GPU를 이용한 실시간 이미지 프로세싱 시스템

Development of Real-Time Image Processing System Using GPU

이 자 용*, 오 재 홍, 강 훈
(Ja-Yong Lee, Jae-Hong Oh, and Hoon Kang)

Abstract : When a real-time image processing application is implemented with a general-purpose computer, CPU (Central Processing Unit) is usually heavily loaded and in many cases that CPU alone cannot meet the real-time requirement at all. Most modern computers are equipped with powerful Graphics Processing Units (GPUs) to accelerate graphics operations. There is a trend that the power of GPU outgrows that of CPU. If we take advantage of the powerful GPU for more general operations other than pure graphics operations, the processing time can be reduced. In this study, we will present techniques that apply GPU to general operations such as image processing procedures. Our experiment results show that significant speed-up can be achieved by using GPU.

Keywords : GPU, real-time, image processing, edge detection, histogram equalization

I. 서론

그래픽 보드는 그래픽 메모리에 저장되어 있는 문자 또는 그래픽 정보를 모니터에 출력할 수 있는 신호로 변환시켜주는 변환 장치에서 시작되었다. 초기의 그래픽 프로세서는 시스템 버스를 통해서 전달되는 그래픽 정보를 그래픽 메모리에 전달하는 기능과 그래픽 메모리의 정보를 램댁(RAMDAC, RAM Access Memory and Digital to Analogue Convertor)에 전달하여 화면으로 출력하는 단순한 기능만을 가지고 있었다. 하지만 가속 기능이 등장하면서 화면의 한쪽 영역을 다른 쪽으로 이동하거나 복사하고 또는 그래픽 메모리 블록을 시스템 메모리에 복사하거나 이동하는 2차원 그래픽 가속 기능이 탑재되기 시작했다. 근래의 그래픽 보드에서는 2차원 그래픽 가속 기능 이외에도 3차원 그래픽 가속 기능과 멀티미디어 가속 기능을 갖추고 있다.

초기에는 단순 변환기 수준이었지만 그래픽 기술이 보다 고성능/다기능화 되면서 그래픽 프로세서는 단순 변환 장치에서 이제는 CPU 수준의 복잡한 프로세서로 변모하고 있다. 또한 성능도 기하 급수적으로 높아지고 있어서 매년 4배 이상 성능이 향상되어지고 있다. CPU수준의 강력한 기능을 갖게 된 그래픽 프로세서는 GPU라고 불리며 발전을 거듭하여 이제는 CPU를 뛰어넘는 성능을 보여주고 있다[4].

최신 GPU들은 쉐이더(shader)라는 기술을 탑재하면서 또 한번의 기술적 도약을 하게 된다. 쉐이더 기술은 GPU 하드웨어에 대한 낮은 수준의 접근 및 조작을 제공함으로써 빠른 실행 속도를 희생하지 않고도 놀라운 유연성을 갖도록 해주었다. 이러한 유연성은 그래픽스 관련 연산에 국한되지 않고, 그래픽스와 무관한 일반적인 연산까지도 GPU에서 처리할 수 있을 만큼 강력한 것이다. 만일 일반 연산을 GPU에서 처리하는 것이 가능하다면, 전체 연산의 일부를 GPU에서 처리하도록 함으로써 CPU에 걸리는 부담을 줄일 수 있으며 GPU의 강력한 연산 능력을 이용하여 전체적인 수행 시간을 줄일 수 있을 것이다. 이는 강력하지만 많은 연산 시간 때문에 사용할 수 없었던 다양한 알고리즘들을 실시간 시스템에 적용할 수 있는 획기적인 방법을 제공할 것이다[8,9].

본 논문은 일반적인 연산을 GPU에서 실행시킴으로써 전체적인 속도 향상을 얻을 수 있는 방법을 제안하고 있다. 제안된 방법의 유용성을 검증하기 위한 테스트로 이미지 프로세싱 연산을 GPU와 CPU에서 각각 계산하도록 하여 연산에 걸리는 시간을 비교하였다. 이미지 프로세싱은 비교적 단순한 연산이 반복되고 분기가 적기 때문에 GPU에서 연산을 수행할 경우 상당한 성능 향상을 기대할 수 있다.

