GPU상에서 동작하는 Ray Tracing을 위한 효과적인 k-D tree 탐색 알고리즘

(An Efficient k-D tree Traversal Algorithm for Ray Tracing on a GPU)

강 윤 식 † 박 우 찬 †† 서 충 원 † 양 성 봉 †††
(Yoon-Sig Kang) (Woo-Chan Park) (Choong-Won Seo) (Sung-Bong Yang)

요 약 본 논문은 GPU상에서 작동되는 ray tracing을 위한 효과적인 k-D tree 탐색 알고리즘을 제안한다. 기존의 k-D tree를 위한 GPU 기반 탐색 알고리즘은 임의의 단말노드에서 교차되는 primitive를 찾지 못한 경우, root 노드 방향으로 bottom-up 탐색하여 부모 노드에서 bounding box 교차검사를 이용해 형제 노드의 기 방문 여부를 판단한다. 이러한 방법은 이미 방문한 부모 노드의 방문과 bounding box 교차검사를 중복적으로 수행한다. 본 논문에서 제안하는 알고리즘은 bottom-up 탐색을 수행 할 때 형제 노드가 이전에 방문했는지를 확인할 수 있는 효율적인 방법을 제시함으로써 형제노드 및 부모노드의 방문을 생략하도록 하고, 또한 아직 방문하지 않은 노드에 대해서만 bounding box 교차검사를 수행함으로써 중복된 연산을 피한다. 결과적으로 본 논문의 실험은 기존 알고리즘 대비 제안하는 알고리즘이 약 30%의 성능 향상이 있음을 보여 준다.

키워드 : 3차원 그래픽스, 광원 추적, 케이디트리, 지오메트리 프로세스 유닛, 레이-프리미티브 교차 알고리즘

Abstract This paper proposes an effective k-D tree traversal algorithm for ray tracing on a GPU. The previous k-D tree traverse algorithm based on GPU uses bottom-up searching from a leaf to the root after failing to find the ray intersected primitive in the leaf node. During the bottom-up search the algorithm decides the current node is visited or not from the parent node. In such a way, we need to visit the parent node which was already visited and the duplicated bounding box intersection tests. The new k-D tree traverse algorithm reduces the brother and parent duplicated visit by using an efficient method which decides whether the brother node is already visited or not during the bottom-up search. Also the algorithm take place bounding box intersection tests only for the nodes which is not yet done. As a result our experiment shows the new algorithm is about 30% faster than the previous.

Key words : 3D Graphics, Ray Tracing, k-D Tree, GPU, Ray Shooting Algorithm, Backtrack Algorithm

· 본 논문은 과학기술부와 산업자원부가 지원하는 국가 반도체 연구개발사업인 "시스템집적반도체기반기술개발사업(시스템 IC 2010)"을 통해 개발된 결과임을 밝힙니다.

† 학생회원 : 연세대학교 컴퓨터과학과
noori@yonsei.ac.kr
snowdeer@cs.yonsei.ac.kr
†† 정 회 원 : 세종대학교 정보처리학과 교수
pwchan@sejong.ac.kr
††† 종신회원 : 연세대학교 컴퓨터과학과 교수
yang@cs.yonsei.ac.kr

논문접수 : 2007년 4월 26일
심사완료 : 2008년 1월 8일

정보과학회논문지: 시스템 및 이론 제35권 제3호(2008.4)

1. 서 론

3D Graphics는 이미 몇몇 entertainment, simulation, medical 용도의 특수한 분야를 넘어서 game, desktop의 windows manager(Microsoft Windows Airo Glass, Compiz, Beryl)등으로 까지 보편화되어 사용되고 있다. 이러한 활용분야의 다양함으로 인해 사용자들은 좀 더 현실적인 화면을 요구하게 되고, 기존의 local illumination 방식과 비교해 좀 더 현실적인 화면을 구현할 수 있는 global illumination 방식에 대한 연구와 관심이 촉발 되고 있다. 이중 ray tracing은 알고리즘이 직관적이고 시스템의 구축이 용이해 많은 관심을 받고 있는 global illumination 알고리즘 중의 하나이다.

Ray tracing은 알고리즘의 특성상 높은 연산 복잡도를 가지고 있으며, 따라서 이의 활용이 대부분 영화와

같은 static scene 분야에 국한되고 있다. 또한 실험적인 실시간 rendering 시스템의 경우에도, 수십 개의 중앙처리 장치를 장착한 고가의 시스템에 국한되고 있다[1-3].

최근 들어 몇몇 실험들은 ray tracing 알고리즘의 성능 향상을 위해 저렴한 GPU를 이용할 수 있는 알고리즘을 소개하고 있다[4,5]. 또한 [6]은 GPU를 이용한 ray tracing 시스템이 CPU대비 가격 효율이 수배에 이름을 증명 하였다. 이들 중 [4]는 가속 알고리즘을 사용하지 않은 ray tracing 알고리즘이 GPU내의 pixel shader에서 수행 가능함을 보였고, [5]는 ray tracing에서 가장 많은 연산 시간을 소비하는 ray-triangle 교차검사 성능을 향상시키기 위해 uniform-grid accelerator 기반 ray tracing 시스템을 구현하였다. 그러나 [5]의 시스템은 uniform-grid accelerator의 문제점인 균일하지 않은 scene data에서 급격히 성능이 저하되는 문제를 가지고 있다. 이후 [7]은 기존의 acceleration structure 중 평균 수행 능력이 가장 우수 한 k-D tree[8]를 GPU에서 수행이 가능함을 보였다. 그러나 stack을 읽기 전용의 texture memory에 구현함으로써, stack push를 위한 연산은 CPU를 이용해야 하고, 따라서 CPU와 GPU의 빈번한 switching을 유발하여 GPU의 성능을 효율적으로 사용하지 못하는 문제점을 가지고 있다.

