論 文
48A-5-21

병렬처리를 위한 고속 Ray Tracing 프로세서의 설계

Implementation of Ray Tracing Processor for the Parallel Processing

崔奎烈* · 鄭德鎭**
(Kyu-Yeoul Choi · Duck-Jin Chung)

Abstract - The synthesis of the 3D images is the most important part of the virtual reality. The ray tracing is the best method for reality in the 3D graphics. But the ray tracing requires long computation time for the synthesis of the 3D images. So, we implement the ray tracing with software and hardware. Specially we design the hit-test unit with FPGA tool for the ray tracing. Hit-test unit is a very important part of ray tracing to improve the speed. In this paper, we proposed a new hit-test algorithm and apply the parallel architecture for hit-test unit to improve the speed. We optimized the arithmetic unit because the critical path of hit-test unit is in the multiplication part. We used the booth algorithm and the baugh-wooley algorithm to reduce the partial product and adapted the CSA and CLA to improve the efficiency of the partial product addition. Our new Ray tracing processor can produce the image about 512ms/F and can be adapted to real-time application with only 10 parallel processors.

Key Words :ray tracing, hit-test unit, ray tracing processor, parallel processing

1. 서 론

미래의 컴퓨터 산업은 가상현실(Virtual Reality)을 기반으로 하는 3차원 영상을 중심으로 발전하게 되며, 이는 보편적으로 사용되는 2차원 평면 그래픽을 이용하여 생성된 영상으로는 가상현실을 구현하기에 역부족이다. 가상현실에서 가장 중요한 것은 인간으로 하여금 가상으로 만들어진 상황을 현실로 인지하도록 하는 것이기 때문에 사실 감이 떨어지는 2차원 그래픽으로는 가상현실을 구현함에 있어 무리가 따르게 된다. 하지만, 3차원 그래픽을 이용할 경우에는 2차원 그래픽에 비해 보다 현실감 있는 가상현실 시스템을 구축할 수 있다는 장점을 가지게 된다. 3차원 그래픽을 구현하는 기법으로는 Ray Tracing, Z-buffering, Ray Casting, Radiosity 등 다양한 기법이 사용되며, 이러한 3차원 그래픽 기법 중 가장 현실감을 갖춘 기법은 Ray Tracing이다. Ray Tracing은 사람의 눈의 관점에서 빛의 방향을 추적하여 화면을 구성하는 기법이다. 사람의 눈으로 들어오는 빛의 세기와 색상을 계산하여 화면을 구성하기 때문에 물체의 밝기, 색상 등이 현실에 가깝게 합성된다는 이점을 가지고 있다. 또한 빛의 방향을 추적하는 기법을 사용하므로 자연스럽게 원근 감이 표현된다. 즉, 인간이 시각을 통해서 보는 영상 그대로를 화면에 표현하는 것이다. 따라서, 다른 기법에 비해 가장 현실에 가까운 화상을 구현할 수 있게 된다. 하지만, Ray Tracing의 가장 큰 문제점은 다른 기법에 비해 화면을 생성시키는데 필요한 시간이 매우 길다는 것이다. 이 문제를 해결하기 위하여 본 연구에서는 Ray Tracing의 원칩화를 위한 전 단계로서 연산의 빈도가 가장 높은 부분인 Hit Test Routine을 고속의 병렬 곱셈기, 덧셈기 등을 사용하여 고속 처리가 가능하도록 설계하였다. 설계한 Unit은 FPGA를 이용하여 제작되었으며, Test를 위하여 Hit Test Unit을 제외한 Ray Tracing의 알고리즘을 소프트웨어로 처리하여 간단한 영상을 구현하였다. 설계한 Hit Test Unit은 병렬처리구조로의 적용이 가능하도록 설계되었다.

* 準 會 員 : 仁荷大學 電子材料工學科 碩士
** 正 會 員 : 仁荷大學 電子材料工學科 敎授 · 工博
接受日字 : 1998년 12월 23일
最終完了 : 1999년 4월 1일

2. Ray Tracing

2.1 Ray Tracing의 특징

3차원 그래픽[1][2]의 최종 목적은 영상을 사진과 같이 사실감 있게 합성해내는 것이다.(Photo Realism) 하지만, 자연은 수학적으로 표현이 불가능하므로 컴퓨터 그래픽을 이용하여 사진과 똑같은 이미지를 구현한다는 것은 거의 불가능하다. 따라서 3차원 그래픽에서 추구하는 것은 실제 사진이 아니라, 사진에 가까운 영상을 구현하는 것이라 할 수 있다. 이러한 조건에 가장 적합한 알고리즘이 바로 Ray Tracing이다. Ray Tracing[3]은 화소에 해당하는 빛 하나 하나를 모두 추적하기 때문에 화면이 표현할 수 있는 해상도 내에서 상당한 수준의 사실적 영상을 합성할 수 있다. 하지만, 각 화소에 해당하는 모든 광선을 추적해야 하므로 연산 량이 증가하는 단점을 안게 된다.

Ray Tracing은 빛의 진행방향을 추적하여 영상을 합성하

는 방법이다. Ray Tracing은 영상 합성에 필요한 빛을 추적하므로 카메라를 통해 보게 되는 영상과 가장 유사한 영상의 합성이 가능하다.

