GPU를 이용한 3차원 텍스쳐 기반 볼륨 렌더링의 속도 향상 기법

이중연°, 구기범
한국과학기술정보연구원 슈퍼컴퓨팅센터
{jylee°,voxel}@kisti.re.kr

Acceleration Techniques for 3D Texture Based Volume Rendering using GPU

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

요 약

최신 GPU는 일반 CPU보다 10배 이상 빠른 연산능력을 갖추고 있는데다가 사용자가 직접 프로그래밍할 수 있기 때문에 이를 이용한 고속 볼륨 렌더링 알고리즘에 대한 연구가 활발하게 진행되고 있다. 그러나 스트림 프로세싱에 특화돼있는 GPU의 특성상 early ray termination과 empty space skipping을 구현하는 것이 쉽지만은 않다. 특히 지금까지 제안됐던, 프록시 도형(proxy geometry)을 사용하는 볼륨 렌더링 알고리즘은 empty space skipping은 비교적 효율적으로 구현하지만 early ray termination의 지원은 상대적으로 미비했다. 본 논문에서는 스텐실 버퍼와 OpenGL 확장(extension)을 이용한 2-pass 알고리즘을 통해서 early ray termination과 empty space skipping을 동시에 구현하는 방법을 제시하고, 그 성능을 측정했다.

1. 서 론

대용량의 수치 데이터를 그림으로 표현하는 방법 중 가장 널리 사용되는 볼륨 렌더링은 3차원이나 그 이상 차원의 볼륨 데이터로부터 의미 있는 정보를 추출해서 그림으로 표현하는 기술로, 의료영상이나 기상, 유체역학 등 다양한 분야에서 활용되고 있다. 특히 대용량 볼륨 데이터의 고품질, 실시간 렌더링에 대한 수요는 지속적으로 증가하고 있으며, 이를 효과적으로 처리하기 위한 시도 역시 꾸준하게 있어왔다.

한편, PC의 그래픽스 하드웨어가 발전하고 다양한 프로그래밍 툴이 발표되면서 3D 그래픽스 하드웨어를 단순한 그래픽스 작업이 아닌 임의의 수치 계산에도 활용할 수 있게 됐다[6,18]. 특히 GPU의 높은 수치 계산능력과 사용자가 직접 GPU를 프로그래밍할 수 있다는 장점을 이용해서 기존의 고성능 병렬 컴퓨터에서나 구현할 수 있었던 실시간 볼륨 렌더링을 PC에서도 구현할 수 있게 됐고, 지금도 이에 대한 연구는 활발하게 진행되고 있다.

본 논문에서는 GPU를 이용한 볼륨 렌더링 알고리즘의 성능향상을 위해 GPU 내에서 early ray termination과 empty space skipping을 동시에 적용하는 기법을 제안하고, 그 결과를 소개한다. 본 논문의 구성은 다음과 같다. 2장에서는 지금까지 소개된 GPU 기반 볼륨 렌더링 기법에 대해 살펴보고, 3장에서는 빠른 볼륨 렌더링을 위한 early ray termination과 empty space skipping 기법을 제안한다. 4장에서는 구현된 알고리즘의 구현 결과를 살펴보고, 마지막 5장에서 결론을 맺는다.

2. GPU 기반 볼륨 렌더링

Akeley에 의해 3차원 텍스쳐 매핑을 이용한 볼륨 렌더링의 가능성이 제기되고 Cullip과 Neumann이 이를 이용한 볼륨 렌더링 알고리즘을 소개한 이후 이와 관련된 연구가 활발히 진행됐다[1,2]. 초창기에는 SGI의 Reality Engine, InfiniteReality 등 고성능 그래픽스 엔진을 이용한 연구가 주를 이뤘지만[2,3,16] 일반 PC 그래픽스 하드웨어의 성능 향상으로 이를 이용한 볼륨 렌더링 연구가 더 높은 비중을 차지하게 됐다. 하지만 초창기의 PC 그래픽스 하드웨어는 3차원 텍스쳐를 지원하지 못하거나, 지원하더라도 성능이 충분히 좋지 않았기 때문에 동일한 볼륨 데이터를 3세트씩 생성하는 2차원 텍스쳐 방식의 볼륨 렌더링 알고리즘이 주로 발표됐다[4,5,15]. 이후 3차원 텍스쳐 매핑 하드웨어의 발달로 이를 이용한 볼륨 렌더링 기술에 대한 연구가 뒤를 이었다[5,14,17].