[9]는 [7]의 문제를 해결하기 위해 stack이 필요 없는 k-D tree 탐색 알고리즘인 restart, backtrack 두 개의 알고리즘을 제안하였다. 이 알고리즘은 단말노드에서 교차 primitive를 찾지 못하면 restart의 경우 root 노드에서 재탐색을 진행하고, backtrack의 경우 현재 단말노드에서 root 노드로의 bottom-up 탐색을 진행한다. 이들의 방법은 이미 방문한 노드에 대한 중복 방문이 필요하다. 예를 들어, 둘 중 더 빠른 탐색 속도를 제공하는 backtrack도 inner 노드에 대해 최대 3번의 방문이 필요하다. 이후 [10]은 restart 알고리즘에 몇 가지 최적화를 통해 성능을 향상시킨 알고리즘을 제안하였다. 우선 가장 큰 성능향상을 얻을 수 있었던 packet tracing은 [1]이 제안한 CPU에서 여러 개의 ray를 하나의 묶음으로 탐색을 수행하는 방법을 GPU로 이식함으로써 얻을 수 있었다. 다음으로 높은 성능을 얻을 수 있었던 short stack은 여분의 register를 이용해 작은 stack을 구현함으로써 부분적인 stack 효과를 얻고 있다. 다음으로 push down 방법은 restart 시 두 개의 자식노드가 모두 ray와 교차되는 지점에서부터 시작함으로써 중복노드를 제거하고 있다. 이중 packet tracing과 short stack은 본 논문의 제안 알고리즘에서도 병행 적용이 가능한 기능으로 [10]과 유사한 성능 향상을 얻을 수 있을 것으로 판단된다. 다음으로 push down은 restart 알고리즘에만 특화된 기법으로 본 논문의 제안알고리즘에 적용이 불가능하나 [10]의 실험결과에 의하면 약 1% 미만의 성능 향상이 있는 것으로 이로 인한 성능차이는 극히 미미할 것으로 판단된다.

[11]은 각 내부노드의 방문도 생략할 수 있는 stack이 필요 없는 알고리즘을 제안하였다. 각 단말노드에 인접한 6개의 노드들에 대한 포인터를 저장하고, 임의의 단말 노드에서 교차하는 primitive를 찾지 못할 경우 다음으로 교차하는 인접한 노드에서부터 재탐색이 가능하도록 하였다. 그러나 이 방법은 dynamic scene 과 같이 k-D tree를 지속적으로 업데이트해야하는 환경에서는 k-D tree 구축성능을 감소시키는 문제를 가지고 있다.

본 논문에서 제시하는 알고리즘은 bottom-up 탐색 시 임의의 노드에서 형제노드의 사전방문 여부를 확인할 수 있는 방법을 제시하여, 형제노드와 부모노드의 중복방문을 생략하고, 부모의형제 노드를 바로 탐색 한다. 이를 통해 내부 노드에 대한 최대 방문 횟수를 평균 1.5회로 줄인다. 또한 노드 방문 시 노드의 사전방문 여부를 판단하여, 이미 방문했던 노드에 대한 ray-bounding box(BB) 교차검사를 생략 하여 각 노드 당 최대 1회의 ray-BB 교차검사를 수행 한다. 이를 위하여 본 논문에서는 top-down 탐색시의 자식노드 방문 순서 결정방법을 bottom-up시에 적용할 수 있는 방법을 제안한다. 제안하는 알고리즘은 부모 노드의 분할 축 정보를 자식노드에도 저장한다. 또한 k-D tree를 저장할 때 breadth first search(BFS) 순서로 하였다. 이를 통해 자식노드에서 그 노드의 방문 순서를 결정할 수 있고, bottom-up 탐색이 가능하다.

우리는 실험을 통해 제안하는 알고리즘과 backtrack 알고리즘의 성능을 비교 분석하였다. 성능평가의 공정성을 유지하기 위해 하나의 k-D tree를 생성하고 이로부터 각각의 알고리즘에서 요구하는 형태의 자료구조로 구성하여 k-D tree의 효율성에 의한 성능 차이는 발생하지 않도록 하였다. 또한 image 크기에 따른 성능 차이를 확인하기 위해 4가지 다른 image 크기를 생성하도록 하고, Top-down, bottom-up 탐색시의 각각에 대한 노드방문 횟수와 탐색에 소요되는 시간을 측정하였다. 이상의 실험을 통해 제안하는 알고리즘이 backtrack 대비 약 20~50%의 성능이 향상됨을 확인하였다. 또한 제안하는 알고리즘은 이미지의 크기가 커질수록 backtrack과의 성능차이가 증가하는 특징을 가지고 있어, 고화질의 영상처리에 더욱 효과적임을 보여주고 있다.