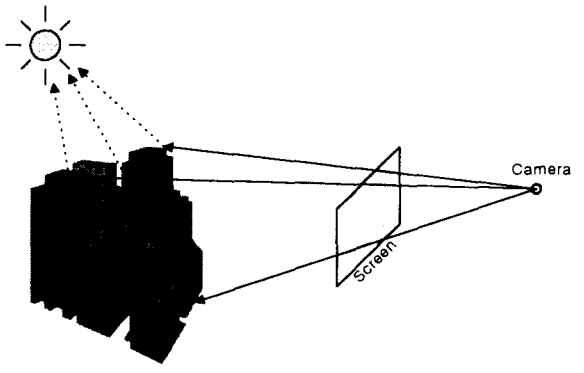

그림 1 Ray Tracing
Fig. 1 Ray Tracing

그림 1에서와 같이 Camera에서 출발한 각각의 시선은 벡터로서 표시된다. Ray Tracing은 각각의 벡터가 만나는 물체의 표면을 검색하고, 가장 가까운 표면에서 그 표면이 받는 빛의 양을 판별하여 색상을 계산하여 각 시선에 해당하는 pixel에 색상의 값을 저장한다.

3. Ray Tracing의 구현

3.1 Ray Tracing의 시선 벡터

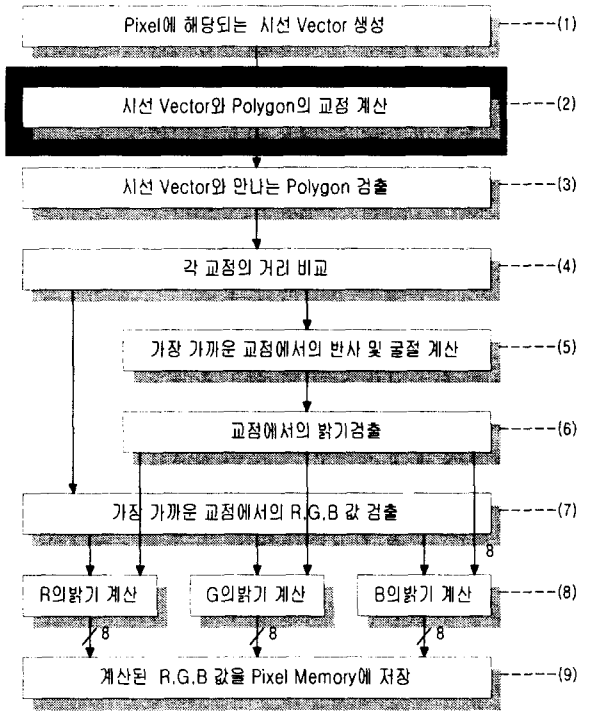

그림 2 Ray Tracing의 블록 다이어그램
Fig. 2 Block Diagram of Ray Tracing

Ray Tracing에서는 영상을 구현하기 위한 최소 단위로서 시선 벡터를 사용한다. 시선 벡터는 사용자의 시야로부터 화면의 각 픽셀 포인트까지의 위치벡터를 이용한다. 식 (1)은 시선 벡터를 구하기 위한 식이다.

$$\vec{I} = \vec{D} - \vec{O} \tag{1}$$

$\vec{I}$는 화면의 각 화소에 해당하는 시선 Vector를 의미한다. $\vec{D}$는 화면의 각 화소의 위치 Vector, $\vec{O}$는 카메라의 위치 Vector이다.

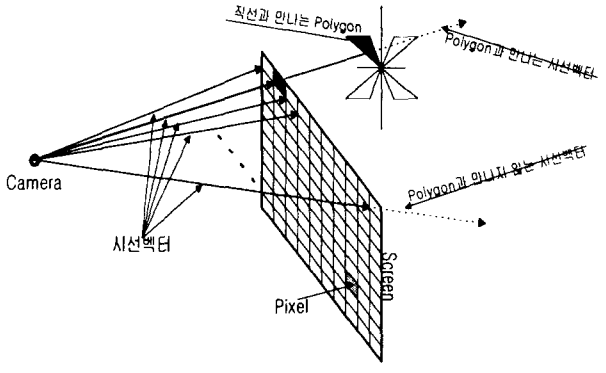

그림 3 Ray Tracing의 시선벡터
Fig. 3 Ray Vector for Ray Tracing

3.2 Hit-Test Unit

Hit Test Unit은 Ray Tracing의 블록 다이어그램(그림 2)에서 회색 박스로 처리된 (2)번 과정에 해당한다. Ray Tracing에서는 하나의 시선벡터와 각각의 Polygon마다 모두 교점을 계산하므로 연산시간이 길어지게 된다. 따라서 Ray Tracing 중 연산시간이 가장 긴 부분 중의 하나가 바로 이 부분이다. 본 논문에서는 이 부분을 하드웨어로 구현하여, Ray Tracing의 전체 연산 시간을 줄이는 방법을 연구하였다. 일반적인 Ray Tracing에서는 다음의 계산 과정을 거쳐 Hit Test를 수행한다. 계산 과정은 식 (2)~(5)와 같다.

