GPU 기반의 Time-Variant 볼륨 렌더링 프로그램과 사용자 친화적인 전이함수 에디터의 설계 및 구현

Design and Implementation of GPU Based Time-Variant Volume Rendering Program and User-Friendly Transfer Function Editor

이중연, Joong Youn Lee, 허영주, Youngju Hur, 구기범, Gee Bum Koo
한국과학기술정보연구원 슈퍼컴퓨팅센터

요약   여러 학계와 산업계로부터 인체영상과 같은 정적인 볼륨 데이터뿐만 아니라, 유체 흐름과 같은 동적으로 움직이는 Time-Variant 볼륨 데이터에 대한 실시간 렌더링의 요구가 계속되고 있다. 일반적으로 Time-Variant 데이터는 그 크기가 정적 볼륨 데이터의 수배에서 수백 배에 이르러, 이를 실시간으로 가시화하는 데에 많은 어려움이 있어왔다. 한편, PC 그래픽스 하드웨어의 급격한 발전에 따라 슈퍼컴퓨터나 다수의 컴퓨터들을 이용한 병렬/분산 렌더링으로나 가능했던 Time-Variant 볼륨 데이터의 실시간 볼륨 렌더링을 한대의 일반 PC에서 수행하려는 시도가 계속되고 있다. GPU의 꼭지점 및 프래그먼트 쉐이더(vertex & fragment shader)는 수치 계산에 최적화된 벡터 연산과 사용자 프로그래밍 기능으로 빠른 볼륨 렌더링을 일반 PC에서도 가능하게 했다. 본 논문에서는 GPU를 이용해서 Time-Variant 볼륨 데이터를 빠르게 가시화하고, 이렇게 개발한 GPU 볼륨 렌더링 프로그램을 사용자가 사용하기 편리하도록 사용자 친화적인 유저 인터페이스를 설계하고 구현하였다. 특히, 시간에 따라 동적으로 변화해야 하는 전이함수를 최대한 편리하게 생성할 수 있도록 전이함수 에디터에 중점을 두었다.

핵심어: Time-Variant Volume Data, Volume Rendering, Transfer Function, User Interface

1. 서론

전통적으로 볼륨 렌더링 분야는 인체영상으로 대표되는 정적인 볼륨 데이터의 렌더링이 주요 관심대상이었다. 그러나 여러 학계와 산업계로부터 정적인 볼륨 데이터(static volume data)뿐만 아니라, 유체 흐름과 같이 동적으로 움직이는 Time-Variant 볼륨 데이터에 대한 실시간 렌더링의 요구가 계속되고 있다. 일반적으로 Time-Variant 데이터는 그 크기가 정적 볼륨 데이터의 수배에서 수백 배에 이르러, 이를 실시간으로 가시화하는 데에 많은 어려움이 있어왔으며 그동안 슈퍼컴퓨터나 다수의 컴퓨터들을 이용한 병렬/분산 렌더링으로나 가능해왔다. 한편, PC 그래픽스 하드웨어의 급격한 발전에 따라 실시간 볼륨 렌더링을 한대의 일반 PC에서 수행하려는 시도가 계속되고 있다. GPU의 꼭지점 및 프래그먼트 쉐이더(vertex & fragment shader)는 수치 계산에 최적화된 벡터 연산과 사용자 프로그래밍 기능으로 빠른 볼륨 렌더링을 일반 PC에서도 가능하게 했다. 본 논문에서는 GPU를 이용해서 개발한 Time-Variant 볼륨 렌더링 프로그램을 사용자가 사용하기 편리하도록 사용자 친화적인 유저 인터페이스를 설계하고 구현하였으며 그 방법에 대해 논하고자 한다. 특히, Time-Variant 볼륨 데이터에 최적화된 사용자 친화적인 전이함수 (Transfer Function) 에디터에 대해 주로 설명한다.