논문 구성은 다음과 같다. 2장에서는 관련 연구에 대해서 설명하고, 다음으로 3장에서는 본 논문에서 제안하는 알고리즘을 설명하고, 이들에 대한 실험 결과를 4장에 설명하고자 한다. 마지막으로, 향후 연구 방향에 대

한 논의와 본 논문의 결론을 맺는다.

2. Background

2.1 k-D tree Ray-Shooting 알고리즘

k-D tree는 공간 분할 트리의 일종으로 빠른 공간 탐색을 가능하게 해주는 자료 구조이며, ray tracing system에서 가장 널리 사용되는 acceleration structure 중 하나이다[12]. Ray와 가장 먼저 intersection되는 primitive를 찾기 위한 k-D tree 탐색은 자식노드 중 ray의 시점에서 가까운 노드에 대하여 먼저 탐색한다는 점을 제외하고는 이진트리의 깊이 우선 탐색과 유사하다. 따라서 k-D tree의 ray-shooting 알고리즘은 이진트리의 깊이 우선 탐색과 같이 두개의 자식 노드 중 나중에 방문할 노드의 저장을 위해 stack을 필요로 한다.

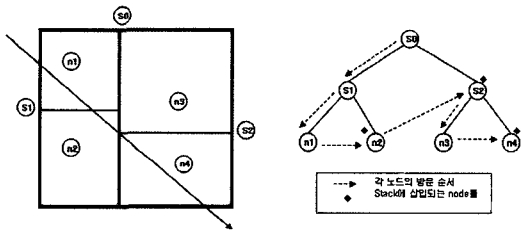

그림 1 CPU 기반 k-D tree 탐색 알고리즘 : Tree의 깊이 우선 탐색 알고리즘과 유사하나, 자식 노드의 방문 순서가 ray의 기울기에 따라 바뀔 수 있기 때문에 ray 시점에서 먼 쪽에 위치한 노드를 stack에 저장한다.

Stack을 이용한 k-D tree 탐색은 2개의 자식노드 중 ray와 교차되는 노드를 찾고, 만약 둘 중 하나만 교차가 된다면 그 노드에 대해 부모와 같이 재귀적으로 탐색을 수행한다. 반면 두개의 자식노드가 모두 ray와 교차되면 자식노드의 방문 순서에 따라 두 번째 방문노드를 stack에 넣고, 첫 번째 방문노드에 대해서 부모 노드와 같이 재귀적으로 교차검사를 진행한다. 이때 ray의 시점에서 가까운 노드는 첫 번째 방문노드가 되고 먼 쪽에 위치한 노드는 두 번째 방문노드가 된다.

자식노드의 방문 순서를 결정하기 위한 효율적인 방법은 [13]이 제안하였다. 이 방법은 부모노드에서 아래와 같은 3가지의 정보를 필요로 한다.

- 분할 축에 대한 ray 기울기 값의 양/음수 여부
- 부모노드의 분할 축(split axis)
- 왼쪽 또는 오른쪽 자식여부

k-D tree는 하나의 voxel을 두개의 공간으로 나눌 때 분할 평면을 기준으로, 이평면 보다 낮은 좌표공간을 왼쪽 자식으로 그 반대의 경우를 오른쪽 자식노드로 결정한다. 따라서 ray의 기울기가 양수 값을 가지면 ray는 왼쪽자식과 먼저 교차하고 다음으로 오른쪽자식과 교차한다. ray의 기울기가 음수 값을 가지면 그 반대가 된다.

그림 1의 S2 노드가 S1보다 ray 시점에서 먼 위치에 있으므로 S2를 stack에 넣고 S1을 방문 한다. 위와 같이 반복적으로 탐색하고 만약 단말노드에 다다르면, 단말노드에 포함된 primitive들에 대한 교차검사를 수행해 교차되는 primitive를 찾는다. 만약, 교차되는 primitive를 찾지 못하면, stack에서 하나의 노드를 pop해서 탐색을 다시 수행한다.

2.2 Restart & Backtrack 알고리즘

[9]는 GPU상에서 stack을 사용하지 않는 restart와 backtrack 두 가지의 ray tracing을 위한 k-D tree 탐색 알고리즘을 제안하였다. 이들의 알고리즘은 stack을 제거하기 위해 기존 알고리즘에서 stack이 저장하던 단말노드 방문이후 다음으로 탐색해야하는 노드를 선택할 수 있는 방법을 제안하고 있다. 이 방법은 top-down 탐색 시에는 기존의 stack 기반 탐색과 같은 방법으로 수행한다. 만약 단말노드에서 교차되는 primitive를 발견하지 못하면 이 노드와 ray의 교차점 중 ray 시점에서 먼 쪽에 위치한 교차점과 ray 시점의 거리를 *t_min*에 저장한다. 그리고 노드와 ray의 교차위치 중 최소 하나이상이 *t_min* 보다 크면 다음으로 방문해야하는 노드로 판단한다.