2차원 텍스쳐 매핑과 3차원 텍스쳐 매핑을 이용한 볼륨 렌더링은 모두 볼륨 데이터를 텍스쳐로 저장하고 프록시 도형(proxy geometry)을 이용한다는 점에서 기본 알고리즘은 동일하다. Kruger와 Westermann은 기존의 하드웨어 볼륨 렌더링 알고리즘과는 전혀 다른, 레이캐스팅(ray-casting) 알고리즘을 GPU 상에서 구현했다[7]. 이 연구는 전통적인 볼륨 렌더링 가속 기법들인 early ray termination과 empty space skipping을 GPU에서 효율적으로 구현했다는 특징이 있다. 그러나 프록시 도형을 사용하는 텍스쳐 매핑 기반 볼륨 렌더링에서의 속도 향상 기법에 대한 연구에서는 대부분 empty space skipping은 많이 구현됐으나 early ray termination에 대한 연구는 상대적으로 미흡했다[9,12,13]. 본 논문에서는 프록시 도형을 이용하면서도 empty space skipping과 early ray termination을 효과적으로 구현할 수 있는 기법을 소개하고, 실제 성능을 측정한다.

3. 속도 향상 기법

프록시 도형을 이용한 볼륨 렌더링 알고리즘은 각 프록시 도형을 샘플링 포인트의 집합으로 간주하면 대체로 기존의 소프트웨어 레이캐스팅과 유사한 형태로 진행된다. 따라서 레이캐스팅 속도 향상을 위해 적용했던 early ray termination과 empty space skipping 기법과 유사한 알고리즘을 GPU에서도 구현할 수 있다면 보다 빠른 속도로 렌더링을 수행할 수 있을 것이다. 특히 양질의 렌더링 이미지를 얻기 위해 프래그먼트 프로그램(fragment program)에서 복잡한 퐁 쉐이딩(Phong shading) 연산을 수행하는 경우가 종종 있는데, 이때는 프래그먼트 프로그램이 병목으로 작용해서 전체 렌더링 성능을 떨어뜨릴 가능성이 매우 높다. 따라서 효율적인 가속 기법을 구현할 수 있다면 성능을 크게 떨어뜨리지 않으면서 좋은 렌더링 이미지를 얻을 수 있을 것이다.

이 장에서는 프록시 도형을 이용한 볼륨 렌더링 알고리즘에 적용할 수 있는 early ray termination과 empty space skipping 기법에 대해 자세히 설명한다.

3.1. Early Ray Termination

Early Ray Termination(이하 ERT)은 대표적인 볼륨 렌더링 방법인 레이캐스팅의 속도 향상 기법 중 하나다. ERT는 이미지 평면의 각 픽셀에서 시작된 광선을 따라 일정한 간격으로 배열되는 샘플링 지점에서 불투명도(opacity)를 누적시키다가 그 값이 임계값(threshold)을 넘어서면 더 이상 계산을 하지 않음으로써 속도 향상을 꾀한다[7,11].

본 논문에서는 프록시 도형을 front-to-back 순서로 그리고, 프록시 도형을 그릴 때마다 픽셀의 누적 불투명도가 임계값을 초과하면 스텐실 버퍼를 1로 설정하는 방식으로 ERT를 구현했다. Front-to-back 순서로 프록시 도형을 그릴 때에는 블렌딩 함수를 back-to-front 순서로

그릴 때와는 다르게 설정해야 한다(수식 1).

