램 가능한 그래픽 하드웨어의 파이프 라인

DirectX나 OpenGL을 통해 구현할 수 있는데, 본 논문에서 사용한 마이크로소프트사의 API인 DirectX에서는 버텍스 쉐이더(VertexShader)와 픽셀 쉐이더(PixelShader/FragmentShader)를 통해 프로그램이 가능하다. 하드웨어 기반의 볼륨 렌더링에서도 음영 계산을 하기 위해 픽셀 쉐이더를 사용할 수 있다. 최신 버전인 DirectX 9.0에서는 픽셀 쉐이더 2.0 버전을 지원하고 있는데 여기에서는 텍스춰나 프레임 버퍼의 원소들이 실수(floating point number) 타입으로 정의될 수 있으며 실수 연산 및 실수 레지스터의 사용이 가능하게 되어서 더욱 정밀한 연산을 수행할 수 있다.

3 스텐실 버퍼를 이용한 픽셀 단위 건너뛰기

서브볼륨 렌더링에서의 스텐실 버퍼를 이용한 픽셀 단위 건너뛰기는 크게 세 단계의 과정을 거친다. 첫번째 단계에서는 넘어온 서브볼륨을 프레임 버퍼가 아닌 텍스춰에 중첩하여 그림으로써 볼륨 렌더링을 수행한다. 이때 이전의 서브볼륨 렌더링에서 갱신된 스텐실 버퍼를 참조하여 각 픽셀을 그릴지의 여부를 판단한다. 두 번째 단계에서는 fragment shader의 z 출력값과 적절한 스텐실 함수를 이용하여 스텐실 버퍼를 갱신한다. 세 번째 단계에서는 첫 번째 단계의 결과 이미지를 2차원 텍스춰 매핑을 이용하여 화면에 그려준다.

이와 같은 세 단계를 거치면 현재의 서브볼륨까지 반영된 영상이 렌더링되며 이 과정은 각 서브볼륨마다 반복적으로 이루어진다. 모든 서브볼륨이 렌더링되어 최종 영상이 완성되면 스텐실 버퍼는 다시 초기화된다.

3.1 법선 벡터의 실시간 계산

기존의 하드웨어 기반의 볼륨 렌더링에서는 3차원 텍스춰를 생성할 때 법선 벡터의 XYZ값과 밀도 값을 저장하기 위해 RGBA, 4개의 채널을 이용했었다. 하지만 이 경우에는 밀도 값만을 저장하는 경우에 비해 4배 많은 메모리를 소모하게 된다.

최신 그래픽 하드웨어는 fragment shader라는 프로그램 가능한 구조를 가진다. 이를 이용한다면 원래의 밀도 값만을 텍스춰에 이용하고 이로부터 실시간으로 법선 벡터를 계산할 수 있다. 그림 3은 fragment shader내에서 법선 벡터 계산을 어떻게 하는가를 대략적으로 표현하고 있다.

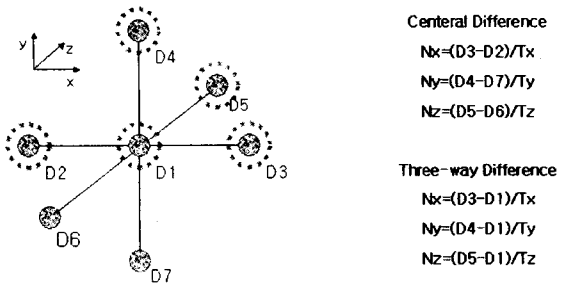

그림 3. 법선벡터 계산법[6]

먼저 입력되는 텍스춰 좌표를 중심으로 각각 복셀 간격만큼의 길이를 더하거나 빼서 인접하는 복셀들의 3차원 텍스춰 값을 얻어온다. 이 값들을 법선 벡터를 계산하는 centeral difference 방법에 적용시킨 후 법선 벡터 값을 계산한다. 하지만 총 6개의 텍스춰 값을 가져와야 하기 때문에 속도가 많이 느려지게 된다. 따라서 D3, D4, D5, D1 위치의 밀도 값을 이용하여 법선 벡터를 계산한다면 세 개의 텍스춰 값만 가져오면 된다.