Restart 알고리즘은 단말노드 방문이후 *t_min*에 ray와 교차되는 두개의 교차거리 값 중 큰 값을 넣고 다음 탐색노드를 찾기 위해 root 에서부터 top-down으로 재탐색을 진행한다. 이는 root에서부터 다음 탐색노드에 이르는 경로상의 이전에 방문했던 모든 inner 노드에 대한 계속되는 중복 방문 및 교차검사가 발생한다.

Backtrack 알고리즘은 Restart 알고리즘의 노드 중복 방문을 감소시키기 위해 마지막 방문한 단말노드에서 이전에 탐색하지 않은 최초의 노드를 발견할 때까지 bottom-up 경로 상에 있는 노드들에 대해 순차적으로 교차검사를 수행해 이 값과 *t_min* 값과의 비교연산을 이용해 다음 탐색 노드를 선택한다. 이를 그림 2의 예를 들어 좀 더 구체적으로 설명 하면, (a)에서 *t_min* 값이 n1의 교차거리 값보다 작으므로 ray 시점에서 가까운 n1을 먼저 탐색 한다. n1이 탐색된 후 *t_min*과 *t_max*를 (b)와 같이 변경 한다. 변경된 *t_min*과 t_max사이에는 n2만이 포함되므로 n2가 다음 탐색 노드로 선택된다.

Backtrack 알고리즘의 bottom-up 탐색 시 가장 큰 문제점은 노드에 대한 방문을 결정하기 위해 부모노드에서 *t_min*이 자식노드의 교차점보다 큰 값을 갖는지를 확인해야 한다. 따라서 최악의 경우 두 자식노드를 탐색

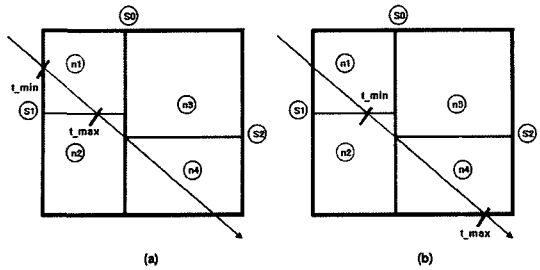

그림 2 Backtrack 알고리즘 개념도 : 노드 n1을 조사한 후 노드 n2를 조사하기 위해 교차 허용 범위를 변경한다. 이후 두 번째 조사인 (b)에서는 n2만이 선택된다.

할 때 첫 번째 자식노드에 대한 방문을 위해 한번, 두 번째 자식노드의 방문을 위해 한번 마지막으로 상위 노드로 이동하기 위해 한번 총 3번의 부모노드 방문이 필요하고 이는 stack 기반 알고리즘과 비교해 inner 노드에 대해 2번의 중복 방문을 필요로 한다. 또한 각 노드 방문 시에 교차 검사가 함께 수행되므로 두개의 자식노드에 대해 3번의 교차 검사가 필요하고, 각 노드 당 1번의 교차 검사만을 수행하는 stack기반 탐색 방법과 비교해 노드 당 평균 0.5회의 중복 교차검사가 필요하다. 본 논문에서는 이상에서 설명한 Backtrack의 문제점을 개선한 알고리즘을 제안한다.

3. 제안 알고리즘

이번 장에서는 본 논문에서 제안하는 알고리즘의 구체적인 설명을 위해 3.1절에 제안한 알고리즘의 노드 탐색방법에 대해 설명한다. 다음으로 3.2절은 이를 위한 노드 방문 순서 결정방법에 대하여 설명한다. 그리고 3.3절은 본 논문에서 제안한 알고리즘의 k-D tree 저장 방법을 설명한다. 마지막으로 제안하는 알고리즘의 노드 방문 횟수를 Backtrack 알고리즘과 비교해 설명한다.

3.1 제안 알고리즘의 노드 탐색 방법

제안하는 알고리즘은 backtrack과 같이 top-down 과 bottom-up 탐색 방법으로 구성된다. 이 중 top-down 탐색 방법은 backtrack과 같은 stack push만이 생략된 전통적인 k-D tree 탐색방법을 사용하고, bottom-up 탐색 방법은 아직 방문하지 않은 노드가 나타날 때까지 형제 노드와 부모의 형제노드를 자식노드에서 부모노드 방향으로 반복적으로 탐색한다.

제안하는 알고리즘은 backtrack 알고리즘의 두 가지 문제점을 개선하였다. 첫 번째로 임의의 노드에서 형제 노드로의 방문 여부를 바로 결정할 수 있으므로 부모노드의 방문이 필요치 않다. 다음으로 형제노드의 방문 필요여부를 교차 검사를 수행하기 이전에 결정하기 때문에 이미 방문이 완료된 노드에 대한 ray-BB 교차 검사를 생략 한다.

```
// n should be a leaf
// we can restart top_down search with top_down_node
top_down_node function bottom_up( ray, n ) {
  if( n == first visit node ) {
    if ( ray-BB_intersection( ray, brother of n ) == true ) {
      return (brother of n)
    }
  }

  node = brother of parent of n
  while ( node != second visit node
          AND ray-BB_intersection( ray, node ) == true ) {
    node = brother of parent of node
  }
  return (node)
}
```

그림 3 다음탐색 노드를 찾기 위한 pseudo code

제안하는 bottom-up 탐색 알고리즘은 아래의 그림 3과 같다. 먼저, 단말노드 n이 이미 방문된 첫 번째 노드인지 판단한다. 여기서 첫 번째 노드는 top-down 탐색 시 n이 형제노드보다 ray의 시점에 가까워 먼저 방문되는 노드를 의미한다. 제안하는 알고리즘의 자식노드 방문순서 결정은 [13]이 제안한 ray의 기울기와 자식노드의 좌/우 여부를 이용하는 방법을 기반으로 한다. 다만 차이점은 부모노드의 방문을 생략하기 위해 임의의 노드 n에서 자신이 첫 번째 노드인지를 판단할 수 있도록 한 것이다. 이에 대한 구체적인 설명은 3.2절에서 한다.

