CUDA를 이용한 효과적인 GPU 광선추적 가속 알고리즘

CUDA를 이용한 효과적인 GPU 광선추적 가속 알고리즘

An efficient acceleration algorithm of GPU ray tracing using CUDA

지중현, Joonghyun Ji*, 윤동호 Dongho Yun** 고광희, Kwanghee Ko***

요약 본 논문은 CUDA를 이용하여 GPU에서의 최적화된 kd-tree 탐색구조 환경과 광선/삼각형의 교차검사 알고리즘을 통한 실시간의 광선추적 시스템을 제안한다. 기존의 GPU 기반 kd-tree 탐색 알고리즘은 대부분 스택이 없는 GPU 하드웨어의 특성상 임의의 단말노드에서 기하요소의 교차검사가 실패할 경우 상위노드로 상향식 탐색을 진행하기 때문에 노드에 대한 중복 방문이 반드시 필요하거나 혹은 불필요한 메모리의 적재가 필요하기 때문에 큰 장면에 대한 광선추적은 어렵게 된다. 본 논문에서 제안하는 알고리즘은 CPU 방식의 kd-tree 탐색과 비슷하게 동작하도록 stack을 CUDA 프레임워크를 이용하여 GPU의 지역메모리로 구현하였기 때문에 기존의 방법 등에서의 문제점을 해결하였다. 또한 탐색구조를 통해서 찾은 말단 삼각형들의 처리를 위해서 최신의 CPU 기반의 교차검사 알고리즘인 Plücker 좌표계 검사를 CUDA로 구현하여 병렬 가속시켰다. Plücker 검사는 기존의 무게중심 좌표 대신에 광선과 삼각형 edge의 관계를 이용하는 간단한 연산만을 이용하는 장점이 있다. 전체 시스템은 단일 커널로 구성되어 있으며 병렬처리를 위한 복잡한 동기화나 광선패킷의 도입 없이 간단하게 구현되었다. 결과적으로 본 논문의 실험은 기존 알고리즘 대비 제안하는 알고리즘이 약 2배의 성능 향상이 있음을 보여 준다.

Abstract This paper proposes an real time ray tracing system using optimized kd-tree traversal environment and ray/triangle intersection algorithm. The previous kd-tree traversal algorithms search for the upper nodes in a bottom-up manner. In a such way we need to revisit the already visited parent node or use redundant memory after failing to find the intersected primitives in the leaf node. Thus ray tracing for relatively complex scenes become more difficult. The new algorithm contains stacks implemented on GPU's local memory on CUDA framework, thus elegantly eliminate the problems of previous algorithms. After traversing the node we perform the latest CPU-based ray/triangle intersection algorithm 'Plücker coordinate test', which is further accelerated in massively parallel thanks to CUDA. Plücker test can drastically reduce the computational costs since it does not use barycentric coordinates but only simple test using the relations between a ray and the triangle edges. The entire system is consist of a single ray kernel simply and implemented without introduction of complicated synchronization or ray packets. Consequently our experiment shows the new algorithm can is roughly twice as faster as the previous.

핵심어: Ray Tracing, GPGPU, CUDA, kd-tree, intersection test

본 연구는 2008년 지식경제부 및 정보통신연구진흥원의 대학 IT연구센터 육성·지원사업의 연구결과로 수행되었음
(IITA-2008-C1090-0804-0002)

*주저자 : 광주과학기술원 기전공학과 석사과정 e-mail: bluesky81@gist.ac.kr
**공동저자 : 광주과학기술원 기전공학과 석박통합과정 e-mail: lipo123@gist.ac.kr
***교신저자 : 광주과학기술원 기전공학과 조교수 e-mail: khko@gist.ac.kr

1. 서론

컴퓨터 영화의 특수효과나 의학적인 용도에 국한되었던 3D Graphics가 게임이나 운영체제의 사용자 인터페이스에 필수적으로 사용되기 시작하면서 최근에는 좀 더 현실적이고 실사적인 이미지의 생성을 요구받게 되었고 그에 따라서 기존의 지역적 조명 (local illumination)을 벗어나 장면의 모

