ETRI

전자통신동향분석 제 22권 제 4호 2007년 8월

렌더링 가속화 기술 동향

The Recent Trends of Rendering Acceleration Technologies

융합 시대를 주도할 디지털콘텐츠 기술 특집

남승우 (S.W. Nam) 렌더링기술연구팀 선임연구원
김해동 (H.D. Kim) 렌더링기술연구팀 선임연구원
김성수 (S.S. Kim) 렌더링기술연구팀 연구원
최진성 (J.S. Choi) 렌더링기술연구팀 선임연구원

목 차

* 본 연구는 정보통신부 및 정보통신연구진흥원의 IT 신성장동력핵심기술개발사업의 일환으로 수행하였음. [2006-S-045-01, 기능 확장형 초고속 렌더러 개발]

컴퓨터 그래픽스를 이용한 디지털 콘텐츠를 제작 및 생산함에 있어서 마지막 단계에서 렌더링 과정을 꼭 거쳐야 하기 때문에 렌더링 부분은 아주 중요하다. 렌더링해야 할 디지털 콘텐츠에는 게임과 같이 실시간성이 아주 중요한 콘텐츠가 있으며, 영화와 같이 영상의 높은 품질을 요구하는 콘텐츠가 있다. 본 고에서는 영화와 같이 고품질을 요구하는 콘텐츠에 대한 렌더링 기술에 대하여 다루고자 한다. 영화의 한 장면과 같이 복잡하며 높은 해상도를 갖는 영상을 기존 단일 CPU 및 소프트웨어 렌더러를 이용하여 렌더링하는 데 아주 많은 시간이 걸린다. 본 고에서는 렌더링 시간을 줄이며 높은 품질의 렌더링 결과를 얻는 기술을 3가지 부분에서 소개하고자 한다. 첫번째 방법에는 수십 개에서 수천 개의 CPU를 이용하거나 PC를 클러스터링하는 방법이고, 두번째는 기존 GPU의 기술이 아주 빨리 발전하여 CPU 보다 빠른 성능을 갖기 때문에 GPU를 활용하여 가속화하는 방법이 있으며, 세번째는 전용 하드웨어를 제작하여 렌더링을 가속하는 방법이 있다. 위의 방법들에 대한 기술 동향에 대하여 살펴보도록 한다.

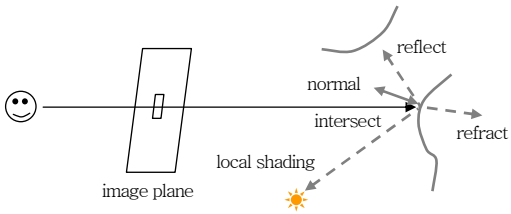
I. 서론

화가는 붓과 물감으로 캔버스에 색을 칠하여 하나의 작품을 만들고, 카메라 작가는 카메라를 이용하여 필름에다가 하나의 작품을 담듯이 컴퓨터를 이용하여 하나의 그림을 만들어 내는 것이 렌더링 기술이다. 컴퓨터 그래픽스 기술은 모델링, 애니메이션, 렌더링 기술로 크게 분류할 수 있다. 모델링 기술은 실제 물체와 크기와 모양이 유사하게 점, 면, 선 등을 이용하여 물체의 형태를 만드는 기술이고, 애니메이션 기술은 프레임(frame)마다 물체의 형태가 움직이게 하는 기술이며, 렌더링 기술은 모델링된 물체의 표면에 색을 입히는 기술이라 할 수 있다. 색을 입히기 위하여 필요한 요소는 광원, 매질, 카메라 특성이다. 여기서 빛을 받아서 물체의 표면이 어떻게 반응하는가와 같은 물체 표면 특성을 매질이라 한다. 그러면 렌더러라 함은 모델링 데이터와 위에서 언급한 색을 입히기 위한 요소를 입력으로 받아서 이차원의 이미지를 출력하는 것이다. 기존 상용 렌더러 중에서 픽사(pixar)의 렌더맨(RenderMan)은 REYES 구조[1]를 계속 발전시켜 왔으나 2005년을 기점으로 전역 조명(global illumination) 렌더링을 위한 광선 추적법을 추가하였다. 여기서 REYES 구조는 광선 추적법(ray tracing) 보다 계산량이 많지 않으므로 속도가 빠른 장점이 있어서 아직도 많이 사용되고 있다. 그러나 최근 이러한 지역 조명(local illumination)에 기반한 렌더링 보다는 전역 조명에 기반한 렌더링이 많이 쓰이고 있고, 전역 조명에 기반한 렌더링이 물리적으로 빛과 매질의 상호작용을 정교하게 해석하기 때문에 지역 조명에 기반한 방법보다 좀더 실사에 가까운 렌더링 효과를 얻을 수 있다. 전역 조명에 기반한 렌더링의 단점은 지역 조명에 기반한 렌더링보다 시간이 많이 소요된다. 빛과 매질의 상호작용을 물리적 기반으로 렌더링하는 대표적인 알고리듬으로는 광선 추적법이 있다. 광선 추적법은 기본적으로 시간이 매우 많이 걸리는 계산이어서 광선 추적법의 시간을 단축시키기 위하여 포톤 맵과 레디언스 캐시(radiance cache) 방법 등이 있다. 포톤 맵을 이용하더라도 마지막에는 final gather를 하게 되어 시간이 많이 걸리는 단점이 있다. 본 고에서는 위의 전역 조명에 기반한 방법(특히 광선 추적법 중심)을 가속하기 위한 세 가지 면에서 기술 동향을 설명하고자 한다. 첫째는 CPU를 이용한 가속 방법, 둘째는 GPU를 이용한 가속 방법, 셋째는 전용 하드웨어를 이용한 가속 방법 등에 대하여 설명한다. 위와 같이 나눈 것은 Wald 등이 유로그래픽스에 발표한 STAR 논문을 참조하였다[2]. II장에서는 병목이 되는 렌더링 알고리듬에 대하여 살펴보고, CPU 기반의 렌더링 가속화 기법을 설명한다. III장에서 GPU를 이용한 렌더링 가속 기술에 대하여 기존 연구들에 대하여 논하고, IV장에서는 병목이 되는 알고리듬을 전용 하드웨어로 설계하여 가속화하는 기술에 대하여 논하고자 한다. 마지막으로 V장에서는 앞으로의 렌더링 가속 부분에서의 전망과 결론을 맺는다.