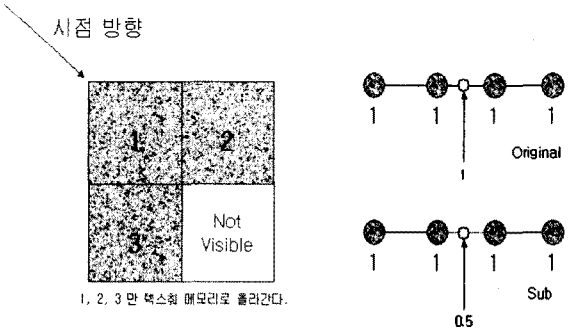

그림 4. 서브볼륨 렌더링

3.2 서브볼륨을 이용한 볼륨 렌더링

기존의 텍스춰 기반 볼륨 렌더링은 각 단면들을 back-to-front 순서로 렌더링한다. 하지만 가려지는 픽셀들을 렌더링에서 제외하기 위해서는 시점으로부터 가까운 데이터들을 먼저 렌더링할 필요가 있다. 따라서 본 논문에서는 볼륨 데이터를 여러 개의 서브볼륨으로 분할하고 이들을 시점에서 가까운 순서대로 렌더링한다. 이렇게 하면 서브볼륨 단위로 차폐 맵을 갱신하여 이후의 서브볼륨 렌더링에 사용할 수 있다.

먼저 원래의 볼륨 데이터를 최대값과 최소값을 인덱스로 가지는 팔진트리(octree)로 재구성한다. 이 때, 가능한 한 각 노드의 최대값과 최소값의 차이가 작도록 팔진트리를 구성한다. 이렇게 하면 주어진 불투명도 변환함수(opacity transfer function)를 이용하여 범위 내에 드는 서브볼륨만을 텍스춰 메모리에 올릴 수 있게 된다. 본 논문에서는 각 서브볼륨의 모양을 정육면체로 제한하였다.

서브볼륨으로 분할할 때에는 경계면에서 문제가 발생할 수 있다. 이는 선형 보간에 의해 텍스춰 값을 가져올 때, 경계면에서는 인접하는 텍스춰에 대한 정보가 없기 때문이다 (그림 4 오른쪽 참조). 이를 해결하기 위해서는 각 서브볼륨의 끝부분을 두 복셀 단위로 반복시킴으로써 해결이 가능하다. 선적분(pre-integration) 방법[5]을 쓰지 않는다면 한 개의 복셀만 반복해도 된다.

각 서브볼륨 내부는 back-to-front 순서로 그려지게 되고 그려진 2차원 이미지를 다시 front-to-back 순서로 합쳐서 최종적인 이미지를 얻는다.

3.3 스텐실 버퍼를 이용한 픽셀 건너뛰기

서브볼륨들을 렌더링할 때 이미 가려진 픽셀들을 렌더링에서 제외하면 속도면에서 많은 이득을 볼 수 있다. 이를 위해 이미 그려진 서브볼륨들에 대한 차폐 맵을 스텐실 버퍼를 이용하여 구현한다. 즉, 이전에 그려진 부분의 불투명도가 1이상이면 스텐실 비트를 1로 바꾸고, 1이 아니라면 0으로 둔다. 이후 선택된 서브볼륨을 그릴 때 스텐실 버퍼의 값이 1이면 해당 픽셀을 렌더링하지 않고 건너뛰게 한다.(그림 5 참조)

