GPU용 Kd-트리 탐색 방법의 성능 분석 및 향상 기법

(Performance Analysis and Enhancing Techniques of Kd-Tree Traversal Methods on GPU)

장 병 준 † 임 인 성 ††
(Byungjoon Chang) (Insung Ihm)

요 약 광선-다각형 교차 계산은 광선 추적법 계산의 상당 부분을 차지하는 중요한 구성요소로서, 보편적으로 정적인 장면에 대해서는 kd-트리와 같은 공간 자료구조를 사용하여 교차 계산을 가속하여왔다. 최근 CPU에 비해 상대적으로 제한된 계산구조를 가지는 GPU에 적합하도록 변형된 kd-트리 탐색 기법이 몇 가지 제시되어 왔는데, 본 논문에서는 이러한 기존 방법을 보완할 수 있는 두 가지 구현 기법을 제안한다. 첫째, 트리 탐색을 위한 스택을 전역 메모리에 할당할 경우 전역 메모리 접근으로 인한 비용을 줄이고자 하는 캐쉬 적용 스택 방법과 둘째, 기존의 로프 방법의 문제점인 상당한 메모리 요구량을 줄이고자 하는 적은 깊이의 스택(short stack)을 사용한 로프 방법을 제시한다. 제안된 방법의 효용성을 보이기 위하여 기존의 GPU용 탐색 방법과의 성능 비교 분석을 수행한다. 이러한 실험 결과는 향후 GPU용 광선 추적법 소프트웨어 개발자들이 상황에 맞는 적절한 kd-트리 탐색 방법을 선택할 수 있도록 해주는 중요한 정보를 제공하게 될 것이다.

키워드 : 실시간 광선 추적법, kd-트리 탐색, 스택 구조, 광선-다각형 교차, GPU 구현

Abstract Ray-object intersection is an important element in ray tracing that takes up a substantial amount of computing time. In general, such spatial data structure as kd-tree has been frequently used for static scenes to accelerate the intersection computation. Recently, a few variants of kd-tree traversal have been proposed suitable for the GPU that has a relatively restricted computing architecture compared to the CPU. In this article, we propose yet another two implementation techniques that can improve those previous ones. First, we present a cached stack method that is aimed to reduce the costly global memory access time needed when the stack is allocated to global memory. Secondly, we present a rope-with-short-stack method that eases the substantial memory requirement, often necessary for the previous rope method. In order to show the effectiveness of our techniques, we compare their performances with those of the previous GPU traversal methods. The experimental results will provide prospective GPU ray tracer developers with valuable information, helping them choose a proper kd-tree traversal method.

Key words : real-time ray tracing, kd-tree traversal, stack structure, ray-polygon intersection, GPU implementation

· 이 논문은 2009년도 정부(교육기술과학부)의 재원으로 한국과학재단의 지원을 받아 수행된 연구임(과제번호: R01-2007-000-21057-0(2009))

† 학생회원 : 서강대학교 컴퓨터공학과
jerrun@sogang.ac.kr
†† 종신회원 : 서강대학교 컴퓨터공학과 교수
ihm@sogang.ac.kr

논문접수 : 2009년 11월 9일
심사완료 : 2009년 12월 28일

정보과학회논문지: 컴퓨팅의 실제 및 레터 제16권 제2호(2010.2)

1. 서 론

고급 렌더링 분야에서 전역 조명 효과 생성을 위한 대표적인 방법으로 광선 추적(ray tracing)법을 들 수 있다. 이 방법은 그림자를 생성할 뿐만 아니라 정반사 효과를 매우 사실적으로 표현하는 반면 많은 양의 연산을 필요로 하는데, 최근의 CPU와 GPU의 성능 향상으로 인하여 실시간 응용으로의 적용이 가능하게 되었다 [1-13]. 특히 수백 개의 스트리밍 코어 프로세서(streaming core processor)에 기반을 둔 대용량 SIMD (Single Instruction, Multiple Data) 연산에서 뛰어난

성능을 보여주는 GPU상에서, 광선 추적법을 실시간 응용으로 구현하기 위하여 요구되는 중요한 구성 요소 중의 하나인 광선-다각형 교차 연산의 효과적인 가속을 위하여 GPU에 최적화된 공간 자료구조 및 탐색 방법의 구현에 관하여 여러 연구가 진행되어 왔다[1-4].

CUDA(Compute Unified Device Architecture) 관점에서 GPU의 대표적인 구조적 특징(GT 200)으로 하드웨어적으로 가속되는 일부 연산들과 접근 속도가 빠르고 읽기/쓰기가 가능한 레지스터(register)와 공유 메모리(shared memory), 그리고 읽기만 가능하나 캐싱 기능이 지원되는 상수 메모리(constant memory)와 텍스처 메모리(texture memory), 마지막으로 읽기/쓰기가 가능하지만 접근 속도가 느린 전역, 지역 메모리(global, local memory)로 이루어진 계층적 메모리 구조를 들 수 있다[14]. 이러한 구조에서 빈번한 전역 메모리의 사용은 전체적인 성능의 저하를 초래하게 되므로 가급적 크기는 작지만 접근 속도가 빠른 레지스터와 공유 메모리를 적절히 사용하여 성능을 향상시키는 것이 중요하다.