II. CPU를 이용한 가속

1. 광선 추적법

3차원 장면(물체, 조명, 카메라 등으로 모델링된 장면)이 카메라를 통해서 2차원 영상을 보여주는 것은 조명과 물체가 상호작용하여 카메라를 통하여 빛이 들어오는 과정이며, 이러한 과정을 역으로 카메라에서부터 물체를 향하여 광선을 쏘아 이를 추적하여 영상에 색깔을 계산하는 알고리듬이 광선 추적법이다. 광선이 물체에 충돌하면 교차된 물체 표면에서 모든 방향으로 광선을 발생시키고(이를 2차광선이라고 하자) 2차 광선들이 또 다른 물체에 교차하여 3차 광선들을 발생시킨다. 이때 차수를 더해갈수록 광선의 수는 기하급수적으로 많아지며 계산량도 기하급수적으로 증가한다. 이러한 계산량을 줄이기 위하여 2차광선은 3가지로 형태로 나누어 발생시킨다. 하나는 섀도 광선을 발생하고 다른 하나는 표면에서 반사하여 반사광을 발생하고, 마지막으로 물체를 투과하는 투과광선을 발생한다. (그림 1)은 광선



(그림 1) 광선 추적법에서의 2차 광선 발생

이 표면이 충돌할 때 발생하는 2차 광선의 예이다.

광선이 물체 표면과 충돌하면 표면에 대한 광량(radiance)을 계산하는데 식 (1)과 같다.

$$L(p,\omega_0)=\int_{s^2} f(p,\omega_0,\omega_1)L(p,\omega_i)\cos(\theta_i)d\omega_i \quad (1)$$

(1)로부터 2차 광선을 3가지 형태로 발생시키되 (그림 1)과 같이 각 방향으로 확률적으로 분산광을 발생시켜서 광량을 계산하면 (1)이 (2)와 같이 된다.

$$\begin{aligned} L(p,\omega_0) = L_{ambient} &+ f_{direct}\sum L(p,\omega_{light}) \\ &+ f_{specular}\sum L(p,\omega_{specular}) \\ &+ f_{trans}\sum L(p,\omega_{trans}) \end{aligned} \quad (2)$$

여기에서 f 값은 BRDF 값이며, 즉 표면에서 입사 빛에 대한 임의의 방향으로 방출되는 빛의 양이 얼마인지를 결정하는 함수이고, L 값은 p 위치에서의 색깔 값이다. (그림 2)는 광선 추적 알고리듬을 의사코드(pseudo code)로 작성한 것이다. 광선 추적법은 각 픽셀마다 샘플링을 통하여 광선을 생성한다.

그리고 교차점이 있으면 교차점에 대한 색깔 값을 가져오고 교차점에서부터 2차 광선을 쏘아 2차 광선이 교차하는 점에서 다시 색깔 값을 가져오고 3차광선도 마찬가지로 반복적으로 수행한다. 각각의 1차 2차 광선에서 가져온 값에 BRDF 함수 값을 곱하여 모두 더해주면 현재 픽셀에서의 최종 색깔 값이 된다. 여기서 kd, kr, kt는 각각 직접광, 반사광, 투사광에 대한 BRDF 함수 값이다.

```
for each pixel
  for each sample
    ray generation
    ray-object intersection
    if(intersect)
      Id = local shading
      if(reflection)
        reflection ray generation
        Ir = trace ray
      endif
      if(refraction)
        refraction ray generation
        It = trace ray
      endif
      I = kd×Id+ kr×Ir+ kt×It
    endif
  end for
end for
```

(그림 2) 광선 추적 알고리듬의 의사코드

2. CPU를 이용한 가속 기술

국내에서는 일반적으로 소프트웨어를 이용한 가속은 첫번째, 소프트웨어를 CPU에 최적화하는 방법이 있으며, 두번째는 CPU를 코어를 많이 사용하여 알고리듬을 병렬로 처리하는 방법이 있다. 최근에는 cell processor의 등장으로 cell로 구현한 사례가 있다. 광선 추적법 자체는 병렬성을 갖고 있으나 메모리와 CPU 간의 데이터 교환 시간과 동기를 맞추는 기본적인 시간이 들어간다. 이를 해결하기 위한 시도로서 수퍼컴퓨터와 같은 공유메모리 시스템을 이용하는 방법이 있고, 다른 방법으로는 PC간에 클러스터링하는 방법이 있다. PC간의 클러스터링 방법은 공유메모리 방법에 비하여 작은 비용으로 구성할 수 있다는 장점이 있으나 CPU간 통신 대역폭과 메모리 대역폭이 낮은 것이 단점이다.

가. Star Ray

Muuss 등은 처음으로 이러한 시도를 하였으며, 비슷한 하드웨어 플랫폼에서 Utah 대학의 Parker 등은 1999년도 논문에 60개의 CPU로 (그림 3)과 같은 512×512 크기의 이미지를 렌더링하는 데 초당 4프레임의 성능을 갖는 star ray를 발표하였다[3].

(그림 3) Utah 대학의 Interactive 광선 추적의 예

나. OpenRT

2001년에 Wald 등이 OpenRT라는 실시간 광선 추적 엔진을 제안하였다. 현재는 웹을 통하여 공개하여 현재 계속적으로 API가 향상되어 가고 있다. OpenRT의 커널(kernel)의 특징은 메모리 접근을 최소화하였고, 프로세서의 캐시 적중률을 높이도록 설계하였으며, 최신 CPU의 SIMD extension을 지원하는 것이다. 또한 광선 추적법에서 광선을 재귀적(recursive)으로 처리하지 않고 패킷 단위로 추적, 충돌처리, 셰이딩(shading)하여 메모리 접근을 최소화하고, SSE 명령어를 사용하였다.

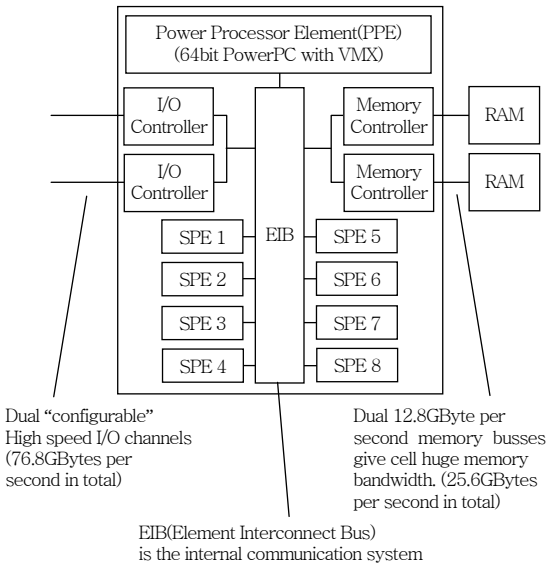
OpenRT는 (그림 4)와 같은 일명 "soda hall" (250만 폴리곤으로 구성)이라는 1024×1024 크기의 영상 한 장을 Pentium-IV 2.5GHz 1개의 CPU에서 초당 4.1 프레임의 속도를 나타내었고, 셰이딩을 포함하면 초당 1.8프레임의 성능을 나타내었다. 이러한 결과로 보면 광선 추적 부분뿐만 아니라 셰이딩 부분도 병목이 되고 있음을 알 수 있다. OpenRT는 또한 분산 병렬 처리를 지원하는데 PC를 클러스터링하여 성능의 향상을 도모할 수 있다. 분산 처리에서 데이터 교환을 위한 라이브러리로 MPI 혹은 PVM 등이 있는데, 실시간으로 성능을 내기 위해서는 이러한 라이브러리를 사용할 수 없어서 UNIX TCP/IP 프로토콜을 이용했다. 다이내믹한 로드 밸런싱을 효율적으로 하기 위하여 이전 업무가 끝이 나면 다음 업무를 미리 가져와서 다음 업무를 시작하기 위해 기다리는 시간을 제거하였다. 이외 프레임간에 차이가 없는 부분은 네트워크를 통하여 전송을 하지 않고 프레임간에 차이가 있는 부분만을 전송하며, multi-threading을 지원하여 성능을 높였다.