만약 임의의 노드 n이 이미 방문된 첫 번째 노드이면 형제 노드인 두 번째 노드가 다음 방문할 노드이므로, 형제 노드에 대하여 교차 검사를 수행한다. 교차 검사가 성공이면 형제 노드를 return한다. 이 과정에서 형제 노드에 대한 위치를 알아야 하며 이를 쉽게 하기 위하여 본 논문에서는 노드들을 BFS 순서로 저장하였다. k-D tree를 BFS 순서로 저장 할 경우 두 형제노드는 연속해서 메모리에 저장 되고 임의의 노드 n에 대해 $index_{(n)}$ = x의 홀수 또는 짝수에 대해 $index_{(brother\ of\ n)}$ = (x ± 1)가 된다.

위의 두 가지 조건을 만족하지 못하는 경우는 형제노드에 대한 방문이 필요치 않고, 이는 자신의 부모 노드를 root로 하는 sub-tree가 탐색이 완료된 것을 의미한다. 따라서 이 경우 부모의 형제 노드로 방문하여 탐색 완료 여부를 판단해야 한다. 여기서, 부모의 형제노드에 대한 위치는 자식노드에 저장한 부모노드 index와 위 단락에서 이야기한 BFS 순서를 이용해 부모의 형제를 쉽게 구한다. 그림 3의 후반부에 제시된 while 문에서는

부모의 형제 노드로 방문과 탐색 완료 여부의 판단을 반복적으로 수행하여 결국 while 문이 수행된 후에는 다음에 방문할 노드의 위치가 결정된다.

3.2 Bottom-up 탐색 시 첫 번째 방문 노드의 결정

k-D tree의 탐색 시 자식 노드의 방문 순서는 ray의 시점에서 가까이 위치한 순서로 진행된다. 만약 ray의 기울기가 양수이면 ray는 왼쪽에서 오른쪽 방향으로 진행하고, 그 반대의 경우 오른쪽에서 왼쪽으로 진행한다. 따라서 ray의 기울기가 양수이면 왼쪽 자식이 첫 번째 방문 노드가 되고, ray의 기울기가 음수이면 오른쪽 자식이 첫 번째 방문 노드가 됨을 알 수 있다. [13]은 이상의 방법을 부모노드에서 자식노드의 방문 순서를 결정하기 위해 사용하였다. 반면 제안하는 알고리즘은 이 방법을 bottom-up 탐색 시에 임의의 자식노드에서 이 노드가 첫 번째 방문노드인지의 판단을 위해 사용하였다. 이를 위해 제안하는 알고리즘은 임의의 노드 n의 방문 시 2.1절에서 제시한 세 가지의 정보를 구할 수 있는 방법이 필요하다.

이 중 ray의 기울기 정보는 k-D tree의 탐색 과정에서 전역 적으로 참조가 가능하므로 임의의 노드에서 쉽게 구할 수 있다. 반면 분할 축 값과 노드의 좌/우 여부는 부모 노드에 저장되므로 이 값을 자식 노드에 복사하거나 또는 필요할 때마다 부모노드를 참조하여 사용하는 것이 요구된다. 제안하는 알고리즘은 분할 축 정보는 자식노드에 복사하는 방법을 사용하였고, 좌/우 판별은 BFS 순서로 저장된 k-D tree의 특징을 이용해 추가적인 overhead 없이 구한다.

BFS로 저장된 k-D tree는 형제노드의 index를 구하는 방법 뿐 아니라 임의의 노드 n이 좌측 또는 우측 노드인지에 대한 판단을 쉽게 할 수 있는 방법도 제공한다. k-D tree는 항상 2개의 자식노드를 갖고, 이들이 BFS 순서에 의해 연속해서 저장되면 좌측노드는 항상 홀수 index 값을 그리고 우측노드는 항상 짝수 index 값을 갖게 된다. 따라서 이 특성을 이용해 임의의 노드에 대한 index가 홀수인지 또는 짝수인지를 검사해 이 노드가 좌측 또는 우측 노드인지를 판단할 수 있다.

3.3 제안 알고리즘의 BFS 순서 k-D tree 저장 구조

제안 알고리즘은 k-D tree를 BFS 순서로 저장을 한다. 반면 대부분의 k-D tree 탐색 알고리즘들은[10,14] depth first search(DFS) 순서로 k-D tree를 저장한다. 이는 모든 ray tracing을 위한 k-D tree의 탐색 알고리즘이 DFS 순서로 이루어지고 있기 때문에 너무나 자연스러운 결과라고 할 수 있다. DFS 순서로 k-D tree의 노드를 메모리에 저장하고 이 순서로 탐색을 할 경우 data의 locality가 증가하고 이는 cache 효율의 증가를 가져올 수 있다[14].