II. GPU

일반적으로는 CPU의 클럭 속도가 GPU의 클럭 속도보다 훨씬 높지만, 특정 연산의 경우에는 GPU를 사용하는 것이 CPU를 사용하는 경우보다 큰 이득을 볼 수 있다. 이것이 가능한 이유 중 한가지는, CPU는 보통 수십 단계의 파이프라인을 가지고 있지만 GPU는 수백의 단계가 있다는 점이다. n개의 파이프라인으로 나누어 연산을 수행하면 이상적으로 n배가 빨라진다는 점을 감안한다면 GPU의 속도 향상은 당연한 것이다. 특히, 그래픽스나 이미지 프로세싱 알고리즘들은 비교적 간단한 연산이 반복되는 경우가 많다. 따라서 다음에 읽어올 메모리 위치를 예측하기 쉽기 때문에 효율적으로 데이터를 미리 가져오는 것이 가능하다.

GPU의 연산 속도 향상을 가능하게 하는 또 하나의 방법은 병렬화(parallelization)이다. CPU는 연산 주도(operation-driven) 방식인데 반하여, GPU는 데이터 주도(data-driven) 방식이며 처리하는 데이터들이 서로 독립적이고 분기가 거의 없다. 이러한 이유 때문에 GPU는 CPU의 경우보다 병렬화가 쉽고, 더 효율적으로 활용될 수 있다.

GPU를 이용한 연산이 CPU에서의 보다 빠른 또 다른 이유는 트랜지스터의 집적량 및 실제 연산에 활용되는 양에서 차이가 있기 때문이다. 최신의 GPU는 CPU에 비하여 더 많은 양의 트랜지스터가 집적되어 있다. ATI R300 GPU의 경우 107 million 이상의 트랜지스터가 집적되어있는 반면, 비슷한 시

* 책임저자(Corresponding Author)
논문접수 : 2004. 8. 30., 채택확정 : 2004. 12. 29.
이자용, 오재홍, 강훈 : 중앙대학교 전자전기공학부
(jalnans@sirius.cie.cau.ac.kr/ojh6t3k@daum.net/hkang@cau.ac.kr)
※ 이 논문은 2004학년도 중앙대학교 학술연구비 지원에 의한 것임.

기에 발표된Intel Pentium4 2.4G CPU의 경우에는 55 million 정도의 트랜지스터를 갖고 있다. 집적된 트랜지스터의 양에도 차이가 있지만 연산에 활용되는 양을 고려한다면 그 차이는 더욱 심해진다. GPU의 경우에는 집적된 트랜지스터의 대부분이 연산에 활용되지만, CPU의 경우에는 60% 이상의 트랜지스터가 단순히 캐쉬(cache)로만 사용된다[1].

또 한가지, GPU에 관련해서 주목해야 할 사실은 그 발전 속도이다. GPU는 매 6개월마다 2배씩 성능이 향상되고 있으며, 이러한 놀라운 성능 향상은 현재도 계속 진행되고 있다. GPU는 이제 CPU의 명령에 따라 그래픽 부분을 처리하는 단순한 그래픽 프로세서의 위치를 뛰어넘어 CPU를 능가하는 성능을 보여주고 있다.

III. 쉐이더

쉐이더 기술이 사용되기 이전의 GPU들은 수행할 수 있는 연산들이 하드웨어적으로 고정되어 있었기 때문에 그래픽스 관련 연산 조차도 제한적으로 수행될 수 밖에 없었다.

그림 1은 GPU 내부에서 3차원 그래픽스 연산을 수행하는 파이프라인을 간단하게 나타낸 개념도이다. 프로그램 가능한 쉐이더들은 하드웨어적으로 고정된 기존의 파이프라인을 완벽하게 대신하면서 동시에 놀라운 유연성을 제공하고 있다.