Kd-트리와 같은 트리 구조를 탐색하기 위해서는 스택을 사용하게 되는데, 일반적인 장면에 대해 생성되는 kd-트리의 최대 깊이를 고려하여 충분한 깊이의 스택을 지역 메모리에 할당할 수 있지만[3], 종종 지역 메모리의 느린 접근 속도로 인하여 성능의 저하가 발생한다. 반면에 접근 속도가 빠른 레지스터와 공유 메모리의 경우 크기 제한으로 인해 적은 깊이의 스택 만이 할당 가능하게 된다[1,2]. 이러한 제약을 극복하고자 kd-트리에 대한 다양한 탐색 방법이 제시되었으며, 각 방법은 GPU 구조의 변화 및 장면의 특성에 따라 적지 않은 성능 차이를 보인다.

본 논문에서는 GPU 상에서 효율적으로 kd-트리를 탐색하기 위한 새로운 변형 기법을 제안하고, 구현과정에 있어서 GPU의 환경에서 성능을 향상시킬 수 있는 최적화 방법을 제시한 후, 기존의 kd-트리 탐색 기법들과 본 논문에서 제안하는 방법들의 성능을 다양한 크기의 장면에 대해 비교 분석하여, 향후 GPU 상에서의 kd-트리 탐색 기법 구현 시 프로그래머가 적절한 선택을 할 수 있는 지표를 제시한다.

2. 이전 연구

실시간 응용을 위한 GPU용 광선 추적법 소프트웨어의구현 시 가장 신경을 써야 하는 부분 중의 하나가 바로 광선-다각형 교차 계산을 위한 공간 자료구조 탐색 부분으로서, GPU상에서는 재귀적(recursive)인 방법을 사용할 수 없거나 비효율적이기 때문에 반복적(iterative)인 탐색을 위한 효과적인 스택 자료구조의 구현이 요구된다.

초기 GPU 상에서의 kd-트리 탐색은 스택 영역 할당에 있어서 메모리 제약으로 인해 스택을 사용하지 않는 방법들이 제안되었다. 장면을 구성하는 다각형들과 광선과의 가장 가까운 교차 지점을 계산하여야 하는 광선 추적법의 특성을 이용하여 광선이 탐색해온 진행거리를 저장하면서 다음 노드로의 탐색이 필요할 경우 루트 노드로부터 다시 탐색하되 저장된 진행거리 이전의 노드에 대해서는 뛰어넘는 재출발(restart, 이하 restart) 방법을 Foley 등이 제안하였다[1].

하지만 이 방법은 최악의 경우 임의의 노드에 대해서 하위 노드의 총 리프 노드 개수만큼 중복 탐색이 발생하게 된다. 이러한 단점을 개선하기 위해 각 노드에 대해 부모 노드로의 링크를 설정하여 중복 탐색을 최대 2번 이하로 발생하도록 개선한 역추적(backtrack, 이하 backtrack) 방법을 제안하였지만[1], 각 노드에 대한 메모리 부담이 발생하고 추가적인 광선과 노드의 AABB (Axis-Aligned Bounding Box)와의 교차 계산이 추가적으로 발생하게 된다. Restart 방법에서 중복된 노드 탐색을 줄이기 위해 루트 노드가 아닌 하위 노드에서도 재탐색을 수행할 수 있는 특수한 경우에 대해서 재탐색을 하는 노드를 변경시키는 푸쉬다운(pushdown, 이하 pushdown) 방법을 Horn 등이 제안하였지만[2], 이 방법은 성능의 향상이 미미하며, 추가적인 비교 연산으로 인해 성능의 저하가 발생하는 경우도 발생한다.

위와 같은 방법들과는 다르게 모든 리프 노드에 대해 각 면을 완전히 포함하는 인접한 노드로의 링크를 설정함으로써 다음으로 탐색하여야 할 노드를 스택 연산 대신에 광선과 리프 노드의 AABB와의 교차 검사를 통해 찾아낼 수 있는 로프(rope, 이하 rope) 방법을 Popov 등이 제안하였다[4]. 장면의 복잡도가 높을 경우 인접한 다음 노드를 찾아내기 위해 수많은 스택 연산을 필요로 하지만 rope 방법의 경우 이러한 연산을 한번의 교차 검사로 대체하기 때문에 상당히 빠른 성능을 보여준다. 하지만 kd-트리 생성시 노드 링크 설정에 관련된 전처리를 필요로 하며 각 리프 노드에 인접 링크 및 AABB 정보를 저장하여야 하므로 장면이 커질수록 메모리 부담이 증가하게 된다.

Restart 및 pushdown 방법의 단점인 중복 노드 탐색의 문제점을 해결하기 위해 GPU의 공유 메모리를 제한된 크기의 스택으로 사용하여, 트리의 하위 노드들에 대해서는 스택 방식의 탐색을 수행하고 만약 스택이 비어있을 경우 재탐색을 수행하는 방법을 Horn 등이 제안하였다[2]. 공유 메모리 스택의 크기를 크게 설정할수록 성능의 향상이 발생하지만 실질적으로 사용 가능한 스택의 크기가 작다는 문제가 있다.

마지막으로 Zhou 등은 GPU의 지역 메모리 상에 충

분한 크기의 스택을 설정하여 전통적인 kd-트리 탐색 방법을 구현하였지만[3], 스택을 접근 속도가 느린 메모리상에 할당함으로써 상대적으로 비효율적인 성능을 보여준다.

