Journal of Korea Multimedia Society Vol. 10, No. 9, September 2007(pp. 1209-1219)

임베디드 시스템을 위한 3차원 그래픽 가속 장치 구동기의 설계 및 구현

김성우[†], 이중화[††], 이종민[†††]

요 약

임베디드 시스템과 같은 제한된 하드웨어에서 3차원 그래픽 기반의 응용 프로그램을 구동하는 것은 쉽지 않다. 그러한 시스템은 그래픽 가속 모듈을 구동하여 다양한 그래픽 기능을 처리할 수 있는 체계적인 3차원 그래픽 처리 구조가 필요하다. 본 논문에서는 임베디드 시스템을 위한 공개 소스 그래픽 윈도우 환경인 Tiny X 체계에서 3차원 그래픽 가속 장치 구동기를 구현하는 방법을 상세히 제시한다. 제안한 방법은 가속 장치 구동기를 단계적으로 초기화하여 직접 렌더링 구조가 이를 적절하게 활용할 수 있도록 한다. 아울러, 3차원 그래픽 처리 성능을 효율적으로 평가할 수 있는 간단한 프로그램을 통하여 구현된 가속 장치 구동기에 대하여 적용하여 그 유용성을 확인하였다.

Design and Implementation of a 3D Graphic Acceleration Device Driver for Embedded Systems

Seong-Woo Kim[†], Jung-Hwa Lee[††], Jong Min Lee[†††]

ABSTRACT

It is difficult to run 3D graphics based application on the embedded system with hardware constraints. Therefore, such a system must have a systematic infrastructure which can process various operations with respect to 3D graphics through any graphic acceleration module. In this paper, we present a method to implement 3D graphics acceleration device driver on Tiny X platform which provide an open source graphics windowing environment. The proposed method is to initialize the driver step by step so that the direct rendering infrastructure can use it properly. Moreover, we evaluated overall 3D graphics performance of an implemented driver through a simple but effective benchmark program.

Key words: 3D Graphics(3차원 그래픽스), Tiny X(Tiny X), Direct Rendering Infrastructure(직접 렌더링 구조), OpenGL(OpenGL)

1. 서 론

임베디드 시스템의 응용분야는 의료장비와 공장 제어 같은 산업분야 및 가전 제품분야와 정보산업 분야, 나아가 우주항공 분야에 이르기까지 그 영향력이 실로 방대하게 확대되어 가고 있으며 새로운 부가가치를 창출하는 중요한 기술로 평가받고 있다[1]. 90년대 이후에 그래픽 하드웨어를 포함한 전자공학

※ 교신저자(Corresponding Author) : 김성우, 주소 : 부산시 부산진구 엄광로 995(614-714), 전화 : 051)890-1728, FAX : 051)890-1724, E-mail : libero@deu.ac.kr
접수일 : 2007년 3월 23일, 완료일 : 2007년 8월 27일
[†] 준회원, 동의대학교 컴퓨터소프트웨어공학과
[††] 정회원, 동의대학교 컴퓨터소프트웨어공학과 (E-mail : junghwa@deu.ac.kr)
[†††] 정회원, 동의대학교 컴퓨터소프트웨어공학과 (E-mail : jongmin@deu.ac.kr)
※ 본 연구는 2006학년도 동의대학교 교내연구비에 의해 연구되었음(2006AA168)

및 컴퓨터 기술의 발달로 휴대폰, PDA, 홈서버, 인터넷 냉장고 등의 다양한 정보단말 기기들이 등장하였고 점차 다양하고 복잡한 기능을 가지게 되었다. 또한, 이러한 포스트 PC 시대에 발맞추어 기존의 VxWorks, QNX, Nucleus 등의 기존의 실시간 운영체제 외에도 윈도우즈 CE, 임베디드 리눅스 등의 운영체제들이 임베디드 시스템에 적용되어 왔다.

이처럼 다양한 임베디드 장치들과 이를 구동시키는 운영체제 환경이 등장함으로써 종래의 범용 그래픽 윈도우 시스템의 화려한 총천연색 화면 장치들과 다양한 입력 장치들을 임베디드 환경에서도 지원할 수 있게 되었다. 또한, 종래의 고성능 개인용 컴퓨터에서 실행되던 그래픽 3차원 게임들이 임베디드 시스템에도 점차 적용되면서 소형 임베디드 그래픽 윈도우 시스템 기술과 3차원 콘텐츠의 개발을 가속시키고 있다[2].

그런데, 복잡한 연산을 많이 수행하는 3차원 게임 등을 원활하게 구동하기 위해서는 3차원 가속 기능을 가지는 그래픽 카드를 포함한 임베디드 하드웨어, 커널, 임베디드 그래픽 윈도우 시스템, 3차원 그래픽 처리 지원 라이브러리, 3차원 지원 응용 프로그램들이 유기적으로 잘 결합되어야 한다[2,3]. 현재의 데스크탑 윈도우 환경에서는 OpenGL이나 DirectX와 같은 표준 API를 통하여 2차원 및 3차원 그래픽 처리를 효율적으로 수행하고 있다. 임베디드 및 모바일 시스템에서는 제한된 자원을 고려하여 시스템의 크기와 전력소모를 줄일 수 있는 가벼운 구조로 변화하고 있으며, 임베디드 및 모바일 장치를 위한 2차원 및 3차원 그래픽 API 표준으로 OpenGL/ES 등이 제안되어 점차 적용되고 있다[4]. 한편, 리눅스와 같은 개방형 플랫폼에서는 표준 그래픽 윈도우 서버와 함께 OpenGL API를 사용하여 3차원 그래픽 처리를 수행하도록 하고 있다[5].

하지만, 임베디드 리눅스와 같은 제한된 환경에서 그래픽 가속 하드웨어를 활용한 3차원 그래픽 처리를 효율적으로 할 수 있는 방안이 아직까지는 많이 미흡한 것이 사실이다. 예를 들면, Eric Anholt는 Tiny X 기반의 그래픽 윈도우 서버 Xati를 개발하였지만 2차원 그래픽 연산에 대해서만 하드웨어 가속 처리가 가능하다[3]. 본 논문에서는 임베디드 리눅스 시스템에서 많이 사용하는 그래픽 윈도우 시스템인 Tiny X 서버 위에서 3차원 그래픽 가속 모듈을 구축하는 방법을 제안하며, 이러한 방법은 OpenGL API를 사용하면서 그래픽 하드웨어의 3차원 그래픽 가속 기능을 활용할 수 있으므로 우수한 성능을 가질 수 있다. 더불어, 임베디드 시스템에서 복잡한 3차원 그래픽 처리 성능을 측정하고 평가할 수 있는 성능 평가 프로그램의 구성 방법을 제시하고, 몇 가지 복합적인 3차원 그래픽 처리 기법을 평가하는 벤치마크 프로그램을 구현하여 특정한 타겟 시스템에 대하여 적용하여 성능을 검증하였다.