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

수식 1. Front-to-back 순서를 위한 블렌딩 함수

3.1.1. glReadPixels의 활용

ERT를 구현하는 가장 쉬운 방법은 프레임버퍼의 알파 값을 메인 메모리로 읽어서 CPU에서 불투명도를 검토한 결과를 직접 스텐실 버퍼에 쓰는 것이다. 다시 말해서 front-to-back 순서로 프록시 도형을 그릴 때마다 glReadPixels 함수로 프레임 버퍼를 메인 메모리로 읽은 후 알파 값을 비교해서 임계값을 초과하면 glDrawPixels 함수로 스텐실 버퍼의 해당 픽셀을 1로 설정하면 된다.

이 방법은 간단하지만 그래픽스 메모리의 내용을 메인 메모리로 옮겨와야 하기 때문에 속도 저하가 일어날 수 있으며 실제로 구현한 결과도 다음에 설명하는 2-패스 알고리즘에 비해 낮은 성능을 보여줬다.

3.1.2. 2-패스 알고리즘 (2-pass ERT)

여기서는 그래픽스 메모리의 내용을 메인 메모리로 읽어 들이지 않고 알파 텍스쳐와 스텐실 버퍼를 이용해서 ERT를 구현하는 2-패스 알고리즘을 소개한다.

첫 번째 패스에서는 프록시 도형을 그릴 때마다 렌더링 결과 중 불투명도를 저장해서 알파 텍스쳐를 생성한다. 이는 OpenGL의 glCopyTexImage 함수나 PBO 확장(Pixel Buffer Object extension)으로 쉽게 구현할 수 있다. 그 다음 생성된 텍스쳐를 다시 프레임버퍼에 그리는데, 이 때 프래그먼트 프로그램에서 해당 프래그먼트의 불투명도가 임계값을 초과하면 깊이 정보를 0으로, 그렇지 않으면 1로 설정한다. 이렇게 Z 버퍼를 설정한 뒤 다시 프록시 도형을 그리면(2번째 패스) 깊이 테스트를 통과하지 못한 픽셀들 - 투명도가 임계값을 초과한 - 에 대해서만 스텐실 버퍼 값을 1로 설정할 수 있다.

이 알고리즘은 그림 1에서 의사 코드(pseudo code)로 나타냈고, 스텐실 함수의 설정은 그림 2와 같다. 이 방법은 3.1.1에서 설명한 기법에 비해 높은 성능을 보여준다.

```
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
    else if (fail depth test)
      stencil = 1
      skip fragment program
    else
      do fragment program (Phong shading)
```

그림 1. 스텐실 버퍼를 이용한 early ray termination

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

그림 2. glStencil 함수의 설정 코드

3.2. Empty Space Skipping

본 논문에서는 early ray termination과 함께 empty space skipping (이하 ESS)도 구현했다. 즉, 전이함수(transfer function)가 적용되지 않는 값을 갖는 구간을 빈 공간으로 취급해서 쉐이딩을 수행하지 않고 건너는 방법으로 렌더링 속도를 향상시키고자 했고, 이를 2-pass 렌더링과 early depth test를 이용해서 구현했다.