2. GPU 기반 볼륨 데이터 렌더링

2.1. 샘플링

볼륨 데이터에서 적절한 복셀(voxel)을 찾아오는 샘플링 과정은 볼륨 렌더링에 필요한 전체 시간 중 매우 많은 부분을 차지하기 때문에 이를 빠르게 처리하는 것은 전통적인 볼륨 그래픽스에서 주요한 연구 주제 중 하나였다. 한편, 그래픽스 하드웨어의 텍스쳐 매핑 기능은 삼선형보간(tri-linear interpolation)을 하드웨어적으로 매우 빠르게 처리해주며 3차원 텍스쳐 매핑은 볼륨 데이터의 복셀 샘플링과 기술적으로 거의 동일하기 때문에 이를 볼륨 렌더링에 이용하면 복셀 샘플링을 실시간으로 처리할 수 있게 된다[1]. 이렇게 텍스쳐 매핑을 이용해서 샘플링을 하기 위해서는 텍스쳐가 입혀질 도형이 필요한데 이를 대리 도형(proxy polygon)이라고 한다. 보통 2차원 텍스쳐를 이용해서 볼륨

렌더링을 구현할 경우에는 대리 도형으로 평면을 텍스처에 평행하도록 배치하고, 3차원 텍스쳐를 이용해서 볼륨 렌더링을 구현할 경우에는 평면을 영상평면과 평행하도록 배치한다[3]. 본 논문에서는 3차원 텍스쳐 방식으로 볼륨 렌더러를 구현했고, GPU의 꼭지점 프로그램을 이용해서 대리 평면이 영상평면과 항상 평행하도록 했다. 꼭지점 프로그램에서는 각 꼭지점에 모델뷰 행렬(modelview matrix)과 투영 행렬(projection matrix)를 곱함으로써 각 꼭지점들에 대한 스크린 좌표를 계산한다. 여기서, 모델뷰 행렬에 회전 행렬을 적용시킬 경우, 꼭지점이 실질적으로 회전하게 되는데, 대리 평면은 항상 영상 평면에 평행해야 하기 때문에 회전하면 안된다. 따라서, 꼭지점 쉐이더에서 꼭지점에 모델뷰 행렬을 무시하고 투영행렬만 곱해지도록 하여 꼭지점 자체는 회전되지 않도록 했고, 대신 각 대리 평면의 꼭지점에 할당되는 텍스쳐 좌표에 모델뷰 행렬을 곱함으로써 볼륨 데이터가 회전하는 것처럼 보이도록 했다.

2.2. 쉐이딩

샘플링된 복셀들은 전이함수(transfer function)을 통해 색깔 및 투명도가 결정되고 여기에 쉐이딩을 통해 음영이 입혀지게 된다. 이러한 작업은 프래그먼트 프로그램에서 수행된다. 전이 함수는 종속 텍스쳐 기법을 이용하면 쉽게 구현이 가능하다[2]. 본 논문에서는 2차원 텍스쳐를 이용한 2차원 전이함수를 사용했는데, 복셀값과 복셀의 법선벡터의 크기를 인덱스로 했다. 법선벡터의 크기가 크면 복셀값이 급격히 변화하므로 물질의 경계 부분이라는 뜻이고 그 크기가 작으면 복셀값의 변화가 거의 없는 균일(homogeneous) 영역이라는 뜻이다. 일반적으로 물질의 경계면이 중요하게 인식되므로 이 부분의 불투명도를 크게 하고 경계면이 아닌 균일 영역은 상대적으로 덜 중요하므로 투명도를 크게 했다[5,7]. 쉐이딩은 퐁의 조명모델(Phong's illumination model)을 사용했는데, 이것을 계산하기 위해 법선벡터를 복셀값과 함께 3차원 텍스쳐에 저장했고 이 때문에 텍스쳐의 크기가 본래 볼륨 데이터의 3배 크기가 되었다.

3. 속도 향상 기법

본 논문에서 구현한 볼륨 렌더러에서는 모든 샘플링되는 복셀에 대해 프래그먼트 프로그램에서 퐁 쉐이딩을 적용했다. 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 이른 광선 단절법과 빈 공간 건너뛰기를 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.

