한국게임학회 논문지
제6권 제4호 2006년 12월

게임그래픽

3D 게임의 실시간 렌더링 속도 향상을 위한 소프트웨어적 기법

황석민°, 성미영*, 유용희**, 김남중***
(주)컴투스°
{인천대학교 컴퓨터공학과}*,**,***
hsm0477@hotmail.com°,{mysung*,yhinfuture**,water09z***}@incheon.ac.kr

A Software Method for Improving the Performance of Real-time Rendering of 3D Games

Suk-Min Whang°, Mee Young Sung*, Yonghee You**, Nam-Joong Kim***
Com2us Corporation°
{Department of Computer Science & Engineering University of Incheon}*,**,***

요 약

그래픽스 렌더링 파이프라인 (응용, 기하, 레스터화)은 컴퓨터 게임에서 가장 중요한 기능인 실시간 그래픽스 렌더링의 핵심이다. 일반적으로 그래픽스 렌더링은 CPU와 GPU의 두 장치의 협조에 의해 완성되며 이 협조 과정에서 병목이 발생할 수 있다. 본 논문에서는 CPU와 GPU 사이에 발생하는 병목현상을 줄이는 데 초점을 맞추어, 보통은 하나의 스레드로 처리되는 CPU 연산을 순수 CPU 연산과 GPU와 연관된 연산의 두 가지로 구분하여 서로 독립적인 스레드로 병렬처리 되게 함으로써 실시간 그래픽스 렌더링의 성능을 향상시키는 방법을 제안한다. 이 방법은 CPU와 GPU 사이의 협조를 위한 전송 과정에서의 병렬성을 극대화한다. 실험을 통하여 제안하는 방법이 기존의 방법 보다 더 빠르게 그래픽스 렌더링을 수행함을 검증하였다. 또한 본 논문에서는 CPU와 GPU의 협조 과정에서 생기는 병목현상으로 인한 유휴시간을 잘 활용하여 렌더링 파이프라인의 균형을 맞추면서 렌더링의 질을 높이는 방법도 제안한다. 제안하는 방법들을 우리가 개발한 네트워크 게임 엔진에 적용하여 실제 시스템에서도 효과가 있음을 확인하였다.

ABSTRACT

Graphics rendering pipeline (application, geometry, and rasterizer) is the core of real-time graphics which is the most important functionality for computer games. Usually this rendering process is completed by both the CPU and the GPU, and a bottleneck can be located either in the CPU or the GPU. This paper focuses on reducing the bottleneck between the CPU and the GPU. We are proposing a method for improving the performance of parallel processing for real-time graphics rendering by separating the CPU operations (usually performed using a thread) into two parts: pure CPU operations and operations related to the GPU, and let them operate in parallel. This allows for maximizing the parallelism in processing the communication between the CPU and the GPU. Some experiments lead us to confirm that our method proposed in this paper can allow for faster graphics rendering. In addition to our method of using a dedicated thread for GPU related operations, we are also proposing an algorithm for balancing the graphics pipeline using the idle time due to the bottleneck. We have implemented the two methods proposed in this paper in our networked 3D game engine and verified that our methods are effective in real systems.

Key Words : real-time graphics rendering, performance, bottleneck, parallel processing

본 연구는 2005년 인천대학교 공모과제 연구비 지원으로 수행되었다.

1. 서 론

일반적인 렌더링 방식은 응용→기하→래스터화로 진행되는 렌더링 파이프라인 상에서 진행된다. 과거에는 파이프라인의 응용단계의 연산과 기하 단계의 연산은 CPU(Central Processing Unit)가 담당하였고 래스터화 단계의 연산은 그래픽 카드가 담당하였다. 그러나 그래픽 카드의 발전으로 기하 단계의 연산을 GPU(Graphics Processing Unit)가 담당함에 따라 CPU 측이 연산을 적게 하여 CPU가 더 많은 연산을 할 수 있게 되었다 [1],[2]. 그러나 이러한 연산의 분배로 그래픽 출력을 위한 일련의 프로세싱(응용, 기하, 래스터화)이 서로 다른 장치에 나뉘어 행해짐으로써 두 장치가 서로 상대편의 연산이 끝나기를 기다리는 병목현상(bottleneck)이 발생하게 되었다. 그래픽 출력의 효율적인 렌더링을 저해하는 병목현상을 효율적인 병렬화를 통하여 유휴시간을 줄여 렌더링 속도를 향상시키는 방법을 제안한다. 또한 병목현상이 발생할 수 있는 장치를 동적으로 체크하여 보다 향상된 그래픽 출력을 제공하는 방법에 대해 논의한다.