$$\vec{N} = a\vec{i} + b\vec{j} + c\vec{k} \tag{2}$$

$$\vec{N} \cdot \vec{P} + d = 0 \tag{3}$$

$$\begin{aligned} t &= -\frac{d + \vec{N} \cdot \vec{O}}{\vec{N} \cdot \vec{D}} \\ &= \frac{d + (a \cdot O_x + b \cdot O_y + c \cdot O_z)}{(a \cdot I_x + b \cdot I_y + c \cdot I_z)} \end{aligned} \tag{4}$$

$$\begin{aligned} P_x &= t \cdot I_x + O_x \\ P_y &= t \cdot I_y + O_y \\ P_z &= t \cdot I_z + O_z \end{aligned} \tag{5}$$

식 (2)는 Polygon의 법선 벡터이다. 식 (3)은 Polygon을 포함하는 평면의 방정식이다. $\overrightarrow{P}$ 는 Polygon 상의 한 점이다. 식 (4)는 교점을 찾기 위한 매개 변수 t를 계산하는 과정이다. (O_x, O_y, O_z)는 카메라의 위치 벡터이고, (I_x, I_y, I_z) 는 시선 벡터를 의미한다. 식 (4)에서 계산된 매개 변수 t 를 식 (5)에 대입하면 시선 벡터와 Polygon을 포함하는 평면간의 교점의 좌표 (P_x, P_y, P_z) 를 얻을 수 있다. 이러한 계산 방법을 이용할 경우 수행해야 하는 과정이 곱셈 9회, 덧셈 8회, 나눗셈 1회의 과정을 거쳐야만 하기 때문에 연산과정이 길어지게 된다.

본 논문에서는 Hit Test Unit을 보다 효율적으로 구현하기 위하여, 알고리즘을 일부 수정하였다. 본 연구에서는 일반적인 알고리즘을 이용하여 계산하되, 카메라의 위치를 원점으로 미리 이동시켜 놓음으로서 계산 횟수를 줄였다. 계산 과정은 식 (5)~(10)과 같이 수행된다.

$$\overrightarrow{H} = a\vec{i} + b\vec{j} + c\vec{k} \tag{6}$$

$$\overrightarrow{I} = l\vec{i} + m\vec{j} + n\vec{k} \tag{7}$$

$$r_h = \sqrt{a^2+b^2+c^2} \tag{8}$$

$$r_i = \sqrt{l^2+m^2+n^2} \tag{9}$$

$$\begin{aligned}\overrightarrow{P} &= \frac{D_P}{\cos\theta}\overrightarrow{I} \\ &= \left(\frac{\frac{d}{r_h}}{\frac{\overrightarrow{I}\cdot\overrightarrow{H}}{r_i r_h}}\right)\frac{\overrightarrow{I}}{r_i} \\ &= \left(\frac{d}{\frac{\overrightarrow{I}\cdot\overrightarrow{H}}{r_i}}\right)\frac{\overrightarrow{I}}{r_i} \\ &= \left(\frac{d\overrightarrow{I}}{\overrightarrow{I}\cdot\overrightarrow{H}}\right)\end{aligned} \tag{10}$$

식 (6)은 Polygon의 법선 벡터를 나타낸다. 식 (7)은 시선벡터를 의미하며, 식 (8)은 법선 벡터의 크기를 의미한다. 그리고, 식 (9)는 시선 벡터의 크기를 의미한다. 식 (10)의 $\overrightarrow{P}$ 는 시선 벡터와 Polygon과의 교점을 나타내는 위치 벡터이다. 식 (10)에서 D_P는 평면과 원점 사이의 거리를 의미하며, θ는 법선 벡터와 시선벡터의 사이 각이다. $\cos\theta$ 는 두 벡터의 내적을 이용하면 계산할 수 있으므로, 식 (10)의 결과와 같은 계산식이 성립된다. 본 논문에서는 Hit-Test Unit을 구현하기 위해서 식 (10)을 사용하였다. 식(10)과 식 (4)(5)사이의 차이는 카메라의 위치를 어디에 잡느냐에 따라서 생긴다. 물론 식 (10)을 사용할 경우 카메라의 위치와 각 Polygon의 위치를 미리 계산해 두어야 한다. 하지만 미리 계산하게 되는 것까지 감안하더라도 식 (4)(5)를 이용한 경우에 비해 연산 횟수를 줄일 수 있다. 식 (10)을 이용하여 하드웨어를 구현할 경우 곱셈 6회, 덧셈 3회, 나눗셈 1회만으로 모든 연산을 수행할 수 있다. 식 (10)을 이용하는 경우와 식 (4)(5)를 이용하는 경우의 연산횟수를 비교하면 표 1과 같다.

