Journal of Korea Multimedia Society Vol. 11, No. 10, October 2008(pp. 1460-1470)

모바일 벡터 그래픽을 위한 OpenVG 가속기 설계

김영옥[†], 노영섭[††]

요 약

최근 휴대용 기기의 성능이 향상되면서 다양한 형태의 메뉴 구성과, 메일 및 이차원 지도 등의 표현에 벡터 그래픽을 많이 도입하고 있다. 본 논문은 모바일 기기에서 많이 사용되고 있는 이차원 벡터의 처리 기술인 OpenVG (Open Vector Graphics)의 하드웨어 가속기를 제안했다. 제안된 하드웨어 가속기는 그래픽에서 처리가 빈번한 렌더링(rendering)의 각 기능을 분석하여 하드웨어 구현에 적합하도록 나누고, 그 알고리즘을 설계 및 검증하여 HDL (Hardware Description Language)로 FPGA (Field Programmable Gate Array)에 이식하여 구현되었으며, 알렉스 처리기에 비하여 약 4배의 빠른 처리속도를 보였다.

Design of Open Vector Graphics Accelerator for Mobile Vector Graphics

Young-Ouk Kim[†], Young-Sup Roh[††]

ABSTRACT

As the performance of recent mobile systems increases, a vector graphic has been implemented to represent various types of dynamic menus, mails, and two-dimensional maps. This paper proposes a hardware accelerator for open vector graphics (OpenVG), which is widely used for two-dimensional vector graphics. We analyze the specifications of an OpenVG and divide the OpenVG into several functions suitable for hardware implementation. The proposed hardware accelerator is implemented on a field programmable gate array (FPGA) board using hardware description language (HDL) and is about four times faster than an Alex processor.

Key words: OpenVG(오픈브이지), Gradient(경사), Stroke(외곽선), Pattern(패턴), SVG(에스브이지)

1. 서 론

벡터 그래픽은 주어진 이차원이나 삼차원 공간에 선이나 형상을 배치하는데 있어 일련의 명령들이나 수학적 표현을 통해 디지털 영상을 만드는 것이다. 벡터 그래픽에서는 사용자들의 창작활동 결과물인 그래픽 파일이, 일련의 벡터 서술문의 형태로 창작되고 저장된다. 예를 들면, 벡터 그래픽 파일에는 선을 그리기 위해 각 비트들이 저장되어 있는 대신에, 연결될 일련의 점들의 위치가 들어 있다. 그로 인해 파일 크기가 작아지는 결과를 가져오게 된다. 또한 기존 비트 맵 등으로 구성된 영상은 확장되거나 변경되는 경우 영상의 외곽 형태에 계단현상이 발생하여 사실적이거나 회화적인 표현이 제한될 수밖에 없다. 하지만 벡터로 구현된 영상은 확대되거나 축소되는 경우에도 수학적인 연산을 통하여 위와 같은 문제점을 극복할 수 있어 자연스럽게 표현될 수 있는 장점이 있다.

이런 이유로 인하여 최근 휴대용 기기에서는 사용자 인터페이스를 기존의 비트 맵 방식에서 벗어나 벡터 위주 혹은 벡터와 삼차원 그래픽의 혼용으로 표현하기도 한다. 이와 같은 벡터 그래픽 처리 기법

※ 교신저자(Corresponding Author) : 노영섭, 주소 : 서울시 강남구 삼성동 37-18(135-090), 전화 : 02)3470-5282, FAX : 02)3470-5282, E-mail : ysroh@suv.ac.kr
접수일 : 2008년 5월 28일, 완료일 : 2008년 9월 26일

[†] 정회원, 서울벤처정보대학원대학교 임베디드시스템학과 (E-mail : sola16@gmail.com)
[††] 정회원, 서울벤처정보대학원대학교 임베디드시스템학과교수

은 휴대폰, PDA(Personal Digital Assistant), PMP(Portable Multimedia Player)등의 모바일 기기 위주의 디지털 융합 제품에서 많이 적용되고 있다.

벡터 그래픽을 처리하기 위한 방법으로는 소프트웨어 렌더링(rendering) 방식의 플래시(Flash)와 W3C(World Wide Web Consortium)의 표준인 SVG(Scalable Vector Graphics)[1] 재생기가 있다. 플래시는 매크로미디어사의 제품으로 국제적인 벡터 표준은 아니며 매크로미디어사만의 독자적인 파일 형식을 이용하여 벡터를 처리한다. SVG 재생기는 W3C의 승인을 거친 국제 표준의 벡터 처리 방식이다.

최근 휴대용 기기의 디지털 융합으로 인해 여러 분야에서 벡터 그래픽의 콘텐츠를 운용하려 하나 소프트웨어 렌더링(rendering) 방식의 느린 처리 속도로 인하여 제한적으로 사용되어 활용도가 떨어지고 있다. 따라서 제한된 연산 능력을 보유한 휴대용 기기에서 빠른 렌더링 처리가 가능한 구조로 설계된 하드웨어 벡터 그래픽 가속기를 사용한다면 활용도가 떨어진 소프트웨어 구동 방식에 비하여 다양한 콘텐츠를 운용할 수 있다[2].

휴대용 기기의 벡터 처리기는 실시간으로 사용자의 입력을 반영하여 콘텐츠를 구동하기 위해 주어진 이차원 좌표의 값을 변형, 가공하여 장면을 생성해 내는 기하 연산 처리기와 최종 생성된 이차원 벡터 패스(path)에 영상(image), 경사(gradient), 패턴(pattern)등의 효과를 합성하는 렌더링 처리기로 이루어진다. 여기서 벡터 패스(path)란 선(line)이나 곡선을 표현하는 단위 벡터의 집합을 의미한다.