이런 논의를 위해 2장에서는 관련연구, 3장에서는 본 논문에서 제안하는 방법, 4장에서는 유효성을 확인하는 실험내용, 마지막으로 5장에서는 결론 및 향후 과제에 대해 기술하겠다.

2. 관련연구

초기의 그래픽 출력 기술의 발전과정에 있어서 대부분 연산을 CPU가 담당했었고, GPU는 이렇게 CPU가 연산한 내용을 화면에 뿌려주는 역할만을 담당 했었다. 따라서 그래픽 출력 속도를 높이기 위해서는 빠른 CPU의 연산만이 필요 했었다. 계속되는 기술의 발전은 하나둘씩 CPU가 하던 연산을 GPU 쪽으로 돌리게 되었고 이제는 상당수 많은 연산들이 이 GPU쪽에서 실행되고 있다. 표 1은 렌더링 파이프라인에서의 CPU와 GPU의 연산 분배 내용을 보여주고 표 2는 이와 같은 연산 분배의 발전과정을 연도별로 보여준다.

(a) 과거의 렌더링 연산 분배

CPU		GPU
AI, Physics, Animation	Transform & Lighting	Rendering

(b) 최근의 렌더링 연산 분배

the CPU	the GPU	
AI, Physics, Animation	Transform & Lighting	Rendering

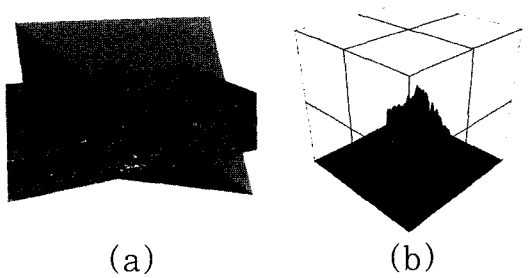
〈표1〉 렌더링 파이프라인에서의 CPU와 GPU의 연산 분배

	1996	1997	1998	1999
장면변화 그래프	CPU	CPU	CPU	CPU
좌표변환	CPU	CPU	CPU	*GPU*
조명	CPU	CPU	CPU	*GPU*
삼각형 설정과 클리핑	CPU	*GPU*	*GPU*	*GPU*
주사변환	*GPU*	*GPU*	*GPU*	*GPU*

〈표2〉 3D 그래픽 장치의 발전과정

이와 같은 CPU와 GPU의 역할 변화는 CPU가 맡고 있던 연산 그래픽 연산들의 상당부분을 GPU가 실행함으로써 CPU는 좀 더 많은 유휴시간들을 효과적인 인공지능, 맵 알고리즘 등의 연산에 더 많은 시간을 할당할 수 있게 해 주었고, 전체적으로 그래픽 렌더링 속도가 향상되는 결과를 가져다주었다(3). 즉 예전에는 CPU가 장면 변화, 좌표변환, 조명처리, 삼각형 설정과 클리핑 등과 같은 연산을 한 후 그 결과를 GPU에게 넘겨주면 GPU가 결과를 화면에 뿌려주는 선형 구조를 취하였기 때문에 빠른 속도를 기대 할 수 없었지만 현재는 CPU쪽에서 하던 위와 같은 연산들의 상당 부분을 GPU 쪽으로 보냄으로써 CPU와 GPU의 병렬화를 통해서 그래픽 출력속도가 향상되는 결과를 이룰 수 있게 되었다. 이러한 병렬화 기술과 그래픽 카드의 발전은 그래픽이 더 빠르게 출력되고 그래픽의 질을 더 높일 수 있게 하였다.

한편 반대로, 더 빠른 그래픽 출력에 대한 요구를 충족하기 위하여, CPU 쪽의 연산을 통해 GPU 쪽 연산의 부담을 줄이려고 하는 여러 가지 알고리즘들도 설계되었다. 그림 1의 그림들은 이런 노력의 가장 대표적인 알고리즘들과 관련된 그림들이다.