본 논문은 다음과 같이 구성된다. 2장에서는 일반적인 리눅스에서 3차원 그래픽 처리를 위한 기술을 살펴보며, 3장에서는 임베디드 리눅스를 위한 그래픽 윈도우 환경인 Tiny X 위에서 3차원 그래픽 가속 처리를 가능하도록 구현하는 방법에 대하여 설명한다. 4장에서는 복합적인 3차원 그래픽 처리를 위한 효율적인 성능 평가 프로그램의 구현 방법에 대하여 서술하고, 실제 타겟 시스템에 적용하여 본 논문에서 제시한 그래픽 가속 모듈에 대한 3차원 그래픽 처리 성능을 평가하였다. 마지막으로, 5장에서는 본 논문의 의의와 향후 연구 과제를 제시한다.

2. 리눅스 상의 3차원 그래픽 처리 기술

3차원 그래픽 응용 프로그램을 구동시키기 위해서는 표준 3차원 그래픽 라이브러리를 이용하는 것이 일반적이며, 3차원 그래픽 가속 기능을 가지는 그래픽 카드를 사용하면 성능을 더욱 높일 수 있다. 현재 전 세계적으로 많이 쓰이는 표준 3차원 그래픽 라이브러리로는 OpenGL과 DirectX를 들 수 있다. 이 중에서 DirectX는 MS Windows 환경에서 멀티미디어 하드웨어를 매우 효율적으로 제어하게 해 주는 인터페이스를 제공하기 위해 개발되었다. 특히, Direct3D 는 3차원 그래픽 하드웨어의 3차원 동작 가속 기능을 이용하는 프로그램을 작성하는데 사용하는 API이다. 현재 판매되는 거의 모든 그래픽 카드는 3차원 가속 기능을 지원하고, 윈도우에서 동작하는 대부분의 3차원 게임은 Direct3D를 사용하여 구현되고 있다[6].

OpenGL은 90년대 초 Silicon Graphics(SGI)에서 개발한 그래픽 하드웨어 제어를 위한 소프트웨어 인터페이스이다. OpenGL은 높은 이식성과 빠른 실행 속도를 가진 3차원 그래픽 및 모델링 라이브러리이

며, 이를 통해 CRT 수준의 시각적인 화질을 가진 뛰어난 3차원 그래픽 영상을 만들 수 있다. 또한, OpenGL은 쉬운 문법과 체계적인 구조로 되어 있으며, 플랫폼 독립적으로 설계되어 MAC, OS/2, UNIX, LINUX, MS Windows 등 다양한 플랫폼을 모두 지원한다[7]. 그러므로 OpenGL은 오래전부터 다양한 3차원 응용 게임들과 과학, CAD 용으로 많이 사용되어 왔다[8]. MS Windows 환경에서는 다음 그림 1에서 보는 바와 같이 OpenGL API 호출이 하드웨어 드라이버를 통해 이루어져 있어 드라이버는 그 결과를 윈도우 GDI (Graphic Device Interface)에 보내는 것이 아니라 그래픽 출력 하드웨어의 드라이버 인터페이스에 보냄으로써 하드웨어 가속을 수행한다.

리눅스에서는 2차원 그래픽 처리를 위해 커널 프래임버퍼를 이용하거나 X 윈도우 시스템 환경을 사용하는 것이 일반적이다. 이런 환경에서 3차원 그래픽 응용을 위해서는 OpenGL과 같은 3차원 그래픽 API 라이브러리가 필요하다. 리눅스에서 이런 역할을 하는 것이 Mesa3D 공개 라이브러리이며, 특히 3차원 그래픽 가속 기능을 소프트웨어적으로 구현하여, 그래픽 하드웨어가 3차원 가속 기능을 지원하지 않더라도 OpenGL 응용의 지원이 가능하다[5,9].

하지만, 게임이나 고성능의 3차원 응용을 위해서는 하드웨어 3차원 가속 기능을 활용하여야 한다. 이런 어려움을 해결하기 위해 DRI (Direct Rendering Infrastructure)라는 통합 구조가 개발되었다[10]. DRI는 커널 및 X 서버에서 그래픽 하드웨어의 3차원 가속 기능을 직접적으로 제어할 수 있도록 함으로써, OpenGL을 사용할 때 더욱 빠른 3차원 가속 수행을 가능하게 한다. 즉, DRI를 사용함으로써 응용 프로그램에서 X 서버로 패킷을 보내지 않고 직접 하드웨어를 제어할 수 있게 된다.

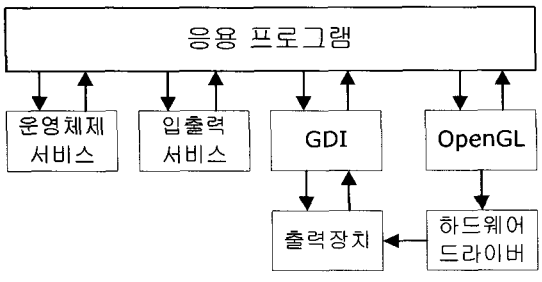

그림 1. 하드웨어 가속 기능 사용 시의 OpenGL

DRI는 X 서버, 직접 렌더링 클라이언트, 커널 장치 구동기 등과 같은 많은 모듈들을 유기적으로 구성한다[10].

모듈들은 각각 3차원 렌더링을 위한 하드웨어 직접 접근을 제공하며, 구성 요소에 대한 설명은 다음과 같다.

클라이언트는 X11을 사용하는 사용자 영역을 나타내며, 하드웨어에 의존적인 직접 렌더링을 지원하기 위한 여러 가지 모듈을 포함한다. 이러한 모듈로는 장치 및 운영체제에 독립적인 가속 기능 및 GLX 조절을 처리하는 표준 OpenGL 라이브러리 libGL.so, DRI를 구동하는 libDRI.so, libHW3D.so 등을 예로 들 수 있다.