OpenRT의 분산 병렬 처리의 결과는 다음과 같다. 48 CPU까지는 테스트를 하였으며 대부분의 장면에 대하여(전역조명의 경우) 선형적으로 성능이 증가하나 복잡한 장면(1천만 폴리곤 이상, 지역조명인 경우)에서는 24 CPU 이상에서 성능이 증가하지 않는 현상을 보인다.

(그림 4) OpenRT 성능 테스트에 사용된 "Soda Hall"

3. Cell 프로세서를 이용한 가속 기술

Benthin 등은 cell 프로세서를 이용하여 광선 추적법을 구현하였다[4]. Cell 프로세서의 구조는 (그림 5)와 같이 8개의 SPE와 1개의 PPE로 구성되어 있다[5]. 각 SPE는 일종의 128bit-SIMD RISC 프로세서로서 순차적으로 명령어를 수행하기 때문에 데이터 순서의 의존성 혹은 분기로 인한 파이프라인 스톨(pipeline stall)이 생길 수 있다. SPE의 명령어들은 SIMD 프로세싱을 할 수 있게 설계되어 있고 1사이클에 1개의 스루풋을 가지며 2~7 사이클의 latency를 갖는다. 각 SPE들은 세 개의 레벨을 갖는 메모리 구조인데, 한 개의 128×128bit 레지스터 파일(register file), 한 개의 256KB의 로컬 스토어



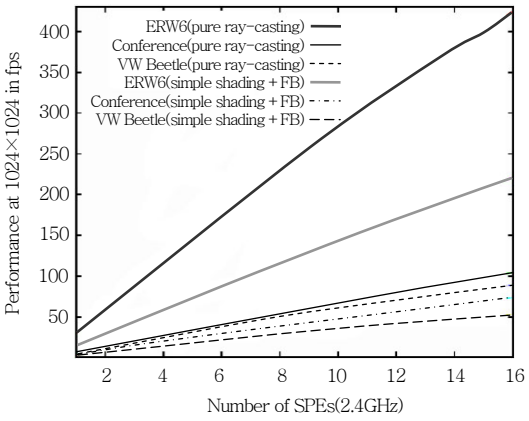
(그림 5) Cell 프로세서 구조

(local store), 메인 메모리로 구성된다. 메인 메모리의 접근은 DMA 컨트롤러를 통해서 이루어지고 25.6GB/s의 대역폭을 갖는다.

메모리 접근 시 latency(수백 SPE 내부 사이클)가 커서 비동기(asynchronous) 전송을 지원하고 광선 추적법과 같은 비규칙적인 메모리 접근이 많이 이루어지는 곳에는 도전적이라 할 수 있다. 프로그래밍 모델은 임의의 커널을 임의의 SPE에서 수행하고 결과를 다른 SPE에 넘겨주는 캐스캐이드(cascade) 형태가 있을 수 있고, 광선 추적 알고리듬 전체를 SPE에서 행할 수도 있다. 후자의 경우는 픽셀마다 하나의 SPE가 할당될 수도 있고 광선마다 SPE가 할당될 수 있어서 병렬 처리가 가능하다. 전자의 단점은 SPE간의 데이터 흐름이 생기는데 SPE간의 내부 대역폭(300GB/s)이 넓긴 하지만 SPE 가운데 하나라도 스톨이 생기면 모든 SPE가 스톨되므로 SPE의 사용성이 떨어질 수 있고 실제 떨어진다. 이러한 cell 프로세서의 특징때문에 장면 데이터를 로딩하기 위한 메모리 접근 방법에 신중해야 하며 장면 데이터 캐싱 방법이 아주 중요하다. Benthin 등은 후자의 방법을 채택하였으며, BVH를 공간 데이터 구조로 채택하였다. 그리드(grid)로 구성하는 것은 cell과 같은 스트리밍(streaming) 프로세서에 적합한 구조이지만 동적인 물체의 처리 및 2차 광선 추적에 그리드보다는 BVH가 더 적합하다고 판단하였다. BVH 노드 캐시를 패어(pair)로 두고, 4-way associative 캐시(a least-recently-used policy)를 사용하여 캐시 적중률을 높였다고 한다. (그림 6)은 cell 프로세서 시스템과 CPU 간의 3개의 장면에 대하여 비교한 데이터이고, (그림 7)은 각 장면에 대하여 SPE 개수에 따라 성능을 나타낸 표이다. (그림 6)에서 보는 바와 같이 cell에서 셰이딩 성능은 광선 교차 테스트와 장면 탐색(traversal)보다 좋지 않은 것으로 나타났다. 실제 실험은 광선 추적이 아니라 광선캐스팅을 한 것으로 하나의 SPE는 CPU와 동등한 성능을 보여 주고 있으며 셰이딩과 같이 메모리 접근이 많은 경우 또 다른 병목이 되고 있다. 지금까지 소프트웨어 기반의 기존 프로세서를 이용한

Scene	ERW6	Conference	VW Beetle
ray casting, no shading			
2.4GHz x86	28.1	8.7	7.7
2.4GHz SPE	30.1 (+7%)	7.8 (-12%)	7.0 (-10%)
Single-Cell	231.4 (8.2x)	57.2 (6.5x)	51.2 (6.6x)
Dual-Cell	430.1 (15.3x)	108.9 (12.5x)	91.4 (11.8x)
PS3-Cell	270.0 (9.6x)	66.7 (7.6x)	59.7 (7.7x)
ray casting, simple shading			
2.4GHz x86	15.3	6.7	6.6
2.4GHz SPE	14.9 (-3%)	5.1 (-23%)	3.5 (-47%)
Single-Cell	116.3 (7.6x)	38.7 (5.7x)	27.1 (4.1x)
Dual-Cell	222.4 (14.5x)	73.7 (11x)	47.1 (7.1x)
PS3-Cell	135.6 (8.9x)	45.2 (6.7x)	31.6 (4.8x)
ray casting, shading & shadows			
2.4GHz x86	7.2	3.0	2.5
2.4GHz SPE	7.4 (+3%)	2.6 (-13%)	1.9 (-24%)
Single-Cell	58.1 (8x)	20 (6.6x)	16.2 (6.4x)
Dual-Cell	110.9 (15.4x)	37.3 (12.4x)	30.6 (12.2x)
PS3-Cell	67.8 (9.4x)	23.2 (7.7x)	18.9 (7.5x)

(그림 6) 장면별 프로세스별 성능



(그림 7) SPE 개수에 따른 성능 테스트

광선 추적법의 동향에 대한 설명을 하였다. 다음 장에는 GPU를 이용한 광선 추적법의 동향을 살펴본다.

