메타 테이블을 이용한 효율적인 레이트레이싱 알고리즘

서충원°, 강윤식, 양성봉
연세대학교 컴퓨터 과학과
{snowdeer°, noori, yang}@cs.yonsei.ac.kr

An Effective Ray Tracing Algorithm Using a Meta Table

Choong-Won Seo°. Y.S. Kang. S.B. Yang.
Department of Computer Science, Yonsei University

요 약

본 논문에서는 메타 테이블(meta table)을 이용한 광선(ray)과 삼각형(triangle)들의 교차검사를 할 수 있는 효율적인 레이트레이싱 알고리즘을 제안한다. 기존의 kd-tree 탐색은 깊이 우선 탐색을 하면서 이미 방문했던 노드들을 방문하지 않기 위해서 스택을 이용하는 방법을 택하고 있는데, 본 논문에서 제안하는 알고리즘은 스택을 사용하지 않고 읽기 전용으로만 쓰이는 메타 테이블을 둠으로써 기존의 트리 탐색 과정보다 효율적으로 트리의 리프 노드들에 접근할 수 있도록 하였다. 실험결과 제안된 레이트레이싱 알고리즘이 기존의 kd-tree의 트리 탐색보다 노드 방문을 5배 이상 적게 하였고, 이미지 렌더링 시간도 총 2배 정도 향상됨을 볼 수 있었다.

1. 서 론

최근 컴퓨터 그래픽 하드웨어의 발달로 컴퓨터 그래픽은 보다 정교하고 실제적인 영상을 보여줄 수 있게 되었다. 그러기 위해서는 글로벌 일루미네이션(global illumination)의 표현이 중요하게 되었다. 글로벌 일루미네이션이란 광원과 물체간의 작용만 고려해서 이미지의 색상을 결정하는 로컬 일루미네이션(local illumination)과는 달리 물체의 광학적인 성질을 이용하여 빛의 반사와 굴절, 그리고 주위 사물들과 빛의 작용에 의한 간접적인 빛까지 계산해서 이미지를 생성하는 방식이다[1]. 이러한 글로벌 일루미네이션을 구현하고 있는 알고리즘들이 많이 있는데 [2]에 의하면 레이트레이싱(ray tracing) 알고리즘이 직관적이고 쉽게 구현이 가능하기 때문에 많이 사용된다.

레이트레이싱은 광선(ray)과 다른 사물들과의 반사, 굴절, 투과 등의 결과를 추적하여 이미지를 생성하는 기법이기 때문에 이 과정에서 광선이 어떤 사물과 교차하는지 알아내는 것이 렌더링 속도에 많은 영향을 미친다. 하나의 장면(scene)은 매우 많은 삼각형(triangle)들로 이루어져 있고, 이 중에서 주어진 광선과 가장 먼저 교차하는 삼각형을 찾아내는 것이 레이트레이싱의 가장 큰 과제라고 할 수 있다[3]. 레이트레이싱은 화면의 각 픽셀(pixel)이나 fragment들 마다 광선을 쏘는 방식이며, 장면마다 모든 광선과 모든 삼각형의 교차검사(ray & triangle intersection test)를 하는 것은 상당한 시간이 요구된다. 이러한 교차검사를 빨리 처리하기 위해 여러 가지 가속화 방법이 사용된다.

Uniform/Non-uniform Grid, BVH(Bounding Volume Hierarchies), octree, kd-tree 등의 많고 다양한 교차검사 가속기 구조들 중에서 [4]는 CPU에서 특정한 영상에서가 아닌 범용적인 목적으로 사용할 경우에는 kd-tree가 가장 좋은 성능을 보인다고 한다.

본 논문에서는 메타 테이블(meta table)을 이용하여 kd-tree를 탐색하는 기존의 트리탐색 방법보다 더 빠른 속도를 보여주는 레이트레이싱 알고리즘을 제안한다. 실험을 통해 제안된 알고리즘이 약 30%~50% 성능을 향상시킴을 확인할 수 있었다.