X 서버는 DRI를 지원하기 위한 사용자 영역의 그래픽 윈도우 서버이며 장치 및 운영체제에 독립적인 코드로 수정되어 있다. X 서버는 3차원 가속 기능을 처리하는 DRI 드라이버의 일부분인 libDRI.so, 하드웨어에 특정한 2차원 렌더링, 3차원 초기화 및 소멸 루틴을 제공하는 libH2D.so 등을 활용한다.

DRM (Direct Rendering Manager) 은 커널 내에서 3차원 하드웨어 드라이버에 해당하는 모듈이며, DMA, AGP 메모리 관리, 자원에 대한 동기화/상호 배제, 하드웨어 접근 보안 등을 처리한다.

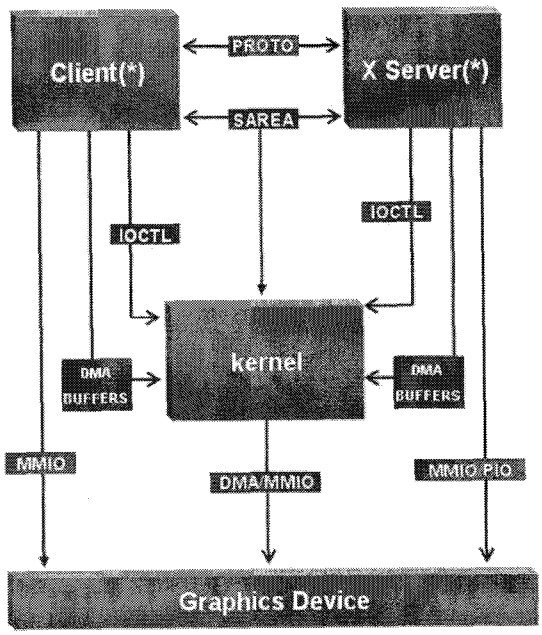

그림 2. DRI 주요 통신 경로

X 서버와 클라이언트는 표준 X 프로토콜 전송 계층 프로토콜 (PROTO)을 통하여 정보를 주고받으며, DRI를 수행하기 위하여 커널 내의 특정한 공유 메모리 영역 (SAREA)을 활용하여 X 서버와 클라이언트 간 정보 전달이나 커널의 상태 정보를 공유한다.

또한, X 서버와 클라이언트는 메모리 매핑 입출력 (MMIO)을 통하여 그래픽 장치로 직접 접근이 가능하다. 하지만, 일반적으로 그래픽 장치는 커널을 통하여 접근하며, DMA 버퍼와 IOCTL 과 같은 방식을 사용한다. 여기서, DMA 버퍼는: DMA를 통해 하드웨어로 보낼 그래픽 장치 명령어를 저장하기 위해 사용되는 메모리 영역이며, IOCTL은 커널 장치 구동기를 위한 특별한 인터페이스로서 시스템 호출과 정보 전달을 위한 메모리 복사에 의한 초과비용을 초래하며, 커널이 장치 입출력을 통해 사용자 영역 응용의 신호 감지 능력도 포함하고 있다.

3. 임베디드 그래픽 가속 장치 구동기

리눅스의 대표적인 그래픽 윈도우 시스템인 X 윈도우는 클라이언트/서버 구조의 그래픽 서버로서, 소스가 공개되어 있고 안정성이 높으며 다양한 툴킷을 수용할 수 있다[11]. 하지만, 범용으로 개발되어 기능이 복잡한 대신 크기가 크고 속도가 느리기 때문에 그동안 임베디드 용으로 적합하지 않았으나, 최근 점차 임베디드 하드웨어의 사양이 고급화되고 3차원 게임과 같은 고급 그래픽 응용에 대한 사용자의 요구가 커지면서, X 서버 크기를 1 메가 바이트 이하로 줄인 Tiny X 서버가 임베디드 시스템 용으로 다시 주목받고 있다[3].

본 논문에서는 Tiny X 환경에서 현재 지원하지 않는 3차원 그래픽 가속 기능을 지원하기 위하여, 표준 X 윈도우에서 OpenGL을 지원하는 방식을 적용하여 3차원 그래픽 가속 장치 구동기를 구현하였다.

```
/* 그래픽 카드 설정 함수들 */
Bool (*cardinit) (KdCardInfo *);                    /* 장치 검색과 메모리 매핑 */
Bool (*scrinit) (KdScreenInfo *);                   /* ScreenInfo 구조체 초기화 */
Bool (*initScreen) (ScreenPtr);                     /* ScreenRec 초기화 */
Bool (*finishInitScreen) (ScreenPtr );  /* ScreenRec를 끝냄 */
void (*preserve) (KdCardInfo *);                    /* 그래픽 카드 상태 저장 */
Bool (*enable) (ScreenPtr);                         /* 그래픽 렌더링 모드 설정 */
Bool (*dpms) (ScreenPtr, int);                      /* DPMS 전원 관리 모듈 설정 */
void (*disable) (ScreenPtr);                        /* 렌더링 모드를 끔 */
void (*restore) (KdCardInfo *);                     /* 그래픽 카드 상태 복구 */
void (*scrfini) (KdScreenInfo *);                   /* 화면을 닫음 */
void (*cardfini) (KdCardInfo *);                    /* 그래픽 카드 해제 */
/* 하드웨어 커서 제어 함수들 */
Bool (*initCursor) (ScreenPtr);                     /* 하드웨어 커서 검색 초기화*/
void (*enableCursor) (ScreenPtr);                   /* 하드웨어 커서 활성화 */
void (*disableCursor) (ScreenPtr);                  /* 하드웨어 커서 비활성화 */
void (*finiCursor) (ScreenPtr);                     /* 하드웨어 커서를 끔 */
void (*recolorCursor) (ScreenPtr, int, xColorItem *); /* 하드웨어커서recolor */
/* 그래픽 카드 가속 함수들 */
Bool (*initAccel) (ScreenPtr);                      /* 그래픽가속 기능 초기화 */
void (*enableAccel) (ScreenPtr);                    /* 그래픽가속 기능 활성화 */
void (*syncAccel) (ScreenPtr);                      /* 그래픽가속 sync 함수 */
void (*disableAccel) (ScreenPtr);                   /* 그래픽가속 기능 비활성화 */
void (*finiAccel) (ScreenPtr);                      /* 그래픽가속 기능 끔 */
/* pseudo color 색 설정 함수들 */
void (*getColors) (ScreenPtr, int, int, xColorItem *); /* pseudo color 얻음 */
void (*putColors) (ScreenPtr, int, int, xColorItem *); /* pseudo color 설정 */
```