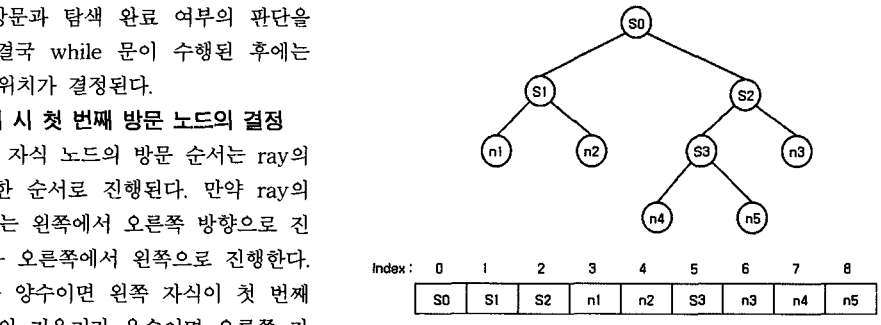

그림 4 k-D tree texture memory의 구조

제안하는 알고리즘은 BFS 순서로 저장 하므로 앞에서 언급한 cache효율 증가의 이점을 갖지 못할 수 있다. 그러나 우리의 실험 결과는 제안하는 알고리즘이 이러한 약점에도 불구하고 기존의 backtrack 알고리즘 대비 충분한 성능 향상이 있음을 보여주고 있다.

그림 4는 BFS로 저장된 k-D tree의 예를 보여 주고 있다. BFS 순서로 저장됨으로 인해 모든 왼쪽 자식(S1, n1, S3, n4)은 홀수 index 값을 갖는다.

[9]는 자식노드의 BB 교차검사를 부모 노드에서 수행한다. 따라서 단말노드의 경우 BB 값을 저장할 필요가 없다. 반면 제안하는 알고리즘은 단말노드에서 형제노드로 바로 방문이 가능하고 방문한 이후 BB 교차검사를 수행하므로, 단말노드에도 BB 정보를 필요로 한다. 이를 위해 Backtrack 알고리즘 대비 각 단말노드 당 24byte의 추가 메모리가 요구 된다. 또한 root노드를 제외한 모든 노드는 3.2에서 언급한 것과 같이 부모 노드의 split 정보를 복사해서 가져야 하므로 추가로 각 노드 당 4byte의 추가 메모리를 사용한다.

3.4 제안 알고리즘의 노드 방문 및 BB 교차 테스트 횟수

제안하는 알고리즘은 Backtrack 알고리즘과 같이 단말노드에 대해서는 최대 1회 방문을 보장한다. 반면 내부 노드는 top-down 탐색 시 1회의 방문과 bottom-up 탐색 시 두개의 형제노드 중 첫 번째 방문한 노드만이 중복 방문 되어 각 노드 당 평균 1.5회의 최대 방문 횟수를 갖는다. 또한 본 알고리즘은 노드에 대한 ray-BB 교차 검사를 기 방문 여부를 판단한 이후에 수행하므로

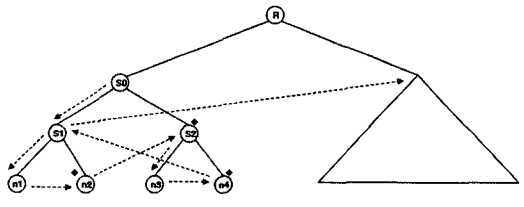

그림 5 Proposed 알고리즘의 노드 방문 순서

각 노드별 중복은 발생하지 않는다.

그림 5를 예로 들면, n1 방문 후 n2를 방문하고 다음에는 부모의 형제인 S2를 방문하므로 단말노드에 대한 중복 방문은 발생하지 않는다. 그리고 S1은 첫 번째 방문 노드이므로 top-down 탐색 시 1회 그리고 bottom-up 탐색 시 n4 노드 탐색 이후에 한 번 더 방문된다.

4. 성능 테스트

우리는 실험의 편의성을 위해 [13]에서와 같이 first ray에 대해서만 실험을 수행하였으며, shadow ray를 포함한 파생 ray에 대한 성능 테스트는 생략하였다. 이는 일반적으로 acceleration structure와 ray의 교차 테스트는 ray의 수에 정비례하여 수행 시간이 소모 되므로, 기존의 알고리즘과 제안하는 알고리즘이 같은 수의 ray 에 대해 k-D tree 교차검사를 수행할 경우 그 성능 비는 동일할 것으로 판단되기 때문이다.

일반적으로 k-D tree의 생성은 Surface Area Heuristic(SAH)[16]가 가장 널리 사용되고 있으며, 우리 또한 SAH에 기반을 둔 [14]의 생성 알고리즘을 사용한다. 다만 생성된 k-D tree는 우리의 알고리즘에 적합한 형태로 메모리 구성이 되어야 한다. 또한 GPU에서 동작 가능하도록 texture 메모리에 로드한다[15].