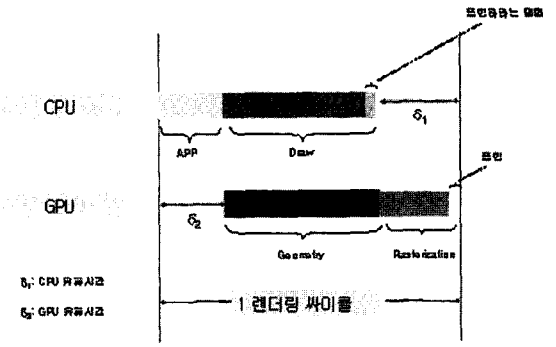
〈그림1〉 뷰 프러스텀 컬링 과 쿼드트리

그림 1 (a)는 뷰 프러스텀 컬링(View Frustum Culling)인데 카메라 안(사각뿔 안쪽 영역)에 들어오지 않는 정점에 대한 연산을 GPU 쪽으로 넘겨주지 않는 알고리즘과 관련된 그림이다. 그림 1 (b)는 쿼드트리(Quadtree) 알고리즘과 관련된 그림으로 지형을 계속적으로 8개의 노드들로 나누어 보일 가능성이 없는 정점들은 미리 제거하여 GPU 쪽으로 더 적은 양의 정점들을 보냄으로써 속도를 향상시키는 알고리즘이다.

본 논문에서 제안하는 기법은 이렇게 GPU 연산을 줄여서 렌더링 속도를 높이는 관점과는 다른 것으로, 렌더링이 진행될 때 CPU와 GPU 간에 생기는 병목현상을 비동기적인 방법을 통해 효율적으로 제거하는 기법이다. 본 논문에서 제안하는 기법도 위의 뷰 프러스텀 컬링과 쿼드트리 알고리즘처럼 CPU쪽 연산을 이용하여 그래픽 출력속도를 높인다는 목적은 서로 같다. 그러나 뷰 프러스텀 컬링과 쿼드트리 알고리즘은 GPU 연산 자체를 직접적으로 줄일 수 있는 반면, 본 논문에서 제안하는 알고리즘은 이런 알고리즘들을 적용하는 일련의 과정에서 발생할 수 있는 병목현상 때문에 생기는 유휴시간을 줄일 수 있다는 차이점이 있다. 따라서 GPU 연산 자체를 줄이는 기법과 본 연구에서 제안하는 병목현상을 줄이는 기법을 같이 쓴다면 그래픽 출력에 더 나은 효과를 얻을 수 있을 것이다.

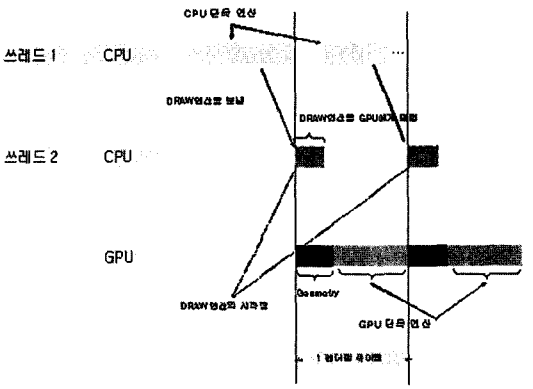
3. 렌더링 병렬 처리

일반적인 렌더링에 있어서 그림 2에서 보는 바와 같이 GPU 연산은 CPU 측의 백버퍼(back buffer)와 프런트 버퍼(front buffer)를 바꾸어 주는 플립연산을 끝으로 한 싸이클을 마치게 되는데[4] 일반적으로 이 플립연산이 끝날 때까지 CPU는 GPU를 기다리게 된다. 따라서 본 논문에서는 그림 2에서 보이는 것처럼 두 장치에서 나타날 수 있는 병목현상을 파악하여 소프트웨어적으로 병렬처리를 해 줌으로써 더 빠르고 효율적인 그래픽 출력을 제공하고자 한다.