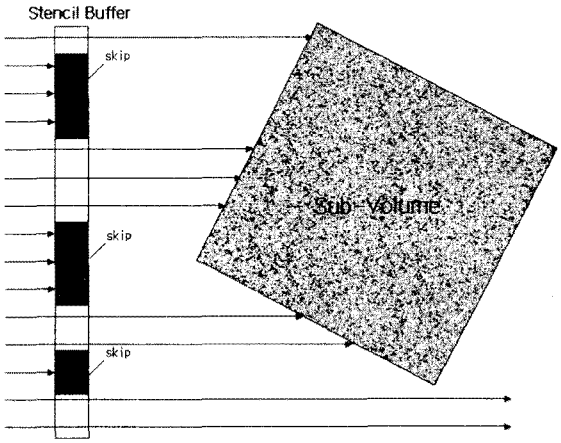

그림 5. 스텐실 버퍼를 이용한 픽셀 건너뛰기

4 구현

본 논문에서 설명한 내용을 구현하기 위해서는 서브볼륨이 그려지는 순서가 시점에서 가까울수록 먼저 그리게 하는 정렬 방법이 필요한데, 이는 시점에서 각 직육면체의 중심까지의 거리를 이용하여 정렬하는 방법으로 단순화하였다.

각 서브볼륨 내부는 세 단계를 거쳐 렌더링이 이루어진다. (표 1참조) 먼저 현재 지정된 서브볼륨을 프레임 버퍼가 아닌 텍스춰에 렌더링한다. 이때 그려지는 위치의 스텐실 버퍼를 참조하여 0으로 설정되어 있을 때만 그리게 한다. 이 때 현재까지 누적된 불투명도 값들도 알

```
for(서브볼륨의 개수)
{
  for(선택된 서브볼륨의 proxy geometry개수)          ┐
  {                                                  │
    if(현재 그려질 픽셀이 차폐 맵에 의해 가려지지 않는가?)  ├ phase1
      픽셀을 텍스춰에 그린다.                          │
  }                                                  ┘
  if(그려진 텍스춰의 알파 값이 1에 가까운가?)          ┐
    스텐실 버퍼를 갱신한다.                           ├ phase2
                                                     ┘
  그려진 텍스춰를 프레임 버퍼에 쓴다.                  ┐ phase3
                                                     ┘
}
```

표 1. 알고리즘

파 채널에 저장되게 된다. 이후 두 번째 단계에서는 첫 단계에서 그려진 텍스춰의 알파 값을 이용하여 스텐실 버퍼를 업데이트한다. 마지막 단계에서는 첫 단계의 결과 이미지를 프레임 버퍼에 알파 블렌딩을 이용하여 그린다.

이와 같이 세 단계로 각 서브볼륨마다 반복적으로 진행이 되며 필요한 모든 데이터가 그려지면 최종적으로 화면에 그려주고 스텐실 버퍼와 프레임 버퍼를 초기화한다. 각 단계의 구체적인 구현 방법은 다음 장에서 자세히 설명한다.

4.1 phase1

먼저 지정된 서브볼륨을 프레임 버퍼가 아닌 따로 생성한 텍스춰에 기존의 볼륨 렌더링 기법으로 그린다. 이 때, 현재의 스텐실 버퍼 값을 보고 0 이 아니라면 그리지 않는다. 이는 스텐실 테스트 옵션을 TRUE 로 하고 스텐실 함수를 EQUAL TO ZERO 로 놓으면 가능하다. 이후 저장된 텍스춰를 phase2 로 넘긴다. 표 2 는 phase1 에 대한 fragment shader 코드이다.

```
ps_2_0        // pixel shader version
def c0, 0.f, 1.f, 0.0f, 0.f       // light
def c1, 0.0078125f, 0.f, 0.f, 0.f // distance
def c2, 0.f, 0.0078125f, 0.f, 0.f // ...between
def c3, 0.f, 0.f, 0.0078125f, 0.f // ...voxels
def c4,  0.8f, 0.8f, 0.8f, 0.8f // for
def c5, 0.2f, 0.2f, 0.2f, 0.2f // ...ambient effect
dcl_volume s0    // volume data
dcl_2d s1        // pre-integration table
dcl t0.xyzw      // front slice texture coordinates
dcl t1.xyzw      // back slice texture coordinates
texld r5, t0, s0 // fetch
texld r6, t1, s0 // ...intensity values
mov r5.y, r6 // coordinates for pre-integ. fetch
texld r5, r5, s1 // fetch pre-integration table
// Nx computation in Equation 1
sub r1, t0, c1
texld r0, r1, s0
add r1, t0, c1
texld r2, r1, s0
sub r1, r0, r2
mov r3.r, r1
// Ny computation in Equation 1
sub r1, t0, c2
texld r0, r1, s0
add r1, t0, c2
texld r2, r1, s0
sub r1, r0, r2
mov r3.g, r1
// Nz computation in Equation 1
sub r1, t0, c3
texld r0, r1, s0
add r1, t0, c3
texld r2, r1, s0
sub r1, r0, r2
mov r3.b, r1
nrm r4, r3       // normalize the normal vector
dp3 r4, r4, c0   // dot product with light
cmp r4, r4, r4, -r4       // absolute value
mul r4, r4, c4   // ambient
add r4, r4, c5   // ...effect
mul r5.rgb, r4, r5  // modulate with shading factor
mov oC0, r5      // output
```