본 논문에서 제안한 가속기는 렌더링 처리에 비해 상대적으로 적은 연산이 필요한 기하 처리 연산의 일부분은 소프트웨어로 처리하고, 연산량이 많고 빈번하게 처리하여야 되는 부분은 하드웨어로 설계했다. 이후 렌더링이 이루어지는 부분은 각 기능별로 구간을 나누어 최종 영상이 프레임 버퍼에 저장될 수 있도록 설계하였다. 그리고 이러한 가속기의 모든 처리 과정(pipeline)은 크로노스 그룹(Khronos group)의 OpenVG(Open Vector Graphic)[3]의 명세서에 따라 설계 되었다.

본 논문의 구성은 2절에서 OpenVG 벡터 그래픽에 대한 관련 연구를 기술하고, 3절에서 OpenVG의 기하 연산 처리기와 렌더링 처리기의 설계를 위해 각 기능별로 구성을 나누어 설명하고, 제 4절에서 설계된 벡터 가속기의 검증 및 성능을 분석 한 후, 5절에서 결론을 내린다.

2. 관련 연구

OpenVG 벡터 그래픽 가속기를 크게 나누면 기하 연산(geometry) 처리기와 렌더링(rendering) 처리기로 이루어진다. 기하 연산 처리기는 렌더링 할 객체(object)를 구성하는 패스의 크기나 위치 변화 및 그에 따른 수학적 연산을 처리하는 부분이고, 렌더링 처리기는 생성된 패스 정보를 이용하여 채우기 부분과 외곽선(stroke) 생성 부분을 만들어 최종적으로 프레임 버퍼에 기록하는 부분이다.

OpenVG의 이상적인 그래픽 처리기는 그림 1과 같이 8 가지의 단계로 처리된다[3]. 기하 연산과 렌더링으로 처리되는 영상은 벡터에서 사용되는 패스로 구성되는데 이의 정보는 그림 1의 1단계(Stage 1)에서 6단계, 8단계의 순서로 처리되고, 영상 정보는 1단계에서 8단계까지의 순서로 처리된다.

OpenVG의 벡터 그래픽 처리의 구현은 명세서에 따라 설계하는 것이 가장 이상적이지만, 최종 렌더링 결과가 OpenVG 명세서의 검증 절차에서 허용하는 범위 내에 있다면 각 단계들은 OpenVG의 명세서와 절대적으로 같지 않아도 된다.

2.1 기하 연산 처리기

기하 연산 처리 단계에서는 이차원의 좌표 정보,

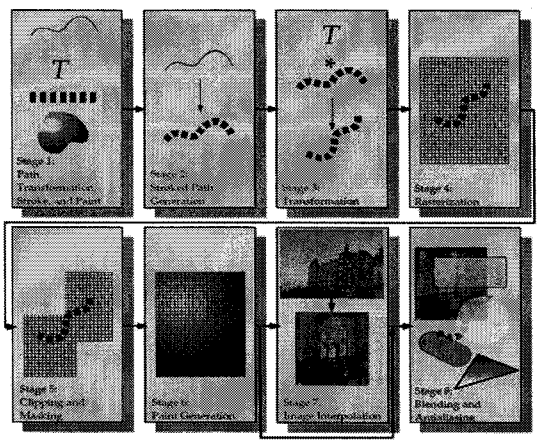


그림 1. OpenVG 벡터 그래픽 처리의 흐름도[3]

즉 패스 정보를 기반으로 수학적 연산을 통하여 모형 변환(model transform)을 수행하는데 이는 그림 1의 1단계에서 3단계에 해당된다.

1단계(Stage 1: Path, Transformation, Stroke, and Paint)에서 변환이 수행되는 대상은 이차원 패스이고, 각 패스를 이루는 좌표를 이용하여 기본적인 외곽선과 대상 객체를 채울 수 있는 페인트(paint) 형태를 만들어 일차적인 선분을 만들어 낸다. 만들어진 선분은 3×3 행렬의 곱셈 연산이 주를 이루게 되는데 행렬의 곱셈 연산은 반복적인 곱셈과 덧셈 연산으로 구성된다. 또한 사용자가 원하는 좌표, 도형 등의 정보를 입력 받아, 그리고자 하는 매개 변수를 결정하기도 한다. 예를 들면, 중심을 기반으로 외곽선의 두께나 채우고자 하는 색상의 결정, 처리 방법의 결정 등이 이에 해당된다.

2단계(Stage 2: Stroked Path Generation)의 외곽선 패스 생성은 OpenVG의 객체를 생성하는데 OpenVG API(Application Program Interface)에서 주어지는 외곽선 매개 변수를 이용하여 생성할 객체의 외곽선을 정의한다. 이때 패스 정보가 외곽선이 되어야 한다면, 입력한 패스 정보를 일정한 두께의 폭으로 차이 값(offset)을 생성하고, 기타 대시(dash) 형태로 그릴 경우는 이를 반영한 새로운 패스를 생성하여야 한다. OpenVG에서 사용하는 패스 정의는 표 1과 같고, 외곽선 형태의 정의는 표 2와 같다.