쉐이더는 어셈블리 언어와 매우 유사한 쉐이더 어셈블리 언어(shader assembly-language)로 작성된 짤막한 프로그램을 사용하여 수행해야 할 연산을 정의한다. 이 프로그램들은 CPU상에서 미리 컴파일 되고 실제 사용을 위해 GPU로 전달된다. 최근에는 C 언어와 유사한 형태의 고수준 쉐이더 언어인 HLSL(High-Level Shader Language)이 개발되어 쉐이더를 더욱 쉽게 이용할 수 있도록 지원하고 있다[5].

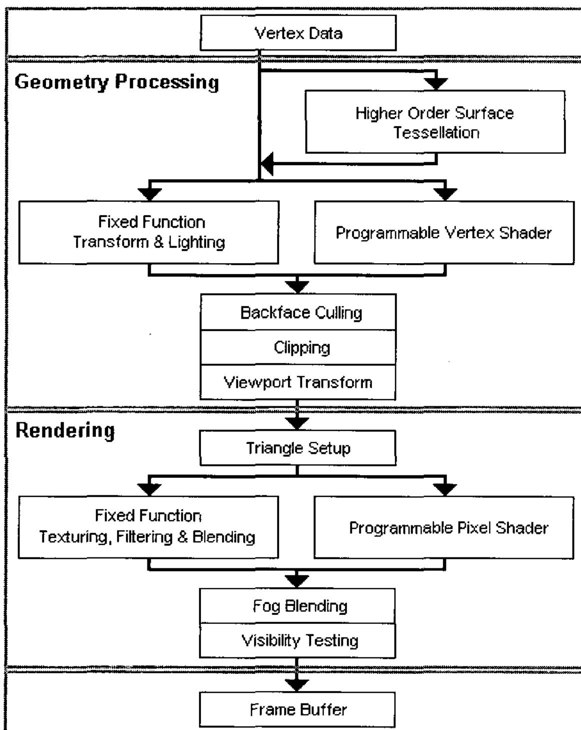


그림 1. GPU의 3차원 그래픽스 파이프라인.
Fig. 1. The 3D graphics pipeline in GPU.

HLSL을 사용하는 것이 편리하기는 하지만, 본 논문에서는 프로그램 최적화를 위해 쉐이더 어셈블리 언어를 이용하였다. 그림 2는 쉐이더 어셈블리 언어로 작성한 Blur연산의 프로그램 예이다. GPU에 탑재된 쉐이더는 정점과 관련된 연산을 수행하는 정점 쉐이더와 렌더링될 픽셀에 영향을 주는 픽셀 쉐이더 2종류가 있다.

1. 정점 쉐이더

정점 쉐이더(vertex shader)는 정점의 색상, 법선 벡터, 텍스처 좌표 및 위치와 같은 각 다각형의 정점과 관련된 값들을 변경시키는 방법을 제공한다. 그림 3에 정점 쉐이더의 개념도를 나타내었다.

```
ps.2.0

dcl t0
dcl t1
dcl t2
dcl t3
dcl t4
dcl t5
dcl t6
dcl t7
dcl_2d s0

texld r0, t0, s0
texld r1, t1, s0
texld r2, t2, s0
texld r3, t3, s0
texld r4, t4, s0

texld r5, t5, s0
texld r6, t6, s0
texld r7, t7, s0
add r8, t0, c20
texld r8, r8, s0

add r9, r0, r1
add r9, r9, r2
add r9, r9, r3
add r9, r9, r4
add r9, r9, r5
add r9, r9, r6
add r9, r9, r7
add r9, r9, r8

mul r00, r9, c0
mov oC0, r0
```

그림 2. Blur 연산을 위한 쉐이더 어셈블리 코드.
Fig. 2. Sample of shader assembly code for blur.

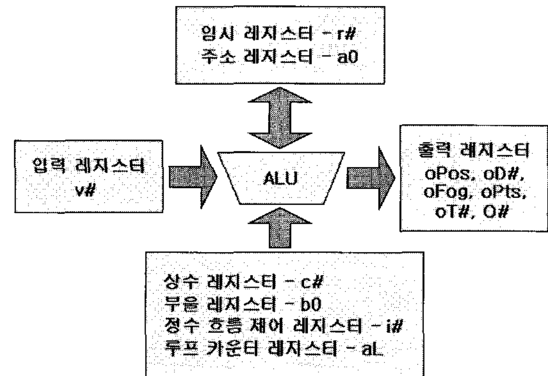


그림 3. 정점 쉐이더 개념도.
Fig. 3. Vertex shader block-diagram.