〈그림2〉 일반적 CPU와 GPU의 병렬처리와 발생가능한 유휴시간

본 논문에서 제안하는 알고리즘에서는 우선 응용 단계의 연산(CPU 연산)을 크게 APP, CULL, DRAW의 3단계로 나누었다[5]. APP 단계와CULL 단계는 CPU만 사용하는 연산으로, DRAW는 CPU와 GPU 모두를 사용하는 연산으로 정의하고자 한다.

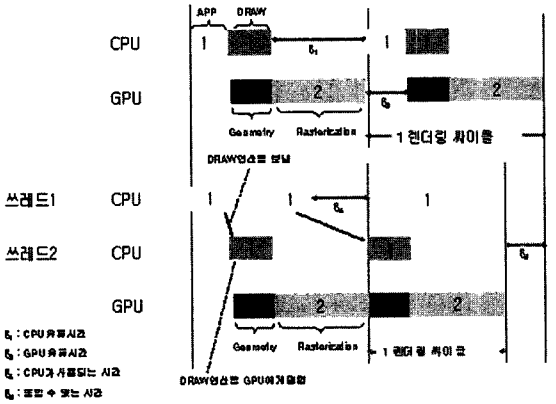


〈그림3〉본 논문에서 제안하는 방법

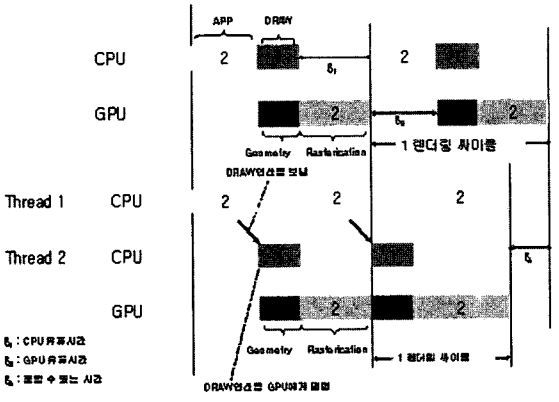
그림 3에서처럼 본 논문에서 제안하는 방법에서는 CPU 측 연산을 두 개의 쓰레드로 수행하게 하였다. 첫 번째 쓰레드는 APP와 CULL을 하는 쓰레드로서 두 번째 쓰레드로부터 플립 완료 이벤트를 받을 때까지 멈춰 있도록 구성하였다. 두 번째 쓰레드는 기존의 연산들 중 DRAW 연산을 실행하도록 구성하였다. 두 번째 스레드는 DRAW 연산에 대한 처리 명령을 이벤트로 받으면 실제 DRAW 명령을 GPU쪽

으로 내려주는 역할을 하고 이런 연산이 끝났을 때는 종료 이벤트를 첫 번째 쓰레드 쪽으로 전송해주는 역할을 한다.

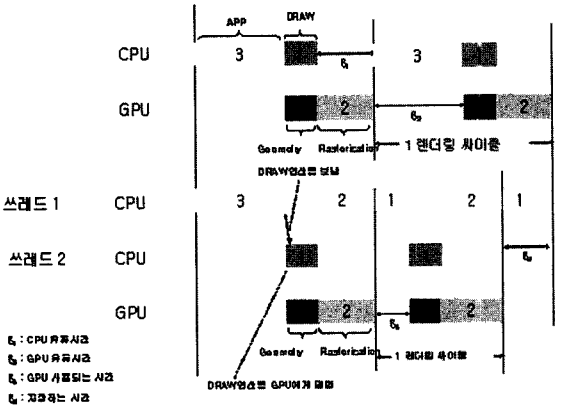
그림 3과 같이 구성하였을 경우에 그림 4, 그림 5, 그림 6와 같은 세 가지 현상이 나타날 수 있다.



〈그림4〉 GPU가 병목인 경우



〈그림5〉 이상적인 경우



〈그림6〉 CPU가 병목인 경우

그림 3은 GPU 연산 시간이 CPU 연산 시간보다 긴 경우로, 이런 상황이 발생한다면 GPU쪽에 병목현상이 생기는 것을 의미한다. 이러한 상황에서 기존의 방법으로 렌더링을 한 하나의 예를 보면, 한 프레임을 렌더링 하는데 걸리는 시간이 4 단위 시간이 걸리고, CPU 쪽에서는 2 단위 시간이, GPU쪽에서는 1 단위 시간이 아무런 연산도 하지 않고 대기 하고 있는 것을 볼 수 있다.