3단계(Stage 3: Transformation)는 사용자 좌표(user coordinate)로부터 주어진 값을 갖고 화면 좌표(screen coordinate)로 변환하는 과정이다. 여기서 좌표 변환이 일어나고 최종 출력하고자 하는 프레임 버퍼의 원점 좌표 간 행렬과 벡터간의 곱셈 연산이 이루어진다. 따라서 위와 같은 형태의 선분과 외곽선 패스의 생성을 이용하여 일차적인 기하 연산 처리를

표 1. OpenVG의 패스 정의

패스 정의	OpenVG의 패스 형태
이동(Moves)	MOVE_TO
직선(Straight line segments)	CLOSE_PATH, LINE_TO, HLINE_TO, VLINE_TO
곡면 (Bezier curves)	QUAD_TO, CUBIC_TO, SQUAD_TO, SCUBIC_TO..
타원 (Elliptical arcs)	SCCWARC_TO, SCWARC_TO, LCCWARC_TO, LCWARC_TO

표 2. OpenVG의 외곽선 형태 정의

외곽선 정의	OpenVG의 외곽선 형태
선두께(Line width)	Line width
끝 모양 형태 (End cap style)	Butt, Round, Square
이어지는 선분 모양 (Line join style)	Miter, Round, Bevel
이음선 제한 (Miter limit)	Miter limit(if using Miter join style)
대시(Dash)	Dash pattern / phase

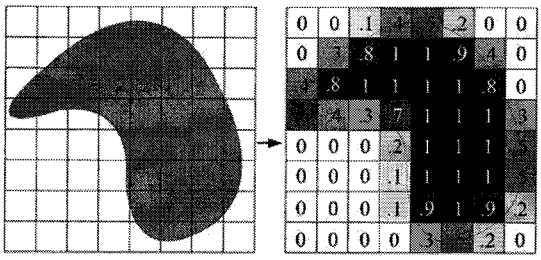


그림 2. 기하 연산 처리기 래스터화

거쳐 현재 패스가 영향을 주는 화소(pixel)의 평균값을 구한다. 여기서 구한 평균값은 저장되어 후 처리 과정에서 생기는 표본화 오류를 제거하는데 사용하게 된다. 그림 2는 기본적인 벡터 정보가 비트 맵 형식의 정보로 변환되는 것을 보여 주고 있다.

2.2 렌더링 처리기

그림 1의 1단계에서 3단계의 기하 연산 처리가 이루어진 후 4단계에서 8단계까지의 렌더링 처리가 진행되며 생성된 패스의 화소에 대한 평균값에 각종 효과(영상 섞기, 잘라내기, 색 섞기 등)를 준다. 렌더링 처리란 벡터 그래픽 객체를 화면에 그리는 동작을 말한다[4].

기하 연산과 연결되는 렌더링의 4단계(Stage 4: Rasterization)에서는 사용자의 경로와 외곽선을 이용하여 실제 화면의 각 픽셀(pixel)을 결정하는데, 이는 각 객체에 대한 프레임 메모리의 알파(alpha) 값을 바꾸어서 진행된다.

5단계(Stage 5: Clipping and Masking)에서는 최종 생성될 패스가 화면의 원하는 영역에만 출력될 수 있도록 잘라내기와 마스킹이 이루어진다. 잘라내기란 주어진 이차원 선분에서 보기 영역(view port)

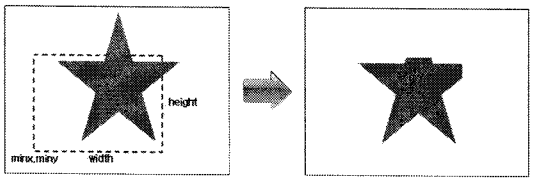


그림 3. 잘라내기

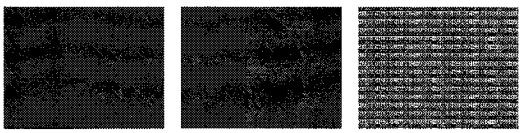
표 3. OpenVG의 마스크 연산 정의

마스크 연산 정의	OpenVG의 마스크 연산
CLEAR_MASK	αnew = 0
FILL_MASK	αnew = 1
SET_MASK	αnew = αmask
UNION_MASK	αnew = 1 - (1 - αmask) * (1 - αprev)
INTERSECT_MASK	αnew = αmask * αprev
SUBTRACT_MASK	αnew = αprev * (1 - αmask)

을 벗어나는 선분을 제거하는 것을 말하고, 마스킹이란 생성될 각 화소에 알파(alpha) 값을 적용하는 것을 말한다. 그림 3은 잘라내기를 보인 것이고, 표 3은 OpenVG에서 정의된 마스크 연산(mask operation)의 목록을 보여주고 있다.

잘라내기와 마스크 연산을 진행한 후 영상의 색상을 결정하기 위하여 6단계(Stage 6: Paint Generation)의 페인트 생성 단계에서 단색으로 칠하기와, 단색 외곽선 경사를 이용한 채우기(fill), 패턴 채우기 등을 수행하게 된다. 페인트 생성 단계는 크게 색상 페인트, 경사 페인트, 그리고 패턴 페인트로 구분 된다. 색상 페인트는 모든 화소에 대해 주어진 색상 값으로 채우는 것이고, 경사 페인트는 선형(linear)과 방사형(radial) 형태로 주어진 선분 영역에 칠하기 효과를 주는 것이며, 패턴 페인트는 OpenVG API에서 주어지는 원본 영상을 이용하여 선분의 사각형 영역에 반복 혹은 단일 형태로 영상을 채우는 것이다. 그림 4는 OpenVG API에서 처리할 수 있는 다양한 형태의 경사 및 패턴 페인트의 예이다.

페인트 생성 과정 이후 7단계(Stage 7: Image Interpolation)에서 OpenVG API를 통하여 들어오는 영상의 각 화소에 색을 더하는 보간 과정을 처리한다. 이 과정은 영상에 적용된 행렬의 역행렬을 사용해 보간된 값으로 각 화소를 계산하는 부분이며, OpenVG API의 그리기 형태에 따라 수행 여부를 판단하여 진행된다.