표 2 연산 횟수 비교

	곱셈	덧셈	나눗셈	Hit Test	선처리	총계
일반적 구조	9	8	1	2.3040*109	1.9200*105	2.3042*109
제안된 구조	6	3	1	1.2800*109	1.9650*105	1.2802*109

표 1에서 Hit Test 항목과 선 처리 항목의 데이터는 해상도 320*200에서 500개의 Polygon을 사용한다고 가정하고, 빛의 반사 또는 굴절을 4회로 제한하였을 때의 수치비교이다. 선 처리 과정은 일반적 구조의 경우 시선 벡터를 계산하는 과정이고, 제안된 구조에서는 각 Polygon의 위치를 카메라의 위치 이동만큼 이동시키는 연산과정이다. 각 폴리곤은 3개의 점을 가지고 있고, 각 점은 3개의 좌표성분을 가지고 있다는 전제하에 계산된 과정이다. 따라서 기존의 연산 방법을 이용하는 경우보다 제안된 구조의 경우, Hit test를 위한 연산을 전체적으로 약 44.4% 줄이는 효과를 가져오게 된다. 연산과정이 줄어든 만큼 처리속도를 높일 수 있게된다. 그림 4는 제안된 Hit Test Unit의 Block Diagram 이다. 곱셈기, 덧셈기, 나눗셈기 각 각 1개씩 사용하였으며, 8개의 32-bit 레지스터와 2개의 32-bit Mux로 구성되어 있다. 각각의 Mux와 레지스터는 Counter를 이용한 Control Unit에 의해 제어된다. 그림 4에서 I-1, I-2, I-3, N, d 그리고 Output은 레지스터로 이루어져 있다. Control Unit에서는 Counter를 이용하여 각 Register와 Mux에 순차적으로 Control Signal을 보내게 된다. Register에는 Enable로 입력이 되고, Mux에는 Selector로서 Control signal이 사용된다. 그림 4에서 입력은 N 레지스터와 d 레지스터로 입력된다. N 레지스터로 입력된 값은 I-3 레지스터 또는 Mux를 거쳐 곱셈기로 입력된다. 초기 3 clock동안 N 레지스터로 (I_x, I_y, I_z) 이 입력되며, 입력된 값은 I-3, I-2, I-1 레지스터에서 순차적으로 Shift된다. 4번째 clock부터는 N 레지스터로 N(a, b, c)값이 순차적으로 입력되어 $a\cdot I_x$, $b\cdot I_y$, $c\cdot I_z$ 의 순서로 연산이 진행된다. 각각의 곱셈 결과는 Accumulator에 누적이 된다. 세 번의 곱셈이 끝나면 곧바로 d 레지스터로 d 값이 입력된다. d 레지스터는 이 순간에만 Enable이 되어 입력을 받으며, 나머지 타이밍에는 입력을 받지 않고 d 값을 유지하게 된다. 4-6번째 clock, 즉 곱셈이 수행되는 동안 I-3, I-2, I-1 레지스터에서는 계속 Shift를 진행한다. 곱셈을 하고 난 I_x 는 I-3 레지스터로 Shift된다. 이렇게 3번의 곱셈을 끝내고 나면, I_x는 다시 I-1 레지스터로 돌아오고, 이 때부터 d 레지스터에 입력되어 있는 d와 곱셈을 수행한다.

(I_x, I_y, I_z) 와 d 의 곱셈 결과는 나눗셈기로 보내져서 나눗셈을 수행하고 나면, 최종 Hit Test Unit의 결과 값인 (P_x, P_y, P_z) 값이 출력된다. 7번째 clock에 d 값이 입력되고 나면 다시 N 레지스터로 (I_x, I_y, I_z) 값이 입력되어 다음번 연산을 기다리게 된다. 첫 번째 입력이 들어가서 마지막 출력이 나오는데 까지는 10 clock이 소요되며, 입력되는 데이터는 7clock의 주기를 가지고 입력된다

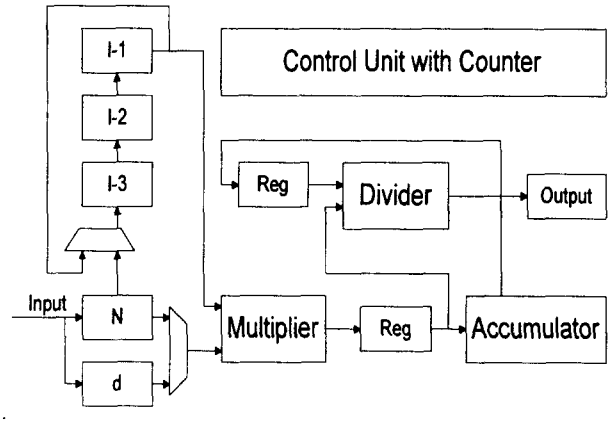

그림 4 Hit Test Unit의 블록 다이어그램
Fig. 4 Hit Test Unit의 Block Diagram

3.3 Hit Test Unit 이후의 연산과정

3.3.1 시선 벡터와 만나는 교점 검출

Hit Test Unit에서 행하는 교점 계산 과정은 시선 벡터와 Polygon이 만나는지의 여부에 관계없이 모든 Polygon에 대해서 수행하게 된다. Ray Tracing에서 필요한 교점은 모든 교점이 아니라, 실제로 Polygon 내부에 있는 교점만을 필요로 하게 되므로, 교점이 Polygon의 내부에 있는지, 외부에 있는지를 판별하는 과정이 필요하다. 이 과정 역시 Hit Test Unit과 마찬가지로 모든 폴리곤에 대해 수행하게 되므로 많은 연산과정을 필요로 하게 된다. 연산 횟수는 Hit Test Unit과 거의 비슷하다.