3. 제안하는 탐색 방법

3.1 캐쉬 적용 스택 방법(Cached Stack)

Kd-트리 탐색 시에는 다음 방문할 노드를 기록하기 위한 스택을 필요로 한다. 하지만 충분한 크기의 스택을 GPU의 지역 메모리에 설정한 kd-트리 탐색 방법[3]의 경우 스택 연산이 발생할 때마다 상대적으로 읽기/쓰기 속도가 느린 지역 메모리에 대한 접근이 발생하게 되므로 적지 않은 성능의 저하가 발생하게 된다. 이러한 스택의 연산은 표 1에서 알 수 있듯이 kd-트리 탐색이 일어나는 동안 대부분의 스택 연산은 스택의 상위 부분에서 빈번하게 발생함을 알 수 있다(푸쉬의 경우 비율이 상대적으로 낮은 이유는 초기에 수십 레벨 아래의 리프 노드까지 내려가기 위해 계속해서 푸쉬를 하기 때문이고, 첫 리프 노드까지 내려간 후에는 스택의 상위 부분에 대해서만 푸쉬가 일어남). 이러한 특성을 통해 GPU의 지역 메모리 스택 방법에서 읽기/쓰기 속도가 빠른 GPU의 공유 메모리 스택을 지역 메모리 스택의 상위 부분에 설정함으로써, 지역 메모리 접근을 줄일 수 있다.

캐쉬 적용 스택(cached stack, 이하 cached stack) 방법의 스택 구조는 그림 1에서와 같이 공유 메모리와 지역 메모리 스택으로 나뉘게 되며 공유 메모리는 순환 스택 구조를 띄게 된다. 스택에 푸쉬 연산이 발생할 경우 우선적으로 공유 메모리 스택에 원소가 푸쉬되며 만약 공유 메모리 스택이 가득 찬 경우 현재 공유 메모리에 저장되어 있는 원소 중 제일 처음에 푸쉬되었던 원소를 지역 메모리 스택에 푸쉬한 후 새롭게 추가되는 원소를 공유 메모리 스택에 푸쉬하게 된다. 스택의 팝 연산이 발생할 경우는 우선적으로 공유 메모리 스택에서 수행하며 만약 공유 메모리 스택이 비어있을 경우 지역 메모리 스택에서 팝 연산을 수행하게 된다.

이러한 스택의 구조는 스택의 상위 부분에서 빈번하게 발생하는 연산을 공유 메모리로 대체함으로써 효율성을 높일 수 있지만 지역 메모리 스택 방법에 비해 스택 푸쉬 연산 시 공유 메모리 스택이 가득 찼는지에 대한 비교 연산과 공유 메모리 스택이 가득 찼을 경우 공유 메모리로부터의 읽기, 팝 연산 시 공유 메모리 스택이 비어있는지에 대한 비교 연산이 추가 되게 된다. 하지만 이러한 푸쉬 연산 비용은 5절에서 확인할 수 있듯이 깊이 2 이상의 공유 메모리 스택을 설정할 경우 푸쉬 연산 비용이 상쇄됨을 알 수 있다.

3.2 적은 깊이의 스택을 사용한 로프 방법(Rope with Short Stack)

미리 저장한 링크 정보를 사용하여 인접한 노드로 바로 이동 가능한 rope 방법[4]의 단점은 장면을 구성하는 다각형의 개수가 많아질수록 적지 않은 메모리를 사용한다는 점이다. 본 논문에서 제안하는 적은 깊이의 스택을 사용한 로프(rope with short stack) 방법은 우선

표 1 Sponza 장면에 대한 cached stack 방법에서 설정된 스택 깊이와 전체 스택 연산 중 캐쉬 적용 스택(cached stack) 상에서 발생하는 스택 연산의 비율. Sponza 장면에 대해서는 캐쉬 적용 스택의 깊이가 6으로 설정될 경우 대부분의 스택 연산이 공유 메모리 상에서 발생함을 알 수 있다.

	Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6
푸쉬	17.06%	24.03%	27.63%	29.56%	30.66%	31.23%
팝	68.94%	90.19%	97.62%	99.25%	99.70%	100%

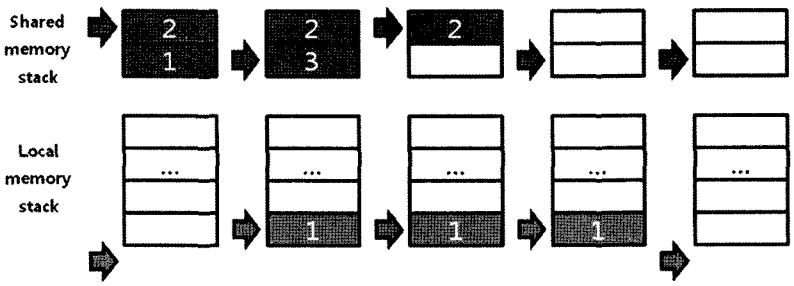

그림 1 캐쉬 적용 스택 방법에서의 스택 구조와 푸쉬, 팝 연산 작동 예. 공유 메모리 스택이 가득 차있는 경우 푸쉬 연산이 발생할 시 공유 메모리 스택 상에 시간상으로 가장 먼저 저장된 원소의 위치에 푸쉬되는 원소를 덮어쓰는 동시에 제거되는 원소를 지역 메모리 스택에 푸쉬하여 준다. 팝 연산은 우선적으로 공유 메모리 스택에서 발생하게 되며 공유 메모리 스택이 비어있는 경우 지역 메모리 스택에서 팝하여 준다.

CPU에서 전처리를 통해 그림 2와 같이 kd-트리의 하단의 노드들에 대해 주어진 깊이(그림 2는 깊이 2인 예를 도시)까지는 로프를 구성하지 않고, 그 상위 노드들에 대해서만 로프 정보를 생성하도록 트리를 구성한 후, GPU를 통한 kd-트리 탐색 시, 로프가 없는 하위 노드에 대해서는 적은 깊이의 스택(short stack)을 사용하여 탐색하고, 나머지 노드들은 로프 정보를 사용하여 탐색함으로써 링크 및 AABB 정보 저장을 위한 메모리 사용량을 줄일 수 있다.