선형 페인트 방사형 페인트 패턴 페인트

그림 4. 각 형태별 페인트 방식

페인트 생성과 영상의 보간이 완료되면 렌더링 수행의 마지막 단계인 8단계(Stage 8: Blending and Antialiasing)의 섞기 단계를 거쳐 완성된 최종 화소를 만들어 낸다. 래스터화(rasterize)에서 계산된 평균값과 새롭게 생성된 색의 값으로 표본화 오류가 제거된 비트 맵 정보를 생성하고, 만약 이 결과와 이전의 화면을 겹치기 위한 섞기 형태가 수행되어야 한다면 섞기를 수행한다.

이상과 같은 각 단계의 처리 과정을 거쳐 출력하길 원하는 패스 또는 영상 등의 객체에 대해 그리기 매개 변수를 설정하고, OpenVG API의 그리기 함수를 호출하면, 사용자가 정의한 그리기 매개 변수를 분석하여 그릴 객체를 비트 맵 정보로 변환한 후(2단계에서 7단계) 이미 출력된 비트 맵과의 합성 여부를 판별하여 합성이 이루어진다(8단계). 이와 같은 처리가 각 객체에 대해 반복적으로 수행되어야 한 장의 영상이 완성된다.

3. 벡터 처리를 위한 OpenVG 가속기 설계

2절에서 언급한 OpenVG의 처리 과정을 토대로 본 논문에서 설계한 이차원 벡터 그래픽 가속기의 전체 구성도는 그림 5와 같다. 본 논문에서는 하드웨어를 이용하여 가속기를 구현하기 전에 벡터 처리를 위한 OpenVG의 각 처리 단계에 따라 C언어로 벡터 그래픽 처리기의 알고리즘을 구현하고 검증하였다. 그림 5에서 기하 연산 처리기(GeometryEngine)로 명명한 것이 기하 연산과 관련된 기하 연산 처리기이고, 렌더링 처리기(RenderingEngine)로 표기된 부분이 각 화소의 색상을 계산하는 부분이다. 그 외 각종 중간 연산 처리의 일시적인 결과 저장을 위해 캐시 처리기(CacheEngine)를 두었고, 외부 메모리를 패스 버퍼(PathBuffer)로 사용한 버퍼 제어기(BufferController) 등도 배치하였다.

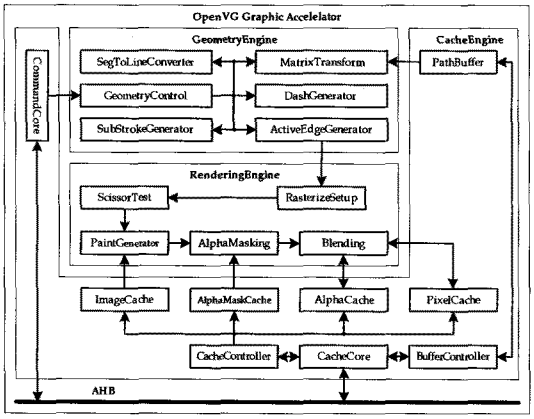


그림 5. 벡터 처리를 위한 OpenVG 가속기의 전체 구성도

기하 연산 처리기는 이차원 객체의 기하 변화를 처리하기 위해 행렬을 이용했다. 변환은 이차원 객체의 기하 변화를 나타내는 3×3 행렬을 만들고 이차원 객체를 변환시킨다. 세그먼트 선 변환기(SegToLine Converter)는 곡선이나 호와 같은 세그먼트를 직선으로 보간한다.

렌더링 처리기는 앞서 기하단계에서 생성된 활성 꼭짓점(active edge)을 이용하여 실제 색상 정보를 구하게 된다. 렌더링 처리기에서 잘라내기 검사(ScissorTest)는 사각형 안의 영역만 그리게 하고 페인트 생성기(PaintGenerator)는 각 페인트의 형태에 따라 객체 안에 채워질 화소 정보를 구한다. 이후 알파 마스킹(AlphaMasking)은 앞서 구해진 화소 정보에 알파 값을 적용한다. 섞기(Blending)에서는 현재 구해진 원본 색상과 프레임 버퍼에 있는 최종 색상 간에 섞기 연산을 한다.

OpenVG의 각 그래픽 객체를 처리하기 위한 과정에서 많은 함수가 사용된다. 함수를 계산하는 방법은 미리 계산된 테이블 값을 참조하는 형태와 유한 차수의 테일러급수를 계산하는 방식이 있다[5]. 전자의 경우는 메모리가 많이 사용되는 반면, 후자의 경우는 연산 시간을 많이 필요로 한다[6]. 본 논문에서는 가속기의 속도 향상을 위하여 테이블 참조 방식을 사용하였다.

3.1 명령어 코어 및 캐시 처리기의 설계

설계한 가속기는 ARM 코어로부터 명령을 받아 처리하는 명령어 코어(CommandCore)와 메모리로부터 정보를 읽고 쓸 때 효율적으로 처리하기 위한 캐시 처리기(CacheEngine)를 내장했다. 그림 6은 명령어 코어에 대한 구성도이고 그림 7은 캐시 처리기의 구성도이다.