제안하는 알고리즘은 k-D tree의 탐색 알고리즘에 국한되므로, k-D tree의 생성에 대한 문제는 고려하지 않았다. 따라서 본 실험에서는 k-D tree 데이터의 생성 시간에 대한 평가는 수행하고 있지 않으며, 이미 생성된 k-D tree 데이터에 대한 교차검사 성능을 실험하였다.

4.1 성능 테스트 환경

본 실험결과에 가장 많은 영향을 주는 요소인 GPU는 Nvidia 사의 Geforce 7800GT를 사용하였다. 7800GT는 24개의 pixel pipe-line과 8개의 vertex pipe-line을 가지고 있으며, GPU의 동작 속도는 400MHz로 동작 한다. GPU 프로그래밍은 NVidia 사의 shader compiler 인 Cg 1.4를 이용하였다.

자세한 테스트 환경은 아래와 같다.

- Windows XP + Service Pack 2
- Nvidia Shader Compiler Cg 1.4
- Nvidia v87.43 Driver
- Pentium4 2.8GHz Dual Core
- 1Gbyte DDR2 Memory
- Nvidia 7800GT with 256Mbytes

4.2 테스트 Scene

테스트를 위한 3D data는 Stanford Bunny(69,451개의 삼각형), Bart Kitchen(110,561개의 삼각형) 그리고 Bart Robot(71,708개의 삼각형)을 사용하였고 모든 k-D tree의 깊이는 최대 16이 되도록 하였다.

4.3 테스트 결과

테스트에 사용된 Windows XP는 multi-threaded OS로 내부적으로 동시에 여러 가지 프로그램이 수행되는 관계로, 실험 결과에 영향을 주는 다양한 현상이 발생할 수 있으므로 실험 결과의 공정성을 위해 총 5회의 실험을 수행하여 평균값을 계산 하였다. 또한 생성되는

표 1 Backtrack 알고리즘과 제안 알고리즘의 노드 방문 횟수 비교

			back track	제안알고리즘
Kitchen	512×512	하향탐색	13,257,222	11,346,984
		상향탐색	10,998,464	9,087,842
		형제탐색		2,040,054
		합계	24,255,686	20,434,826
		감소율(%)	7.34	
	1024×1024	하향탐색	52,750,665	45,203,797
		상향탐색	44,222,727	36,674,963
		형제탐색		8,030,096
		합계	96,973,399.34	81,878,760
		감소율(%)	7.28	
Robots	512×512	하향탐색	15,181,713	12,708,480
		상향탐색	11,282,520	8,814,507
		형제탐색		2,693,197
		합계	26,464,240.28	21,522,987
		감소율(%)	8.49	
	1024×1024	하향탐색	61,185,870	51,334,328
		상향탐색	45,597,597	35,746,195
		형제탐색		10,899,659
		합계	106,783,475.49	87,080,523
		감소율(%)	8.24	

표 2 Backtrack 알고리즘과 제안알고리즘의 BB 교차검사 횟수 비교

		Backtrack	제안알고리즘	감소율(%)
Bunny	512×512	10,315,940	7,073,306	31.43
	1024×1024	40,035,115	25,632,385	35.97
Kitchen	512×512	24,255,686	17,036,337	29.76
	1024×1024	96,973,392	67,315,729	30.58
Robots	512×512	26,464,233	18,921,364	28.50
	1024×1024	106,783,467	76,135,200	28.70

표 3 Backtrack 알고리즘과 제안알고리즘의 수행속도 테스트 결과(초)

	Bunny		Kitchen		Robots	
	Backtrack	제안알고리즘	Backtrack	제안알고리즘	Backtrack	제안알고리즘
128	0.781	0.609	0.932	0.813	1.313	1.125
256	1.343	1.109	1.863	1.594	3.172	2.312
512	2.812	2.062	4.032	3.047	7.406	4.500
1024	6.641	4.969	9.853	7.141	20.328	10.844

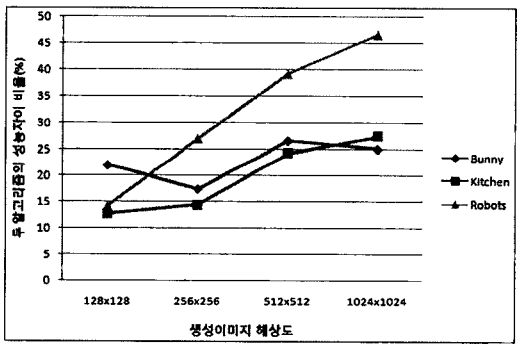

그림 6 두 알고리즘의 성능차이 비율

이미지의 크기에 따른 성능 차이를 확인하기 위해 4가지의 크기가 다른 이미지로 생성하여 실험하였다.

표 1은 Backtrack과 제안하는 알고리즘의 노드 방문 회수를 비교하고 있다. 제안 알고리즘은 Backtrack과 달리 형제노드로의 바로탐색이 가능하므로 이에 대한 항목이 추가되었다. 전반적으로 scene의 복잡도가 높은 Robots 이 좀 더 높은 감소율을 보여주고 있다. 이는 복잡한 scene일수록 한 번의 단말노드 방문에 교차되는 primitive를 찾을 확률이 낮으므로 계속되는 탐색을 위한 상향 및 하향 탐색이 상대적으로 빈번히 발생한다.