든 빛의 속성을 고려하는 전역적 조명 (global illumination)에 대한 관심이 증대되고 있다. 전역적 조명의 알고리즘 중에 하나인 광선추적법 (ray tracing)은 비교적 단순하여 구현하기 쉽고 그림자 및 반사와 굴절효과를 복잡한 절차없이 물리적으로 정확하게 묘사한다. 그러나 광선추적법은 화면의 각각의 픽셀에서 개별적인 광선을 쏘아 추적하는 방식이므로 연산의 복잡도가 기본적으로 매우 높기 때문에 고품질의 화면이 필요한 영화나 애니메이션의 제작에 치중하고 있으며 병렬 PC 클러스터같은 특수한 장치를 이용해야 효율적으로 고품질의 실시간 광선추적이 가능한 실정이었다.

최근 GPGPU(General Purpose computing on GPU)의 등장으로 GPU를 그래픽 분야가 아닌 범용 연산등에 활용할 수 있게 되면서 거대행렬 계산, 물리 모델링 및 시뮬레이션 등의 가속화가 급격히 진행되고 있다. 이로 인해 여러가지 GPGPU 플랫폼이나 툴킷들이 제작되었고 그 중 가장 활발히 연구되고 있는 것이 NVIDIA 사의 CUDA (Compute Unified Device Architecture) 이다.

CUDA는 자사의 G80이상의 GPU를 위한 병렬 연산처리용 소프트웨어 개발 도구로서 사용되며 커널 (kernel)에 의해 실행될 병렬 데이터들을 쓰레드 (thread), 블록 (block)및 그리드 (grid)의 단계로 구분하여 동시적으로 처리한다. 이 밖에도 효율적인 데이터 처리를 위한 공유 메모리및 여러 기술들을 제공함으로서 기존의 GPGPU의 처리 방법보다 훨씬 개선된 연산 효율을 보여주고 있다.

광선추적은 주로 삼각형의 메쉬에 대한 개별 광선의 교차 테스트를 통해 물체를 형성하는 기하요소의 색을 결정하게 되는데 일반적으로 수 천개 이상의 삼각형으로 이루어진 물체에 대해서 교차검사를 수행하는 것은 연산 비용이 굉장히 크므로 공간 가속구조를 이용하게 된다. 공간 가속구조들은 물체들의 하위 기하요소를 최적화된 방법으로 계층화하거나 분해하여 불필요한 연산을 제거하게 된다.

일반적으로 kd-tree같은 공간분할 기법이 가장 빠른 성능을 보여주고 있지만 근본적으로 기존 그래픽스 하드웨어에서와 GPGPU 플랫폼에서는 스택(stack)의 지원이 미약하고 병렬처리를 위한 SIMD(Single Instruction Multiple Data) 대응 및 프로그래밍이 어렵기 때문에 GPU를 통해 비교적 거대한 장면에 대해서 kd-tree를 이용하여 실시간으로 광선을 추적하는 것은 많은 한계가 있다. 본 논문은 광선추적의 핵심 모듈인 공간 가속구조와 광선/삼각형 교차검사의 가속을 위하여 CUDA를 이용한 알고리즘을 구현하였다.

2. 관련 연구

[1]은 기존의 시스템에서 성능이 가장 뛰어난 kd-tree를 GPU의 텍스쳐 메모리에 생성및 구축하였다. 하지만 텍스쳐 메모리는 읽기전용이기 때문에 stack의 push작업을 위한 쓰기 연산은 CPU를 사용하기 때문에 빈번히 발생하는 I/O 오버헤드로 인한 효율의 저하가 큰 문제점이 된다. 이를 해결하기 위해 [2]는 stack이 필요없는 두 개의 알고리즘(backtrack, restart)을 제안하였다. 트리의 단말노드에서 광선과의 교차가 발견되지 않을때에 약간의 차이는 있지만 위 두 알고리즘은 이미 방문한 노드들을 재방문해야하는 단점이 있기 때문에, [3]는 각 내부노드들의 재방문을 효과적으로 제거한 빠른 탐색법을 제안하였다. 각 단말노드에 'rope'라고 불리는 6개의 인접노드에 대한 포인터를 별도로 저장한 뒤, 임의의 단말 노드에 대해 교차된 삼각형이 발견되지 않았을 경우 다음의 인접한 노드 중 교차가 발생된 노드부터 재탐색하도록 하였다. 하지만 이 방법은 메모리공간의 효율성이 떨어지는 단점이 있다.