이후 논문의 구성은 다음과 같다. 2장에서 레이트레이싱과 기존의 kd-tree 탐색 과정에 대해서 설명하고, 3장에서는 본 논문이 제안하는 알고리즘을 설명하고자 한다. 4장에서 이들에 대한 비교 실험결과를 설명하고, 마지막으로 5장에서 논문의 결론과 향후 연구 방향에 대해 설명하였다.

2. 관련연구

2.1 레이트레이싱

레이트레이싱은 빛의 광학적인 성질을 이용한 렌더링 기법이다. 빛이 직진하는 성질을 이용해서 광선의 경로를 추적하다가 광선이 물체를 만났을 때 발생하는 반사, 굴절 또는 투과 현상을 고려하여 이미지의 색을 결정한다. 그리고 이 과정에서 생성되는 광선들에 대해서도 같은 계산을 반복하여 보다 사실적인 이미지를 생성할 수 있다. 이 때 광선의 추적은 광원에서부터 시작하는 것이 아니라 관찰자의 눈에서부터 역추적(backward tracing)하여 계산한다.

2.2 Uniform Grid 가속 구조

Uniform grid를 사용하는 가속 구조는 장면을 x, y, z의 3개의 축에 따라 균일한 크기의 격자(grid)로 나누어 광선이 지나가는 격자에 속하는 삼각형에 대해서만 교차검사를 하는 방식이다[그림1].

이 때 광선이 지나가는 격자들의 인덱스(index)는 [그림1]과 같이 d_x와 d_y를 이용하여 단순한 수학적인 계산[5]만을 이용하여 쉽게 구해낼 수 있다.

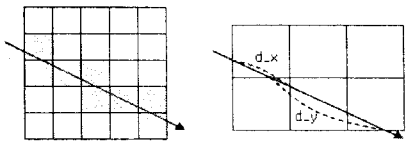

그림 1 : *Uniform Grid 가속 구조*

Uniform grid 가속 구조는 레이트레이싱의 가속 구조들 중에 가장 간단하여 널리 활용되고 있지만, 삼각형의 밀집도가 고르지 않을 경우 성능이 저하되는 문제점을 가지고 있다.

2.3 기존의 kd-tree 탐색 과정

현재 레이트레이싱에서 사용되는 kd-tree는 특정 축을 따라 공간을 분할하여 자식 노드(node)들을 생성한 형태로 되어 있다. 이 때 분할되는 분할 평면(split-plane)의 좌표는 공간내의 삼각형의 개수와 삼각형을 포함하는 공간의 면적을 고려해서 분할하는데, 보통 넓은 면적에는 삼각형 개수를 적게 할당하고 좁은 면적에 삼각형 개수를 많이 할당하는 방식의 알고리즘인 SAH(Surface Area Heuristic)를 많이 사용한다[5].

이러한 과정을 거쳐 생성된 kd-tree를 이용하면 특정 광선을 쏘았을 때 광선이 지나가는 공간 속의 삼각형들에 대해서만 교차검사를 하면 되기 때문에 공간을 나누지 않았을 때보다 검사 횟수를 많이 줄일 수가 있다. [그림2]를 보면 kd-tree에서 광선이 지나가는 영역인 n0, n3, n4에 속하는 삼각형들에 대해서만 교차검사를 실행하면 되는 것을 알 수 있다.

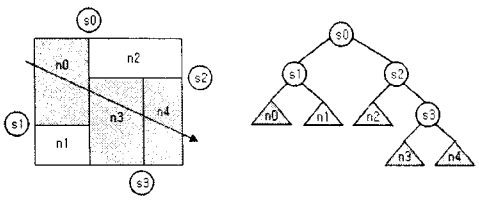

그림2 : *kd-tree에서의 광선&삼각형의 교차검사*

기존의 kd-tree 탐색 과정을 살펴보면[5], 맨 처음은 광선과 루트 노드(root node)의 바운딩 박스(bounding box)와의 교차검사로부터 시작된다. 만약 서로 교차되지 않은 경우는 광선이 kd-tree내의 어떠한 삼각형과도 교차되지 않으므로 결과값을 false로 하고 검사를 종료하면 된다. 만약 교차된 경우는 처음으로 광선과 교차하는 삼각형을 찾기 위해 트리를 탐색한다.

kd-tree 탐색에서 현재 방문하고 있는 노드가 리프 노드(leaf node)가 아닌 경우에는 [그림3]과 같은 방법에 따라 탐색을 진행하며, 리프 노드인 경우는 [그림4]와 같은 방법에 따라 탐색을 진행한다.