표 2. phase1 fragment shader 코드

4.2 phase2

phase1에서 넘어온 텍스춰를 이용하여 스텐실 버퍼를 갱신한다. 이를 위해 fragment shader를 이용하여 넘어온 텍스춰의 알파 값을 z 값으로 출력한다. Z 테스트를 활성화시키고 정해진 임계값보다 z 값이 크면 스텐실 버퍼를 1로 바꾼다. 즉, 넘어온 텍스춰의 알파 값이 지정된 임계값보다 크다면 뒷부분을 가리는 것이라 가정하고 스텐실 버퍼를 갱신한다. 이 때 마지막으로 테스트된 결과가 그려지면 안되므로 fragment shader내에서 출력 알파 값을 0으로 한다. 표 3는 phase2에 대한 fragment shader 코드이다.

```
ps_2_0          // version instruction

def c7, 0.f, 0.f, 0.f, 0.0f
dcl_2d s4        //render target texture
dcl t0.xyzw      //polygon

texld r7, t0, s4
mov r4.x, r7.a
mov r7, c7
mov oC0, r7
mov oDepth, r4.x
```

표 3. phase2 fragment shader 코드

4.3 phase3

phase1의 결과 텍스춰를 이용하여 화면에 그대로 뿌려 준다. 서브볼륨 내부에서는 중첩시키는 방식이 back-to-front 순서이지만 각 서브볼륨의 결과 이미지의 중첩은 front-to-back 순서이므로 중첩에 알파 값이 필요하게 되는데, 이 값은 현재 프레임 버퍼의 알파 값을 사용한다. 표 4는 phase3에 대한 fragment shader 코드이다.

```
ps_2_0          // version instruction

def c7, 0.f, 0.f, 0.f, 0.0f
dcl_2d s4        //render target texture
dcl t0.xyzw      //polygon

texld r7, t0, s4
mov oC0, r7
```

표 4. phase3 fragment shader 코드

5 실험 결과 및 토론

본 논문에서 제안하는 알고리즘을 사용한 렌더링 결과는 그림 6과 같다. 실험 환경은 P4 2.8에 그래픽 카드는 ATI Radeon 9800 256MB이다. 사용한 데이터는 심장 CT 데이터이고 512*512*249*16bit의 크기를 가진다. 사용된 서브볼륨의 개수는 8개와 32개이며 각 서브볼륨의 크기는 256*256*256과 128*128*128이다. 렌더링 결과, 기존의 볼륨 렌더링 기법에 비해서 약 1.5~2배 정도의 속도 향상을 얻을 수 있었다. (표 5 참조)

실험에 사용된 데이터의 크기는 최대 512*512*512이다. 텍스춰 메모리가 128MB인 그래픽스 하드웨어에서 기존의 볼륨 렌더링 기법으로 만들 수 있는 볼륨 텍스춰의 크기는 256*256*512이며 이는 약 67MB의 텍스춰 메모리를 차지한다. 메모리가 128MB임에도 불구하고 더 크게 못 만드는 이유는 각 축이 가지는 해상도가 반드시 2의 승수가 되어야 하기 때문이다. 따라서 서브볼륨으로 나누어 처리를 하게 되면 좀더 효율적으로 텍스춰 메모리를 이용할 수 있다.