3.3.2 교점의 거리 비교

시선 벡터와 Polygon이 만나는 교점은 1개만 있는 것이 아니다. 1개만 존재하는 경우도 있겠지만, 이 경우는 극히 드물다. 화면에 보여지게 되는 물체는 입체로 이루어진 물체이기 때문에 무한한 길이의 시선 벡터의 관점에서 보게 되면 만나게 되는 Polygon의 교점은 1개 이상이 될 수 있다. 이러한 경우 시선의 원점으로부터 가장 가까운 거리에 있는 교점을 찾아낸다.

3.3.3 반사 및 굴절

교점에서 이루어지는 시선 벡터는 Polygon에 부딪혀 반사와 굴절현상이 일어나게 된다. 반사 또는 굴절이 일어날 경우에는 반사된 빛의 밝기가 일정한 수준 이하로 떨어질 때까지 반사 또는 굴절을 계산하게 된다.

3.3.4 교점에서의 R, G, B 값의 검출

3차원 영상의 각 Pixel에 해당하는 Color를 계산하기 위한 과정이다. 먼저, 교점을 가지는 Polygon에 지정된 R, G, B 세 색상을 읽어서, 밝기를 계산한다. 입력, 출력 모두 사람이 인지 가능한 Color수의 한계로 알려진 24-bit으로서 R, G, B 각각 8-bit을 가진다. 각 화소에 해당하는 메모리에 저장된 값은 화면의 지정된 위치에 바로 표시된다.

3.4 연산장치의 구조

Hit Test Unit은 연산의 반복에 의해 연산시간이 오래 걸리는 만큼 하드웨어의 성능이 연산을 직접 수행하는 곱셈기와 덧셈기의 성능에 크게 의존한다. 본 연구에서는 이러한 연산장치의 성능을 개선함으로서 Hit Test Unit의 처리속도를 높이고자 하였다. 본 연구에서는 모든 연산장치를 IEEE가 지정한 32-bit Floating Point 규격을 지원하도록 설계하였다. Hit Test Unit의 Critical Path는 곱셈기이므로 본 연구에서는 곱셈기의 성능을 높이는데 중점을 두었다.

3.4.1 곱셈기

곱셈기는 수정된 Booth 알고리즘[4][5][6][7]을 사용하여 부분곱의 수를 줄이고, Baugh-Wooley알고리즘[8]을 사용하여 Sign-bit의 확장을 억제함으로서, 곱셈에 필요한 덧셈의 수를 줄이고, 하드웨어의 크기를 줄였다. 곱셈기의 부분곱을 계산하는 과정은 Carry Save Adder Tree를 사용하여 Carry의 전파를 축소하였으며, CLA(Carry Look Ahead)구조를 CSA(Carry Select Adder)와 함께 적용하여 최적화[9] 시킴으로서 고속 곱셈기를 구현하였다. 그림 5는 Floatng Point 곱셈기의 블록다이어그램이다. 24-bit Unsigned Multiplier와 Exponent Unit, 그리고 Sign Unit과 후처리기로 이루어져 있다. Sign Unit의 경우 XOR 게이트 하나로 구현되며, Exponent Unit는 8-bit 덧셈기로, 후처리기는 Mux set으로 구성되어 있다. Floating Point의 경우 표현 가능한 수의 범위가 2-127 ~ 2128이므로 이는 5.8775*10-39 ~ 3.4028*1038에 해당하므로 매우 큰 수도 표현 가능하다.

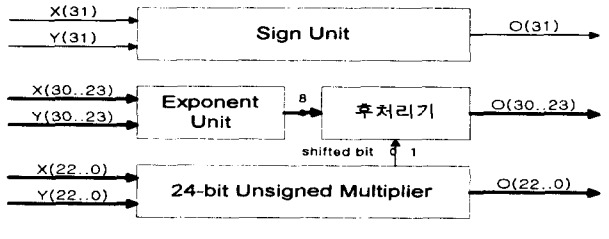

그림 5 Floating Point 곱셈기의 구조
Fig. 5 Structure of Floating Point Multiplier

그림 6은 Floating Point 곱셈기에 포함되어 있는 24-bit 곱셈기의 구조이다. Booth Encoder와 Booth Decoder에서

부분 곱을 생성시키면, 그 부분 곱을 CSA Adder Tree에서 부분 곱의 수를 2개로 줄이게 된다.

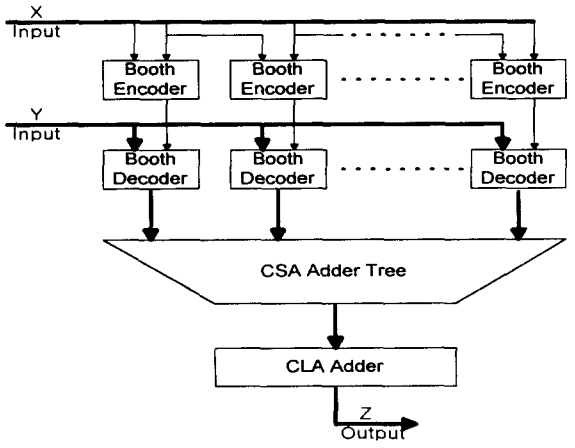

그림 6 Floating Point 곱셈기의 24-bit 곱셈기
Fig. 6 24-bit Multiplier in Floating Point Multiplier