본 논문에서 제안 하는 방법을 사용하면 한 프레임을 렌더링 하는데 걸리는 시간이 3 단위시간이 되어 기존의 방법으로 렌더링 한 경우보다 1 단위시간을 줄여 전체적으로 렌더링 속도를 빠르게 해주는 결과를 얻을 수 있다. 또한 GPU쪽의 유휴시간은 제거하고, CPU쪽의 유휴시간은 1 단위시간 동안에 CPU가 쉬지 않고 다른 연산을 할 수 있어 1단위시간을 줄이는 효과가 있었다.

그림 5는 플립하는데 걸리는 시간과 CPU 단독연산 시간이 같은 이상적인 경우로서 유휴시간이 전혀 발생하지 않는 경우이다.

그림 6는 플립 하는데 걸리는 시간이 CPU 단독시간보다 짧아 CPU쪽에 병목현상이 발생하는 경우이다. 한 프레임을 렌더링 하는데 걸리는 시간이 6 단위시간이 걸렸으며, CPU쪽에서는 2단위시간이, GPU쪽에서는 3 단위시간이 대기하고 있는 것을 볼 수 있다. 그러나 본 논문이 제안하는 방법을 사용하면 한 프레임을 렌더링 하는데 4 단위시간이 되어 2 단위시간을 줄여 전체적인 렌더링 속도를 빠르게 할 수 있었다. 또한 CPU쪽의 유휴시간은 제거하고, GPU쪽은 1 단위시간 동안 더 많은 연산을 할 수 있는 효과가 있었다.

본 논문에서는 DRAW 연산만을 위한 독립된 별도의 스레드를 사용하여 병목시간을 줄이는 방법뿐 아니라, CPU와 GPU의 협조 과정에서 생기는 병목현상으로 인한 유휴시간을 잘 활용하는 알고리즘도 제안한다. 위에서 살펴본 것처럼 만약 매 프레임마다 병목현상이 발생하는 장치와 병목시간을 알 수 있다면 병목현상이 발생하지 않는 장치에 병목시간 만큼의 연산을 추가로 할당하여 효율적인 그래픽 렌더링이 가능하게 할 수 있다. 지금부터 그 방법을 살펴보고자 하며 렌더링 성능 측정의 단위로는 프레임율 (FPS; Frames Per Second)를 사용하겠다.

만일 CPU 연산이 증가되었음에도 불구하고 렌더링 프레임율이 감소한다면 이 경우는 CPU에 병목현상이 생긴 경우이며, GPU는 CPU의 연산이 끝나기를 기다리는 동안 조명

의 계산을 더 섬세하게 한다든지 하는 고급 조명 처리와 세이딩 연산을 더 많이 할 수 있다. 반대로, 만일 CPU의 연산이 증가하였음에도 렌더링 프레임율이 감소하지 않는다면 이 경우는 GPU에 병목현상이 생긴 경우이므로, CPU는 GPU의 연산이 끝나기를 기다리는 동안 인공지능 알고리즘이나 물리 알고리즘 연산을 더 많이 할 수 있다.

그림 7은 실시간으로 병목현상이 일어나는 장치를 조사하여 병목현상이 없는 장치에 부가적인 연산을 할당하는 알고리즘을 요약한 것이다.

```
Increase the amount of calculations;
if ( previous FPS > current FPS ) /* FPS: Rendered Frames per Second */
  then Increase the GPU operations /* Bottleneck in the CPU case */
  else Increase the CPU operations /* Bottleneck in the GPU case */
```

〈그림7〉 병목 유휴시간 활용 알고리즘

4. 실험내용

본 절에서는 3절에서 소개한 방법에 대해 실험을 통하여 그 효과를 확인해 보고자 한다. 실험을 위하여 그림 2의 일반적인 경우와 그림 3의 제안하는 기법을 적용한 경우를 측정 할 수 있는 프로그램을 각각 작성하였다. 실험환경은 표 3과 같다.