3.1. 이른 광선 단절법

이른 광선 단절법(Early Ray Termination: 이상 ERT)은 대표적인 볼륨 렌더링 방법인 광선 추적법(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[8]. 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 1과 같다.

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

수식 1. 앞-뒤 순서를 위한 블렌딩 함수

본 논문에서는 불투명한 픽셀의 스텐실 버퍼를 1로 설정하는 것을 두 가지 방법을 사용해서 구현했다.

3.1.1. 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 메인 메모리로 다시 읽어 들여야 하기 때문에 속도 저하가 일어날 수 있으며 실제로 구현한 결과도 뒤에 설명할 다중패스 렌더링 방식에 비해 성능이 낮았다.

3.1.2. 다중패스 렌더링

두 번째 방법은 알파 텍스처와 다중 패스 렌더링을 통해

서 그래픽스 메모리의 내용을 메인 메모리로 읽어 들이지 않고 스텐실 버퍼를 설정하는 방법이다. 우선, 매번 대리 도형을 그릴 때마다 렌더링된 결과 중 알파값을 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 1과 같고, 스텐실 함수 설정 방법은 그림 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 1. 스텐실 버퍼를 이용한 이른 광선 단절법 알고리즘

```
glStencilFunc(GL_NOTEQUAL, 1, 1);

glStencilOp(GL_KEEP, GL_REPLACE, GL_KEEP);
```

그림 2. glStencil 함수의 설정 코드

3.2. 빈 공간 건너뛰기

본 논문에서는 이른 광선 단절법과 함께 빈 공간 건너뛰기(Empty Space Skipping : 이상 ESS)도 구현했다. 즉, 전이함수가 적용되지 않는 복셀 값을 가지는 복셀에 대해서는 빈 공간으로 취급해서 쉐이딩을 수행하지 않고 건너뛰도록 하는 방법으로 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[6, 10]. 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 버퍼를 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 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 3. 빈 공간 건너뛰기 알고리즘

4. Time-Variant 데이터 렌더링

일반적으로 Time-Variant 데이터는 정적 데이터(static data)에 비해 그 크기가 커서 일반적으로 최대 1GB의 크기를 가지는 GPU 메모리는 물론, 훨씬 큰 용량을 지닌 메인 메모리에조차 전체 데이터를 다 올릴 수 없을 수도 있다. Time-Variant 데이터는 n프레임의 정적 데이터들이 모여서 이루어진 것이기 때문에 크기가 프레임 수만큼 커질 수 밖에 없다. (n프레임이라면 정적 데이터의 n배) 반면 GPU 메모리는 그 크기에 한계가 있어 Time-Variant 데이터의 모든 프레임을 한꺼번에 로드(load)할 수 없다. 이러한 문제를 극복하기 위해서는 볼륨 데이터를 메인메모리 또는 하드디스크에 저장하고 있다가 가시화에 필요한 프레임이 있으면 그때그때 GPU 메모리로 복사하는 방법으로 렌더링하는 것이 필요하다.

4.1. 데이터 생성

GPU 볼륨 렌더링에서는 모든 볼륨 데이터를 3차원 텍스쳐로 생성한다. 이때, 필요한 3차원 텍스쳐에는 크게 3가지가 있는데, 볼륨 텍스쳐, 그라디언트(gradient) 텍스쳐, norm 텍스쳐이다. 한편, 여러 프레임으로 구성된 Time-Variant 데이터의 경우에는 모든 프레임들을 미리 텍스쳐로 생성할 필요가 있다. 렌더링에 필요한 데이터가 볼륨 텍스쳐 뿐일 경우에는 하드디스크 또는 메인 메모리에 저장된 원시 데이터(raw data)를 읽어서 실시간으로 텍스쳐를 생성할 수 있지만, 그라디언트 또는 norm 데이터가 필요할 경우에는 이

를 실시간으로 계산하며 텍스처로 생성할 수 없다. 이러한 이유로 모든 프레임들에 대한 그라디언트 및 norm 데이터를 미리 생성해야 한다.