하지만 바로 다음 살펴봐야 할 노드를 찾아내기 위해 스택을 사용하는 연산이 추가되게 되므로 성능의 저하가 발생하지만 이러한 단점을 접근 속도가 빠른 공유 메모리에 스택을 설정함으로써 성능의 저하를 최소화 하였다. 트리의 탐색 과정은 로프가 설정된 노드에 도달할 경우 스택을 사용한 탐색을 수행하게 되며 스택이 비어 있고 탐색이 완료되지 않을 경우 로프 노드의 AABB와 광선과 교차검사를 통해 인접한 노드를 찾아내어 해당 노드부터 탐색을 반복하게 된다.

4. 메모리 접근에 대한 최적화

4.1 Kd-트리 구조에서의 최적화

Kd-트리에서 사용하는 스택의 원소는 일반적으로 스택 연산이 발생하는 임의의 노드에서의 1. 광선의 최소 범위 2. 광선의 최대 범위, 3. 노드 주소를 저장하게 되며 이는 총 12 바이트의 메모리를 요구한다. 하지만 스택으로부터 팝 연산이 일어나게 되는 시점의 광선의 최대 범위와 스택으로부터 팝 연산으로 가져오게 되는 원소에 저장되어 있는 광선의 최소 범위 값은 동일하기 때문에 8 바이트를 사용하여 스택의 원소를 설정하는 것이 가능하다. 스택의 원소 크기를 줄인 결과 지역 메모리 스택의 경우 메모리 접근량의 감소, 그리고 공유 메모리 스택의 경우 작아진 스택 원소 크기로 인해 더 큰 깊이의 스택 할당으로 상당한 성능의 향상이 이루어졌다.

4.2 GPU 환경에서의 최적화

GPU의 중요한 특징중의 하나로 고성능의 SIMD 기반의병렬 처리 연산과 그에 특화된 메모리 접근 구조를 들 수 있는데, 이러한 병렬 처리 연산은 32개의 쓰레드가 하나의 와프(warp)라 불리는 단위로 처리가 이루어진다. 이는 다시 16 개의 반와프(half warp) 단위로 메모리 접근이 발생하게 된다. NVIDIA사의 GeForce GTX 280의 최신 GPU 환경을 기준으로 우선 지역 메모리의 경우를 살펴보면 전역 메모리와 동일한 메모리 전송 패턴을 따르며 반와프가 64바이트의 단위 구역내의 전역 메모리를 접근하려 할 때 최적의 성능을 발휘하게 된다.

각 쓰레드마다 설정되는 지역 메모리는 그림 3과 같이 이러한 최적의 전송 패턴을 따르기 위해 각각 하나의 쓰레드 단위가 아닌 16개의 쓰레드 단위로 순차적인 전역 메모리 주소 영역이 할당되게 된다. 이것은 결국 반와프가 동일한 스택의 위치를 참조할 때 가장 효율적인 메모리 전송 패턴이 발생하게 되며 이는 반와프가 동일한 트리 노드 탐색을 하게 될 경우 발생하게 된다. 이러한 메모리 전송 패턴의 최적화는 쓰레드 블록의 형태를 달리하여 조절이 가능한데 하나의 커다란 쓰레드 블록의 작은 단위인 반와프가 정사각형 형태를 띄게 될 때 각 쓰레드가 kd-트리 내에서 동일한 노드 탐색이 발생할 확률이 극대화되게 된다. 그러므로 쓰레드 블록의 크기는 4 × 4의 배수 형태로 설정하여 이러한 조건을 만족시킬 수 있다.

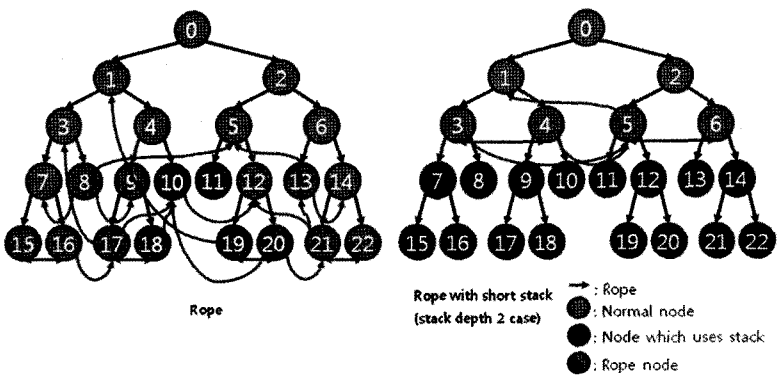

그림 2 Rope 방법과 깊이 2의 공유 메모리 스택을 설정한 rope with short stack 방법의 노드 구조. Rope 방법의 경우 모든 리프 노드에 대해 연결 링크가 설정되며 트리의 리프 노드의 수가 증가할수록 그에 따른 메모리 사용 부담량이 늘어난다. Rope with short stack 방법의 경우 약간의 스택 만을 사용하여도 설정하여야 할 포인터의 수가 상당히 줄어들었음을 알 수 있다.