CPU	Pentium4-2.54Ghz
RAM	1.00GB
VGA	ATI RADEON 9700 PRO
LIBRARY	DIRECTX 9.0[2]
개발환경	VC++ 6.0
사용언어	C++
모드	1280*944
사용색	32비트 컬러

〈표3〉 실험 환경

CPU 연산은 Sleep() 함수를 사용하여 이 함수의 인자 값을 조정함으로써 연산의 처리에 소요되는 CPU 시간의 변화량을 나타내었고, GPU 연산에 대하여는 그려지는 삼각형의 개수를 조정함으로써 연산의 변화량을 나타냈다. 동일한 CPU 시간과 GPU 시간을 설정한 후에 일반적인 방법으로 렌더링 하는 경우와, 본 논문에서 제안하는 방법인 DRAW 전담 쓰레드를 사용하여 렌더링 경우의 각각에 대하여 작성한 두 개의 프로그램을 실행시켜 FPS(Frames Per Second)를 측정해 보았다. 실험은 일정한 수의 삼각형 그리기(800,000개, 1,000,000개, 1,500,000개, 그리고 2,000,000개)에 대하여, 즉 정해진 GPU 연산량에 대하여 CPU 연산량을 0ms, 10ms, ...100ms 등으로 단계별로 변화시켜 이때 변하는 프레임 수를 측정하는 방법으로 진행하였다.

테스트 프로그램의 핵심 내용은 아래와 같다.

○ 2개 쓰레드 생성

· 쓰레드1 : APP와 CULL 단계를 수행하며 이벤트1(DRAW completion event)을 받을 때까지 블록 되었다가 다음 프레임의 DRAW 명령을 실행하는 쓰레드2에게 이벤트2(DRAW start event)를 보낸다.

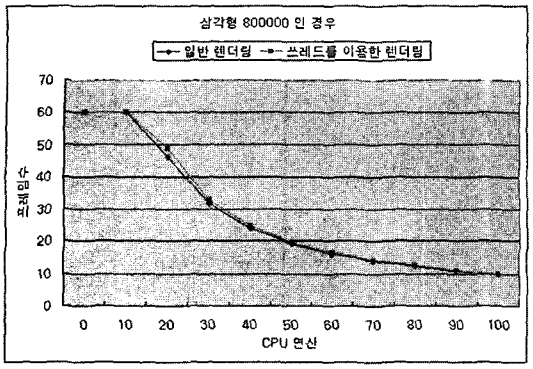
· 쓰레드2 : DRAW 단계만을 전담한다. 쓰레드1으로부터 이벤트1(DRAW start event)을 받은 즉시 실행을 시작하고 GPU가 DRAW 오퍼레이션을 수행하도록 명령을 보내며, GPU가 DRAW 동작을 완료하면 블록되어 있던 쓰레드1이 재시작 되도록 이벤트2(DRAW completion event)를 쓰레드1에게 보낸다.

○ 2개 이벤트 생성

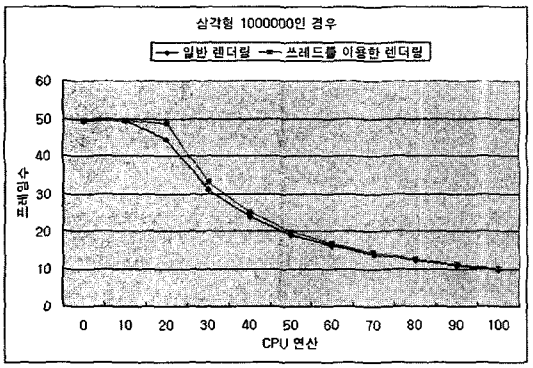
· 이벤트1 : 쓰레드1이 쓰레드2에게 보내는 DRAW start event 로서 쓰레드2로 하여금 GPU에게 DRAW 명령을 보낼 수 있게 한다.

· 이벤트2 : 쓰레드2가 쓰레드1에게 보내는 DRAW completion event로서 쓰레드1가 블록된 상태를 해제하고 다음 프레임을 위한 계산을 시작하게 한다.