그림 3. Tiny X 의 그래픽 장치 구동 함수들

3.1 자료 구조

Tiny X는 작은 메모리 용량에서도 잘 동작하도록 기존의 표준 XFree86 서버의 하부를 대폭 수정하고 줄여 작은 용량을 가지도록 구현된 임베디드 시스템용 X 서버이다[11,12]. 특히, 표준 XFree86 서버가 지원하는 모듈 기능을 없앴으며, 지원하는 X 확장 기능도 한정되어, 일반적인 Tiny X 서버의 크기는 x86 CPU에서 수행할 때 1 메가 바이트 정도이다. 현재 Tiny X는 1, 4, 8, 16, 24, 32 bpp(bit per pixel) 흑백/칼라 모드를 실행하는 기존 cfb/mfb 코드보다 더 작은 새로운 프레임버퍼 코드를 사용한다. 기본적인 Tiny X 서버는 프레임버퍼 또는 VESA 표준 그래픽 입출력에 기반한 Xfbdev 서버와 Xvesa 서버 2가지가 있으며, 기타 igs, mach64, iPAQ, trident, pcmcia 등의 그래픽 장치를 위한 서버를 지원한다.

Tiny X 그래픽 장치 구동기는 간단한 자료 구조를 제공하는데, 카드 정보 구조체와 화면 정보 구조체를 설계하는 것이 필요하다. 카드 정보 구조체는 그래픽 카드가 기본적으로 제공하는 드라이버(VESA, fb 등), 그래픽 레지스터들(VGA 등의 일반 그래픽 레지스터들 및 기타 전용 그래픽 레지스터들), 그래픽 카드의 IO 및 프레임버퍼 주소, 프레임버퍼 크기 등의 정보를 가진다. 화면 정보 구조체는 하드웨어 커서를 사용할 경우 관련된 정보들과 그래픽 가속을 위한 오프스크린(offscreen) 패턴 및 타일 정보 등을 포함할 수 있다.

3.2 기본 기능

Tiny X 서버가 실행되면 호출되는 장치 수준의 원 (stub) 함수는 크게 세 가지인데, 그래픽 장치 구동기의 일반 함수들을 등록하는 InitCard(), 출력 장치 구동기를 초기화하는 InitOutput(), 입력 장치 구동기를 초기화하는 InitInput() 루틴들이다.

그래픽 장치 구동기의 주요 모듈은 KdCardFuncs 구조체에 등록되는 함수들인데, 기본 형식은 다음 그림과 같다.

3.3 2차원 그래픽 가속 기능

Tiny X 그래픽 장치 구동기의 2차원 그래픽 가속 기능은 표준 XFree86 서버의 그래픽 가속 구조인 XAA (X Accerelation Architecture)를 대폭 간소화한 KAA (Kdrive Acceleration Architecture) 구조를 이용하여 구현하였다[3]. 가속 기능은 주로 화면 상의 창과 같은 Drawable이나 그래픽 컨텍스트(Graphic Context)를 통하여 구현할 수 있다.

ScreenRec에 등록되는 함수들 중 CreateGC를 제외하고는 화면 창 단위로 그래픽 가속 기능을 수행한다[9]. 대표적으로, CopyWindow() 함수는 창 이동 시 실행되는 함수이고, PaintWindowBackground()와 PaintWindowBorder() 함수는 창의 배경색 또는 배경 그림, 테두리 등을 그리는 기능이다[9].

그래픽 컨텍스트를 통해 구현되는 그래픽 가속에 관련된 함수들은 CreateGC() 함수에 의해 생성된 GC와 GCFuncs 구조체에 의해 GCOps 구조체로 등록되어 그래픽 컨텍스트 내에서 구현된다[3].

3.4 3차원 그래픽 가속 기능

Tiny X에서는 Mesa3D OpenGL 라이브러리와 X 서버와 3차원 그래픽 응용 프로그램 사이의 프로토콜인 GLX 모듈을 이용하여 간접 렌더링 방식을 구현할 수 있다. 이러한 구조는 다음 그림 4와 같다.

하지만, 이런 구조는 3차원 그래픽 처리를 위한 하드웨어 가속 기능을 사용하지 않기 때문에 빠른 연산을 기대하기 힘들다. 그래서 앞서 살펴본 직접 렌더링을 위한 DRI 구조를 Tiny X 상에서 구현함으로써 3차원 그래픽 가속 성능을 향상시킬 수 있다. 즉, Tiny X가 구동된 상태에서, 2차원 및 그래픽 하

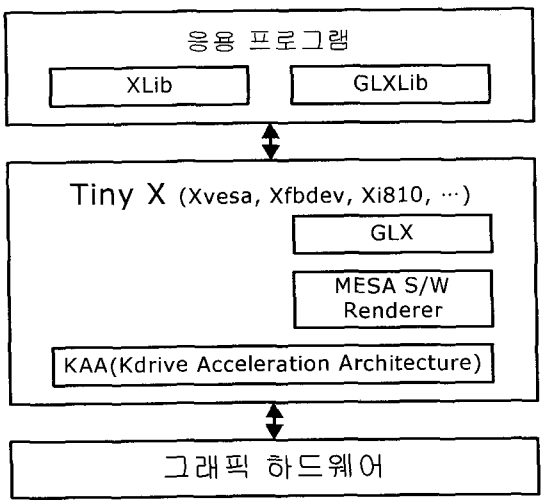

그림 4. 간접 렌더링 Tiny X 환경

드웨어에서 지원하지 않는 일부 3차원 그래픽 처리에 대해서는 앞서 설명한 KAA 구조를 통하여 실현 가능하고, 3차원 그래픽 가속 기능을 지원하는 3차원 그래픽 처리 요청에 대해서는 Tiny X 서버 내부에 구성한 DRI 및 DRM 초기화 루틴을 통하여 등록된 처리 함수들을 통하여 커널 및 그래픽 하드웨어에 연결되어 처리된다. 다음 그림 5 는 본 논문에서 구현한 Intel i830 계열 그래픽 하드웨어를 위한 Tiny X 용 Xi830 서버의 구조를 보여준다.

3차원 그래픽 서버의 핵심은 초기화 과정을 통하여 2차원 및 3차원 그래픽 처리를 위한 준비 작업을 수행하는 것이다. 본 논문에서 구현한 Xi830 서버의 초기화 동작은 그래픽 장치를 초기화하는 것, 그래픽 처리를 위해 비디오 메모리를 할당하는 것, DRI 구조 초기화, 그래픽 장치를 활성화 시키는 것 등 크게 네 가지로 구분할 수 있다. 다음 그림 6 은 Xi830 서버의 드라이버 함수 흐름을 보여준다. 그림에서 화살표는 호출 관계를 나타낸다.