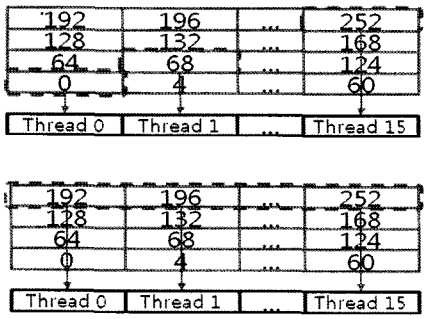

그림 3 16개의 쓰레드에 설정된 지역 메모리 스택의 주소 영역과 두 가지 전송 패턴. 첫 번째 그림은 각 쓰레드가 서로 다른 스택 영역을 접근하려 하는 경우로 256 바이트의 전역 메모리를 필요로하므로 총 4 번의 64 바이트의 메모리 전송이 발생하게 된다. 두 번째 그림은 동일한 지역 메모리 스택 위치를 접근하는 경우로 한 번의 64 바이트 메모리 전송이 발생한다.

다음으로 공유 메모리의 경우 각 64바이트의 메모리 단위 구간으로 이루어지게 되는데 각 구간은 4바이트 간격의 16개 뱅크로 나뉘어지게 되며 16개의 각 쓰레드가 각기 다른 뱅크 영역의 공유 메모리에 접근하게 될 때 최적의 성능을 발휘하게 된다. 그러므로 공유 메모리 스택의 경우 동일한 뱅크 영역의 공유 메모리 접근을 방지하기 위해 지역 메모리 스택과 유사한 구조를 필요로 한다. 하지만 공유 메모리의 특성상 하나의 쓰레드 블록이 공유하게 되므로 그림 4와 같이 16개의 쓰레드가 사용하는 공유 메모리 스택을 하나의 단위로써 공유 메모리 상에 순차적으로 배열되는 형태로 구성을 하도록 하였다.

5. 성능 비교 결과 및 분석

본 논문에서 제안한 방법 이외에 기존의 다양한 탐색 방법을 구현하여 비교함으로써 공유 메모리를 사용한 스택의 성능과 장면의 특성에 따른 각 탐색 방법의 장단점을 확인할 수 있었다. 본 실험은 NVIDIA사의 GeForce GTX 280 GPU에서 수행하였다. 그림 5는 실험에 쓰인 각 장면들이며 표 2는 이러한 장면들에 대해서 SAH(Surface Area Heuristic) 방법을 적용하여 CPU를 통해 생성된 kd-트리의 특성들을 보여주고 있

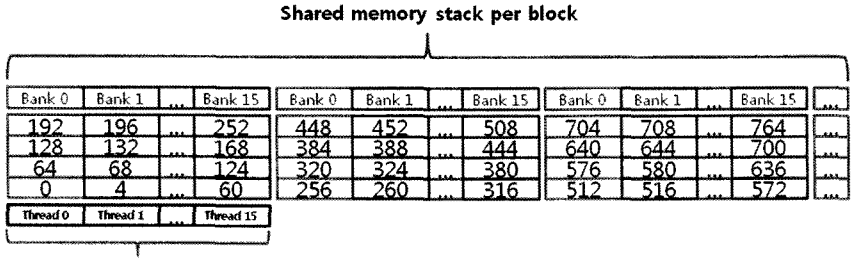

그림 4 Half warp 단위의 공유 메모리 스택의 모습과 하나의 쓰레드 블록의 스택 구조

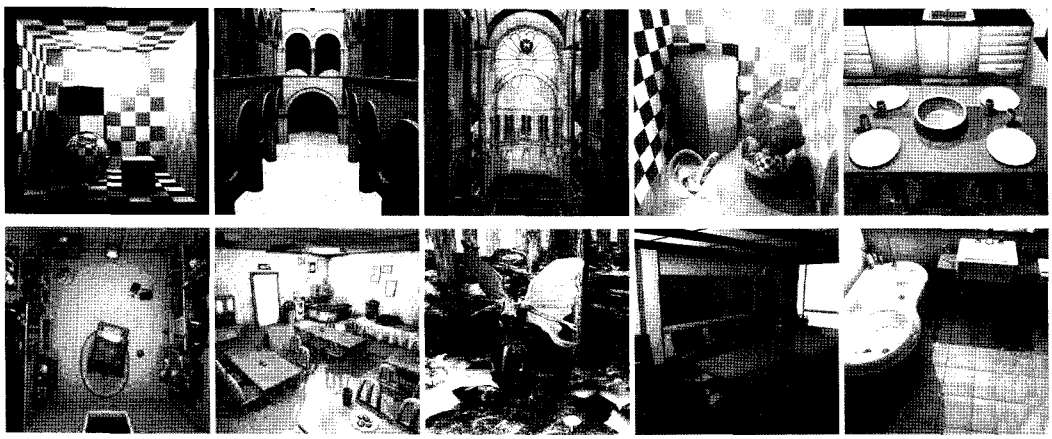

그림 5 kd-tree 탐색 성능 측정 실험에 사용된 장면. (윗줄: 왼쪽에서 오른쪽 순으로 Cbox, Sponza, Sibenik, Adv cbox, Kitchen, 아랫줄: 왼쪽에서 오른쪽 순으로 Room, Café, Fairy, Conference, Bathroom)

표 2 각 장면의 다각형 수와 구성된 kd-트리의 특징. 대체적으로 다각형의 수가 많을수록 생성된 트리의 최대 깊이와 리프 노드의 수가 증가하며 장면의 다각형의 공간 밀집도가 높을수록 각 리프 노드에 겹치는 삼각형의 수 및 리프 노드에 포함되는 최대 삼각형, 구성 시간이 증가한다.