공간 가속구조를 통해 빠르게 노드탐색이 완료되었다고 하더라도 각 노드가 참조하는 삼각형의 개수만큼 광선/삼각형 교차검사를 실시하여야 한다. 광선/삼각형 교차검사는 광선추적에서 90% 정도의 연산에 해당하기 때문에 굉장히 중요하다. 종래의 삼각형 무게중심 좌표계를 이용하는 알고리즘은 삼각형의 법선을 실행 중에 계산해서 삼각형을 둘러싸는 평면과 광선의 거리를 구해야만 하는 단점이 있었다. [4]는 광선의 원점을 삼각형의 무게중심 좌표축에 대하여 변환하는 projection 메소드를 사용하여 평면의 방정식의 계산 없이 메모리 사용량과 연산량을 모두 줄일 수 있었다. [5]는 삼각형의 꼭지점과 광선 충돌지점을 삼각형을 둘러싸는 평면에 직교하지 않은 평면으로 투사하면 충돌지점의 무게중심 좌표가 바뀌지 않는다는 사실을 이용하여 2D 공간에서 간략화된 연산들이 수행되도록 하였다. 하지만 복잡한 장면에 대하여 실시간 광선처리를 위해서는 연산들이 더욱 간단해져야 하며 GPU에서 수행되는 병렬 광선/삼각형 교차검사의 경우에는 중복 처리 및 병목현상을 제거하기 위한 세심한 스레드 처리가 반드시 필요하다.

3. 시스템 개관

본 논문에서 제안하는 광선추적 알고리즘을 포함한 전체적인 모듈의 구조는 <그림 1>과 같이 구현되어 진다. host(CPU)에서는 장면의 계층적 구성과 삼각형 메쉬에 대한 사전 연산을 수행하고 CUDA 텍스쳐 버퍼로 필요한 모든 정보를 device(GPU)에 넘기는 것으로 광선추적이 시작된다. GPU에서는 모든 광원에 대하여 스택 기반의 탐색을 통해 비교적 복잡한 장면을 지원하도록 하였으며 완료된 노드가 참조하는 삼각형에 대하여 효과적인 교차검사를 실시하였다. 1차 광선의 탐색을 시작하고 혹은 그림자를 생성하기 위해 부가적인 계산을 하여 픽셀당 쉐이딩 결과를 획득하는데 이때 반사나 굴절의 효과를 위해 2차 광선이 필요할 수도 있게 된다. 완료된 이미지는 다시 host에 OpenGL 픽셀 버퍼

오브젝트(PBO)로 전달이 되어 최종 렌더링이 마무리되게 된다.

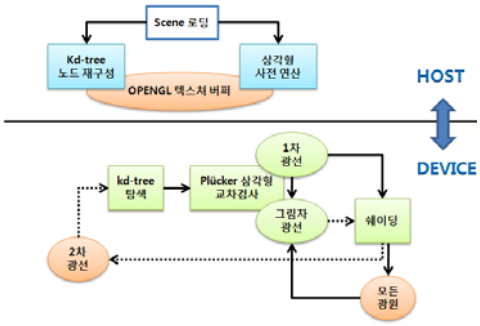


그림 1. 광선추적 시스템 개관

4. kd-tree 노드 탐색방법

이번 장에서는 제안된 스택 기반의 트리 탐색에 대한 구체적인 설명을 위해 4.1절에서 노드의 배치를 설명하고 4.2절에서는 이를 기반으로 노드의 탐색방법을 설명한다.

4.1노드 구성

트리의 순회 단계에 비해 메모리는 한정적이기 때문에 효율적인 캐시효과의 증대를 위한 노드의 설계가 필요하다.