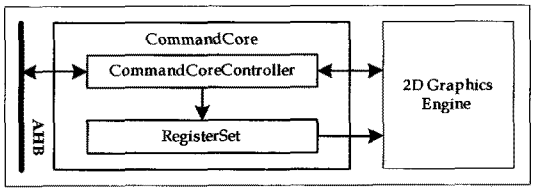


그림 6. 명령어 코어의 구성도

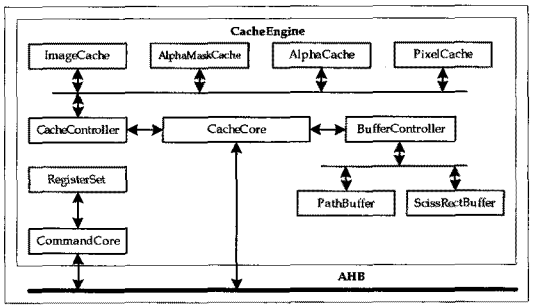


그림 7. 캐시 처리기의 구성도

명령어 코어는 ARM 코어로부터 명령을 받아 벡터 처리기를 동작시키는 데, OpenVG의 명세서에 따라 자르기, 사각형, 페인트, 알파 마스킹 표시(flag), 섞기 등을 제어할 수 있는 제어기(CommandCore Controller)와 캐시의 기본 주소를 설정하는 레지스터(RegisterSet)들로 구성된다.

설계된 가속기의 API에서는 입력되는 명령을 사전 검증하여 명세서에 부합되는 값들은 드라이버 단에서 API의 매개 변수를 레지스터에 설정하고 부합되지 않는 값들은 오류 처리한다. 따라서 벡터 처리에 적합한 것들만 API를 거쳐 해당 명령이 수행된다. 이 명령어 레지스터 집합은 소프트웨어 드라이버를 통해 설정되는 값이다.

캐시 처리기는 각각의 용도에 따라 알파 캐시(AlphaCache), 알파 마스크 캐시(AlphaMaskCache), 영상 캐시(ImageCache), 화소 캐시(PixelCache)를 조절하는 캐시 제어기(CacheController)와 패스 버퍼(PathBuffer), 자르기 사각형 버퍼(ScissRectBuffer)를 조절하는 버퍼 제어기(BufferController)로 나누어진다.

3.2 기하 연산 처리기 설계

벡터 그래픽 가속기의 기하 연산 처리기는 벡터 정보를 처리하는데 있어 기하적인 연산을 담당하는 부분이다. 그림 8은 설계된 기하 연산 처리기의 전체 구성도이다. 기하 연산 제어기(GeometryController)에서는 기하 연산에 관련된 각 모듈을 관리하며 행렬 변환에서는 읽어 들인 패스 좌표를 행렬로 변환하고, 대시 생성기(DashGenerator)는 외곽선 페인트를 처리할 때 대시 값을 확인하여 새로운 선 좌표들을 생성한다[7].

부분 외곽선 생성기(SubStrokeGenerator)는 선 두께, 선 끝의 모양, 선이 이어지는 부분의 모양에 따라 새로운 선들을 생성하고, 활성 꼭짓점 생성기(ActiveEdgeGenerator)는 앞에서 생성된 좌표에 따라 렌더링 처리기에서 사용될 시작점 x축, 시작점 y축, 끝점 x축, 그리고 끝점 y축 값을 가지는 선의 좌표를 만든다. 이때 OpenVG의 벡터 그래픽 처리기에서 패스와 영상을 처리할 때 그림 1의 1단계에서 3단계는 객체 단위로 처리한다. 즉, 벡터 콘텐츠에서 사용된 각종 OpenVG API의 패스, 행렬, 선 두께, 합쳐지는 모양, 정점 모양 등의 여러 속성들은 매개 변수로 기하 연산 처리기에 전달되어 수행된다.

다음 구성은 행렬 변환(MatrixTransform)으로 모든 패스 정보는 화면에 표시하기 위하여 변환 단계를 거쳐야 한다. 변환을 위해서는 3×3 행렬이 사용된다. 그림 9는 행렬 연산을 처리하기 위한 구성도로 3×3 행렬을 저장하기 위해 행렬 버퍼(MatrixBuffer)를 두었고, 행렬 생성기(MatrixBuilder)에서는 읽어 들인 행렬 간에 연산을 수행한다. 패스 변환기

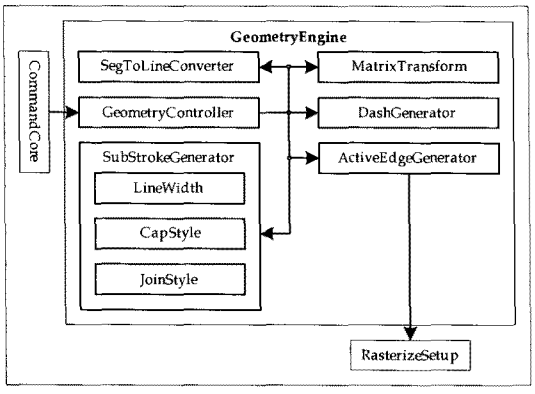


그림 8. 기하 연산 처리기의 구성도

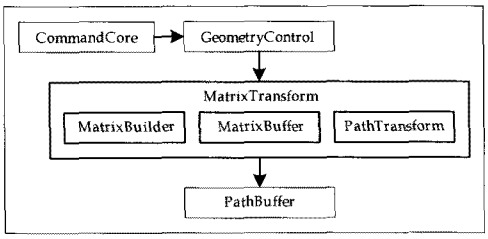


그림 9. 기하 연산 처리의 행렬 연산 구성도

(PathTransform)에서는 패스 버퍼로부터 읽어 들인 세그먼트들의 좌표 값을 변환한다.