[그림5(a)]를 보면 루트 노드 s0 와 광선이 t_min, t_max의 위치에서 교차하며 t_split의 위치에서 분할 평면과 교차한다. [그림3]에 따라 FarNode에 해당하는 s2를 스택에 저장하고 NearNode에 해당하는 s1에 대해서 검사를 계속 진행한다. [그림5(b)]에서는 t_split > t_max 이므로 NearNode에 해당하는 n0에 대해서만 검사를 계속해서 진행하고 n1은 건너뛰게 된다. n0의 경우는 리프 노드이므로 [그림5(c)]와 같이 스택에 저장되어 있는 s2를 꺼내어서 s2에 대한 검사를 계속해서 진행한다. [그림5(d)]와 같이 s2에서 t_split < t_max 이므로 FarNode에 해당하는 s3에 대해서 검사를 진행한다. [그림5]의 (e)와 (f)의 경우에도 같은 방법으로 진행하여 kd-tree의 트리 탐색을 완료하게 되며, 이 과정 중에 어느 과정에서라도 광선과 교차되는 삼각형을 만나게 되면 그 삼각형의 정보를 반환하며 검사는 종료된다.

이와 같이 기존의 kd-tree 탐색방법은 스택을 이용하여 이미 방문했던 노드들을 재방문할 필요 없이 빠르게 트리의 노드들을 탐색하는 방법을 취하고 있다.

t_min, t_max	*현재 노드와 광선의 교차점. 광선의 시작점과 가까운 교차점이 t_min, 먼 교차점이 t_max이다.*
t_split	*현재 노드의 분할 평면과 광선의 교차점.*
NearNode, FarNode	*현재 노드의 자식 노드들. 광선의 시작점과 가까운 자식 노드가 NearNode, 먼 자식 노드가 FarNode이다.*

```
if ( t_split >= t_max or t_split < 0 )
   search_continue(NearNode);
else if ( t_split <= t_min )
   search_continue(FarNode);
else
   stack.push(secondNode);
   search_continue(NearNode);
```

그림3 : *현재 노드가 리프 노드가 아닌 경우*

triangles	*현재 리프 노드에 속해있는 삼각형들*

```
if (currentNode == leafNode)
   if (intersect_test(triangles))
      return true & triangle;
   else
      if (stack.is_empty() == true)
         return false & null;
      else
         nextNode = stack.pop();
         search_continue(nextNode);
```

그림4 : *현재 노드가 리프 노드인 경우*

3. 제안하는 알고리즘

본 논문에서는 kd-tree에서 트리 탐색 과정을 줄이기 위해서 스택을 사용하지 않고, 트리의 리프 노드들의 주소값을 갖고 있는 메타 테이블을 사용하여 트리 탐색 속도를 향상시켰다. Uniform grid를 사용하는 가속 구조에서 광선이 지나가는 격자들의 인덱스를 구할 때 단순한 수학적인 공식에 의해 쉽게 구할 수 있다는 장점과 공간의 면적과 삼각형의 개수를 이용하여 효율적으로 삼각형들의 정보를 관리할 수 있는 kd-tree의 장점을 이용하였다.

메타 테이블의 모습은 [그림5]와 같이 각각의 노드들이 같은 크기로 이루어져 있으며, 각 노드들은 kd-tree의 리프 노드들이 있는 위치에 각각 대응하여 위치하고 있다. 각 노드들은 배열로 이루어져 있으며, 각각 대응되

는 kd-tree의 리프 노드들의 주소값을 가리킨다.