그림 7과 같이 CSA Adder Tree는 역 피라미드 형태로 구성되어 있기 때문에 CSA의 단수를 줄일 수 있다. 총 부분 곱의 수가 13개에 Baugh-Wooley Constant를 포함하면 모두 14개의 부분곱을 더하게 된다. 일반적인 CSA를 쓰게 되면 12개의 CSA와 1개의 CLA를 필요로 하게 된다. Tree 구조로 구현할 경우 사용되는 CSA의 수는 같지만 총 거치게 되는 CSA의 단은 $\log_{3/2}13 \fallingdotseq 6.33$ 이 되므로 6개의 CSA단과 1개의 CLA 단만 거치면 되므로 속도를 보다 높일 수 있게 된다.

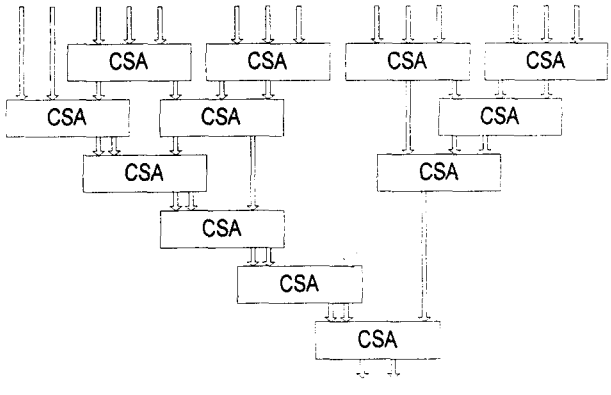

그림 7 CSA Tree
Fig. 7 CSA Tree

고속 연산을 위하여 CLA(Carry Look Ahead)구조를 Carry Select Adder와 함께 적용하였다. CLA는 bit수가 커질수록 Fan out이 커지고 차지하는 면적이 커지게 되므로 크게 만들기 어렵다. Fan out이 커지게 되면 그 주위 회로의 크기가 커지고 그와 함께 속도도 떨어지게 된다. 본 논문에서는 이러한 단점을 보완하기 위해 Carry Select Adder를 채용하여 Critical Path를 줄이는 동시에 Fan out 문제까지도 해결하였다.

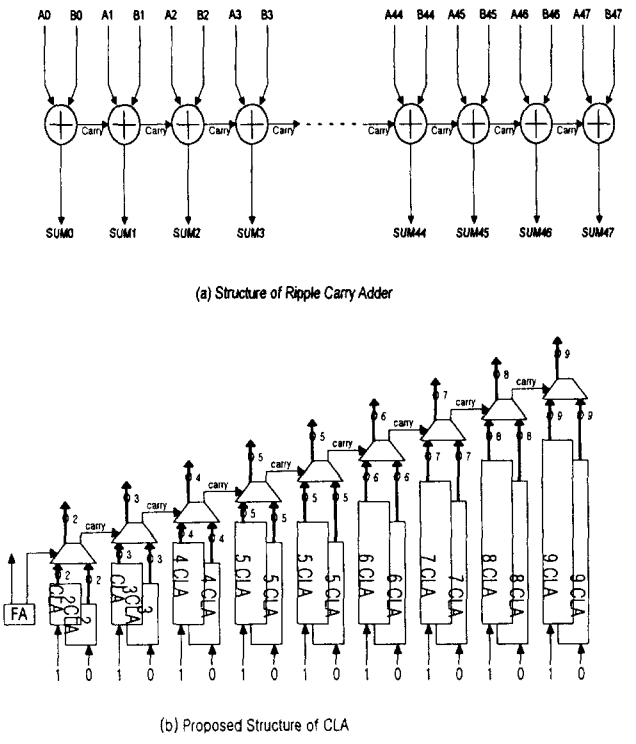

그림 8 CLA 구조의 비교
Fig. 8 Compare CLA Structures

그림 8은 Carry Select Adder의 개념을 설명한 그림이다. 일반적인 Ripple Adder에서는 Critical Path가 길어지게 된다. 이를 해결하기 위해 본 논문에서는 전단의 Carry를 기다리는 것이 아니라 전단의 Carry가 전달되기 전에 미리 결과를 계산해 놓았다가, 전단의 Carry가 전달되는 즉시 Mux를 통해서 결과를 내보내는 구조로 되어 있다. 전단의 Carry는 '0'과 '1' 두 가지 경우이므로 두 가지 경우의 계산을 먼저 해놓고 전단의 Carry가 넘어오기를 기다리는 것이다. 그림 8의 (a)는 일반적인 Ripple carry adder이다. (a)에서는 Critical Path가 48개의 FA가 되기 때문에 연산시간이 길어지게 된다. (b)는 CLA를 사용한 일반적인 Carry Selective Adder이다. (a)에 비해 하드웨어는 2배 필요하지만, Critical Path는 14FA로 70%정도 향상된다. 본 논문에서는 이 구조를 좀더 개선하여 점진적으로 CLA의 bit수를 늘림으로서 Critical Path를 줄였다. (b)와 비교하였을 때 하드웨어의 크기는 차이가 없지만, Critical Path를 10 FA로 줄일 수 있다. 이러한 Carry Select Adder를 이용할 경우 CLA의 크기가 과도하게 커지는 것을 방지할 수 있을 뿐만 아니라, Carry의 전파에 의한 Critical Path도 줄어들게 되어 보다 빠른 속도의 덧셈기를 구현할 수 있다.