본 알고리즘은 아직 방문하지 않은 노드에 대해서만 BB 교차검사를 수행한다. 반면 Backtrack 알고리즘은 방문노드의 판단을 위해 BB 교차검사를 수행하므로 이미 교차검사를 수행했던 노드에 대해서도 중복 BB 교차검사를 수행한다. 표 2에서는 제안 알고리즘이 Backtrack 알고리즘 대비 약 30% 정도의 BB 교차검사 감소 효과가 있음을 보여준다.

표 3과 같이 제안된 알고리즘은 backtrack 알고리즘 대비 최대 약 2배의 성능 향상 효과가 보이고 있고, 또한 그림 6은 이미지의 크기가 커질수록 성능 차이가 커짐을 보여주고 있다. 이러한 성능 향상의 차이는 backtrack 알고리즘이 제안된 알고리즘 대비 작은 크기의 data를 사용하므로 GPU로의 data 전송속도에서 우수하고, 반면 탐색속도는 제안하는 알고리즘이 우수 하다. 따라서 고정된 data 크기에서 이미지의 크기가 커질수록 연산량이 증가하므로 두 알고리즘 간 성능 차이가 증가하게 된다.

5. 결 론

본 논문에서는 스택을 사용하지 않는 k-D tree 탐색 알고리즘을 제안하고 있다. 이 알고리즘은 기존 알고리즘 대비 노드방문 횟수가 약 7% 가량 감소되고, 중복 BB 교차검사는 약 30% 정도 감소한다. 이미지의 크기가 커질수록 그 성능 향상 폭이 증가돼는 본 알고리즘의 특징은 고화질의 영상처리에서 더욱 효과적인 알고리즘 이라고 할 수 있다. 또한 Stack을 사용하지 않으므로 향후 Ray-Tracing 가속기의 개발 시 단순한 하드웨어 요구 사항으로 인해 다중의 pipe-line구성이 용이하게 함은 물론이고, 좀 더 효율적이고 단순한 시스템 설계가 가능하도록 할 수 있을 것으로 믿는다.

참 고 문 헌

[1] I. Wald, C. Benthin, M. Wagner, and P. Slusallek, "Interactive Rendering with Coherent Ray Tracing," *EUROGRAPHICS*, Vol.20 No.3 pp. 153-164, 2001.

[2] I. Wald, T. J. Purcell, J. Schmittler, C. Benthin, and P. Slusallek, "Realtime Ray Tracing and its Use for Interactive Global Illumination," *EUROGRAPHICS State of the Art Reports*, pp. 85-122, 2003.

[3] S. Parker, W. Martin, P.-P. J. Sloan, P. Shirley, B. Smits, and C. Hansen, "Interactive ray tracing," *Proceedings of the 1999 symposium on Interactive 3D graphics*, pp. 119-126, 1999.
[4] A. Nathan, D. Jesse, and C. John, "The Ray Engine," *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pp. 37-46, 2002.
[5] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan, "Ray tracing on programmable graphics hardware," *ACM Trans. Graph*, pp. 703-712, 2002.
[6] N. A. Carr, J. D. Hall, and J. C. Hart, "The Ray Engine," *Tech. Rep. UIUCDCS-R-2002-2269*, Department of Computer Science, University of Illinois, 2002.
[7] M. Ernst, C. Vogelgsang, and G. Greiner, "Stack implementation on programmable graphics hardware," *Vision Modeling and Visualization*, pp. 255-262, 2004.
[8] V. Havran, "Heuristic Ray Shooting Algorithms," Ph.D. thesis, *Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague*, 2000.
[9] T. Foley, and J. Sugerman, "KD-Tree Acceleration Structures for a GPU Raytracer," *Proceedings of Graphics Hardware*, pp. 15-22, 2005.
[10] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan, "Interactive k-d tree GPU raytracing," *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pp. 167-174, 2007.
[11] S. Popov, J. Gunther, H. P. Seidel, and P. Slusallek, "Stackless KD-Tree Traversal for High Performance GPU Ray Tracing," *EUROGRAPHICS*, Vol.10, No.1, 2007.
[12] J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Communications of the ACM*, pp. 509-517, 1975.
[13] F. W. Jason, "Data structures for ray tracing," *Proceedings of the workshop on Data Structures for Raster Graphics*, pp. 57-73, 1986.
[14] M. Pharr, and G. Humphreys, *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, 2004.
[15] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream computing on graphics hardware," *Proceedings of ACM SIGGRAPH*, 2004.
[16] J. D. MacDonald, and S. B. Kellogg, "Heuristics for ray tracing using space subdivision," *The Visual Computer*, 1990.

강 윤 식
연세대학교 컴퓨터과학과 석사(1999년) 취득. 연세대학교 컴퓨터과학과 박사과정(2004년~) 재학 중

박 우 찬
연세대학교 컴퓨터과학과 학사(1993년), 석사(1995년), 박사(2000년) 학위 취득. 세종대학교 부교수 재직

서 충 원
연세대학교 컴퓨터과학과 학사(2001년), 석사(2007년) 학위 취득. 현재 삼성전자 정보통신총괄 선임연구원

양 성 봉
Oklahoma 주립대학교 컴퓨터과학과 박사학위(1992년) 취득. 현재 연세대학교 컴퓨터과학과 정교수