	Triangles	Triangle offset number in leaf node	Increased triangle ratio in leaf node	Construction time(sec)	Max tree level	Node (inner+leaf)	leaf node	empty leaf node	empty leaf ratio	max triangle in leaf node
Cbox	996	2,359	236.84%	0.033916	58	2,251	1,126	461	40.94%	8
Sponza	66,454	271,373	408.36%	3.684146	72	209,643	104,822	26,620	25.39%	33
Sibenik	80,479	317,981	395.11%	5.015748	70	211,297	105,649	19,492	18.44%	27
Adv cbox	89,763	469,834	523.41%	7.061467	54	456,513	228,257	86,217	37.77%	20
Kitchen	101,015	414,134	409.97%	7.910624	57	308,277	154,139	38,427	24.93%	61
Room	117,855	842,658	714.99%	10.186898	74	649,139	324,570	79,083	24.36%	134
Café	171,425	1,410,079	822.56%	16.549731	86	960,561	480,281	95,414	19.86%	32
Fairy	174,117	1,546,463	888.17%	20.520823	72	1,113,621	556,811	139,984	25.14%	50
Conference	190,947	2,736,322	1433.02%	23.506568	80	1,377,297	688,649	98,606	14.31%	129
Bathroom	268,725	2,095,631	779.84%	28.00563	95	1,449,125	724,563	185,209	25.56%	595

표 3 각 장면에 대한 kd-트리 탐색 방법의 성능 비교. 대체적으로 rope 방법이 가장 빠른 성능을 보여준다. 640 × 480 해상도에서 주광선(primary ray)에 대해 시험을 수행하였다. (단위: Mrays/sec)

	Cbox	Sponza	Sibenik	Adv cbox	Kitchen	Room	Cafe	Fairy	Conference	BathRoom
kd-restart	76.16	20.25	32.99	80.85	27.43	28.95	24.08	13.89	33.96	21.01
kd-restart stack 1	78.70	22.71	37.74	81.39	30.31	32.11	27.17	14.76	36.59	22.73
kd-restart stack 2	80.20	25.70	43.29	83.18	34.18	35.20	30.39	16.28	38.37	25.66
kd-restart stack 3	82.03	27.84	46.00	83.24	35.29	37.50	33.40	18.04	40.77	27.61
pushdown	73.75	20.44	33.64	78.73	26.85	30.10	25.03	14.57	34.98	22.74
pushdown stack 1	75.43	22.20	37.00	82.03	29.89	31.67	27.26	16.32	36.40	23.41
pushdown stack 2	76.45	24.85	42.69	82.47	33.13	34.50	29.38	18.38	38.96	24.90
pushdown stack 3	82.52	27.25	45.79	83.10	34.57	36.90	32.26	20.14	39.98	26.90
local stack	64.14	23.39	39.93	62.92	30.69	33.39	24.55	17.50	36.42	31.11
cached stack 1	64.13	23.51	38.27	64.57	29.57	32.49	23.41	16.73	34.96	30.14
cached stack 2	66.14	23.84	39.00	65.57	32.56	33.48	26.32	17.35	35.61	31.05
cached stack 3	66.88	25.28	40.48	66.22	33.70	34.52	30.80	18.03	36.89	31.80
rope	76.76	32.85	49.56	81.44	41.69	41.99	38.45	26.74	41.72	38.58
rope with stack 1	75.34	33.02	49.00	80.49	40.75	41.12	37.80	25.92	42.11	38.71
rope with stack 2	76.10	31.86	48.36	81.97	40.22	40.40	37.06	25.21	40.98	37.75
rope with stack 3	75.80	29.74	47.09	80.21	40.20	39.87	36.28	24.72	40.80	36.86

다. 표 3은 각 장면의 kd-트리에 대해 GPU상에 하나의 멀티프로세서당 512개의 쓰레드를 활성화하여 kd-restart, kd-restart with short stack, pushdown, pushdown with short stack, local stack, cached stack, rope, rope with short stack 순으로 탐색을 수행하여 초당 처리 가능한 광선의 수를 보여주고 있으며, 표 4는 rope 방법 사용 시 요구되는 메모리의 크기와, 본 논문에서 제안한 rope with short stack 방법 사용 시 스택 깊이에 따른 요구 메모리 크기의 변화량을 보여주고 있다.

표 3을 통해 restart과 pushdown 방법은 성능상에서 서로 별다른 차이를 보여주고 있지 않지만 두 방법에 공유 메모리 스택을 사용할 경우 스택의 크기를 늘릴수록 성능 향상이 발생함을 알 수 있다. 그에 비해 local stack 방법은 restart 및 pushdown 방법보다는 전반적으로 빠르지만 깊이 3의 공유 메모리 스택을 사용하는 방법(cached stack 3)에 비해 느린 것을 알 수 있다. Cached stack 방법의 경우 비교 연산의 추가로 1 깊이의 공유 메모리 스택을 추가할 경우 느려지게 되지만 2 이상의 공유 메모리 스택을 설정하게 될 경우 점차적인 성능 향상이 발생한다.

Rope 방법의 경우 어느 정도 이상의 복잡도를 가지는 장면에 대해서는 평균적으로 가장 좋은 성능을 보여줌을 알 수 있지만 rope 방법에서 추가되는 광선과 노드의 AABB 교차 연산에 비해 약간의 스택 연산만으로 인접한 다음 노드를 탐색 가능한 간단한 장면에 대해서는 다른 방법에 비해 효율성이 떨어지는 경우를 확인할 수 있다(그림 6).