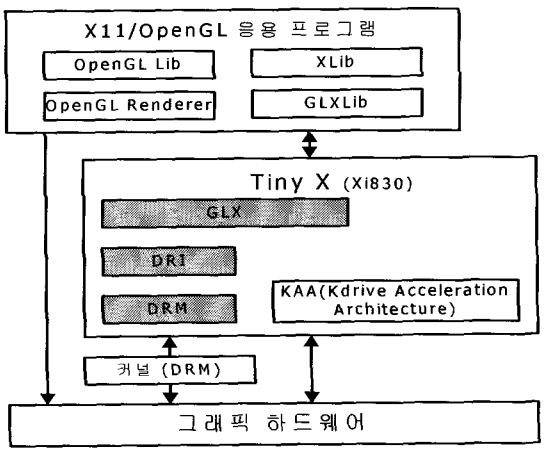

그림 5. 직접 렌더링 Tiny X 환경

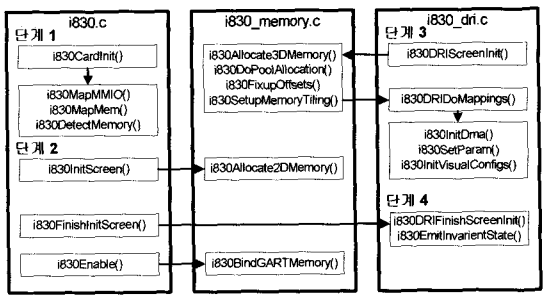

그림 6. Xi830 드라이버 함수 흐름도

다음은 Xi830 서버의 초기화 단계를 설명한 것이다.

가. 그래픽 장치 초기화 (단계 1)

Xi830 서버가 실행되면 가장 먼저 그래픽 장치를 초기화하고 사전 할당 메모리(Pre-Allocated Memory)의 크기와 프레임버퍼의 최대 크기를 계산하고 MMIO를 설정한다. Intel 82855 GME 칩셋의 사전 할당 메모리(Pre-Allocated Memory) 크기는 1MB, 8MB, 16MB, 32MB 중에서 System BIOS 설정으로 선택되는데, 그래픽 장치 초기화를 수행하는 i830CardInit() 에서 i830DetectMemory()에 의해 사전 할당 메모리 크기를 얻어낸다. 커서에 대한 물리 주소 필요여부와 링 버퍼의 낮은 주소 할당 필요 여부를 결정한 후, 프레임버퍼의 최대 공간을 계산하는데 그 수식은 아래와 같다.

프레임버퍼 최대용량 = 비디오메모리 크기 - (하드웨어커서 + 오버레이 + 링버퍼 + 스크래치 버퍼) 크기

i830MapMMIO( )와 i830MemMap( ) 함수들을 통해 프레임버퍼 비디오 레지스터 영역에 대한 MMIO(Memory Mapped IO)를 설정한다.

나. 화면 초기화 (단계 2)

그래픽 장치가 초기화된 후 2차원 그래픽 처리를 위한 비디오 메모리 할당 및 VESA 표준 모드 화면 초기화 과정을 수행한다. 화면 초기화는 i830-ScreenInit()에서 시작되는데, X 서버 부팅 시 설정한 화면 크기와 깊이(Depth)/bpp 값을 받아 설정할 VESA 모드에 따라 RandR (Rotation and Resize)과 그림자(Shadow) 기능 초기화를 수행하고, 깊이 값에 따른 시각 (Visual) 모드를 선택한다. 그런 후에 메모리 Aperture(그래픽 메모리에 접근을 시작할 연속적인 가상 메모리 공간)를 지정하고, 우선 할당 메모리를 고정시킬 영역을 설정한다.

위와 같이 메모리 영역이 정해지면, 다음으로 가용한 2차원과 3차원 버퍼 메모리들을 할당한다. 먼저, 2차원 비디오 메모리 할당은 i830Allocate-2DMemory()를 호출함으로써 이루어지는데, 이는 프레임버퍼, 링(Ring) 버퍼, 스크래치(scratch) 메모리, 하드웨어 커서에 대한 초기화 과정을 포함한다. 이러한 과정들을 통하여 VESA 표준 모드의 화면 초기화 과정이 완료된다.

다. DRI 초기화 (단계 3)

화면 초기화 과정이 완료되면 DRI 화면 모드로 다시 초기화하여, 3차원 그래픽에 필요한 각종 버퍼 등에 대한 메모리를 확보하고, 시각 설정 초기화를 통해 GLX 컨텍스트에 필요한 과정을 완료한다. DRI 초기화는 i830DRIScreenInit()에서 시작되는데, 먼저, DRI 버전을 검사하고 커널의 DRM 드라이버 이름과 응용 드라이버 이름을 지정한 다음, 앞의 화면 초기화에서 계산한 수치들로 DRI 구조체의 자료를 채우는데, 이 때 채워지는 자료들은 버스 ID, DDX 드라이버 버전, 프레임버퍼의 물리적인 주소, 프레임버퍼 크기, DDX draw 영역 테이블, SAREA 크기와 같은 값과 컨텍스트(Context) 생성 및 파괴, 버퍼 초기화, 이동, 전체 화면, 2차원/3차원 변환, 단일/다중 창 변환 등 그래픽 처리에 필요한 함수들을 지정한다.

함수 지정이 완료되면 DRM 버전을 검사한 후, i830Allocate3DMemory()를 호출하여 가용할 3차원 버퍼들의 메모리 할당을 시작한다. 먼저, 후면(Back) 버퍼를 할당하고, 메모리 타일을 최적화한 다음, 후면 버퍼와 같은 크기로 깊이(Depth) 버퍼를 할당한다. 그런 다음 컨텍스트를 위한 논리 공간과 텍스처(Texture) 버퍼를 할당한다.

i830DoPoolAllocation()을 호출하여, 우선 할당된 영역을 제외한 다른 공간에 메모리 풀(Momery Pool)을 할당한 다음, i830FixupOffsets()을 호출하여, 메모리 풀의 가장 상위 지점인 우선 할당 메모리의 끝에서 전면 (Front) 버퍼, 커서 메모리, 링 버퍼, 스크래치 메모리, 오버레이 메모리, 후면 버퍼, 깊이 버퍼, 컨텍스트 메모리, 텍스처 메모리의 오프셋을 고정시킨다. 그리고 i830SetupMemoryTiling()을 호출하여 메모리 타일을 설정하는데, 이 메모리 타일은 후면 버퍼와 깊이 버퍼를 위해 동적으로 생성된다.

i830DRIDoMappings()를 호출하여, 레지스터, 후면 버퍼, 깊이 버퍼, DMA 버퍼, AGP 버퍼, 링 버퍼, 텍스처 버퍼를 최종적으로 초기화한다. 다음으로 그래픽 장치의 버스 ID로부터 인터럽트를 수령하여 인터럽트 핸들러를 지정하고, AGP 버스 관리를 위한 간단한 메모리 관리자를 생성한다.