Ⅲ. GPU를 이용한 가속

보통 그래픽스 카드라고 불리는 그래픽스 처리 장치(GPU)가 마치 가전제품처럼 각 가정의 PC에 설치되기 시작한 지가 약 10년 이상 되었다. 최근에

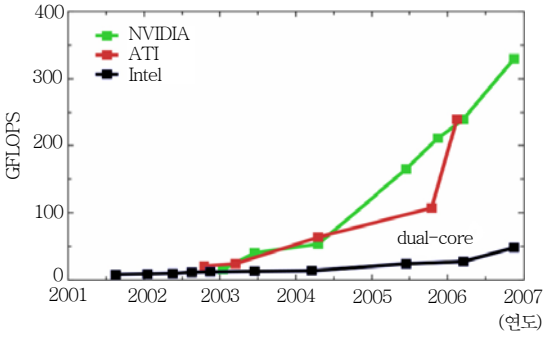
〈표 1〉 CPU와 GPU의 항목별 비교

항목	CPU(Intel Core2 Duo 3.0GHz)	GPU(GeForce 8800 GTX)
계산능력	48 GFLOPS	330 GFLOPS
메모리 대역폭	21GB/s	55.2GB/s
가격	874달러	599달러
매년 성능 향상폭	1.4x	1.7x(픽셀처리기) 2.3x(정점처리기)

<자료>: Kurt Akeley, Stanford GPUbench Project

소개되고 있는 GPU들은 프로그램 가능한 속성(programmability)과 정밀성(precision) 그리고 소비 전력(power)면에서 보다 유연하고 강력한 프로세서로 진화해 나가고 있다.

<표 1>에서 볼 수 있듯이, CPU에 비해 계산능력과 메모리대역폭이 우수하면서, 오히려 가격 측면에서도 경제적이다. 또한, 1년에 두 배씩의 성능향상이 있다는 무어의 법칙(Moore's Law)이 일반적인 칩 디자인 패턴을 이미 능가해서, GPU의 성능향상 추세는 6개월에 두 배씩의 성능향상 패턴을 보이고 있다. (그림 8)은 주요한 프로세서 제작업체별 제품의 성능향상 추이를 보여주고 있다. 여기서 주목할 부분은 앞서 이야기했듯이, GPU의 성능향상 추이가 CPU 성능향상 추이와 비교하여 현저한 차이로 향상되고 있다는 것을 알 수 있다. 이런 성장의 주된 요인으로는 DirectX와 같은 게임 API쪽의 강력한 드라이브가 가장 크며, 고성능 컴퓨팅(HPC) 분야의 성장이다. 이러한 GPU의 성능향상은 앞으로도 계속될 것이라는 것이 일반적 전망이다.



<자료>: SUPERCOMPUTING 2006 Tutorial

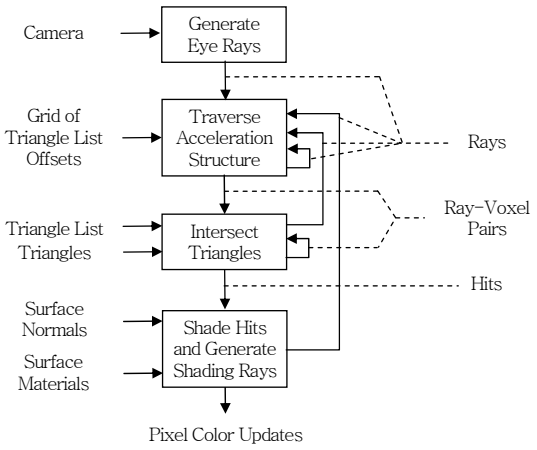
(그림 8) 제조사별 GPU 성능향상 추이

1. 스트리밍 광선 추적기

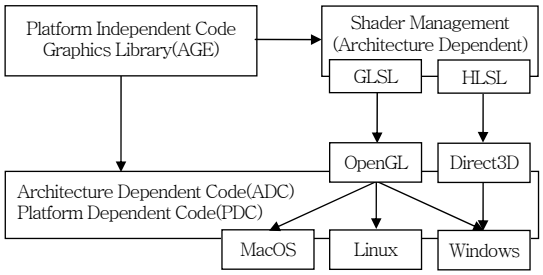
고품질 렌더링을 위해서 주로 많이 이용되는 광선 추적법은 렌더링을 위해서는 많은 계산 시간을 필요로 하기 때문에 최근에 GPU의 고속 계산 성능을 활용한 GPU 기반 렌더러 구현이 이뤄지고 있다.

Purcell 등이 최소의 호스트 상호작용으로 광선 추적 계산을 전부 GPU에서 효율적으로 할 수 있다는 발상으로 제안한 것이 (그림 9)에 보인 스트리밍 광선 추적기(streaming ray tracer)이다[6]. (그림 9)에는 광선 추적 과정을 나타내었다. 광선 추적 과정은 몇 개의 코어 커널인 시선 광선 발생기(eye ray generation), 가속 구조 탐색, 삼각형 교차기 그리고 색을 결정하는 셰이더로 나뉘고, 커널들 간에 전달되는 스트림 데이터(stream recodes)는 점선으로 표시된 것이다.

Christen 등은 Purcell의 방법을 근간으로 GPU를 이용한 광선 추적기를 구현하였다[7]. 기본적으로 광선 추적 알고리듬에 쓰이는 요소를 조그만 여러 개 커널들을 픽셀 셰이더들로 작성하였으며, 각 커널들은 순서화되어 수행되며, 같은 커널들이 여러 차례 호출될 수 있도록 하고 있다. 여러 차례 반사되는 것은 커널 정보를 저장하여 재귀 호출 없이 반복적으로 처리했다. 공간 데이터 구조로는 GPU의 제한된 자원에서 구현하기 쉬워 주로 이용되는 그리드 구조가 사용되었고, 물체 탐색은 3DDA 알고리듬을



(그림 9) 스트리밍 광선 추적 처리 과정도

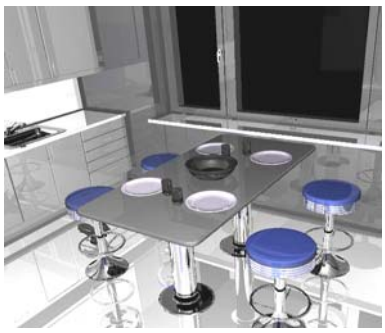


(그림 10) 개발 프로그램 모델

이용했다. 모든 장면 데이터는 실수 값 텍스처 값으로 변환하여 텍스처 메모리를 이용하여 저장된다. 특히, 반사나 굴절을 위해 추가적인 정보 저장을 위해 광선-텍스처를 추가할 수 있으나 구현하지는 않았다. (그림 10)은 개발 프로그램 모델을 보인 것으로, HLSL이나 GLSL 구조를 이용하여 개발 기술의 셰이더 프로그램을 작성하였다.