Rope with short stack 방법의 경우 대부분 성능 저

표 4 각 장면에 대한 rope 방법과 rope with short stack 방법의 메모리 사용량. 스택의 깊이를 1만큼 증가시킬수록 이전 메모리 사용량의 20%가 감소한다. (단위: Mbyte)

Scene	Traversal method	Memory usage for rope	Scene	Traversal method	Memory usage for rope
Cbox	rope	1.09	Sponza	rope	25.76
	rope with stack 1	0.90		rope with stack 1	21.92
	rope with stack 2	0.72		rope with stack 2	18.47
	rope with stack 3	0.59		rope with stack 3	15.80
Sibenik	rope	13.78	Adv cbox	rope	16.00
	rope with stack 1	11.28		rope with stack 1	13.47
	rope with stack 2	9.25		rope with stack 2	11.07
	rope with stack 3	7.71		rope with stack 3	9.31
Kitchen	rope	17.75	Room	Rope	20.34
	rope with stack 1	14.44		rope with stack 1	16.52
	rope with stack 2	11.53		rope with stack 2	13.29
	rope with stack 3	9.38		rope with stack 3	10.90
Café	rope	40.56	Fairy	rope	32.81
	rope with stack 1	33.66		rope with stack 1	26.54
	rope with stack 2	27.83		rope with stack 2	21.24
	rope with stack 3	23.30		rope with stack 3	17.29
Conference	rope	55.17	Bath room	Rope	49.31
	rope with stack 1	44.58		rope with stack 1	39.82
	rope with stack 2	36.16		rope with stack 2	31.75
	rope with stack 3	29.79		rope with stack 3	25.85

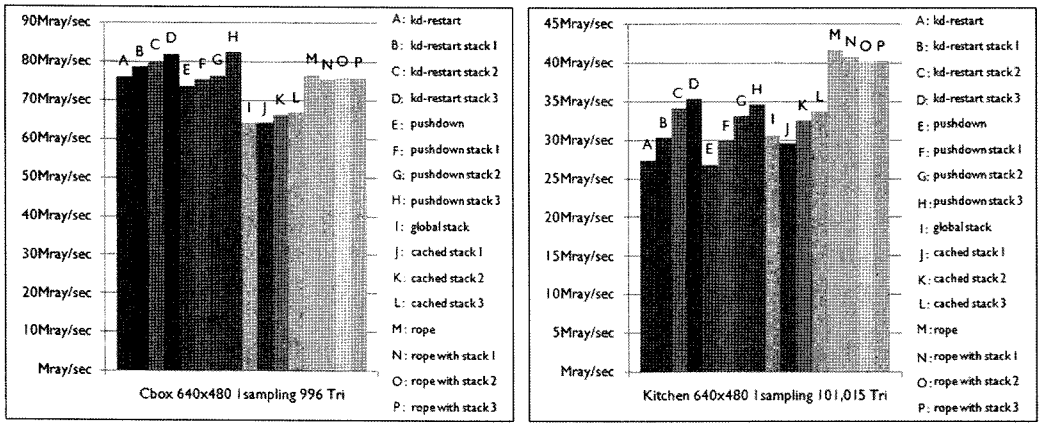

그림 6 실험에 사용한 장면 중 Cbox(삼각형 수: 996)와 Kitchen(삼각형 수: 101,015) 장면에 대한 각 방법의 성능 비교 그래프. 공유 메모리를 사용한 스택을 크게 할당할수록 더 좋은 성능을 보여준다. Kitchen과 같이 복잡한 장면에 대해서는 스택 연산이 상대적으로 많은 비중을 차지하기 때문에 스택 연산 없이 바로 인접한 노드를 찾아낼 수 있는 rope 방법이 뛰어난 성능을 보여주지만, Cbox와 같이 간단한 장면에 대해서는 스택 연산에 비해 인접 노드를 찾아내기 위한 연산 비용이 부담이 되기 때문에 상대적으로 효율적이지 못한 결과를 보여주고 있다.

하가 미미하게 발생하거나 비슷한 반면 표 4를 통해 공유 메모리 스택의 깊이를 1만큼 증가시킬수록 rope 방법으로 인한 메모리 사용량은 20%씩 감소함을 알 수 있다.

일반적인 광선 추적법의 구현에는 쉐이딩 및 그림자 효과등 다양한 계산이 포함되므로 그만큼 사용되는 레지스터의 수가 증가하게 된다. 그러므로 실질적으로 GPU 상의 하나의 멀티 프로세서에서 256개의 쓰레드만이 활성화되게 된다. 그만큼 성능의 저하가 발생하는 반면 쓰레드당 가용 공유 메모리가 늘어나게 되므로 최대 7 깊이의 공유 메모리 스택을 할당할 수 있게 된다. 이 경우 표 5에서와 같이 cached stack 방법은 같은 깊이의 restart with short stack 방법보다 더 좋은 성능을 발휘하게 되며 rope with short stack 방법의 경우

표 5 각 장면에 대하여 스택 크기를 7로 설정할 경우 cached stack 방법과 restart with short stack 탐색 방법의 성능 비교. 간단한 장면을 제외한 대부분의 경우 cached stack 방법이 비교적 좋은 성능을 보여준다. 640 × 480 해상도에서 주광선(primary ray)에 대해 실험을 수행하였다. (단위: Mrays/sec)

	Cbox	Sponza	Sibenik	Adv cbox	Kitchen	Room	Café	Fairy	Conference	BathRoom
cached stack 7	65.19	21.87	34.24	65.07	27.03	29.01	26.25	16.33	30.29	26.78
kd-restart stack 7	77.24	14.77	25.24	73.04	20.71	22.29	18.56	11.22	26.50	20.18

rope 방법으로 인해 사용되는 메모리의 약 80%를 줄일 수 있게 된다.