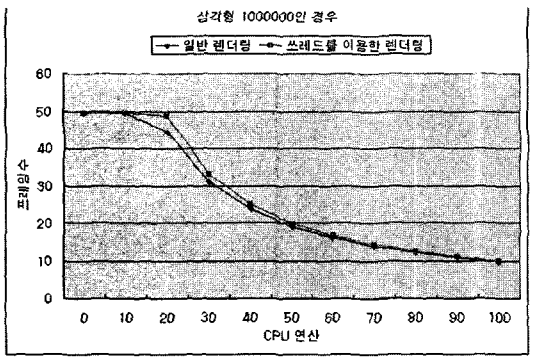
그림 8, 그림 9, 그림 10, 그림 11에서 보는 바와 같이, DRAW 전용 쓰레드를 사용한 경우 렌더링 되는 프레임수가 증가하는 것을 볼 수 있었다. 위의 네 가지 실험 모두에서, CPU 연산량이 늘어남에 따라 GPU 병목 상황에서 이상적인 상황을 거쳐 CPU 병목 상황으로 전환됨을 볼 수 있었다. 또한 삼각형의 개수를 늘릴수록 프레임수가 떨어지는 시점이 늦춰짐을 확인할 수 있었는데, GPU 쪽 연산이 증가하여 CPU 쪽 유휴시간이 더 많이 생길 수 있기 때문에 나타나는 현상으로 볼 수 있다.



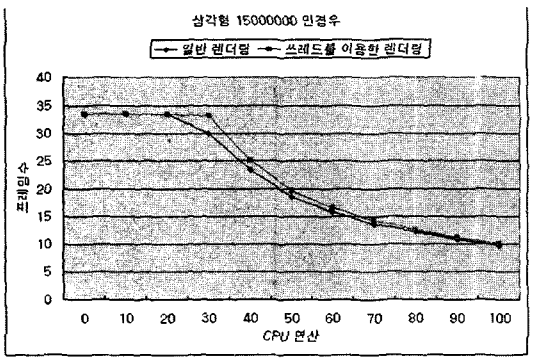
〈그림8〉 렌더링하는 삼각형 수가 800,000개인 경우



〈그림9〉 렌더링하는 삼각형 수가 1,000,000개인 경우

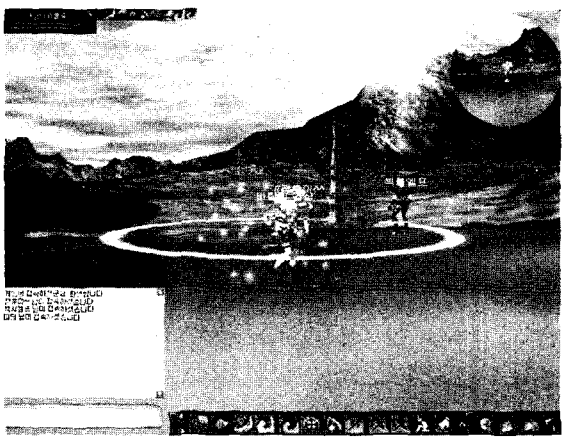


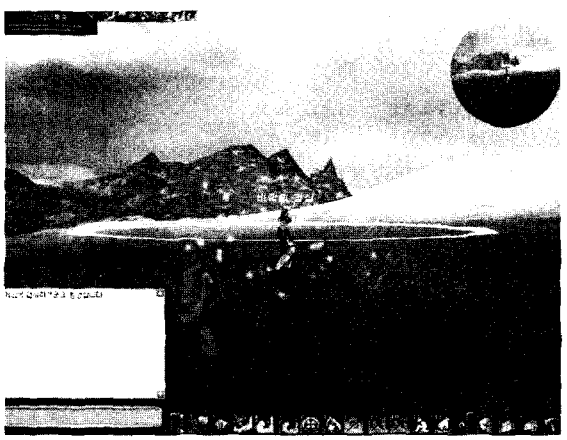
〈그림10〉 렌더링하는 삼각형 수가 1,500,000개인 경우



〈그림11〉 렌더링하는 삼각형 수가 2,000,000개인 경우

그림 7에 요약된 본 논문에서 제안하는 병목 유휴시간 활용 알고리즘을 본 연구실에서 개발한 네트워크 3D 게임 엔진에 적용하여 보았다. 이 게임 엔진에서는 렌더링 성능(프레임율)을 주기적으로 측정하여 CPU와 GPU의 연산량을 병목 유휴시간 활용 알고리즘에 따라 조절하게 하였다. GPU에 병목현상이 생기면, 네트워크 게임에 충돌 탐지(collision detection)와 운동학 알고리즘(kinematics algorithms) 연산을 CPU에 부가하고, CPU에 병목현상이 생긴 경우에는 퐁 전반사 알고리즘(Phong' s specular highlighting algorithm)과 체적 그림자 알고리즘(volumetric shadow algorithm)을 GPU에 부가하였다.