각각의 노드는 [4]와 비슷하게 8 byte의 메모리 소비량을 가지지는 구조체로 구현하였으며 UNION을 쓰지 않고 마스크 비트를 통한 연산으로 내부노드와 단말노드를 구분하게 된다. 내부 노드에서는 자신이 내부노드인지 단말노드인지 확인하도록 1비트를 할당하고 오로지 한 쪽의 방향으로 노드들이 저장되도록 하여 <그림 2> 두 개의 자식 노드들에 대해서 포인터 한 개만을 사용했다. 나머지 자식에 대한 오프셋으로 28비트를 쓰고 분할 면의 축(x, y, z)을 위한 2비트, 분할면의 좌표를 4byte 할당한다. 단말노드의 경우는 전체의 삼각형을 포함하는 인덱스에 대해 4byte를 할당하고 나머지를 각 단말노드당 삼각형의 개수를 저장하도록 하였다. 그리고 탐색 이전에 구축된 트리에 대해서 CUDA에서 조금 더 유용한 array 방식의 접근을 위해서 임의의 하위트리들을 BFS(Breadth First Search) 방식으로 메모리에 연속되게 넣어 트리 전체를 다시 할당했다. 이 방법으로 첫 번째 자식 노드를 가리키는 것으로 나머지 자식노드까지 모두 접근할 수 있으므로 캐시 효율을 증가시킬 수 있다.

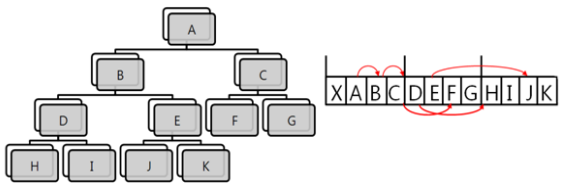


그림 2. 제안하는 kd-tree 노드의 메모리 레이아웃

4.2 스택기반의 탐색 방법

CUDA 플랫폼에서 제공하는 NVIDA의 하드웨어 스펙에는 global, shared, local, register 메모리가 있다. 이중 shared 메모리가 가장 빠르고 한 블록안에 있는 모든 쓰레드들이 접근 가능하기 때문에 유용하게 쓰인다. 그러나 한 블록당 16KB 밖에 되지않는 shared 메모리에 스택을 만들게 되면 가용범위에 문제가 발생하고, 다른 스레드들이 수시로 공유를 요청하고 갱신해버리는 경우가 생기므로 동기화가 필요하며 이 때 너무 많은 동기화 작업은 탐색의 효율성을 절감시키게 된다. 실험을 통해 shared 메모리에 스택을 만들었을 때 G80 하드웨어의 경우 모든 쓰레드에게 할당된 메모리는 오직 42바이트 뿐이기 때문에 활용할 수 있는 트리의 깊이가 10 밖에 되지 않았다.

```
STRUCT __stack ⇐ __data
        STRUCT __data ⇐ Node, Near, Far
    __data[MAXDEPTH]
    __stack.init()
    integer idx
    integer push();;return ++idx
    integer pop();;return --idx
-------------------------------------------------------
loop
    if (inner node) then
        loop
            bit ← getOffset(Node), dim ← dim & 3
            offset ← bit & CHILDBIT >> 3
            axis ← getAxis(Node)
            ray_pos ← getRay(dim, ray_pos)
            ray_dir ← getRay(dim, ray_dir)
            dist ← (axis − ray_pos) / ray_dir
            if ray_dir ≥ 0 then offset+sign ^ 0
            endif
            idx←offset+sign ^ 1
            if dist < near then offset+sign ^ 0 endif
            else if dist ≤ far then
                __stack__data.Node ← offset+sign ^ 0
                __stack__data.Near ← dist
                __stack__data.Far ← far
                far←dist
                __stack__data.push()
            endif
        end loop
    else
        loop
            perform intersection test
        endloop
        if __stack__data.pop() ≠ empty and ray is hit!
            __stack__data.Near ← near
            __stack__data.Far ← far
        endif
endloop
```

그림 3. 탐색 알고리즘

따라서 다른 쓰레드에 의해 자주 스택이 갱신되어야 할 필요가 없고 쓰레드 내의 연산이 많은 광선추적의 경우는 다른 메모리보다 쓰레드 안에서만 쓰이는 local 메모리가 효율적이다.