GeForce 6시리즈에서 HLSL과 GLSL 모두로 구현할 수 있으나, 하드웨어가 GLSL 2.0 이상의 기능을 드라이버에서 지원하지 않아 HLSL에서 정상 동작하는 것만을 보였다.

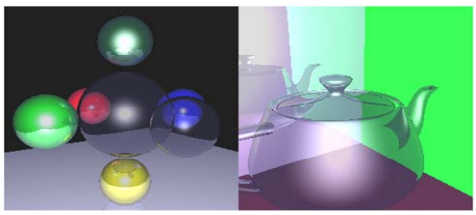
광선 추적 방식은 여러 차례 광선이 표현하려는 공간을 탐색해서 원하는 정보를 계산하여야 하므로 장면의 공간내 물체와 광선과의 빠른 교차 탐색이 필요하다. 이러한 교차 탐색은 초기에 자원의 제약으로 구현이 쉬웠던 grid 탐색에서 빠른 kd-tree 탐색으로 연구가 이뤄지고 있다[8],[9]. Foley는 스택에 저장되는 구조를 "kd-restart", "kd-backtrack"을 이용하여, 스택 구조가 없는 GPU에서도 빠르게 kd-tree 탐색을 수행할 수 있도록 하였다. 그러나 여전히 스택 저장을 위한 메모리 접근이 많아 빠른 계산 능력에도 불구하고, 스택 구조가 있는 CPU 보다 느리다는 단점이 있었다. 그래서, Horn 등은 "short-stack"을 도입하여 처리하는 일부 스택 내용을 스택 구조로 처리함으로써 빠른 kd-tree 탐색 방법을 제시하고 있다. (그림 11)은 1024×1024 크기의 반사가 있는 장면을 1회 바운스(bounce)만 고려하고 퐁 셰이딩(phong shading)으로 렌더링하는데, 8fps 정도 렌더링 속도를 나타냈다. (그림 11)은 렌더링 결과 영상이다.



(그림 11) 반사가 있는 장면

디지털 애니메이션은 고정된 장면이 아니라 장면 내 객체의 움직임을 포함하므로 이와 같은 경우에도 빠르게 렌더링 될 수 있는 방법들이 연구되고 있다. 애니메이션을 고려한 동적인 장면에서는 GPU의 제한적인 자원에 적합한 grid 방식이 선호되고 있다[10]. 그러나, GPU의 발전과 더불어 그 제한이 많이 줄어들고 있어 새로운 방법론들이 연구되고 있다. 그 중에 하나로 GPU에도 멀티 코어 시스템이 개발되고 있는데, 이러한 멀티 코어와 BVH 방법을 이용해 동적인 객체를 빠르게 갱신하면서 탐색할 수 있도록 하는 것도 연구되고 있다[11].

그외 스위스 바젤(Basel) 대학에서 GLSL과 HLSL를 이용하여 GPU를 통해 광선 추적을 수행하는 광선 추적기(ray tracer)를 구현하였다. (그림 12)는 GLSL/HLSL 기반의 GPU 광선 추적기의 수행결과를 보여주고 있다.



(그림 12) GLSL/HLSL 기반 GPU 광선 추적기

2. 젤라토

젤라토(Gelato)는 엔비디아(Nvidia)에서 개발한 GPU를 이용한 고품질 렌더러이다[12]. 연구 수준의 GPU 렌더러들이 간단한 장면이나 간단한 수준

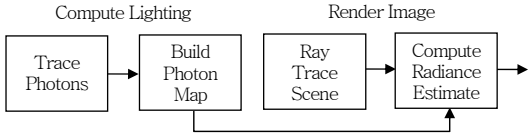
의 셰이딩 정보를 이용하여 GPU를 이용해 빠르게 렌더링하는 방법에 대해 연구되었다면, 젤라토는 GPU를 이용해 기존 상용 소프트웨어(Maya, 3DS MAX)와의 호환성을 유지하면서 실제 영화 제작에 활용할 수 있도록 다양한 셰이더를 지원하여 만들어진 제품이다. 현재 버전 2.1이 출시되었으며, 기본 버전은 무료이고, 프로버전은 구매를 하여야 한다.

젤라토 2.1의 주요 특징으로, 빠른 광선 추적 및 ambient occlusion 기능, 마야 8.5 및 3DS MAX 9 지원, 마야를 위한 망고(Mango) 플러그인과 텍스처 베이크 세트(Maya texture bake sets) 지원, 털이나 모발 표현이 뛰어난 셰이브 앤 헤어컷 4 지원, 대용량 이펙트를 위한 안개, 빛 지원 등이 있다. 특히, 젤라토 프로 2.1은 소베토(Sorbetto)의 재조명(re-lighting)과 다이내믹 섀도 기능, 네트워크 병렬 렌더링, 다중 스레드 렌더링, 그리고 현재는 현재 리눅스만 가능한 64비트 마야 지원 등을 추가로 처리할 수 있다. 그러나, 현재 젤라토에서 지원하지 않는 것은 광선 추적 기능에 의해 자동으로 자연스런 그림자를 생성할 수 있는 면적 광원(area lights)과 자체 셰이딩 언어의 한 규격인 GSL texture 3D는 지원하지 못하고 있다.

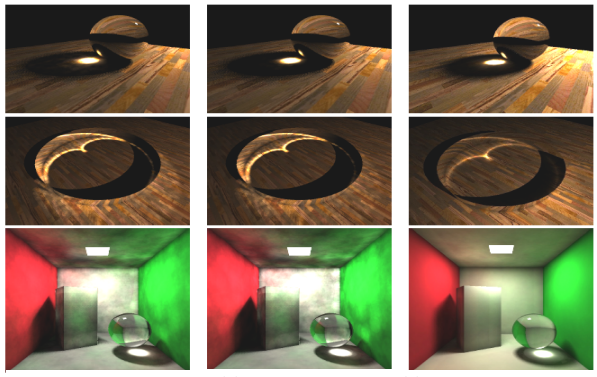
3. 포톤 맵 가속

포톤 맵(photon map)은 Jensen 등이 Computer & Graphics에 1995년도에 처음 발표하였고[13], 2003년 Purcell 등이 Graphics Hardware를 통해 GPU를 이용하여 포톤 맵 전체를 구현하여 실시간 시뮬레이션으로 결과를 보여주었다[14]. 포톤 맵을 이용한 렌더링 시스템은 (그림 13)과 같이 포톤을 조명으로부터 추적하여 포톤 맵을 생성한다. Purcell 등은 두 가지 방법으로 포톤 맵을 생성하는데, 한 가지는 bitonic sorting을 이용해서 grid cell 형태로 텍스처 메모리에 포톤을 저장하는 방법과 stencil 라우팅을 이용하여 grid cell 형태로 저장하는 방식을 사용한 것이 특징이다. 마지막으로 래디언스 추정(radiance estimation)은 k-NN 알고리듬을 사용하였다.