마지막으로, i830InitVisualConfigs()를 호출하여 깊이 값(16 혹은 24)에 따른 RGB 자료를 설정하여, Visual 초기화를 완료한다.

라. 그래픽 장치 활성화 (단계 4)

이 단계에서는 화면과 전력관리 모드(DPMS)를 설정하고 그래픽 장치를 활성화시킨다. i830-FinishInitScreen()에서 i830DRIFinishInit()을 호출하여 DRI 그림자 버퍼를 갱신하고, 페이지플립(PageFlip)을 활성화시켜 DRI 기능을 위한 초기화 과정을 완료한다. 그리고 i830EmitInvarientState()를 호출하여 컨텍스트 주소를 지정함으로써 최초 컨텍스트를 위한 초기화 과정이 완료된다.

vesaFinishInitScreen()이 호출되어 최종적으로 화면 설정 및 계층(layer)을 생성하고, RandR 기능을 초기화한 다음 i830Enable()에서 VESA 그래픽 모드 활성화 및 MMIO 설정 재확인을 수행한 후, i830BindGARTMemory()를 호출하여 우선 할당 메모리를 다시 결합한다. 마지막으로 DPMS(전력 관리) 모드를 설정하면, 그래픽 장치가 최종적으로 활성화된다.

4. 3차원 그래픽 처리 성능 평가

그래픽 하드웨어 가속 기능에 따르는 3차원 그래픽 처리 성능을 평가하기 위하여, 일반적으로 그래픽 자료를 처리하는 능력을 측정하는 하드웨어적인 성능에 관한 측정과 종합적인 3차원 그래픽 처리와 관련된 렌더링 처리에 관한 성능 측정을 수행할 수 있다. 본 논문에서는 종합적인 3차원 렌더링 처리에 관한 성능을 측정하고 평가하였다.

4.1 성능 평가 프로그램

본 연구에서는 그래픽 처리 시뮬레이션을 통해 3차원 그래픽 처리 성능을 평가할 수 있는 성능 평가 프로그램을 작성하였다. 프로그램은 각각의 평가요소에 필요한 작업을 수행하여 1000여 개의 큐브로 표현하는 능력을 초당 프레임 수, 즉 FPS (Frame per seconds) 수치로 측정하도록 하였으며, 전체적인 프로그램 구조와 수행 과정은 다음 그림 7 과 같다.

가. 렌더링 선처리 모듈

선처리 모듈은 FPS를 측정하기 위한 환경을 구축한다. 실제 FPS 측정에 포함되지 않거나반복 수행이 필요없는 작업을 수행하며 환경 구축이 완료되면, 다음 모듈로 제어를 옮겨 작업을 수행하게 된다.

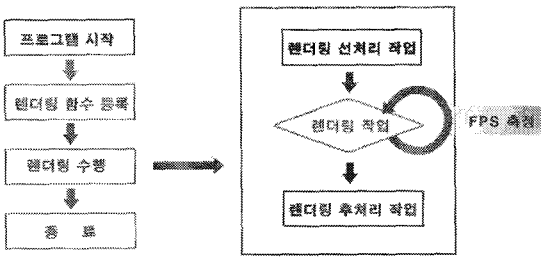

그림 7. 성능 평가 프로그램 수행 과정

아래 예에서는 반복 수행이 필요없는 작업들을 나열하고 있다.

- ○ 광원을 위치시키고, 광원의 값을 설정한다.
- ○ 알파블랜딩(Alphablending) 기능을 켠다.
- ○ 텍스처 파일을 읽어오고, 텍스처 기능을 설정한다.
- ○ 압축 텍스처 기능을 켠다.
- ○ 화면 상에 위치시킬 물체의 위치를 초기화한다.
- ○ 파일로부터 출력될 메시지의 내용을 설정한다.
- ○ FPS를 측정하기 위한 타이머를 초기화한다.
- ○ 디더링 (Dithering) 기능을 켠다.

나. 렌더링 측정 모듈

측정 모듈은 선처리 모듈에서 구축된 환경에서 측정 작업을 수행한다. 일정시간 동안 FPS를 측정하며, 측정이 끝나면 후처리 모듈로 제어를 옮겨 작업을 수행하게 된다. 측정 모듈에서는 다음과 같은 일들을 수행하게 된다.

- ○ 후면 버퍼의 내용을 지운다.
- ○ 색상 버퍼의 내용을 지운다.
- ○ 물체의 색상을 지정한다.
- ○ 각각의 큐브들이 위치할 좌표를 계산한다.
- ○ 각각의 물체를 후면 버퍼에 그린다.
- ○ 전면 버퍼와 후면 버퍼의 교체작업을 수행한다.
- ○ FPS를 측정하기 위해 프레임 카운터를 증가시킨다.
- ○ 일정시간 간격으로 FPS를 창에 출력한다.

다. 렌더링 후처리 모듈

후처리 모듈은 측정된 FPS에서 평균을 계산한 다음 출력 모듈을 통해 평균 FPS를 기록하게 된다. 그 외에도 해당하는 렌더링 처리 작업에서 사용한 자원을 회수하는 작업과 다음에 검사할 요소에 영향을 미치지 않도록 환경을 마련해 주어야 한다.

- ○ 측정된 FPS로부터 평균 FPS를 구하여 로그 파일에 출력할 수 있도록 한다.
- ○ 광원 기능을 끈다.
- ○ 알파블랜딩 기능을 끈다.
- ○ 텍스처와 관련된 자원을 회수하거나 텍스처 기능을 끈다.

라. 렌더링 함수 등록 모듈

등록 모듈은 실행 모듈 목록의 비어 있는 주소를 가져와 선처리 모듈, 측정 모듈, 후처리 모듈을 등록한다. 실행 모듈 목록은 세 가지로 구성되며, 선처리 모듈을 등록하는 목록, 측정 모듈을 등록하는 목록, 후처리 모듈을 등록하는 목록으로 구성된다.