행렬 변환에서 변환된 패스 좌표 정보의 곡선이나 호는 관련된 세그먼트 선으로 보간 되어야 한다. 곡선이나 호의 길이가 길어질수록 더운 세분화된 선을 생성하여 부드러운 곡선을 만든다. 대시 생성기는 외곽선 패스에 대시 on/off 개수가 2 이상인 경우 대시 패턴을 생성한다. 대시 생성기는 16개의 대시 세그먼트를 지원하며 그림 10은 대시 패턴의 예이다.

대시 패턴이 완료된 후 부분 외곽선을 생성한다. 그림 11은 부분 외곽선 생성의 개념도이며 선의 두 좌표를 받아 새로운 좌표들을 생성한다.

위와 같은 처리 과정 이후 기하 처리의 마지막 과정으로 그림 8의 활성 꼭짓점 생성기(ActiveEdge Generator)에서 앞서 생성된 좌표 값으로 방향성을 가지는 선들을 생성한다. 방향성이 필요한 이유는 이후 렌더링 처리기 단에서 채우기 규칙을 적용하기 때문이다. 그림 12는 활성 꼭짓점을 생성하는 규칙을 보여준다.

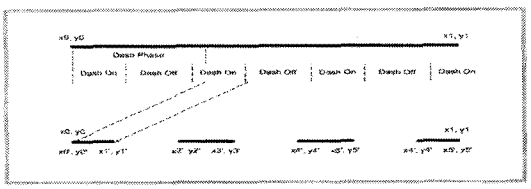


그림 10. 기하 처리 형태의 대시 생성 예

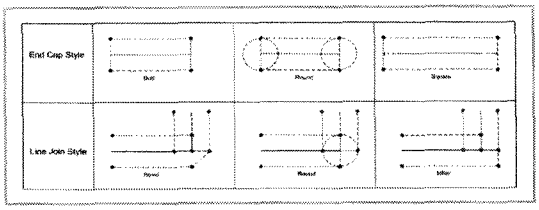


그림 11. 기하 처리 형태의 부분 외곽선 생성

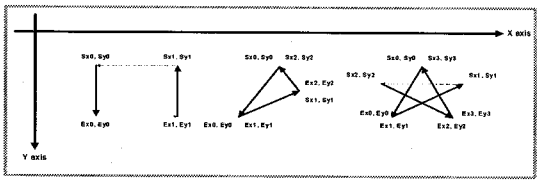


그림 12. 활성 꼭짓점의 생성 규칙

3.3 렌더링 처리기 설계

기하 연산 처리기(RenderingEngine)에서 생성된 활성 꼭짓점은 래스터화 설정(RasterizeSetup) 단에서 채우기 규칙을 검사하고 확장(span)될 최저 x축, 최고 x축, 확장의 단위, 그리고 알파 값을 구한다. 이후 처리 단계에서 자르기 검사를 거치고 각 페인트 형태에 따른 화소 정보를 구한다. 구해진 화소 정보는 알파 마스킹과 섞기 연산을 거쳐 최종 화면에 표시된다. 그림 13은 렌더링 처리기의 구성도이다.

OpenVG에서 정렬은 격자무늬(tessellation)를 기반으로 하는 렌더링 알고리즘을 사용한다. 즉, 주사선을 지나는 정점을 x좌표 순서로 정렬할 때와 여러 개의 자르기 사각형을 정렬하는 경우가 있다. 본 논문에서는 패스 생성 구성에서 이차원 x, y의 크기를 병합 정렬 알고리즘을 이용하여 오름차순으로 정렬하고, 화소 생성을 위해 선과 선 형태로 변형하여 벡터의 여러 요소인 객체를 표현한다. 따라서 이를 위해서는 하드웨어에서 정보 버퍼에 저장된 정보를 읽어 각 좌표의 거리 값에 따라 선별 처리할 수 있는 구성이 필요하다[8].

그림 13에서 래스터화 설정 단계 이후 물체의 자르기 시험(ScissorTest) 단은 x축, y축, 너비, 그리고 높이의 집합으로 구성되어 있는 32개의 자르기 사각

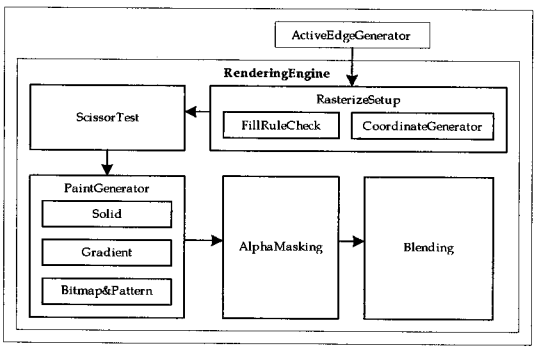


그림 13. 렌더링 처리기의 구성도

형을 지원한다. 각 자르기 사각형은 래스터화 설정 단계에서 넘어오는 좌표에 대하여 32개의 자르기 검사를 동시에 처리한다. 자르기 시험 이후에는 페인트 생성이 시작된다.

페인트 생성기(PaintGenerator)에서는 각 화소의 정보를 가지고 솔리드 색상, 경사, 및 비트 패턴에 대한 각 좌표의 화소 정보를 계산한다. 솔리드(Solid)에서는 각 객체 내에서 매개 변수에 의해서 주어진 단일 색상으로 색을 칠한다. 경사(Gradient)에서는 각 화소의 x축과 y축의 좌표, 너비 및 높이에 대해 256가지의 색상을 가지고 있는 색상 표시기에서 색상 정보를 읽어와 선형 경사의 경우는 주어진 (x0, y0)와 (x1, y1) 간에 선형적으로 경사 효과를 내며, 방사형 경사의 경우는 주어진 가운데 점, 초점, 그리고 반경의 매개 변수를 이용하여 원형의 경사 효과를 낸다.