각 노드의 정보에 비트 인코딩된 오프셋과 분할 축을 참조하여 해독된 노드의 정보는 광선의 위치와 방향과 함께 사용되어 하향식의 kd-tree 탐색을 진행하게 된다. 광선 세그먼트 [near, far]를 장면의 전체 바운딩 박스에 클리핑 시킨 다음 업데이트를 하면서 왼쪽, 오른쪽 중 어떤 자식노드에 광선이 교차되었는지를 판단하기 위해 탐색구조 안에서 탐색된 노드의 각각에 분할면에 대한 거리(dist)를 계산하여 그 거리를 현재의 광선 세그먼트와 비교하게 된다. 광선 세그먼트가 near보다 작거나 far보다 크게 되면 현재 노드들의 하위 트리는 검색할 필요가 없지만 그렇지 않다면 각 노드를 순서대로 탐색하여야 한다. 이때 현재 탐색되지 않는 노드에 대한 정보를 push하게 되는데 본 논문에서는 CUDA를 통해 지역메모리에 배열 형식으로 트리의 깊이에 기반한 고정된 사이즈의 읽기/쓰기 가능한 스택을 구현하고 사용하였다. 본 알고리즘은 CPU내에서 수행되는 탐색방법과 매우 비슷하다. 한 개의 광선을 담당하는 쓰레드마다 스택을 모두 사용하였고 각 압축된 각 노드들은 1D의 텍스쳐 캐시를 통해 빠르게 읽어들일 수 있다.한 쪽 노드에서 말단 노드에 진입 후 교차되는 삼각형이 없을 경우 탐색되지 않은 노드에 대한 부가적인 상향 재방문없이 pop하여 탐색을 계속 진행 할 수 있게 하였다. <그림 3>는 제안된 탐색 알고리즘의 간략한 모습을 보여주고 있다.

5. 삼각형 교차검사 가속 방법

본 논문은 최근 CPU 기반의 광선/삼각형 교차검사 알고리즘인 Plücker 좌표계 검사 알고리즘[8]을 이용하여 구현하였다.

5.1 Plücker 좌표계 기반 광선/삼각형 교차검사

Plücker 좌표계는 6개의 수를 이용하여 3차원 공간에 있는 직선들을 기술하는 대체적인 방법이다. 이 숫자들을 가지고 6차원의 내적의 순열을 통해 두 직선이 교차하는지 알 수 있다(내적이 0일 때). 또한 한 직선이 다른 직선에 대해 어느 면으로 통과하는지 확인하는 방법으로 광선이 삼각형을 통과하는지 아닌지도 확일 할 수 있다. <그림 4>

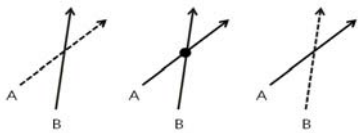


그림 4. 두 개의 직선의 3가지 가능성

edge와 광선을 정의하는 6차원의 벡터가 다음과 같다면

$$e_0 = \{v - v_0, v \times v_0\} \tag{1}$$

$$e_1 = \{v_1 - v, v_1 \times v\} \tag{2}$$

$$e_2 = \{v_0 - v_1, v_0 \times v_1\} \tag{3}$$

$$R = \{d \times o, d\} \tag{4}$$

6차원의 벡터로부터 다음 3개의 내적값이 같은 부호를 따르는지를 판별하면 광선/삼각형의 교차판정이 가능하게 된다.

$$t_0 = (e_0, R),\ t_1 = (e_1, R), t_2 = (e_2, R). \tag{5}$$

내적 연산은 곱셈과 덧셈만을 이용하기 때문에 이 방법은 연산이 굉장히 간편해지는 잇점이 있다. 그러나 약간의 사전연산을 통해 삼각형의 edge (e_0, e_1, e_2)들을 적절한 자료구조에 삽입함으로서 교차검사를 더 빠르게 처리할 수 있게 된다. 사전연산되는 삼각형의 정보에는 9개의 float형 데이터 및 3개의 integer형 데이터가 필요하다.