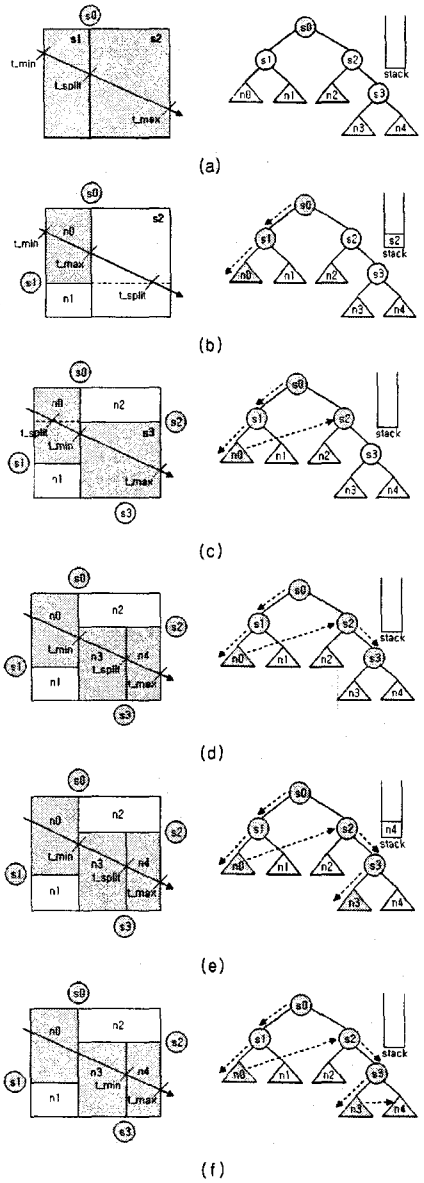

그림5 : *기존의 kd-tree 탐색 과정*

이러한 형태의 메타 테이블을 생성하는 방법은 크게 두 가지가 있다. 첫 번째 방법은 테이블을 구성하고 있는 하나의 노드의 x축, y축, z축으로의 단위 길이를 먼저 정한 다음 그 단위 길이를 이용해서 kd-tree의 분할 평면의 좌표를 구하는 방법이 있다. 이 단위 길이는 장면마다 다르게 할 수 있다. 두 번째 방법은 기존에 미리 생성되어져 있는 kd-tree의 노드들의 분할 평면 좌표를 모두 검사한 다음 각 축별로 좌표들의 최대공약수를 구하여, 그 값을 해당 축의 단위 길이로 정해서 메타 테이블을 생성하는 방법이 있다. 어느 방법에서나 중요한 것은 kd-tree의 분할 평면의 좌표는 단위 길이의 배수인 위치에서 정해져야 한다는 점이다.

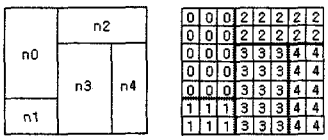

그림 5 : 메타 테이블

이렇게 메타 테이블의 노드들의 크기가 균일하게 생성되면 간단한 수학적인 공식만을 이용해 공간의 좌표들의 위치에 존재하는 메타 테이블의 배열 인덱스를 찾을 수 있다.

메타 테이블을 이용한 kd-tree의 탐색과정은 다음과 같다. 기존의 kd-tree의 탐색과정과 마찬가지로 시작은 광선과 장면 전체의 바운딩 박스와의 교차검사로부터 이루어진다. 만약 검사가 실패했을 경우는 광선과 교차하는 삼각형이 존재하지 않는 경우이므로 결과값을 false로 하고 검사를 종료하고, 그렇지 않은 경우에는 [그림6]과 같은 방법을 통해 검사를 계속해서 진행한다.

getFromXYZ	*현재 좌표를 이용해서 메타 테이블의 인덱스를 구하는 함수.*
MT	*메타 테이블의 배열정보.* *MT[k].addr 은 MT내의 인덱스가 k인 배열이 갖고 있는 kd-tree의 리프 노드의 주소 정보.*
get_tmax	*광선과 바운딩 박스를 이용해서 t_max 값을 구하는 함수*

```
while (t_max <= global_t_max)
   t_min = t_max;
   Array_index = getFromXYZ(t_min);
   currentNode = kdTree[MT[Array_index].addr];
   if (intersect_test(currentNode.triangles))
      return true & triangle;
   else
      t_max = get_tmax(ray, currentNode.boundingBox);
```