페인트 생성기에서 비트 맵과 패턴(Bitmap& Pattern)의 경우는 지정된 비트 맵 정보를 객체에 적용하여 처리한다. 비트 맵 정보는 영상 캐시를 통하여 읽어오며, 패턴 형태에서 비트 맵의 크기가 객체보다 작으면 채우기, 패드, 반영 및 반복을 지원하고, 채우기 형태에서 객체가 비트 맵 사이즈보다 큰 경우 레지스터로 설정된 색상으로 객체의 나머지 부분을 채우며, 나머지의 경우에는 앞서 설명한 경사(gradient)와 동일하다. 영상 형태일 경우에는 앞서 행렬 변환에서 구해진 3×3 행렬 정보를 이용하여 영상 보간을 수행한다.

알파 마스킹(AlphaMasking) 단계에서는 페인트 생성기에서 구해진 화소 정보의 알파 값을 알파 마스크 캐시로부터 읽어 각 화소에 적용하는 데 화면 전체 영역에 대해서도 알파 마스킹을 적용할 수 있도록 구성하였다.

페인트 생성기에서 구해진 색상 정보는 알파 마스킹 단을 거친 후 섞기(Blending) 단계에서 OpenVG 명세서에 명시된 여러 가지의 섞기와 영상 페인트 형태에서의 섞기인 기본 영상, 곱셈 영상, 그리고 스텐실(stencil) 영상 형태로 프레임 버퍼의 화소 정보와 혼합하여 최종 화면에 보이게 될 색상 정보를 프레임 버퍼에 다시 쓰게 된다. 섞기 단계에서는 현재 화소 정보를 읽고 쓰기 위하여 알파 캐시(AlphaCache)와 화소 캐시(PixelCache)를 두었으며, 섞기에서 중복된 연산을 피하기 위하여 공유 연

산기를 두었다.

4. 실험 및 검토

본 논문에서 설계된 OpenVG API의 알고리즘은 먼저 윈도우즈(Windows XP) 하에서 OpenVG의 명세서에 대한 적합성 여부와 동작을 검증하고, 검증된 알고리즘을 기반으로 설계된 하드웨어 가속기를 FPGA에 이식하여 OpenVG의 명세서에 대한 검증과 처리 속도를 측정했다.

4.1 설계된 OpenVG API의 검증

가속기의 기능 검증을 위해 먼저 소프트웨어로 설계한 OpenVG API를 윈도우즈에서 구동하여 검사했다. OpenVG API의 검증에 있어 소수점에 민감한 기하 연산 처리와 오차의 누적 발생이 가능한 부분의 코드를 일부 수정하여 부동 소수점 연산 방식을 고정 소수점 연산 방식으로 변경했다[9]. 이는 기하 연산 처리에서 20.8 형식의 고정 소수점 형식을 사용해도 패스 보정이 가능하므로 불필요한 연산을 줄일 수 있다.

본 논문에서 제안한 가속기의 C 언어 기반의 알고리즘에 대한 검증은 두 가지로 진행하였다. 먼저, ㈜휴원[10]에서 제작하여 배포하고 있는 OpenVG 1.0에 대한 검사용 콘텐츠를 사용하여 본 논문에서 제안한 가속기가 OpenVG의 명세서에 적합한지를 검사하였다. 표 4의 시험 결과를 보면 (주)휴원의 검사용 콘텐츠에서 제공하는 모든 시험 항목에 대하여 본 논문의 알고리즘이 적합하다는 것을 알 수 있다.

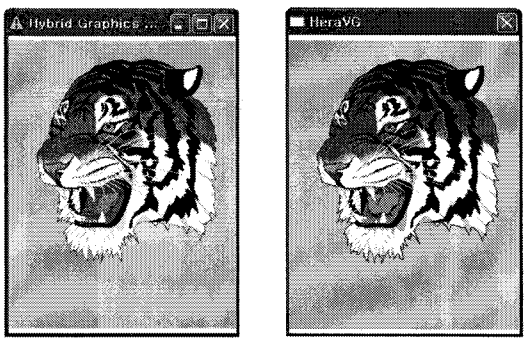
표 4. OpenVG 1.0 명세서에 대한 적합성 검사

시험 항목	OpenVG 1.0 명세서에 대한 적합 여부
매개변수(Parameter)	적합
행렬(Matrix)	적합
방사형 채우기(Radial)	적합
자르기(Scissoring)	적합
지우기(Clearing)	적합
페인트(Paint)	적합
이미지(Image)	적합
마스킹(Masking)	적합
섞기(Blending)	적합
휘감기(Convolve)	적합
선형 채우기(Linear)	적합

또한, 주어진 콘텐츠의 처리 결과를 검증하기 위하여 하이브리드(Hybrid)사[11]의 호랑이 영상을 사용하였다. 검증에 사용된 호랑이 영상은 SVG Tiny용으로 만들어진 벡터 콘텐츠를 OpenVG API에서 사용하기 위해 변환한 것으로 305개의 궤적을 비롯하여 솔리드(solid) 및 다양한 외곽선 형태가 포함되어 있다. 하이브리드사의 래스터 처리기를 구동하여 얻은 그림 14(a)의 호랑이 영상과 본 논문에서 제안한 가속기의 OpenVG API를 사용하여 얻은 그림 14(b)의 호랑이 영상은 동일하다.