삼각형의 법선 (n_0, n_1, n_2)의 경우 이 알고리즘에서는 특정한 길이 값을 필요로 하지 않으므로 원소 3개를 저장할 필요 없이 2개의 값 (n_u, n_v)만으로 표현이 가능해진다. 삼각형의 버텍스의 경우는 무게중점 좌표인 u와 v의 인덱스를 이용한 v_u, v_v, 그리고 버텍스 v와 위의 노말값을 내적한 np를 구하여 저장한다. 삼각형의 edge의 경우 앞서 버텍스의 경우와 마찬가지로 u와 v의 두 개의 인덱스를 이용하여 e_0, e_1에 적용하여 4개의 float $(e_{u0}, e_{v0}, e_{u1}, e_{v1})$만으로 나타낼 수 있게 된다. 추가적으로 법선의 원소 n_u와 n_v가 0이 되는 축에 정렬된 삼각형에 대한 플래그를 표시하도록 하여 적응형의 가벼운 간편한 연산을 수행하도록 한다.

실제로 교차검사를 수행시에는 다음과 같은 임시 변수들을 이용해서 효율적인 연산의 결과를 재사용한다.

$$\det = d_u \times n_u + d_v \times n_v + d_w \tag{6}$$

$$\det t = np - (o_u \times n_u + o_v \times n_v + o_w) \tag{7}$$

$$D_u = d_u \times \det t - (v_u - o_u) \times \det \tag{8}$$

$$D_v = d_v \times \det t - (v_v - o_v) \times \det \tag{9}$$

$$\det_u = (e_{v1} D_u \times e_{u1} D_v) \tag{10}$$

$$\det_v = (e_{u0} D_v \times e_{v0} D_u) \tag{11}$$

임시 값들을 계산한 후에 최종적으로 $\det_u$, $\det_v$, 그리고 $\det - \det_u - \det_v$이 모두 같은 부호를 취한다면 광선이 삼각형안에 교차한다는 의미이므로 교차된 위치를 구하기 위해 다음과 같은 식을 이용한다.

$$\det_{inv} = 1/\det \tag{12}$$

$$t = \det t \times \det_{inv} \tag{13}$$

$$u_{tb} = \det_u \times \det_{inv} \tag{14}$$

$$v_{tb} = \det_v \times \det_{inv} \tag{15}$$

최종적으로 Plücker 검사를 통해 도출되는 것은 삼각형의 무게중심 좌표값 (u_{tb}, v_{tb})과 교차거리인 t로서 차후 교차된 삼각형의 쉐이딩을 수행할 수 있다.

5.2 CUDA를 이용한 병렬 교차검사

기본적으로 Plücker 교차검사를 하는 것은 하드웨어로부터 지원되지 않는 복잡한 연산에 매우 효율적이며 특히 단정도(single precision)의 부동소수점 처리만으로도 Plücker 좌표계를 저장하고 내적연산을 하기 충분하기 때문에 GPU에서 구현하기 적합하다. 일반적으로 GPU의 메모리가 메인메모리의 크기보다 작은 문제점이 있지만 이미 kd-tree 탐색 구조를 생성할 때 간략화된 구조를 통해 여분의 메모리 대역폭을 확보하였기 때문에 각 삼각형을 위한 12개의 데이터에 대한 자료구조를 그대로 업로드하여 사용할 수 있다.

CPU에서 사전 연산된 원소들은 캐시의 잇점을 볼 수 있는 1D의 CUDA 텍스처로 저장된 후 커널함수에서 트리 탐색이 완료된 후 반복적으로 호출되어 광선 1개와 삼각형 1개마다 처리된다. 1차 광선의 경우는 Plücker 검사의 2단계를 모두 진행하여 교차점까지 찾는 과정을 수행하였지만 그림자 광선에 경우에는 쉐이딩이 목적이 아니므로 무게중심 좌표를 돌려줄 필요가 없으므로 간단히 hit에 대한 정보만 돌려주는 것으로 최적화를 수행하였다.