4. 하드웨어의 설계 및 Simulation

4.1 하드웨어의 설계

본 논문에서는 Altera사의 FPGA Tool을 이용하여 설계,

검증하였다. 설계에는 VHDL과 Maxplus2의 Schematic 기능을 이용하였다.

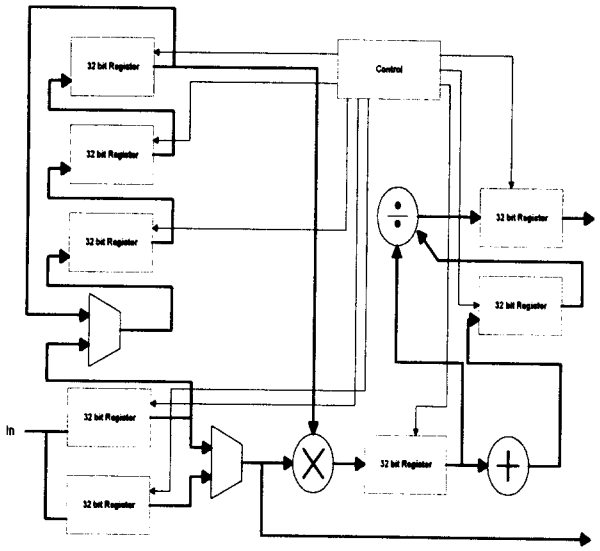

그림 9 The Logic Diagram of Hit Test Unit
Fig 9 The Logic Diagram of Hit Test Unit

그림 9는 Hit-Test Unit을 FPGA를 이용하여 설계한 Logic Diagram이다. 1개의 곱셈기, 1개의 덧셈기, 1개의 나눗셈기로 구성되어 있다. 동작을 제어하기 위한 Control Unit과 8개의 레지스터를 사용하였다. 각각의 연산장치는 IEEE의 32-bit Floating Point 표준을 준수하도록 설계되었다.

4.2 Simulation

Simulation 역시 Altera사의 FPGA Tool인 MaxPlus2를 이용하여 수행하였다. Simulation에서 사용한 동작 주파수는 10MHz 이고 Fig. 7에 보인 화면은 알아보기 쉽게 하기 위하여 5MHz로 동작시킨 화면이다. O[31..0] 항목은 Hit-Test Unit의 최종 결과 값이고, H[31..0] 항목은 Input Signal이다. 한 개의 시선벡터와 한 개의 Polygon과의 교점 계산을 수행하는 데에는 8clock이 소요된다. 이러한 계산과정을 자세히 살펴보면 시선 벡터와 Polygon사이의 교점을 나타내는 위치 벡터는 다음과 같다.

$$\overrightarrow{P} = \left(\frac{d\,\overrightarrow{I}}{\overrightarrow{I} \cdot \overrightarrow{H}} \right) \qquad (11)$$

이것의 계산 과정은 다음과 같다. 계산 과정의 수는 모두 32-bit Floating point number로 되어 있으며, 2진수와 10진수를 같이 표시하였다.

d ⇒
(00111111111100000000000000000000 → 1.75)

$\overrightarrow{I}$ ⇒
(01000000010101000000000000000000 → 3.3125,
01000000101111000000000000000000 → 5.875,
01000000001110110000000000000000 → 2.921875)

$\overrightarrow{H}$ ⇒
(01000001011100110000000000000000 → 7.1875,
01000001011100000000000000000000 → 0.46875,
01000000000001110000000000000000 → 2.109375)

Hit Test Unit에서 가장 먼저 계산하는 것은 시선벡터 $\overrightarrow{I}$ 와 법선벡터 $\overrightarrow{H}$ 의 내적이다. $\overrightarrow{I}$ 와 $\overrightarrow{H}$ 의 내적은 다음과 같다.

$$\overrightarrow{I} \cdot \overrightarrow{H} = 01000001001101100111001110100000 \rightarrow 59.22583008$$

다음 계산과정은 Polygon의 인수 중 하나인 d 와 시선벡터 $\overrightarrow{I}$ 와의 곱셈과정이다. 곱셈 후 바로 나눗셈을 통해 Output으로 출력된다.

$d\,\overrightarrow{I}$ =
(01000000101110011000000000000000 → 5.796875,
01000001001001001000000000000000 → 10.28125,
01000000101000111010000000000000 → 5.11328125)

마지막계산 과정은 $d\,\overrightarrow{I}$ 를 $\overrightarrow{I} \cdot \overrightarrow{H}$ 으로 나누는 과정이다. 이 계산의 결과값이 Hit Test Unit의 최종 출력 값이 된다.