〈그림12〉 응용 예의 스크린 샷

5. 결론

본 논문에서는 CPU와 GPU의 병렬처리 과정에서 발생하는 병목현상을 줄여 실시간 렌더링에서 그래픽 출력을 빠르게 하는 소프트웨어적인 비동기적 처리 기법을 제안하였다. 렌더링을 위한 CPU 연산과 GPU 연산의 병렬처리에 있어서 CPU쪽의 연산을 순수 CPU 작업(APP와 CULL)을 하는 쓰레드와 GPU 제어 연산(DRAW)을 제어하는 쓰레드로 나누어 두 가지 연산을 비동기적으로 수행하는 방법을 제안하였고, 실험을 통하여 제안하는 방법이 그래픽 출력 과정에서 나타날 수 있는 CPU와 GPU 장치의 병목현상을 줄일 수 있음을 확인하였다. 향후에는 그림 5와 같은 이상적인 상황이 발생하는 GPU 연산량과 CPU 연산량의 비율을 계산하여, 실시간으로 GPU 연산량에 따른 CPU 연산량의 최적치를 유지하도록 조절함으로써 더욱 빠른 실시간 렌더링 효과를 내게 하는 연구를 진행하고자 한다.

참고문헌

[1] Akenine-Moller, T., Haines, E.: Real-Time Rendering. 2nd edition, A K PETERS (2002)

[2] Gray, K.: The Microsoft DirectX 9 Programmable Graphics Pipeline. Microsoft Press (2003)

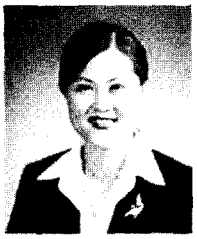
[3] Wimmer, M., and Wonka,P.: Rendering Time Estimation for Real-Time Rendering. In proceedings on Eurographics Symposium on Rendering 2003 (2003) 118-129

[4] Cox, Michael, Sprague D., Danskin., Ehlers., Hook B., Lorensen B., and Tarolli G.: Developing High-Performance Graphics Applications for the PC Platform. In Course 29 notes at SIGGRAPH 98 (1998)

[5] Rohlf, J., Helman, J.: IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. In proceedings on SIGGRAPH 94 (1994) 381-394

황 석 민 (Suk Min Whang)

2006년 인천대학교 컴퓨터공학사
2006년~ 현재 (주) 컴투스 개발부 프로그래머
관심분야: 소프트웨어공학, 컴퓨터그래픽스

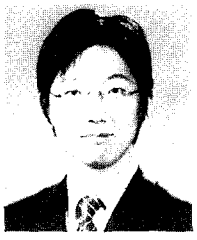


성 미 영 (Mee Young Sung)

1982년서울대학교 학사
1987년프랑스 INSA de Lyon 컴퓨터공학 석사
1990년프랑스 INSA de Lyon 컴퓨터공학 박사
1990년 ~ 1993년한국전자통신연구소 인공지능연구실 선임연구원
1993년 ~ 현재인천대학교 컴퓨터공학과 교수
관심분야 : 네트워크 멀티미디어/가상현실, 게임프로그래밍

유 용 희 (Yong hee You)

2007년 인천대학교 컴퓨터공학사
2007년~현재 인천대학교 대학원 컴퓨터공학과 석사과정
관심분야 : 네트워크 멀티미디어/가상현실, 게임프로그래밍



김 남 중 (Nam-Joong Kim)

2006년 인천대학교 컴퓨터공학사
2006년~ 현재 인천대학교 대학원 컴퓨터공학과 석사과정
관심분야: XML, 데이터베이스