마. 제어 모듈

제어 모듈은 등록된 모듈을 반복해서 호출하며, 제어권 이동 명령을 받으면, 목록에 등록된 다음 모듈을 반복 호출을 하게 된다. 목록에 등록된 마지막 모듈이 호출되고, 제어권 이동 명령을 받게 되면, 프로그램을 종료하는 모듈을 호출하게 된다.

바. 출력 모듈

출력될 결과를 하나의 파일에 기록이 가능하도록 하며, 파일 열기, 파일 닫기, 결과 저장 작업 등을 대신 수행한다.

사. 렌더링 평가 요소

구체적으로 하드웨어 성능 평가의 바탕인 지오메트리 연산 능력을 측정하기 위해 객체의 이동, 크기 변환, 회전을 통한 변환 (Transform) 요소와 반사광, 분산광, 주변광, 재질광을 통한 광 처리 (Lighting) 요소를 평가하고, 3차원 그래픽 응용 분야의 필수 요소인 텍스처 검사, 멀티텍스처 검사, 메모리 효율을 위한 압축된 텍스처 검사, 투명 효과에 사용되는 알파블렌딩 기능을 평가 요소로 선정하였다.

4.2 성능 평가 결과

본 논문에서 제안하는 3차원 그래픽 가속 장치 구동기의 성능 평가를 위해 인텔사의 i855 GME 임베디드 칩셋이 장착된 i855GMEm-LFS 보드를 사용하여 Pentium-M 모바일 프로세서 기반의 임베디드 PC를 사용하였다. 이 보드에는 인텔의 익스트림 그래픽스 2 가 사용되어, 2차원, 3차원 그래픽 및 동영

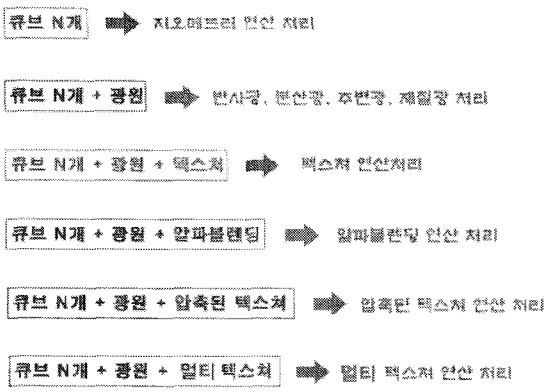

그림 8. 성능 평가 요소

상 압축 가속 기능을 구현하고 있다. 본 논문에서는 리눅스 커널 버전 2.6.9 위에서 인텔 i830 계열 그래픽 칩셋을 위한 Tiny X 서버인 Xi830을 개발하여 구동시키고, 앞서 제작한 성능 평가 프로그램을 실행하였다. 다음 그림 9 는 텍스쳐 렌더링을 수행하는 모습을 캡쳐한 것이다.

각 평가 요소로 구성된 렌더링을 수행하면서 표준 XFree86 서버의 직접 렌더링 16/24 비트 모드, Xi830 서버의 간접 렌더링 16/24 비트 모드, 직접 렌더링 16/24 비트 모드 환경 하에서 FPS 수치를 측정하였다. 측정 결과는 다음 그림 10, 11, 12 와 같으며, 각 그림 당 2개의 평가 요소를 포함하고 있다.

Xi830 24 비트 모드에서의 간접 렌더링 모드와 직접 렌더링 모드를 비교한 결과, 직접 렌더링 모드가 큐브, 광원, 텍스쳐, 알파블렌딩, 압축된 텍스쳐, 다중

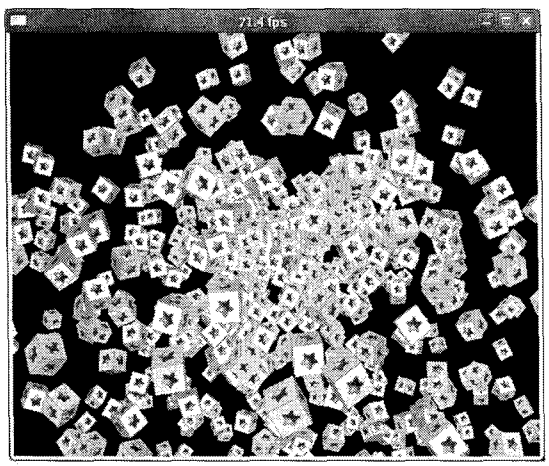

그림 9. 성능 평가 프로그램 실행 화면

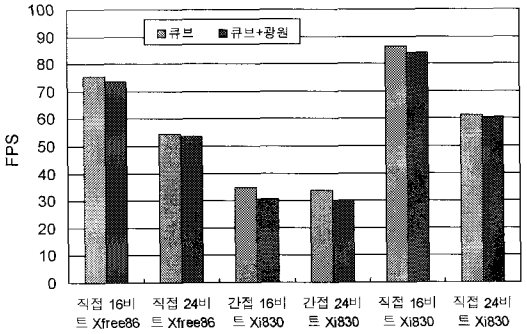

그림 10. 큐브, 큐브+광원 렌더링 측정 결과

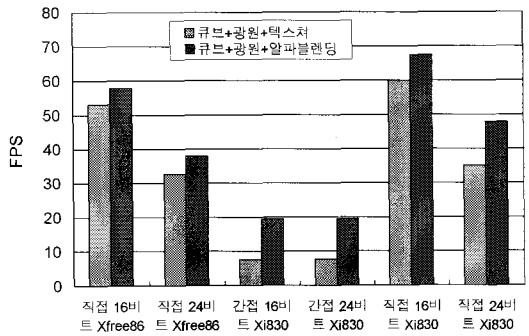

그림 11. 큐브+광원+텍스쳐, 큐브+광원+알파블렌딩 측정 결과

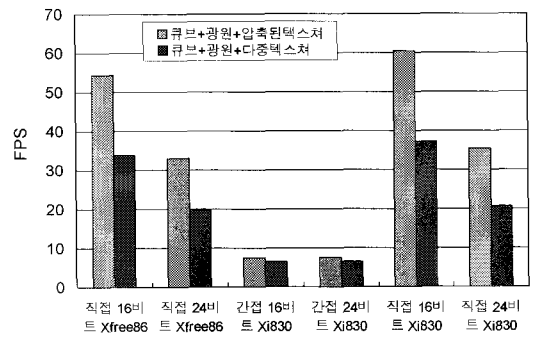

그림 12. 큐브+광원+압축된텍스쳐, 큐브+광원+다중텍스쳐 측정 결과

텍스쳐 테스트에서 각각 82%, 102%, 359%, 145%, 371%, 219% 만큼 향상된 수치를 보였다. 즉, 간접 렌더링 모드에 비해 직접 렌더링 모드가 3차원 그래픽 처리 성능이 월등히 높아진 것을 알 수 있다. 이것은 간접 렌더링 시에 소프트웨어적으로 순차적으로 수행되는 OpenGL 연산들에 비해 직접 렌더링 시에

는 그래픽 가속 하드웨어에서 수행되는 고속의 3D 그래픽 연산으로 대체되어 수행되기 때문이다.