(그림 14)는 Purcell 등이 구현한 결과인데 테스



(그림 13) 포톤 맵을 이용한 렌더링 시스템 흐름도



(a) Bitonic Sort (b) Stencil Routing (c) Software Reference

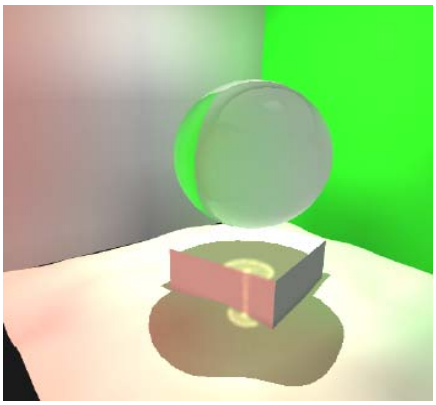
	Bitonic Sort				Stencil Routing			
Scene Name	Trace Photons	Build Map	Trace Rays	Radiance Estimate	Trace Photons	Build Map	Trace Rays	Radiance Estimate
Glass Ball	1.2s	0.8s	0.5s	14.9s	1.2s	1.8s	0.5s	7.8s
Ring	1.3s	0.8s	0.4s	6.5s	1.3s	1.8s	0.4s	4.6s
Cornell Box	2.1s	1.4s	8.4s	52.4s	2.1s	1.7s	8.4s	35.0s

(그림 14) 테스트 장면((a) GPU상에서 Bitonic Sort를 이용한 결과 (b) GPU상에서 Stencil Routing을 이용한 결과 (c) 소프트웨어로 구현한 결과)과 성능 비교표

트 장면과 성능을 나타내었다. 실험 결과에 대한 시스템 환경은 GeForce FX 5900 Ultra, Pentium 4 CPU 3GHz, 2.0GB RAM이다.

그 외 GPU 가속을 통한 포톤 매핑 시뮬레이션 기법을 덴마크 공대 연구진이 2004년에 소개하였고, (그림 15)에서 그 결과를 보여주고 있다.

또한, 래디언스 캐시를 GPU를 이용하여 구현하고 프라이머리 광선 바운스되는 환경에서 전역 조명을 구현하여 SIGGRAPH 2005에서 소개하였다.



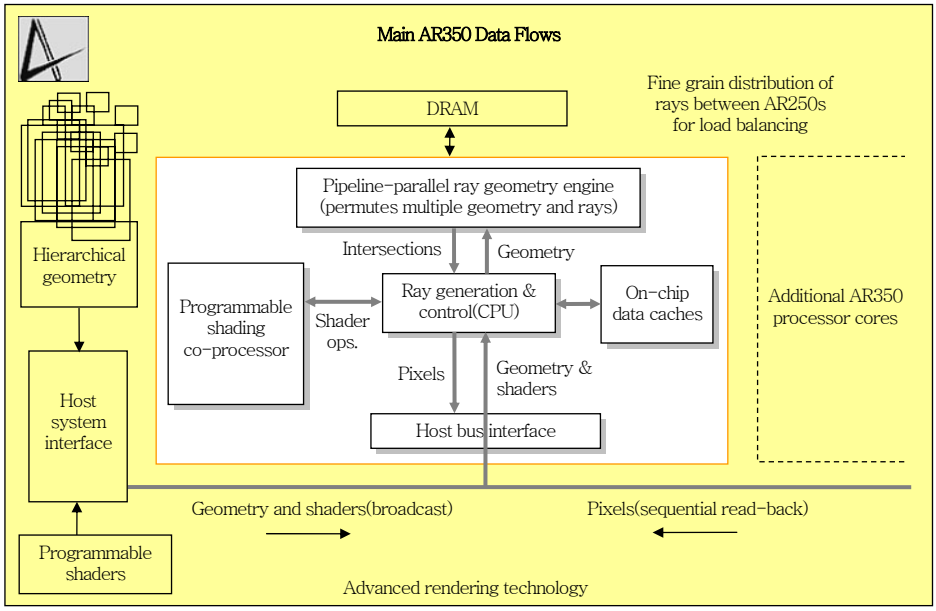
(그림 15) GPU 포톤 맵을 통한 Caustics

Ⅳ. 전용 하드웨어 가속 기술

1. AR350 프로세서

ART에서는 AR350 프로세서를 개발하였고 8~16개의 AR 350 프로세서를 사용하여 PURE라는 PCI 인터페이스의 보드와 48개의 AR350 프로세서를 사용한 RenderDrive를 개발하였다[15],[16]. 현재는 각각 RayBox와 RenderServer라는 이름으로 상품으로 판매하고 있으며 AR350 프로세서도 향상되어 다수의 AR500 프로세서가 장착되어 있다. 본 고에서는 2001년 Graphics Hardware라는 워크샵에서 발표된 AR350에 대한 구조에 대하여 설명하고[17] 2004년에 GRAPHITE에서 발표된 논문[15]을 바탕으로 RenderDriver에 대한 성능도 살펴보도록 한다.

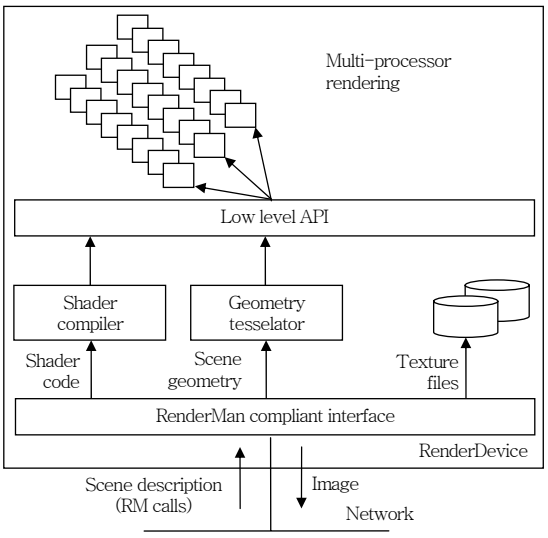
(그림 16)은 AR350 프로세서의 데이터 흐름도 및 구조도이다. AR350은 다섯 개의 부분으로 나누어져 있다. 첫째로 광선과 지오메트리간의 충돌을 병렬로 처리할 수 있는 충돌 처리 엔진(pipeline-parallel ray geometry engine)을 가지고 있고, 둘