4.2. 데이터 로딩

볼륨 렌더링에 필요한 텍스쳐들을 모두 GPU 메모리에 올리기에는 텍스쳐들의 크기가 너무 큰 경우가 많다. 특히, 복셀(또는 텍셀)의 포멧이 부동소수점(floating point)일 경우에는 복셀 포멧이 unsigned char인 경우에 비해 4배의 크기가 필요하다. 이러한 이유로 볼륨 렌더링을 위한 텍스쳐들을 미리 생성해 놓되, 메인 메모리 또는 하드디스크에 생성해 놓고 필요에 따라 데이터를 GPU 메모리에 올리는 방법으로 최대한 필요한 GPU 메모리의 크기를 줄여야 한다. 이러한 이유로 본 논문에서는 하드디스크 – 메인 메모리 – GPU 메모리로 연결되는 메모리 계층 구조를 만들고, 메인 메모리와 GPU 메모리에, 또 하드 디스크와 메인 메모리에 캐쉬를 두어 최대한 데이터 교환에 따른 속도 저하를 없애도록 하였다. 이를 구현하기 위해 GPU에 현재 렌더링에 필요한 볼륨 데이터 이외에 앞, 뒤 프레임의 볼륨 데이터를 미리 GPU에 올리도록 했다. 즉, 다음 프레임에 렌더링할 볼륨 데이터를 미리 텍스쳐로 만들어서 텍스쳐 메모리에 올려둔 것이다. 이러한 정책은 일반적으로 프레임 순서대로 렌더링하는 Time-Variant 데이터의 특징을 이용한 것으로 현재 프레임을 디스플레이하는 동안 미리 다음 프레임을 GPU에 올려두기 때문에 전체 렌더링 속도의 향상에 도움을 주었다. 그러나 만약 순서대로 렌더링하지 않고 랜덤하게 디스플레이해야 할 필요가 있을 대는 오히려 속도에 저하를 가져왔다. 마찬 가지로 현재 프레임의 앞 뒤 10 프레임씩을 하드 디스크에서부터 메인 메모리로 미리 올려놓고 렌더링하도록 했다.

Time-Variant 데이터의 각 프레임을 로드할 때, 매번 glTexImage3D 함수를 호출해서 각 프레임마다 3D 텍스쳐를 생성하면 OpenGL에서 매 프레임마다 새로운 3D 텍스쳐 객체를 생성해야 하기 때문에 시간이 오래 걸릴 뿐만 아니라 리소스의 낭비도 심해진다. 이러한 점을 해결하기 위해서는 각 프레임을 로드할 때, glTexSubImage3D를 이용해서 기존의 3D 텍스쳐에서 텍셀버퍼의 내용만을 변경하도록 하면 된다. glTexImage3D를 사용할 때보다 glTexSubImage3D를 사용할 때 렌더링 성능이 훨씬 향상된 것을 느낄 수 있다.

5. 유저 인터페이스와 전이함수 편집기

5.1. 유저 인터페이스

본 논문에서 구현한 볼륨 렌더러를 사용자가 쉽게 사용할 수 있도록 유저 인터페이스를 설계, 구현했다. 전체적인 유저 인터페이스의 모습은 그림 4, 6, 7에서 볼 수 있다. 메인 렌더링 화면은 마우스 조작으로 볼륨 데이터의 회전 및 축소, 확대가 가능하게 했다. 오른쪽의 라디오 박스로 조작할 대상을 선택할 수 있도록 했으며, 직교투영(parallel projection)과 원근투영(perspective projection)을 선택할 수 있도록 했다. 또, 아래쪽의 슬라이드를 이용하면 샘플링 레이트(sampling rate)를 변경할 수 있다. Time-Variant 데이터의 경우에는 Navigation 버튼들을 이용해서 플레이할 수 있다. 광원은 렌더링 윈도우 상단의 Light 메뉴를 이용해서 추가/삭제/변경할 수 있다. 광원은 최대 5개까지 추가할 수 있으며 현재는 앰비언트 광원(ambient light)과 방향 광원(directional light)을 지원한다. 광원 추가 및 변경 윈도우는 그림 5와 같다. 이 윈도우를 이용해서 광원의 종류, 색깔 및 방향을 지정할 수 있다. 이렇게 추가한 광원들은 메뉴의 Light On/Off 메뉴를 이용해서 임시로 키고 끌 수 있다. 그림 6을 보면 광원을 변경한 화면을 볼 수 있는데, 동일한 전이함수를 사용했음에도 광원이 다르기 때문에 렌더링 결과가 다른 것을 알 수 있다.