그림6 : *메타 테이블을 이용한 탐색 과정*

[그림7(a)]를 보면 광선과 장면 전체의 바운딩 박스와의 교차검사를 통해 global_t_min, global_t_max 값을 구할 수 있다. 이 때 구한 global_t_min(t_min) 값을 이용해 [그림7(b)]와 같이 메타 테이블의 인덱스 값을 구하고, 그 인덱스의 배열이 가리키는 주소값을 얻는다. 이 주소값을 이용해 [그림7(b)]에서 kd-tree의 리프 노드에 해당하는 n0에 접근하여 n0에 속하는 삼각형들과 광선의 교차검사를 시행한다. 만약 교차하는 삼각형이 발견되면 그 삼각형 정보를 결과값으로 반환하며 검사를 종료하게 되고 그렇지 않은 경우는 역시 [그림6]의 과정을 통해 검사를 계속 진행한다. [그림7(c)]를 보면 광선과 n0의 바운딩 박스와의 교차검사를 통해 t_max 값을 구할 수 있고 이 값은 메타 테이블에서 t_min 값으로 이용할 수 있다. 이 값을 이용해서 역시 메타 테이블의 인덱

스를 구할 수 있고 kd-tree의 리프 노드인 n3에 접근할 수 있다[그림7(d)]. 이러한 과정은 t_max 값이 [그림7(a)]에서 구한 global_t_max의 값과 같거나 클 때까지

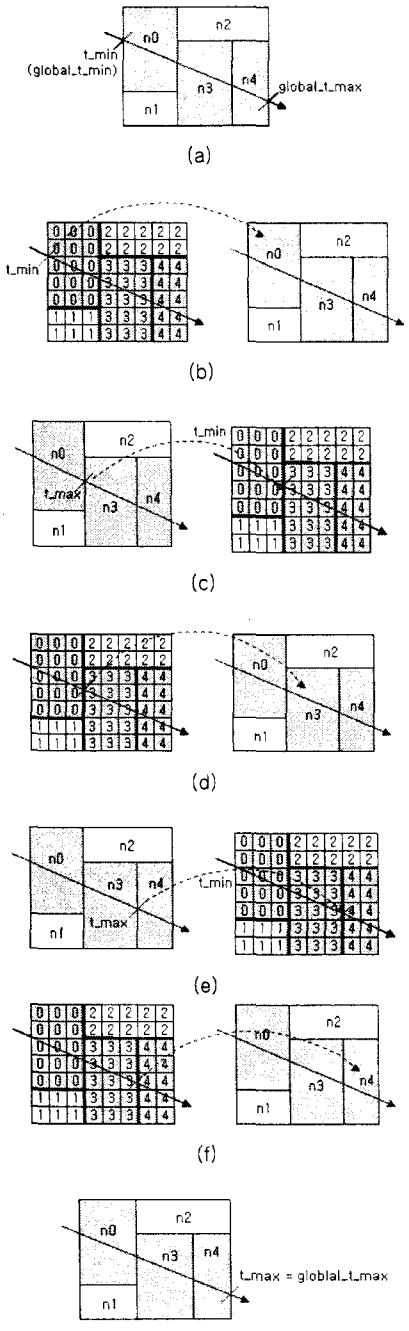

그림 7 : *메타 테이블을 이용한 탐색 과정*

반복되어 [그림7(g)]와 같은 상태가 되면 검사를 종료하게 된다. 검사를 종료할 때까지 광선과 교차하는 삼각형을 하나도 발견하지 못한 경우는 검사 결과를 false로 하고, 그렇지 않은 경우는 광선과 교차하는 삼각형 정보를 반환하면서 도중에 검사를 종료한다.

본 논문이 제안하는 이 탐색과정은 kd-tree에서 리프 노드를 찾기 위해 내부 노드(inner node)들을 탐색하는 과정대신 메타 테이블을 한 번만 검사하면 바로 리프 노드에 도달할 수 있기 때문에 탐색 시간을 줄일 수 있게 된다. kd-tree에서 트리의 깊이가 깊어질수록 그 차이는 많이 나게 되며 이 때, 사용하는 메타 테이블은 읽기 전용으로만 사용되기 때문에 스택을 사용하지 않는다.