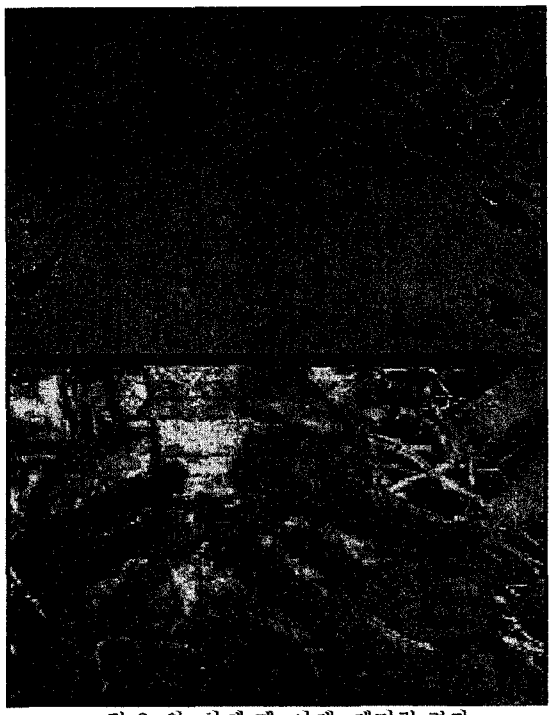

그림 6. 위: 차폐 맵, 아래: 렌더링 결과

데이터 크기	서브볼륨 크기	서브볼륨 개수	렌더링 속도	기존방법
512*512*249 *16비트	128*128*128	32	4.2fps	2.8fps
512*512*512 *16비트	256*256*256	8	2.1fps	1.4fps

표 5. 기존 방법과의 렌더링 속도 비교

6 결론 및 향후 연구 과제

본 논문에서는 볼륨 데이터를 서브볼륨들로 분할하고 이들을 렌더링할 때마다 차폐 맵을 갱신하여 가려진 픽셀

들을 렌더링에서 제외시키는 기법을 제안하였다. 그리하여 기존의 텍스춰 기반 볼륨 랜더링 기법에 비해 큰 속도 향상을 얻을 수 있었다.

향후 연구과제로는 하드웨어의 텍스춰 메모리보다 큰 볼륨 데이터를 렌더링할 때의 속도 개선을 들 수 있다. 텍스춰 기반 볼륨 렌더링의 큰 제약점은 사용할 수 있는 텍스춰 메모리의 양이 현재 나오고 있는 고해상도 CT나 MRI 데이터를 수용하기에 부족하다는 것이다. 이같은 대용량 데이터를 처리하기 위해선 서브볼륨으로 나누어 메인 메모리와의 연동을 통해 렌더링하는 방법이 필요하다. 하지만 데이터 이동의 대역폭이 데이터 양에 비해 굉장히 좁기 때문에 속도가 많이 느려지게 된다. 이를 보완하는 방법으로 이 논문에서 제시하는 방법을 사용할 수 있다. 대부분의 볼륨 데이터가 시점과 가까운 부분에서 불투명도가 1에 가깝게 되기 때문에 실제적으로는 뒷부분의 데이터는 필요하지 않다. 차폐 맵을 이용하여 완전히 가려진 서브볼륨 데이터들을 걸러내어 텍스춰 메모리에 올리지 않는다면 속도는 많이 향상될 것이다.

참고 문헌

[1] F. Dachille, K. Kreeger, B. Chen, I. Bitter, and A. Kaufman. High-Quality Volume Rendering Using Texture Mapping Hardware. In SIGGRAPH Eurographics Graphics Hardware Workshop, 1998.

[2] R. Westermann and T. Ertl. Efficiently Using Graphics Hardware in Volume Rendering Applications. In Proc. of SIGGRAPH, Comp. Graph. Conf. Series, 1998.

[3] William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A real-time procedural shading system for programmable graphics hardware, 2001.

[4] N. Max. Optical models for direct volume rendering. IEEE Transactions on Visualization and Computer Graphics, pages 99- 108, 1995.

[5] K. Engel, M. Kraus, and T. Ertl. High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In Eurographics / SIGGRAPH Workshop on Graphics Hardware, pages 9-17, 2001.

[6] Roni Yagel, Daniel Cohen, and Arie Kaufman Normal estimation in 3D discrete space The Visual Computer,1992