5. 결 론

본 논문에서는 GPU상에서의 효과적인 광선 추적법 구현에 있어 필수적인 요소인 광선-다각형 교차 연산을 위한 kd-트리에 대해 탐색 방법의 구현 기법에 대해 살펴보았다. 특히 GPU의 공유 또는 지역 메모리 상에서의 스택 구현 시 효율적인 접근을 위한 구체적인 스택 구조 및 최적화 방안을 제안함으로써 공유 메모리 상의 최대 가용 스택 깊이를 40% 이상 증가시킬 수 있었고, 지역 메모리 스택의 경우 메모리의 접근 회수를 줄임으로써 성능을 약 20% 향상시킬 수 있었으며, GPU에서 제공하는 공유 메모리를 최대한으로 사용함으로써 지역 메모리 접근을 최소화하며 기존 스택 기반 방법에 비해 향상된 성능을 보여주는 cached stack 방법 및 성능의 저하를 최소화하며 메모리 사용을 줄일 수 있는 rope with short stack 방법을 제안하였다(표 3-5).

또한 새로 제안한 탐색 방법들과 다양한 특성을 가진 장면들 상에서 기존에 제안된 탐색 방법들과의 성능 비교를 통해 임의로 주어진 장면의 kd-트리의 특성에 따라 최적의 성능을 발휘할 수 있는 탐색 방법 선택의 지표를 제시하였다. 이는 향후 개선될 GPU의 성능과 공유 메모리의 크기, 지역/전역 메모리 접근 구조 변화 등에 대해서도 상황에 맞는 최적의 탐색 방법의 선택 및 변형에 대한 올바른 지표를 제시할 수 있을 것이다.

참 고 문 헌

[1] T. Foley and J. Sugerman, "KD-tree acceleration structures for a GPU raytracer," *Proc. of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pp.15-22, 2005.

[2] D. R. Horn and J. Sugerman and M. Houston and P. Hanrahan, "Interactive k-d tree GPU ray tracing," *Proc. of the 2007 Symposium on Interactive 3D Graphics and Games*, pp.167-174, 2007.

[3] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-Time KD-Tree Construction on Graphics Hardware," *ACM Transactions on Graphics*, vol.27, no.5, pp.126:1-126:11, Dec 2008.

[4] S. Popov, J. Günther, H. Seidel, and P. Slusallek, "Stackless KD-Tree traversal for high performance GPU ray tracing," *Computer Graphics Forum*, vol.26, no.3, pp.415-424, 2007.

[5] I. Wald, "Realtime Ray Tracing and Interactive Global Illumination," *PhD thesis, Computer Graphics Group*, Saarland University, 2004.

[6] I. Wald, S. Boulos and P. Shirley, "Ray tracing deformable scenes using dynamic bounding volume hierarchies," *ACM Transactions on Graphics*, vol. 26, no.1, pp.1-18, 2007.

[7] J. Günther, S. Popov, H. -P. Seidel, P. Slusallek, "Realtime Ray Tracing on GPU with BVH-based Packet Traversal," In *IEEE Symposium on Interactive Ray Tracing*, pp.113-118, 2007.

[8] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan, "Ray tracing on programmable graphics hardware," *ACM Transactions on Graphics*, vol.21, no.3, pp.703-712, 2002.

[9] M. Shevtsov, A. Soupikov and A. Kapustin, "Highly parallel fast KD-tree construction for interactive ray tracing of dynamic scenes," *Computer Graphics Forum*, vol.26, no.3, pp.395-404, 2007.

[10] I. Wald, W. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. Parker and P. Shirley, "State of the Art in Ray Tracing Animated Scenes," In *Eurographics 2007 State of the Art Reports*, 2007.

[11] I. Wald, C. Benthin and S. Boulos, "Getting rid of packets - efficient SIMD single-ray traversal using multibranching BVHs," In *IEEE/EG Symposium on Interactive Ray Tracing*, pp.49-57, 2008.

[12] R. Overbeck, R. Ramamoorthi and W. Mark, "Large ray packets for real-time Whitted ray tracing," In *IEEE/EG Symposium on Interactive Ray Tracing*, pp.41-48, 2008.

[13] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," *ACM Transactions on Graphics*, vol.27, no.3, pp.1-15, 2008.

[14] NVIDIA. NVIDIA CUDA Compute Unified Device Architecture: Programming Guide (Version 2.3), 2009.

[15] 오상락, *GPU를 사용한 효과적인 Kd-Tree 탐색 알고리즘*, 서강대학교 석사학위 논문, 2009년 1월.

[16] 장병준, 임인성, "GPU상에서의 광선 추적을 위한 Kd-Tree 탐색 기법 비교 분석", 2009 한국컴퓨터그래픽스학회 하계학술대회 논문집, pp.71-73, 2009년 10월

장 병 준

2005년 8월 서강대학교 컴퓨터공학과 졸업(공학사). 2007년 2월 서강대학교 대학원 컴퓨터공학과 졸업(공학석사). 2007년 3월~현재 서강대학교 대학원 컴퓨터공학과 박사과정. 관심분야는 컴퓨터 그래픽스, 실시간 렌더링, GPU 프로그래밍

임 인 성

1985년 2월 서울대학교 자연과학대학 계산통계학과 졸업(이학사). 1987년 5월 Rutgers-The State University of New Jersey 컴퓨터학과 졸업(이학석사). 1991년 7월 Purdue University 컴퓨터학과 졸업(이학박사). 1999년 7월~2000년 7월 University of Texas at Austin의 TICAM의 연구 교수 1993년 3월~현재 서강대학교 컴퓨터공학과 교수. 관심분야는 컴퓨터 그래픽스, 과학적 가시화, 고성능 계산