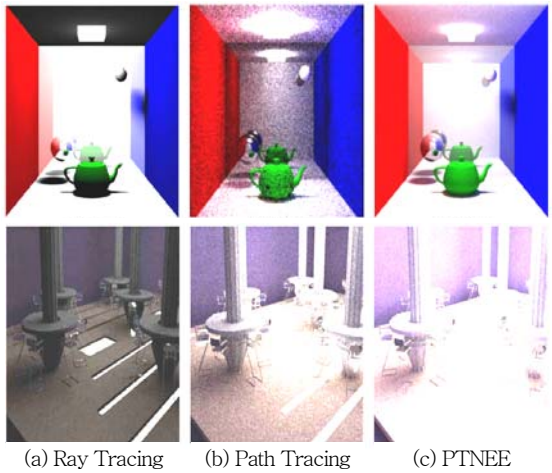
(그림 16) AR350의 데이터 흐름도 및 구조도

째로 광선을 생성할 수 있는 광선 생성기 및 신호 제어기(ray generation & control)가 있으며, 셋째로 데이터를 캐싱할 수 있는 on-chip 캐시를 가지고 있으며, 넷째로 컴파일된 셰이더를 처리할 수 있는 셰이더 보조프로세서가 있고, 마지막으로 버스 인터페이스로 구성되어 있다. AR350 프로세서의 데이터 흐름은 다음과 같다. 호스트(host)로부터 지오메트리 데이터와 컴파일된 셰이더가 AR350으로 전달되고 AR350에서는 광선을 생성하고 생성된 광선과 지오메트리를 충돌 처리 엔진으로 전달하고 충돌 처리 엔진에서 충돌이 일어나면 셰이딩 보조 프로세서에 셰이딩 처리를 요구하여 셰이딩된 픽셀을 호스트로 전달한다.

다음은 48개의 AR350 프로세서를 사용한 RenderDrive에 대하여 알아보도록 한다. (그림 17)은 RenderDrive를 상에서의 데이터 흐름을 나타낸 그림이다. 렌더맨 포맷의 셰이더를 RenderDrive에 네트워크를 통해 보내면 셰이더는 RenderDrive에서 컴파일되어 텍스처와 함께 하드디스크에 저장된다. 장면 지오메트리는 GT에서 계층적으로(hierarchical) 분화되고 삼각형으로 쪼개어진다. 그러면 렌더링이 시작되는데, 셰이더와 지오메트리가 각 프로세서에 전달되고 프로세서마다 광선의 분포도 균등하게 할당된다. 각 프로세서는 할당된 광선을 추적하게 되고, 각각 프로세서에서 하나의 픽셀의 색깔 값을 출력하게 된다. 프로세서로부터 출력된 픽셀을 모아서 결과 영상이 되고 이를 네트워크를 통하여 호스트로 되돌려 준다. RenderDrive의 특징은 렌더맨 셰이더 포맷을 지원하는 것과 패스 추적(PT)을 하는 것, 그리고 패스 추적 시 다음 단계를 예측(PTNEE)하는 것이다. 광선이 생성되고 충돌하는 것을 무한 반복해도 광원과 만나지 않으면 의미가 없어지기 때문에 이러한 계산을 최대한 줄이기 위하여 PTNEE 방법을 사용하여 광선의 발산을 줄이는 것이다. (그림 18)은 "cornell box"와 "coffee shop"이라는 장면 데이터에 대하여 각각 방식을 사용한 성능을 나타낸 것이다. RenderDrive에서 PTNEE를 사용하여 "coffee shop"을 렌더링하는 데 걸린 시간은 42분 31초이고, "cornell box" 같은 경우는 5분 47초였다. 현재 ART에서 판매하는 상용 제품으로 ray box나 RenderServer는 이보다 성능이 훨씬 좋을 것으로 판단된다.



(그림 17) RenderDrive의 데이터 흐름도

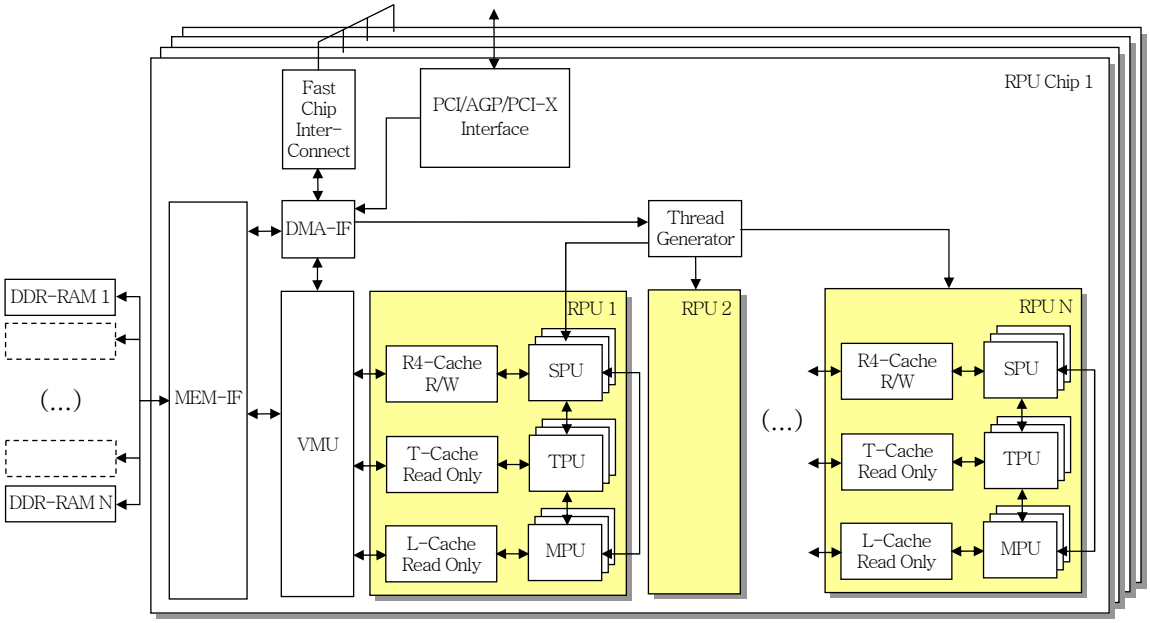


(a) Ray Tracing (b) Path Tracing (c) PTNEE

(그림 18) "Cornell Box"와 "Coffee Shop"을 RenderDrive를 이용하여 렌더링한 결과

2. RPU

Saarland 대학의 Woop 등은 실시간 광선 추적을 위한 광선 추적 프로세서를 제안하고 SIGGRAPH



(그림 19) RPU의 구조도 및 데이터 흐름도

05에 발표하였다[18]. RPU의 구조는 (그림 19)와 같이 셰이딩을 처리하기 위한 다수의 SPU와 트리 탐색을 위한 다수의 TPU 그리고 MPU로 구성되어 있고, 스레드 생성 및 관리를 위하여 스레드 생성기(thread generator)가 있고 PCI/PCI-X 인터페이스로 호스트와 통신하며 메모리 접근을 위한 DMA 컨트롤러를 가지고 있다. 또한 캐싱을 위한 VMU로 구성되어 있다. RPU의 구조는 기존의 Saarland 대학에서 개발한 SaarCOR라는 광선 처리 프로세서 구조를 향상시킨 것이며 OpenRT와 연동되게 설계되어 있다. 현재는 한 개의 RPU 프로토타입을 FPGA로 구현하여 성능은 OpenRT과 비교하여 0.5~1.6배 정도로서 빠르지 않지만 이러한 구조를 ASIC으로 구현하여 여러 개의 코어로 구성한다면 훨씬 뛰어난 성능을 보일 것으로 기대한다.