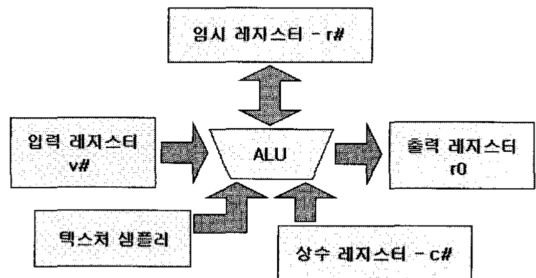


그림 4. 픽셀 쉐이더 개념도.
Fig. 4. Pixel shader block-diagram.

쉐이더 어셈블리 언어로 미리 정의된 일련의 연산은 산술 장치(ALU)에서 수행된다. 산술 장치는 입력 레지스터에서 데이터를 읽어 적절한 연산을 수행하고 결과를 출력 레지스터를 통해 다음 단계로 전달한다. 이때 상수 레지스터, 부울 레지스터 등의 내용을 참조하여 연산에 이용할 수 있고, 중간 계산 값을 임시 레지스터에 저장하거나 반대로 읽어올 수 있으며, 주소 레지스터를 이용하여 간단한 분기 명령을 처리할 수도 있다.

2. 픽셀 쉐이더

픽셀 쉐이더(pixel shader)는 정점 쉐이더와 같이 GPU가 어떻게 데이터를 다루는가에 대한 섬세한 지배권을 허락한다. 픽셀 쉐이더는 화면에 렌더링될 모든 픽셀에 영향을 준다. 그림 4에 픽셀 쉐이더의 개념도를 나타내었다.

픽셀 쉐이더는 정점 쉐이더와 구조는 거의 비슷하지만 두 가지 큰 차이점이 있다. 우선 픽셀 쉐이더에는 주소 레지스터가 없기 때문에 분기 명령을 수행 할 수 없다. 픽셀 쉐이더에서 분기 명령을 사용할 수 없다는 점은 프로그램 작성시 상당한 제약 조건으로 작용한다. 또한 픽셀 쉐이더는 텍스처 데이터를 제공하는 텍스처 샘플러가 제공되는데 이를 이용하면 대용량 데이터를 쉽게 참조 할 수 있다.

IV. GPU 의 제약 조건

GPU는 본래 그래픽스 관련 연산을 처리하도록 설계되어 있다. 쉐이더 기술을 사용하면 GPU를 하드웨어 레벨에서 조작할 수 있기 때문에 원래 설계된 의도와는 달리 그래픽스와 무관한 연산도 처리하도록 만들 수 있지만 상당한 제약 조건들을 수반하고 있다. 정점 쉐이더나 픽셀 쉐이더에서 수행할 연산들은 어셈블리 언어와 매우 유사한 쉐이더 어셈블리 언어로 작성된 짤막한 프로그램을 사용하여 정의 할 수 있다. 그런데, 이 프로그램에 사용할 수 있는 명령어 수가 너무 적다. 그나마 그래픽스 분야에 특화된 명령어를 제외한다면 간단한 산술 연산이 전부이다. 특히, 픽셀 쉐이더에서는 분기 명령을 사용할 수 없기 때문에 분기 명령이 포함된 연산은 픽셀 쉐이더에서 처리하는 것이 불가능하다. 또한, 쉐이더에서 사용하는 프로그램에는 최대 지원 가능한 라인 수의 제한이 있다. 쉐이더 버전마다 약간의 차이가 있고, 최신 버전의 경우 라인 수가 비약적으로 늘어나긴 했지만 복잡한 알고리즘을 구현할 경우에는 아직도 많이 부족하다.

표 1. 정점 쉐이더의 최대 지원 명령어 라인 수.
Table 1. Maximum code line for vertex shader.

버전	최대 명령어 라인 수
vs_1_1	128
vs_2_0	256
vs_3_0	512

표 2. 픽셀 쉐이더의 최대 지원 명령어 라인 수.
Table 2. Maximum code line for pixel shader.