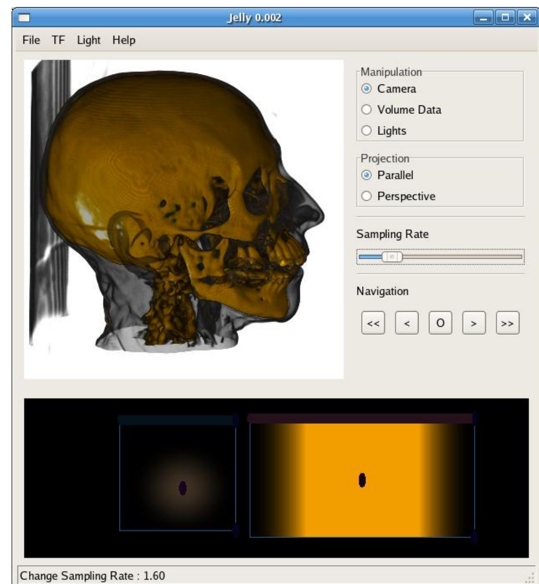

그림 4. 볼륨 렌더러 유저 인터페이스

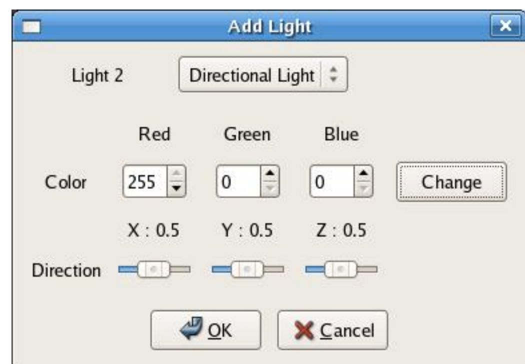

그림 5. 광원 추가 윈도우

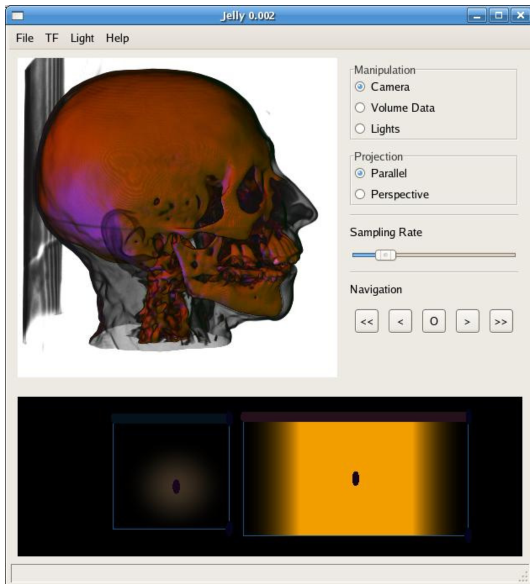

그림 6. 광원 변경 화면

5.2. 전이함수 에디터

본 논문에서 구현한 볼륨 렌더링 프로그램에는 사용자가 손쉽게 전이함수를 변경할 수 있도록 하는 전이함수 에디터를 추가했다. 이 전이함수 에디터에서는 변경된 전이함수가 실시간으로 렌더링에 반영되어 즉각적인 피드백을 사용자에게 줌으로써 매우 효율적으로 효과적인 전이함수를 생성할 수 있도록 했다. 이 전이함수 에디터에서는 1D, 타원, 삼각형, 무지개 등 4가지 종류의 위젯을 지원하며 사용자는 이 위젯들을 자유롭게 추가, 배치함으로써 전이함수를 생성할 수 있다. 전이함수의 변경에 따른 렌더링 결과의 변경은 그림 7에서 볼 수 있다.