4. 실험 결과

기존의 kd-tree의 탐색 속도와 메타 테이블을 이용한 탐색 속도 차이의 비교를 위해 다음과 같은 환경에서 실험을 진행하였다.

- 인텔 펜티엄4 CPU 2.8GHz
- 1GB DDR2 RAM
- 마이크로소프트 윈도우즈 XP 프로페셔널 서비스 팩 2
- Nvidia Geforce 6200 TM(Turbo Cache)

본 실험은 제안하는 알고리즘이 기존의 kd-tree 보다 탐색 속도면에서 우수함을 증명하기 위한 실험이기 때문에 메타 테이블 데이터의 생성 및 kd-tree의 생성 시간은 고려하지 않았다. 그리고 레이트레이싱의 특성상 첫 번째 광선(first ray)이 삼각형과 만나서 생성되는 광선들에 대해서도 재귀적으로 같은 과정을 반복하는 방식이기 때문에 실험의 편의를 위해 첫 번째 광선에 대해서만 실험을 수행했다.

실험에 사용한 3D 모델 데이터는 [그림8]과 같이 2,904개의 정점들과 5,804개의 삼각형들로 이루어진 cow mode과 35,947개의 정점들과 69,451개의 삼각형들로 이루어져 있는 Stanford Bunny를 사용했으며, 이 때 사용된 광선의 개수는 76,800(320x240해상도)개이다.

kd-tree를 생성하기 위해 [5]의 SAH를 이용하였으며, 이 때 kd-Tree의 최대 깊이를 16으로 했다. 이 때 메타 테이블의 노드 하나의 단위 길이는 각각 0.02(총 30,528개의 배열)와 0.002(총 371,124개의 배열)를 이용했으며 이는 모델 데이터를 이루고 있는 삼각형 개수와 모델 전체의 바운딩 박스의 각 축별 길이 정보 및 [5]에서 Uniform grid 방식으로 구현할 때 격자 하나의 단위 길이를 구하는 공식을 참고하여 구한 값이다. 실험에서는 총 4가지의 시점(앞, 위, 좌, 우)에서 모델을 바라볼 때 사용된 노드 탐색 횟수와 걸리는 시간을 측정하였다.

[그림9]에서 보이는 것과 같이 메타 테이블을 이용한 탐색방법은 기존의 kd-tree 탐색방법에 따라 노드 탐색 횟수가 80%~85% 이상 줄어든 것을 알 수가 있다. 실제 이미지 렌더링 시간을 비교해보면 [그림10]과 같이 30%~50% 정도의 시간 단축이 있었음을 확인할 수가 있다. 노드 탐색 횟수 단축에 비해 시간 단축이 크지 않

그림 8 : *실험에 사용된 Cow Model과 Stanford Bunny.*

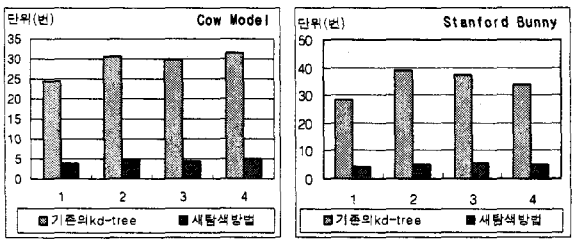

그림 9 : *광선 하나당 노드 탐색 횟수 비교*

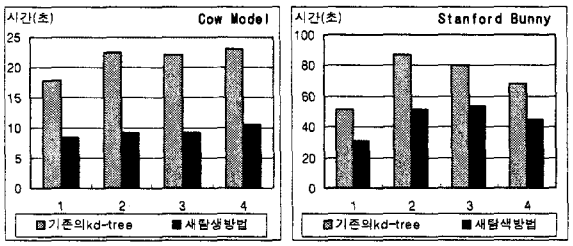

그림 10 : *이미지 렌더링 시간 비교*

은 이유는 광선과 삼각형의 교차검사는 기존의 kd-tree 탐색과 메타 테이블을 이용한 탐색 모두에서 kd-tree의 리프 노드에서 이루어졌기 때문에 그 횟수가 동일하기 때문이며, 노드 방문에 드는 시간적 비용보다는 교차검사에 드는 시간적 비용이 더 크기 때문이다. [5]에서는 대략 교차 검사에 드는 시간적 비용을 노드 방문에 드는 시간적 비용의 80배 정도 차이로 계산하고 있다.