버전	최대 명령어 라인 수
ps_1_1	8
ps_2_0	64
ps_3_0	512

최대 명령어 라인수가 비약적으로 증가한 상위 버전이 도입된다면 이러한 문제점이 어느 정도 해결되어 추가적인 속도 향상이 기대된다.

쉐이더의 입출력은 8bit로 한정되어 있기 때문에 정확도 측면에서 문제가 발생하게 된다. 내부적으로는 32bit 부동소수점 방식으로 처리하기 때문에 중간 과정은 제약이 덜 하지만 처리된 결과를 8bit로 밖에 받을 수 없다는 점은 정확도 측면에서 치명적인 제약 조건이 된다. 실제 실험 결과에서, GPU를 사용한 경우와 CPU를 사용한 경우의 출력 영상은, 육안으로는 확인 불가능 하지만, 전체 영상의 20~30%에서 ± 1정도의 오차를 보이고 있다. 이러한 오차는 내부적으로 32bit 부동소수점 방식으로 처리되던 데이터가 출력 과정에서 8bit정수로 변환되면서 발생한다. 이러한 정확도의 문제는 각 픽셀을 128bit 부동소수점 값으로 표현하는HDRI(High Dynamic Range Imaging) 방식이 도입된다면 해결될 것으로 예상된다.

V. 실험 결과 및 분석

본 논문에서 제안하고 있는 방법의 유용성을 검증하기 위해 동일한 연산을 GPU와 CPU에서 각각 처리하도록 하여 연산에 사용된 시간을 비교하였다. 실험 결과는 각각 10만 번씩 연산을 수행하여 얻은 데이터의 평균이다. 실험에는 이미지 프로세싱의 대표적인 연산인 윤곽선 검출과 히스토그램 평활화 알고리즘이 사용되었다. 실험에 사용된 GPU는 ATI R300이고 CPU는 비슷한 시기에 발표된Intel Pentium4 2.4G를 사용하였다. 실험에 사용된 입력 영상의 크기는320*240, 640*480, 1024*768 이다.

1. 윤곽선 검출

윤곽선 검출(edge detection)은 아주 간단하면서 유용한 방법으로 그 응용분야가 매우 넓다. 다양한 윤곽선 검출 알고리즘 중, 실험에 사용된 것은 Sobel 마스크와 Laplacian of Gaussian 마스크 이다. 실험 결과는 다음과 같다.

GPU를 사용하여 윤곽선 검출을 수행한 경우 CPU를 이용하는 경우보다 적게는 3배에서 많게는 5배 이상 속도가 향상되었다. 일반적으로 실시간 시스템에서는 최소한 초당 6프레임 정도를 처리 할 수 있어야 하고, 초당 15프레임을 처리한다면 확실한 실시간 시스템이라고 할 수 있다. 즉, 1프레임의 처리속도가 166ms 이하가 되어야 실시간 시스템이라고 할

표 3. 연산 시간(sobel 윤곽선 검출).
Table 3. Processing time of sobel edge detection(ms).

input size	GPU	CPU
320*240	2.446	7.423
640*480	8.436	44.668
1024*768	20.630	116.142

표 4. 연산 시간(Laplacian of Gaussian 윤곽선 검출).
Table 4. Processing time of Laplacian of Gaussian ED(ms).

input size	GPU	CPU
320*240	2.662	8.456
640*480	9.143	46.125
1024*768	22.334	117.348

수 있고, 66ms 이하가 되면 사람이 처리 시간을 거의 인지하지 못할 정도가 된다. 윤곽선 검출을 GPU에서 연산하도록 한 경우 1024*768 정도의 고해상도 영상까지도 처리 시간에 충분한 여유가 있음을 알 수 있다.