기존의 GPGPU 혹은 CPU 프로그래밍에서는 병렬처리 효과를 위해 광선 패킷을 이용하여 다수의 광선을 동시에 추적하는 SIMD mapping을 명시적으로 지정하기 때문에 구현이 어렵고 광선의 coherency가 감소할수록 성능의 향상은 줄어드는 문제가 있다. 반면에 단일 광선을 사용하는 CUDA의 구현 방법을 이용하게 되면 성능 프로파일링을 통해 스레드와 블록의 적절한 수를 산출해내고 명시적인 광선패킷 없이도 SIMD 프로그래밍을 구축할 수 있기 때문에 좀 더 최적화가 용이하게 된다. 이렇게 구축된 스레드와 블록의 설정을 고려하여 광선을 추적하게 되면 다음과 같은 픽셀의 x, y쌍의 인덱스로 광선과 물체 안에 있는 삼각형을 대응할 수 있게 된다.

```
int bw = blockDim.x;
int bh = blockDim.y;
int x = blockIdx.x*bw + threadIdx.x;
int y = blockIdx.y*bh + threadIdx.y;
```

bockDim, threadIdx 그리고 blockIdx는 CUDA의 내부 변수로서 블록의 모양 및 그것들의 인덱스를 나타내며 각 인덱스의 값에 대응하는 스레드나 블록의 구별을 위한 것으로서 x,y는 장면의 모든 위치에 대해 광선을 대응시키는 과정이라고 할 수 있다.

연산이 완료하게 되면 각 스레드는 다시 그 결과를 커널의 개수에 따라 장치 매트릭스 변수에 있는 자신의 행과 열에 인덱스에 저장하고 커널이 종료되는데 자신의 결과를 다른 커널이 이용하도록 하거나 혹은 호스트(CPU)로 건네주게 된다. 본 논문에서는 광선추적 시스템이 커널 함수 1개로 되어있기 때문에 직접적으로 호스트로 전해주게 된다.

GPU에서는 각각의 스레드가 광선과 대응하여 독립적으로 교차테스트를 실시하므로 파이프라인에서 병목현상이 일어나지 않도록 하는 것이 중요한데 이를 위해서 추가적으로 분기를 발생시키지 않는 교차테스트를 구현하였다. 만약 잘못된 분기를 통해서 교차테스트가 수행이 된다면 실행의 길이에 따라서 경우에 따라서는 심각한 성능저하를 일으킬 수 있기 때문이다. 분기를 발생시키지 않도록 하기 위해서 교차가 발생했을 때의 연산과 교차가 발생하지 않았을 때의 연산에 대한 양쪽의 경우를 추가적인 비트 마스크를 사용하여 논리 비트연산으로 제어하여 잘못된 분기를 막아 커널의 coherency를 유지하였다.

5. 실험 결과

광선 추적의 성능 검사에 주로 쓰이는 장면을 포함한 몇 가지 장면들에 대한 실험 결과 본 논문에서 제안한 방법이 효과적으로 수행됨을 확인 하였다. 본 실험은 Intel Quadcore 3.2Ghz, 3.25GB의 메인메모리와 NVIDIA Gefordce 8800GTX를 사용한 PC에서 이루어졌다. 성능 비교를 위해 기존의 스택이 없는 kd-tree 탐색 알고리즘[3]과 projection 메소드 기반[4]의 교차검사 알고리즘을 직접 구현하여 동일한 조건에서 실험하였다.

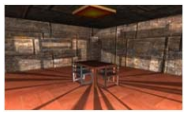
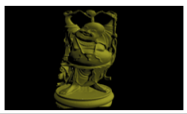
제안하는 탐색 방법은 생성 및 구축에 관한 문제는 포함하지 않으므로 본 실험에서는 생성시간에 대한 평가는 실험하지 않았고 이미 구축된 트리에 대한 성능 검사를 실시하

였다. 테스트는 1차 광선에 대해 phong 모델의 쉐이딩을 사용하였다.

5.1 테스트 결과

<표 1>과 같이 모든 장면에 대하여 제안된 스택기반의 kd-tree가 그래픽스 하드웨어에 제한된 메모리에 적합한 사용량을 보여주고 있다. 노드들을 압축하지 않는 경우에도 3배 이상의 메모리 절감 효과를 보여주고 있기 때문에 좀 더 복잡한 장면의 로딩을 위해서 본 알고리즘이 좀 더 바람직하다고 볼 수 있다.