Early depth test는 프래그먼트 프로그램을 수행하기 전에 미리 깊이 테스트를 해서 테스트를 통과하는 프래그먼트에 대해서만 프래그먼트 프로그램을 실행하는 기법을 말한다[7,9,19]. 이를 이용해서 빈 복셀에 대해서는 퐁 쉐이딩을 수행하지 않도록 했는데, 이중 패스 렌더링 시 똑같은 프록시 도형을 같은 위치에 두 번 그리도록 하고, 처음 그릴 때는 복잡한 퐁 쉐이딩을 적용시키지 않고 단순히 복셀 체크만 수행해서 그 복셀이 비었는지의 여부만을 판단한다. 복셀이 비었을 경우에는 깊이 버퍼를 0으로 설정해서 두 번째 렌더링 시 깊이 테스트에서 건너뛰도록 하고 복셀이 비어있지 않았을 경우에는 깊이 버퍼를 1로 설정해서 두 번째 렌더링 시 복잡한 퐁 쉐이딩을 수행하도록 했다. 전체적인 알고리즘은 그림 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
  else do fragment program (Phong shading)
```

그림 3. Empty space skipping

4. 구현 결과

본 논문에서 제안하는 알고리즘은 듀얼 인텔 제온(3.06GHz), 4GB의 메인 메모리, NVIDIA GeForce 6800GT(AGP 8x, 256MB) 그래픽스 카드를 장착한 시스템에서 구현됐고, 표 1의 데이터를 이용해서 성능을 측정했다. 그리고 렌더링을 할 때에는 각 볼륨 데이터에 대해 2가지의 전이 함수를 적용했으며, 그 결과는 그림 4에서 볼 수 있다.

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

표 1. 각 볼륨 데이터의 크기

ERT는 앞에서 제안한 두 가지 기법을 모두 구현했는데 3.1.2에서 제안한 알고리즘의 성능이 더 좋게 나타났다. 이는 프레임버퍼를 메인 메모리로 읽어 들일 때 그래픽스 메모리와 메인 메모리 간의 인터페이스가 병목으로 작용하기 때문이다. 이러한 병목현상은 정도의 차이만 있을 뿐, 동일한 GPU를 장착한 PCI Express 16배속 방식의 그래픽스 카드에서도 확인할 수 있다. 표 2는 bighead1을 렌더링할 때 ERT의 구현 방식과 그래픽스 카드 인터페이스에 따른 속도 차이를 보여준다.

ERT 구현 방식	glReadPixels	2-pass
AGP 8X	14.91	16.52
PCI Express 16X	17.24	18.57

표 2. ERT 구현 방식과 그래픽스 인터페이스에 따른 성능 비교

그림 4. 볼륨 데이터의 렌더링 결과 (왼쪽부터 bighead1, bighead2, engine1, engine2)

데이터 종류	bighead1		bighead2		engine1		engine2	
렌더링 해상도	256×256	512×512	256×256	512×512	256×256	512×512	256×256	512×512
기본	13.25	4.48	13.25	4.48	14.64	4.52	14.64	4.52
ERT	16.52 (x1.25)	5.75 (x1.28)	20.99 (x1.58)	7.45 (x1.66)	14.85 (x1.01)	4.60 (x1.02)	18.26 (x1.25)	5.70 (x1.26)
ESS	25.14 (x1.90)	10.04 (x2.24)	25.13 (x1.90)	10.04 (x2.24)	37.59 (x2.57)	15.80 (x3.50)	37.61 (x2.57)	15.81 (x3.50)
ERT + ESS	34.50 (x2.60)	15.61 (x3.48)	41.63 (x3.14)	20.40 (x4.55)	34.89 (x2.38)	14.54 (x3.22)	40.70 (x2.78)	18.48 (x4.08)

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

표 3은 각 볼륨 데이터와 전이 함수(transfer function)에 대해 속도 향상 기법을 적용할 때와 그렇지 않을 때의 성능을 보여준다. 표의 수치는 초당 프레임 수(frames/sec)를 의미하고, 괄호 안의 숫자는 기본 렌더링 알고리즘 대비 속도 향상의 정도를 나타낸다.

본 논문에서 제안하는 ERT는 기본적으로 2-pass 알고리즘이기 때문에 실행에 추가의 부하가 걸린다. 이러한 이유로 매 프록시 도형(proxy geometry)마다 ERT를 적용할 경우, 퐁 쉐이딩을 생략해서 속도를 향상시키는 정도보다 부하로 인한 속도 저하 부분이 더욱 커서 오히려 속도가 더 떨어진다. 데이터의 대부분이 투명한 engine1의 경우에는 그 정도가 더욱 심하다. 이러한 점을 극복하기 위해서 본 논문에서는 ERT 계산을 매 프록시 도형마다 실행시키지 않고 16개의 프록시 도형을 그릴 때 마다 한번만 실행하도록 했다. 표 3을 보면 ESS에 비해 ERT로 인한 속도 향상은 상대적으로 그 비중이 낮음을 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.

5. 결론 및 향후 연구

본 논문에서는 프록시 도형을 사용하면서도 전통적인 볼륨 렌더링 가속 기법인 ERT와 ESS를 동시에 구현하는 알고리즘을 제안했다.

ERT는 스텐실 테스트를 이용하는데, GPU 내에서 스텐실 버퍼와 프레임 버퍼를 직접 접근할 수 없기 때문에 2-패스 방식의 알고리즘으로 구현했다. 후에 GPU 내에서 해당 버퍼를 직접 액세스할 수 있게 되면 보다 효율적인 ERT를 구현할 수 있을 것이다.