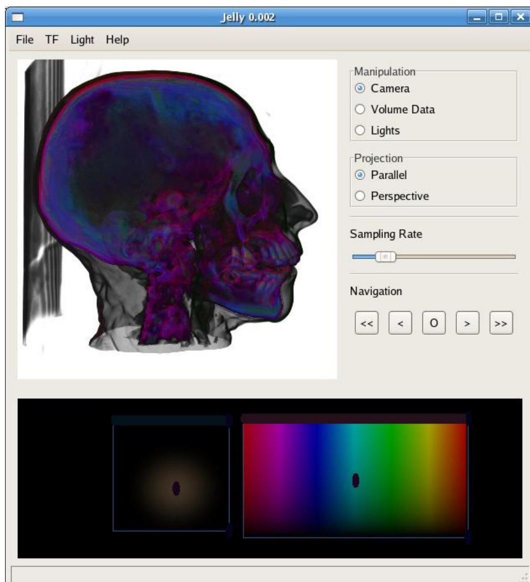

그림 7. 전이함수 변경 화면

여러 프레임으로 구성된 Time-Variant 볼륨 데이터는 그 특성상 시간 프레임이 진행될수록 데이터의 특징이 지속적으로 변화하므로 이를 위한 전이함수 역시 데이터의 특징이 변화함에 따라 함께 변화해야 한다. 그러나 수백 프레임으로 이루어진 볼륨 데이터의 전이함수를 일일이 변경하기란 불가능하다. 이를 위해 본 논문에서는 사용자가 중간의 특정 프레임의 전이함수만 생성해주면 자동으로 전체 전이함수를 생성하도록 하는 반자동방식(Semi-Automatic)을 사용해서 보다 편리하게 전이함수를 생성할 수 있도록 했다. 사용자가 전이함수를 지정한 프레임을 렌더링할 때는 물론 그 전이함수를 사용해서 렌더링하도록 했고, 중간 프레임을 렌더링 할 때는 앞, 뒤의 전이함수를 선형보간하여 생성한 전이함수를 이용해서 렌더링하도록 했다. 그림 8, 9, 10은 turbulent vortex 데이터를 렌더링한 화면이다. 이 데이터는 128X128X128 크기에 32비트의 부동소수점 복셀로 구성되어 있고 총 100 타임 스텝을 가지고 있다. 그림 8은 이 데이터의 첫 번째 프레임이고 그림 9는 20번째 프레임이다. 그림 10은 10번째 프레임으로, 앞의 두 프레임에서의 전이함수를 보간(interpolation)해서 생성한 전이함수를 적용한 화면이다.

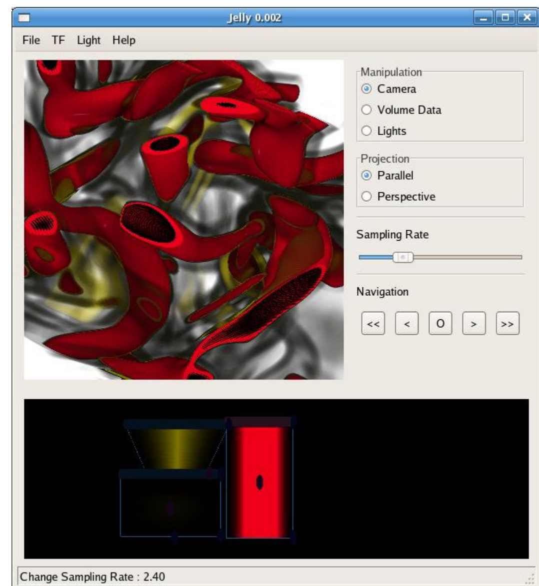

그림 8. Turbulent Vortex 데이터 (첫번째 프레임)