2. 히스토그램 평활화

표 5. 연산 시간(histogram equalization).
Table 5. Processing time of histogram equalization(ms).

input size	mask size	GPU	CPU
320*240	21*21	42.00	182.68
	31*31	76.27	293.77
	41*41	122.99	457.74
640*480	21*21	166.79	698.63
	31*31	305.83	1189.51
	41*41	497.50	1805.25
1024*768	21*21	401.82	1855.65
	31*31	756.66	3077.21
	41*41	1246.88	4720.21

(a) Original input image

(b) Local HE result by GPU (c) Local HE result by CPU

그림 5. 히스토그램 평활화 결과 이미지.
Fig. 5. Result image of load histogram equalization.

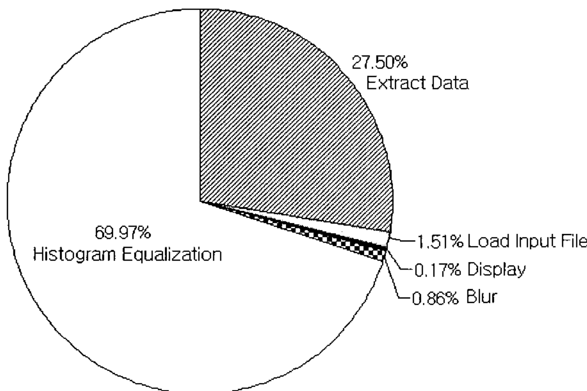


그림 6. 히스토그램 평활화에 대한 부하 분석.
Fig. 6. Load profiling of histogram equalization.

히스토그램 평활화(histogram equalization)는 명암 값의 분포가 한쪽으로 치우치거나 균일하지 못한 영상을 명암 값의 분포가 균일해지도록 영상을 향상하는 이미지 프로세싱 기법이다. 즉, 명도 값 분포를 수정함으로써 전체적인 영상의 콘트라스트 밸런스가 개선된다[2].

히스토그램 평활화는 크게 2종류로 분류된다. 영상 전체의 히스토그램 분포를 이용하는 전역 히스토그램 평활화(Global HE)와 매 픽셀마다 인접 지역의 히스토그램 분포를 조사하여 이를 이용하는 지역 히스토그램 평활화(local HE)가 있다. 전역 히스토그램 평활화는 간단하지만 부분적으로 명암 값이 치우친 영상은 개선하지 못하는 단점이 있다. 이를 개선한 방법이 지역 히스토그램 평활화 기법이지만 너무 많은 연산량을 요구하기 때문에 사용이 제한된다. 지역 히스토그램 평활화를 빠른 시간 안에 처리할 수 있다면 보다 다양한 분야에 적용할 수 있을 것이다. 표 5에 지역 히스토그램 평활화에 대한 실험 결과를 나타내었다. GPU를 사용한 경우가 CPU를 사용한 경우에 비해 평균적으로 4배 이상 연산 속도가 향상됨을 알 수 있다. 그림 5는 히스토그램 평활화의 결과 이미지이다.

3. 부하 분석

그래픽 보드는 시스템 버스를 통해서 전달되는 그래픽 정보를 화면으로 출력하는 일을 담당한다. 즉, 그래픽스 관련 연산에서는 GPU를 통해 가공된 데이터가 화면으로 출력될 뿐, 다시 시스템 메모리로 가져와야 하는 경우가 극히 드물다. 이러한 이유 때문에, 그래픽 보드는 시스템 메모리의 데이터를 그래픽 메모리로 전달하는 부분은 최적화가 되어 있지만, 반대로 그래픽 메모리의 데이터를 시스템 메모리로 가져오는 부분은 최적화가 덜 이루어져서 시간이 많이 걸린다.

그림 6은 GPU를 사용하여640*480 크기의 이미지에 21*21 크기의 마스크를 사용한 지역 히스토그램 평활화에 대한 각 부분별 부하량을 분석한 그래프이다. 전체 연산 시간에서 그래픽 메모리의 데이터를 시스템 메모리로 가져오는 부분(extract data)이 차지하는 비율이 27.5% 정도 차지한다. CPU에서 모든 연산을 실행할 때와는 달리, GPU를 사용할 경우 실제 연산과는 무관하게 연산된 결과를 가져오는데 드는 시간이 상당히 많은 부분을 차지하는 것은 큰 문제점이라고 할 수 있겠다. 이러한 문제점은 그래픽 메모리와 시스템 메모리 사이의 대역폭을 증가 시킴으로써 어느 정도 개선할 수 있다. 실험에 사용된 시스템에는 AGP 4x가 사용되고 있으며 대역폭은 1.06GB/sec이다. 현재 AGP 8x를 지원하는 GPU들이 발표되어있고, AGP 8x의 대역폭은 AGP 4x의 거의 2배에 해당하는 2.1GB/sec이다. 그래픽 메모리와 시스템 메모리 사이의 데이터 전달 속도가 빨라진다면 GPU에서의 연산 시간은 더욱 짧아지게 될 것이다.