$\overrightarrow{P}$ =
(00111101110010000111001111111000 → 0.09787748,
00111110001100011100001010100000 → 0.173594021,
00111101101100001101000010010000 → 0.086335324)

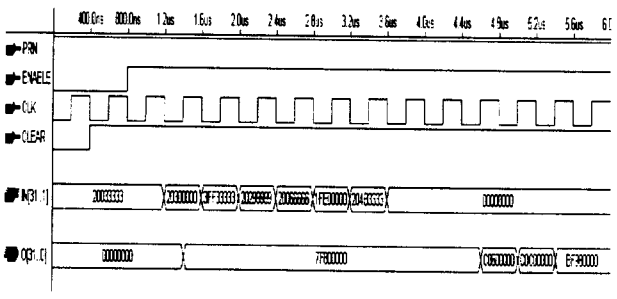

그림 10 Timing Diagram of the Hit Test Unit
Fig 10 Timing Diagram of the Hit Test Unit

5. 결론

Ray Tracing은 앞에서 언급한대로 연산과정이 반복적이기 때문에 연산시간이 매우 오래 걸리는 알고리즘이다. Ray Tracing의 모든 과정을 소프트웨어로만 구현할 경우 320 * 200의 화면크기에서 약 20여 개의 Polygon을 사용하여 화면을 구성할 경우 Pentium 133MHz 시스템에서 약 10분이 소요된다. 본 논문에서 설계한 Hit Test Unit은 10MHz로 동작이 가능하므로, Hit-Test Unit에서 데이터를 처리하는 시간은 약 512ms 정도 소요된다. Hit Test Unit이 Ray Tracing의 전체 연산 소요시간 중에서 차지하는 비중을 최소 20%로 가정할 경우 약 2.5초에 한 Frame의 생성이 가능해진다. 하지만 Pipeline 구조로 Ray Tracing을 설계할 경우에는 전체 Flow대로 진행하면 되므로, 512ms에 한 Frame의 영상을 생성할 수 있게 된다. 보통 Animation의 Frame rate는 18 Frame/sec이므로 약 9개의 칩을 병렬 처리하는 것만으로 동영상의 구현이 가능하다. 하지만 현실적인 3차원 그래픽을 위해서는 수백 개 이상의 Polygon이 소요되므로 이때에는 FPGA대신 고속 동작이 가능한 ASIC 칩을 사용하게 되면 보다 적은 수의 칩만으로 동영상의 구현이 가능해질 것이다. 여기에 요즈음 대두되는 Morphing 기법 등을 응용할 경우에는 더 적은 칩만으로도 동영상의 구현이 가능해진다. 기존의 경우 DSP를 이용하여 Ray Tracing을 구현한 사례[5]가 있었지만, 이 경우 512개의 Processor를 병렬로 연결하여 사용하였기 때문에 시스템으로 구현하는 데에는 어려움이 있다. 차후 칩의 고집적화를 통하여 Ray Tracing의 모든 과정을 고속의 ASIC 칩으로 구현할 경우에는 10개 이하의 칩만으로도 동영상의 실시간 처리가 가능해 질 수 있다.

이 논문은 1997년도 교육부 반도체분야 학술 연구 조성비에 의하여 연구되었음.

참고문헌

[1] A. Watt, 3D Computer Graphics, Addison-Wesley, 1993

[2] 신영수 외, 3차원 그래픽-C언어로 배우는 알고리즘, 가남사, 1995

[3] R. Pulleyblank, "The Feasibility of a VLSI Chip for Ray Tracing Bicubic Patches", IEEE CG&A, March 1987, pp. 33-44.

[4] Israel Koren, "Computer arithmetic algorithms", Prentice-Hall International, Inc. , 1993

[5] Naresh R. Shanbhag and Pushkal Juneja, "Parallel implementation of a 4x4 multiplier using modified Booth's algorithm", IEEE Journal of solid state circuits, vol. 23, no.4 Aug. 1988

[6] Philip E. Madrid, Brian Millar, and Earl E Swartzlander, "Modified Booth algorithm for high radix fixed-point multiplication", IEEE Trans. on VLSI systems, vol. 1, no. 2, pp.164-167, June 1993

[7] I-Chen. Wu, "A fast 1-D serial-parallel systolic multiplier", IEEE Trans. on computers, vol.C-36, no.10, pp.1243-1247, OCT.1987

[8] Jalil Fadavi-Ardeakni, "MxN Booth encoded multiplier generator using optimized Wallace trees", IEEE Trans. on VLSI systems, vol.1, no.2, pp.120-125, June 1993

[9] N. Ohkubo et al., "A 4.4 ns CMOS 54 × 54-b Multiplier Using Pass-Transistor Multiplexor", IEEE J. Solid-State Circuits, vol. 30, pp. 251-257, March 1995

[10] C. Scott Ananian, "The TigerSHARK Architecture", 1996

저 자 소 개

최 규 열 (崔 奎 烈)

1996년 인하대 전자재료공학과 졸. 1998년 인하대 전자재료공학과 석사

Tel : (032) 874-1663

정 덕 진(鄭德鎭)

1948년 2월 8일생. 1970년 서울대 공대 전기공학과 졸업. 1984년 미국 Utah State University 졸업(석사). 1988년 미국 University of Utah 졸업(공박). 1989년 ~ 인하대 전자재료공학과 교수

Tel : (032) 860-7435

E-mail : djchung@dragon.inha.ac.kr