5. 결론 및 향후 계획

본 논문에서는 기존의 CPU에서 레이트레이싱을 구현할 때 범용적인 목적으로 쓰기에 가장 우수하다고 알려져 있는 kd-tree를 이용한 트리 탐색보다 더 성능이 우수한 메타 테이블을 이용한 탐색방법을 제안하였다. 탐색방법의 특성상 kd-tree가 아닌 육면체들의 집합으로 이루어져 있는 어떠한 구조에 대해서도 적용이 가능하며, 장면의 특성에 따라 [그림11]과 같은 특수한 형태의 구조에 대해서도 메타 테이블을 이용할 수 있다.

레이트레이싱은 광선들간의 연산들이 독립적으로 수행되기 때문에 병렬화에 용이하다. 그래서 다양한 병렬 처리 시스템에서 연구가 되어왔고, 요즘은 가격대 성능이 우수한 병렬 처리 프로세서인 GPU를 레이트레이싱의 연산에 이용하는 연구가 이루어지고 있다[6,7]. 실 예로

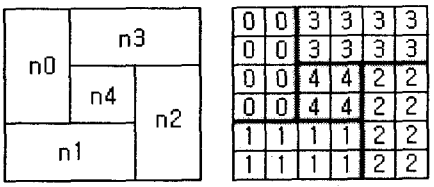

그림 11 : *kd-tree로 표현할 수 없는 구조*

Nvidia 사의 Geforce 7800GT 같은 경우는 24개의 픽셀 파이프라인과 8개의 정점 파이프라인으로 구성되어 있다.

그러나 현재의 GPU의 경우에는 읽기와 쓰기가 둘 다 필요한 스택을 갖고 있지 않기 때문에 CPU기반에서 가장 우수하다고 알려져 있는 kd-tree를 그냥 쓸 수가 없다. 그래서 스택을 사용하지 않는 kd-Restart 알고리즘이나 kd-Backtrack 알고리즘들이 연구되어져 왔다[7]. 그러나 이 알고리즘들은 스택이 없는 GPU에서 레이트레이싱을 실행하기 위해 만들어진 알고리즘들이기 때문에 기존의 스택을 이용하는 kd-tree 탐색 알고리즘보다 성능이 떨어진다는 단점이 있다.

본 논문에서 제안한 메타 테이블을 이용한 알고리즘은 메타 테이블이 읽기 전용으로만 사용되기 때문에 스택이 필요가 없다. 따라서 이 연구는 향후 GPU에서 구현 가능성을 갖고 있으며 그 구현을 목표로 연구를 계속할 계획이다.

6. 참고 문헌

[1] Alan Watt, *3D Computer Graphics*, Addison-Wesley, 2000.

[2] William R.Mark, Donald Fussell, *Real-Time Rendering Systems in 2010*, Technical Report, Department of Computer Sciences, University of Texas at Austin, May. 2005.

[3] Niels Thrane, Lars Ole Simonsen, *A Comparison of Acceleration Structures for GPU Assisted Ray Tracing*, M.S.thesis, Department of Computer Science, University of Aarhus, Aug. 2005.

[4] Havran V., *Heuristic Ray Shooting Algorithms*, Ph.D. thesis, Department of Computer Science and Engineering, Czech Technical University in Pragure, Nov. 2000.

[5] Pharr M. Humphreys G, *Physically Based Rendering*, Morgan Kauffman, 2004.

[6] The Foley and Jeremy Sugerman, "KD-Tree Acceleration Structures for a GPU Raytracer," *Proceedings of the Graphics Hardware*, 2005.

[7] Timothy J.Purcell, Ian Buck, William R.Mark, Pat Hanahan, "Ray Tracing on Programmable Graphics Hardware," *ACM Transactions on Graphics*, 2002.