표준 XFree86 24 비트 모드와 Xi830 24 비트 모드에서 수행된 3D 그래픽 처리 성능 비교에서는 Xi830 24 비트 모드가 각각의 테스트에서 13%, 13%, 7%, 26%, 8%, 4% 가량 향상된 성능을 보여주는데, 이것은 두 모드에서 수행되는 3D 그래픽 처리 과정은 비교적 유사하지만 Xi830 그래픽 서버의 세부 구조와 기능이 표준 XFree86 서버에 비해 훨씬 경량화되어 더욱 간결하게 그래픽 처리가 수행된 결과임을 알 수 있다. 뿐만 아니라, 같은 Xi830 서버이더라도 16 비트 모드로 구동하는 경우가 24 비트 모드에 비해 기본적인 픽셀 처리 단위가 작으므로 일반적으로 더 우수한 성능을 나타낸다.

5. 결론 및 추후 연구 과제

본 논문에서는 임베디드 시스템을 위한 3차원 그래픽 응용 프로그램을 지원하기 위하여 임베디드 리눅스 기반의 Tiny X를 위한 3차원 그래픽 가속 장치 구동기를 구현하는 방식을 제안하였다. 실제로 구현한 모듈은 단계적인 초기화되어 그래픽 하드웨어가 제공하는 다양한 2차원 및 3차원 그래픽 가속 기능들을 활용하여 소프트웨어적으로 처리하는 간접 처리 방식에 비해 그래픽 처리 성능을 2배 이상 크게 향상시켰으며, 다양한 3차원 그래픽 평가 요소에 대해 간단하면서도 효율적으로 구성된 성능 평가 프로그램을 통하여 성능을 평가하고 검증하였다.

본 논문을 통해 개발된 핵심 요소 기술은 임베디드 리눅스 기반의 3차원 그래픽 처리에 관한 기반 기술로서 사용 가능하며, 제안된 3차원 그래픽 처리 성능 평가 프로그램은 임베디드 시스템에서 다양한 3차원 그래픽 처리 기능을 복합적으로 평가하는 데 활용할 수 있다.

향후에는 구현된 시스템 상에서 OpenGL 기반의 고속 3차원 그래픽 처리 및 게임 응용 프로그램을 개발하고 향후 임베디드 시스템의 3차원 그래픽 처리에 대한 평가체계를 더욱 발전시킬 예정이다. 또한, 현재 2차원 그래픽 처리를 포함한 모든 그래픽 처리를 OpenGL과 같은 표준 그래픽 API로 구현하는 방법들이 진행 중에 있으므로 본 논문에서 제안한 방식도 향후에는 이러한 방식에 적용 가능하도록 개선시켜 나갈 것이다[13].

참 고 문 헌

[1] 김선자, 김홍남, 김채규, "정보가전용 임베디드 운영체제 기술," 한국통신학회지, 제18권 제12호, pp. 72-81, 2001.

[2] 김성우, 권오준, 김태석, "임베디드 그래픽 윈도우 시스템 기술," 한국멀티미디어학회지, 제6권, 제2호, pp. 29-35, 2002.

[3] E. Anholt, "High Performance X Server in the Kdrive Architecture," *Proceedings of the USENIX Annual Technical Conference*, pp. 41-50, 2004.

[4] 이범렬, 류성원, 이은주, 박재형, "모바일 3D API 기술 표준화 연구," 전자통신동향분석, 제20권, 제4호, pp. 110-119, 2005.

[5] The Mesa 3D Graphic Library Homepage, http://www.mesa3d.org.

[6] P.J. Kovach, *Inside Direct3D*, Microsoft Press, 2000.

[7] M. Segal, K. Akeley, and J. Leach (ed), *The OpenGL Graphics System: A Specification*, SGI, 1999.

[8] R.S. Wright and B. Lipchak, *OpenGL Super-Bible, 3rd ed.*, Sams, 2004.

[9] S. Angebranndt, R. Drewry, P. Karlton, T. Newman, B. Scheifler, K. Packard, and D. P. Wiggins, *Definition of the Porting Layer for the X v11 Sample Server*, X Window Consortium, 1994.

[10] DRI Homepage, http://dri.freedesktop.org.

[11] X.org Foundation Homepage, http://www.x.org.

[12] XFree86 Homepage, http://www.xfree86.org

[13] K. Packard, "Getting X Off The Hardware," *Proceedings of the Linux Symposium*, pp. 381-390, 2004.

김 성 우

1991년 한국과학기술원 전기 및 전자공학과 졸업 (공학사)
1993년 한국과학기술원 전기 및 전자공학과 졸업 (공학석사)
1999년 한국과학기술원 전기 및 전자공학과 졸업 (공학박사)
1999년~2001년 한국전자통신연구원 선임연구원
2002년~현재 동의대학교 컴퓨터소프트웨어공학과 조교수
관심분야 : 임베디드 그래픽스, 센서 네트워크, 지능제어

이 종 민

1992년 경북대학교 컴퓨터공학과 졸업 (공학사)
1994년 한국과학기술원 전산학과 졸업 (공학석사)
2000년 한국과학기술원 전자전산학과 졸업 (공학박사)
1997년~2002년 삼성전자 무선사업부 책임연구원
2005년 University of California at Santa Cruz, Research Associate
2002년~현재 동의대학교 컴퓨터소프트웨어공학과 조교수
관심분야 : 모바일 컴퓨팅, 라우팅, 센서 네트워크

이 중 화

1992년 부산대학교 전자계산학과 졸업 (이학사)
1995년 부산대학교 전자계산학과 졸업 (이학석사)
2001년 부산대학교 전자계산학과 졸업 (이학박사)
2002년~현재 동의대학교 컴퓨터소프트웨어공학과 조교수
관심분야 : 데이터베이스, XML, 시맨틱 웹 등