V. 결론

컴퓨터 그래픽스를 이용한 콘텐츠 제작자들의 뛰어난 품질에 대한 갈구와 제작시간 단축 및 비용 절감이라는 두 마리 토끼를 다 잡기 위한 노력이 계속되고 있는 가운데 CPU를 기반으로 하는 상용 소프트웨어, GPU를 기반으로 하는 렌더링 시스템, 전용 하드웨어를 기반으로 하는 시스템 등이 존재하고 있다. CPU, GPU 등은 나날이 발전해가고 있고, 새 버전의 출시 주기가 점점 더 빨라지고 있는 상황에서 렌더링에 최적화된 전용하드웨어까지 가세하여 앞으로의 렌더링에 대한 향방을 점치기 어렵게 하고 있다. 분명한 것은 현재 소프트웨어만의 렌더링으로는 제작 시간을 단축하기 힘들다는 판단에 따라 GPU를 이용하거나 전용하드웨어를 설계하기에 이르렀으며 앞으로 이 분야도 하나의 큰 연구 흐름이 될 것이라 판단된다. 물론 인텔이 CPU 코어를 늘려가고 있고 멀티코어 멀티스레딩 프로그램이 유행할 때까지는 GPU 및 전용 하드웨어에 대한 연구는 계속될 것으로 보인다.

지금까지 CPU 기반의 소프트웨어, GPU 및 전용 하드웨어를 이용한 렌더링 가속 기술에 대하여 살펴보았다. 향후 GPU 및 전용 하드웨어를 이용한 가속 기술은 렌더링 알고리듬 가속뿐만 아니라 다른 여러 분야에 사용될 것으로 판단되며, 멀티코어 CPU 기반 가속 분야도 또 다른 연구의 줄기가 될 것이다.

약 어 정 리

3DDA	3D Differential Analyzer
ART	Advanced Rendering Technologies
BRDF	Bidirectional Reflectance Distribution Function
BVH	Bounding Volume Hierarchy
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DMA	Direct Memory Access
GLSL	OpenGL Shading Language
GPGPU	General-Purpose computation on GPUs
GPU	Graphics Processing Unit
GT	Geometry Tesselator
HLSL	High-Level Shading Language
HPC	High-Performance Computing
MPI	Message-Passing Interface
MPU	Mailboxed list Processing Unit
NN	Nearest Neighbor
PPE	PowerPC Processor Element
PT	Path Tracing
PTNEE	Path Tracing with Next Event Estimation
PVM	Parallel Virtual Machine
RPU	Ray Processing Unit
SIMD	Single Instruction Multiple Data
SPE	Synergistic Processing Elements
SPU	Shader Processing Unit
STAR	State of art
VMU	Virtual Memory Unit

참 고 문 헌

[1] Robert L. Cook, Loren Carpenter, and Edwin Catmull, "The Reyes Image Rendering Architecture," *ACM SIGGRAPH Computer Graphics*, Vol.21, No.4, July 1987.

[2] Ingo Wald, Timothy J. Purcell, J. Schmittler, Carsten-Benthin, and Philipp Slusallek, "Realtime Ray Tracing and Its Use for Interactive Global Illumination," *Eurographics 2003*, 2003.

[3] Steven Parker, Michael Parker, Yaren Livnat, Charles Hansen, and Peter Pike Sloan, "Interactive Ray Tracing," *In Proc. of Interactive 3D Graphics(I3D)*, Apr. 1999, pp.119-126.

[4] Carsten Benthin, Ingo Wald, Michael Scherbaum, and Heiko Friedrich, "Ray Tracing on the Cell Processor," *IEEE Symp. on Ray Tracing 2006*, 2006, pp.15-23.

[5] http://www.blachford.info/computer/Cell/Cell1_v2.html

[6] Timothy J. Purcell, "Ray Tracing on a Stream Processor," Thesis for the degree of doctor of philosophy in Stanford University, 2004.

[7] Martin Christen, "Ray Tracing on GPU," University of Applied Sciences Basel, thesis 2005.

[8] T. Foley and J. Sugerman, "Kd-tree Acceleration Structures for a GPU Raytracer," *In HWWS '05: Proc. of the ACM SIGGRAPH/EUROGRAPHICS Conf. on Graphics Hardware*, ACM Press, New York, NY, USA, 2005, pp.15-22.

[9] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan, "Interactive k-D Tree GPU Ray-tracing," *Proc. of the 2007 Symp. on Interactive 3D Graphics and Games*, 2007, pp.167-174.

[10] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G Parker, "Ray Tracing Animated Scenes Using Coherent Grid Traversal," *ACM Transactions on Graphics 2006*, 2006, pp.485-493.

[11] Thiago Ize, Ingo Wald, and Steven G Parker, "Asynchronous BVH Construction for Ray Tracing Dynamic Scenes on Parallel Multi-Core Architectures," *Proc. of the 2007 Eurographics Symp. on Parallel Graphics and Visualization*, 2007.

[12] http://www.nvidia.com/page/gz_home.html

[13] Henrik Wann Jensen and Niels Jorgen Christensen, "Photon Maps in Bidirectional Monte Carlo Ray Tracing of Complex Objects," Computers & Graphics, Vol.19, No.2, Mar. 1995, pp.215-224.

[14] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan, "Photon Mapping on Programmable Graphics Hardware," *Proc. of the ACM SIGGRAPH/EUROGRPHICS Conf. on Graphics Hardware*, 2003, pp.41-50.

[15] Christophe Cassagnabère, Francois Rousselle, and-Christophe Renaud, "Path Tracing Using the AR350 Processor," *Proc. of GRAPHITE'04*, 2004, pp.23-29.

[16] http://www.artvps.com

[17] http://www.graphicshardware.org/previous/www_2001/presentations/Hot3D_Daniel_Hall.pdf

[18] Sven Woop, Jörg Schmittler, and Philipp Slusallek, "RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing," *SIGGRAPH 2005*, 2005.