VI. 결론

최근에 발표된 GPU는 CPU를 능가하는 성능을 보이고 있어 이를 이용한다면 상당한 성능 향상을 기대할 수 있다. 본 논문에서는 쉐이더 기술을 이용하여 일반적인 연산을GPU에서 수행하도록 함으로써 시간적인 측면에서 성능을 향상 시킬 수 있는 방법을 제안하고 있다.

제안된 방법의 유용성을 검증하기 위해 대표적인 이미지

프로세싱 알고리즘인 윤곽선 검출과 히스토그램 평활화를 GPU와 CPU에서 각각 처리하도록 하여 연산에 사용된 시간을 비교하였다. CPU를 이용한 경우와 비교할 때, GPU를 사용한 경우 비약적으로 실행 속도가 향상된다는 것을 실험을 통해 확인 할 수 있다. 이러한 속도 향상은 GPU의 놀라운 발전 속도와 신기술의 도입으로 더욱 큰 차이를 나타낼 것으로 예상된다. GPU를 이용하는 방법은 강력하지만 연산 속도의 제한 때문에 구현할 수 없었던 다양한 분야의 알고리즘을 실시간 시스템에 탑재할 수 있는 획기적인 방법을 제공하고 있다.

참고문헌

[1] T. A. Moller and E. Haines, *Real-Time Rendering,* A. K. Peters, Natick, 2002.

[2] R. C. Gonzalez and R. E Woods, *Digital Image Processing,* Addison Wesley, Reading, 1992.

[3] G. Shen, L. Zhu, S. Li, H. Y. Shum and Y. Q. Zhang, "Accelerating video decoding using GPU", *Proc. of the Conf. ICASSP'03,* vol. 4, pp. 6-10., April, 2003.

[4] M. Macedonia, "The GPU enters computing's mainstream", *Computer,* vol. 36, issue. 10, pp. 106-108, Oct, 2003.

[5] K. Gray, *Microsoft DirectX9 Programmable Graphics Pipeline,* Microsoft Press, 2003.

[6] W. F. Engel, *Direct3D ShaderX,* Wordware Publishing, Plano, 2002.

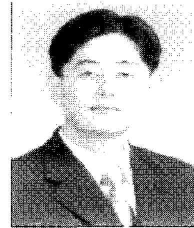
[7] K. Dempski, *Real-Time Rendering Tricks and Technique in DirectX,* Premier, Boston, 2002.

[8] http://www.developer.nvidia.com

[9] http://www.ati.com

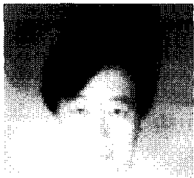
이 자 용

2001년 중앙대학교 전자전기공학부(공학사). 2003년 중앙대학교 전자전기공학부(공학석사). 2003~현재 중앙대학교 전자전기공학부 박사과정. 관심분야는 인공 지능, 지능 제어, 머신 비전, 컴퓨터 그래픽스.



오 재 홍

2003년 중앙대학교 전자전기공학부(공학사). 2003~현재 중앙대학교 전자전기공학부 석사과정. 관심분야는 로보틱스, 지능 제어, 머신 비전.



강 훈

1982년 서울대학교 전자공학과(공학사). 1984년 서울대학교 전자공학과(공학석사). 1989년 Georgia Institute of Technology(공학박사). 1992년~현재 중앙대학교 전자전기공학부 교수. 관심분야는 퍼지 및 신경망, 진화 알고리즘, 인공 생명, 머신 비전, 로보틱스, 지능 제어.