표 1. CUDA kd-tree 노드에 대한 텍스쳐 크기 비교

Scene (삼각형수)	스택이 없는 구조[5]	제안된 스택기반 구조
Shirley 6 (804) 	268.5KB	13KB (40KB)
Fairy Forest (174K) 	48.03MB	3.17MB (9.4MB)
Happy Buddha (1.08M) 	74.8MB	6.94MB (20.8MB)

<표 2>는 최종적인 시스템의 광선추적 수행능력을 각각 CUDA로 구현하여 결과는 10번의 반복을 통한 평균을 제시하였다. 제안된 시스템은 기존 스택이 없는 kd-tree를 이용한 시스템에 비해서 노드 재방문이나 중복된 바운딩 박스의 교차검사를 하지 않는 단순한 구조로 되어있기 때문에 여분의 자원을 더 많은 CUDA 프레임워크 상에서 쓰레드 동시 수행이나 거대 장면의 렌더링을 지원하기 위해 배분할 수 있으므로 가동이 가능한 멀티 프로세서들이 증가됨에 따라서 성능효율 또한 약 두 배 가까이 증대되었다. 또한 기존의 교차검사에 비하여 Plücker 기반의 교차검사를 수행하였을 때 10-20% 이상의 추가적인 성능의 향상이 있었다.

표 2. [5]의 기존의 가속 알고리즘과 제안 알고리즘의 수행속도 비교 (fps)

Scene (삼각형수)	스택이 없는 구조[5]		제안된 스택기반 구조	
	Projection 메소드	Plücker 좌표계 검사	Projection 메소드	Plücker 좌표계 검사
Shirley 6 (804)	36.9	41.1	66.3	74.1
Fairy Forest (174K)	10.2	15.9	20.3	24.3
Happy Buddha (1.08M)	18.6	23.7	34.2	40.4

6. 결론

본 논문에서는 CUDA를 이용해 지역 메모리에 스택을 구현하였고 그에 맞는 자료구조 및 가속화 방법을 제안하고 있다. 본 논문의 알고리즘은 별도의 coherency를 이용하는 광선패킷이나 특정한 하드웨어에 대한 자료구조를 요구하는 가속화 방법을 쓰지 않고 병렬 처리시에 발생하는 스레드간에 발생하는 빈번한 동기화를 필요로하지 않는다. 또한 다른 노드들을 참조하기 위한 별도의 저장공간이 필요치 않으므로 좀 더 복잡한 장면을 로딩하는 데 수월하게 되며 남는 자원은 GPU에서 좀 더 효율적으로 사용될 수 있게 된다. 또한 광선추적의 대부분의 시간이 소요되는 교차검사의 향상을 위해 최신의 Plücker 검사방법을 GPU에서 최적화하여 사용하여 가속화시킴으로서 기존 알고리즘 조합(스택이 없는 kd-tree + projection 메소드)에 비하여 약 두 배 이상의 성능 향상을 도출하였다.

그러나 동적으로 물체가 움직이거나 변형되는 장면에서는 본 논문에서 사용된 사전연산과 트리의 빌드과정이 탐색의 성능에 치중된 관계로 비교적 느리기 때문에 트리의 업데이트의 효율성의 문제가 발생하였다. 따라서 탐색과 갱신이 모두 가벼운 Surface Area Heuristic 기반의 장면 구축 알고리즘 개발 및 GPU 자체에서 kd-tree를 실시간으로 빌드에 대한 연구를 수행하고 있다.

참고문헌

[1] M. Ernst, C. Vogelgsang, and G. Greiner, "Stack implementation on programmable graphics hardware," Vision Modeling and Visualization, pp. 255-262, 2004.

[2] T. Foley, and J. Sugerman, "KD-Tree Acceleration Structures for a GPU Raytracer," Proceedings of Graphics Hardware, pp. 15-22, 2005.

[3] 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.

[4] I. Wald, "Realtime Ray Tracing and Interactive Global Illumination," PhD thesis, Saarland University, 2004.

[5] T. Möller, B. Trumbore, "Fast minimum storage ray-triangle intersection", Journal of Graphics Tools, Vol.2 No.1, pp. 21-28, 1997.

[6] M. Shevtsov, A. Soupikov and A. Kapustin, "Ray-triangle Intersection Algorithm for Modern CPU Architectures", GraphiCon', 2007.