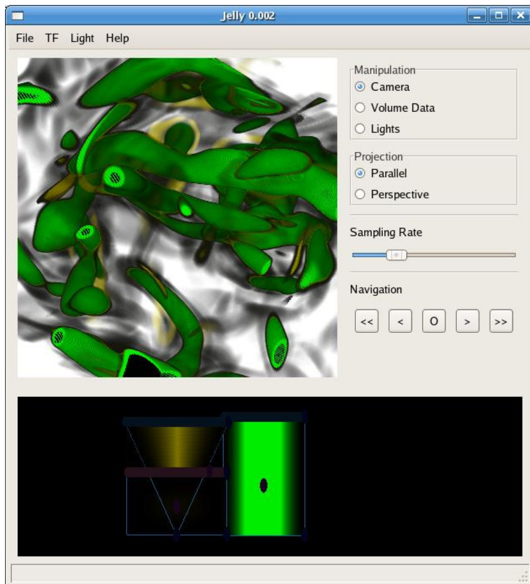

그림 9. Turbulent Vortex 데이터 (20번째 프레임)

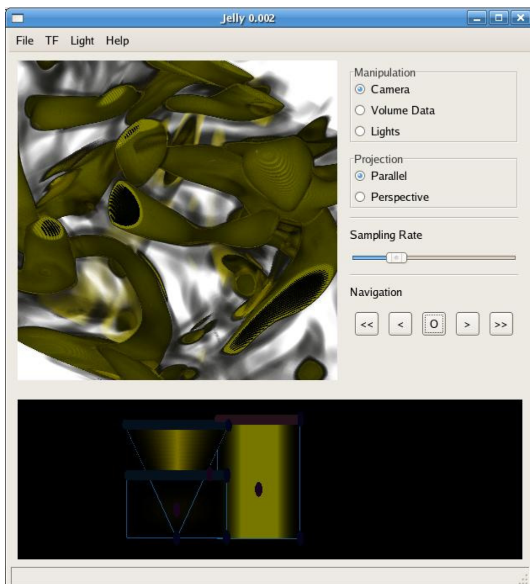

그림 10. Turbulent Vortex 데이터 (10번째 프레임)

6. 결론

본 논문에서는 시간에 따라 동적으로 변화하는 Time-Variant 볼륨 데이터를 빠르게 렌더링하기 위한 렌더링 기법을 소개하고, 편리하게 전이함수 세트를 생성할 수 있도록 하는 전이함수 에디터를 소개하였다. 그러나 Time-Variant 데이터 렌더링의 경우 한 프레임이 GPU 메모리 크기를 넘어서는 경우 제안한 알고리즘을 사용하지 못한다는 문제가 있다. 전이함수 에디터의 경우 편리하게 전이함수 세트를 생성할 수 있도록 하는 반자동 전이함수 에디터를 소개했다. 그러나 앞, 뒤의 전이함수를 단순히 선형보간하는 방식으로는 정확한 중간 프레임의 전이함수들을 생성할 수 없었다. 보다 정확한 전이함수들을 생성하기 위한 보다 추가적인 연구가 필요하다고 하겠다.

참고문헌

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

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

[3] Hadwiger, Kniss, Engel, Rezk-Salama, "High-Quality Volume Graphics on Comsumer PC Hardware", Course Notes 42, SIGGRAPH2002, July, 2002.

[4] Kilgariff, Fernando, "The GeForce 6 Series GPU Architecture", Chapter 30, GPUGEMS2, 2005, pp. 471-491

[5] Kniss, Kindlmann, Hansen, "Multi-Dimensional Transfer Functions for Interactive Volume Rendering", IEEE Transactions on Visualization and Computer Graphics, Vol. 8, No. 3, pp. 270-285, July, 2002

[6] Kruger, Westermann, "Acceleration Techniques for GPU Based Volume Rendering", Proceedings on IEEE Visualization 03, 2003

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

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

[9] GPGPU.ORG, http://www.gpgpu.org

[10] NVIDIA, "NVIDIA GPU Programming Guide version 2.4, 2005

[11] 이중연, "GPU를 이용한 3차원 텍스쳐 기반 볼륨 렌더링의 속도향상 기법", KCC2006, 한국정보과학회, 2006.

[12] Hiroshi Akiba, Nathan Fout, and Kwan-Liu Ma, Proceedings of Eurographics Visualization Symposium, 2006.