ESS는 early Z-test를 이용해서 구현했고, ERT보다 속도 향상에 더 도움이 됐다. 하지만 본 논문에서 구현한 ESS는 픽셀 단위의 작업을 수행하기 때문에 속도 향상에 한계가 있다. 앞으로 팔진 트리(octree)나 BSP 트리 등의 자료구조를 이용해서 보다 효율적인 ESS를 구현할 계획이다.

6. 참고 문헌

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

[2] Cullip and Neumann, "Accelerating Volume Reconstruction with 3D Texture Hardware", TR93-027, University of North Carolina, Chapel Hill, N.C.

[3] Dachille et al., "High-Quality Volume Rendering using Texture Mapping Hardware", Proceedings of EG/SIGGRAPH Workshop on Graphics Hardware 98, August, pp.69-76, 1998.

[4] Engel, Kraus, Ertl, "High-Quality Pre-Integrated Volume Rendering using Hardware-Accelerated Pixel Shading", Proceedings of EG/SIGGRAPH Workshop on Graphics Hardware, 2001.

[5] Hadwiger et al., "High-Quality Volume Graphics on Comsumer PC Hardware", Course Notes 42, SIGGRAPH 2002, July, 2002.

[6] Kilgariff, Fernando, "The GeForce 6 Series GPU Architecture", Chapter 30, GPU Gems 2, pp.471-491, 2005.

[7] Kruger, Westermann, "Acceleration Techniques for GPU Based Volume Rendering", Proceedings of IEEE Visualization 03, 2003.

[8] LaMar, Hamann, Joy, "Multi-resolution Techniques for Interactive Hardware Texturing based Volume Visualization", Proceedings of IEEE Visualization 1999, pp.355-361, 1999.

[9] Lee, "3D Texture based Fast Volume Rendering using Vertex and Pixel Shaders", Proceedings of KIPS Conference, Vol.12, No.1, 2005.

[10] Levoy, "Display of Surfaces from Volume Data", IEEE Computer Graphics and Applications, Vol. 8, No. 3, pp.29-37, May, 1988.

[11] Levoy, "Efficient Ray Tracing of Volume Data", ACM Transactions on Graphics, Vol. 9, No. 3, July, pp.245-261, 1990.

[12] Li, Mueller, Kaufman, "Empty Space Skipping and Occlusion Clipping for Texture-based Volume Rendering", Proceedings on IEEE Visualization 2003, October, 2003.

[13] Li, Kaufman, "Texture Partitioning and Packing for Accelerating Texture-based Volume Rendering", Proceedings of IEEE Visualization 2003, October, 2003.

[14] Meissner, Hoffmann, Strasser, "Enabling Classification and Shading for 3D Texture Mapping based Volume Rendering using OpenGL and Extensions", Proceedings of IEEE Visualization 1999, pp.207-214, 1999.

[15] Rezk-Salama et al., "Interactive Volume Rendering on Standard PC Graphics Hardware using Multi-Texturing and Multi-Stage Rasterization", Proceedings of EG/SIGGRAPH Workshop on Graphics Hardware 2000, 2000.

[16] Van Gelder, Kim, "Direct Volume Rendering with Shading via Three-Dimensional Textures", Proceedings of IEEE Symposium on Volume Visualization 96, pp.23-30, 1996.

[17] Westermann, Ertl, "Efficiently Using Graphics Hardware in Volume Rendering Applications", Proceedings of SIGGRAPH 98, 1998.

[18] http://www.gpgpu.org

[19] NVIDIA, "NVIDIA GPU Programming Guide", 2005.