4.2 설계된 하드웨어 가속기의 검증

먼저 설계된 하드웨어 가속기의 성능 측정을 위해 가속기의 구성별 기능들에 부합되는 콘텐츠를 동일한 환경의 FPGA 보드에서 소프트웨어만으로 구동한 것과 가속기를 구동한 것을 비교하여 시험했다. 사용한 FPGA는 10MHz로 동작하며, 내장된 ARM 프로세서의 코어도 같은 주파수로 동작한다. 가속기의 속도 측정은 앞에서와 같이 ㈜휴원[10]에서 제작하여 배포하고 있는 OpenVG 1.0에 대한 검사용 콘텐츠를 사용했다. 검사용 콘텐츠의 시험 항목에 대한 성능 평가는 표 5와 같다. 표 5에서 "S/W 구동"은 본 논문에서 제안한 OpenVG의 알고리즘을 소프트웨어만으로 구동하기 위하여 FPGA 안에 있는 ARM 프로세서 코어만 사용하여 측정하였고, "가속기 구동"은 설계된 하드웨어 가속기를 FPGA에 이식하여



(a) 하이브리드사의 래스터 처리기 (b) 본 논문에서 설계한 C 언어 기반의 처리기

그림 14. 설계된 OpenVG API의 검증 실험 결과 : 하이브리드사의 호랑이 영상

표 5. 본 논문의 소프트웨어 가속기와 하드웨어 가속기의 항목별 처리 속도

시험 항목	처리 속도 (sec, 초)		
	SW 구동	가속기 구동	S/W 구동과 비교한 가속기 구동의 처리 속도
매개변수(Parameter)	2.731	0.435	약 6.3배 빠름
행렬(Matrix)	2.461	0.200	약 12.3배 빠름
방사형 채우기(Radial)	14.028	0.219	약 64.1배 빠름
자르기(Scissoring)	4.533	1.068	약 4.2배 빠름
지우기(Clearing)	15.706	1.863	약 8.4배 빠름
페인트(Paint)	13.793	1.167	약 11.8배 빠름
이미지(Image)	9.210	1.171	약 7.8배 빠름
마스킹(Masking)	5.010	1.570	약 3.2배 빠름
섞기(Blending)	6.849	1.641	약 4.2배 빠름
휘감기(Convolve)	27.772	0.409	약 67.9배 빠름
선형 채우기(Linear)	9.289	2.993	약 3.1배 빠름

측정했다.

표 5의 실험 결과를 보면 방사형(radial)과 휘감기(convolve)의 경우 "가속기 구동"이 "S/W 구동"보다 약 60배 이상 빠른 것을 볼 수 있는 데, 이는 영상처리에 있어 두 가지의 처리 과정에서 병목 현상이 발생한다는 것을 말해 주고 있다. 따라서 방사형과 휘감기 연산의 곱셈 및 나눗셈 연산이 병렬로 처리되도록 하드웨어 가속기를 설계했다. 또한 실험의 페인트 항목은 단순 콘텐츠가 아닌 복합적인 콘텐츠를 처리하기 위한 것이며, 단순 영상 처리와 복잡한 영상을 반복 처리한 결과이다.

표 6은 표 5에서 실험한 항목들의 연산량을 측정한 것이다. 표 6의 결과를 보면 검사 항목에 가장 많은 시간이 소요된 휘감기는 소프트웨어로만 구동하였을 때 약 16MIPS의 연산량을 필요로 하는데 비하여 가속기를 구동하여 측정한 경우는 약 0.235MIPS가 소요되어 약 68배의 차이를 보였다.

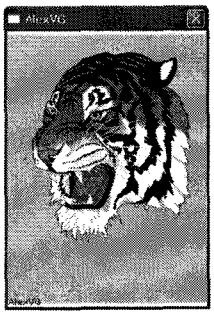
그리고 하드웨어 가속기를 통하여 처리된 영상에 대한 검증을 위하여 (주)휴원에서 판매하고 있는 OpenVG용 알렉스 처리기(소프트웨어로 된 그래픽 처리기)의 영상 출력과 본 논문에서 제안한 하드웨어 가속기의 영상 출력을 비교하였으며, 그 결과를 그림 15와 그림 16에 보였다.

그림 15(a)와 그림 15(b)에서는 호랑이 영상을 사용하여 두 처리기의 영상 출력을 비교함으로써 복잡한 궤적과 솔리드 항목을 검증했고, 그림 15(c)와 그림 15(d)에서는 표 4에서 사용된 섞기의 여러 형태 중 영상과 영상을 곱하여 나온 결과를 보기 위한 것

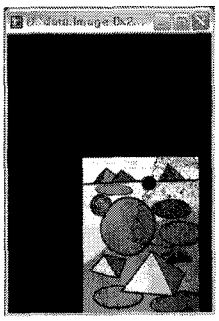
표 6. 본 논문의 소프트웨어 가속기와 하드웨어 가속기의 항목별 연산량 비교

시험 항목	연산량(MIPS)	
	S/W 구동	가속기 구동
매개변수(Parameter)	1.568	0.245
행렬(Matrix)	1.411	0.117
방사형 채우기(Radial)	8.085	0.117
자르기(Scissoring)	2.606	0.646
지우기(Clearing)	9.054	1.068
페인트(Paint)	7.951	0.715
이미지(Image)	5.301	0.676
마스킹(Masking)	2.881	0.921
섞기(Blending)	3.939	0.940
휘감기(Convolve)	16.000	0.235
선형 채우기(Linear)	5.350	1.724

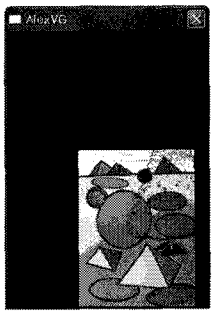
(a) 본 논문의 호랑이



(b) 알렉스 처리기의 호랑이

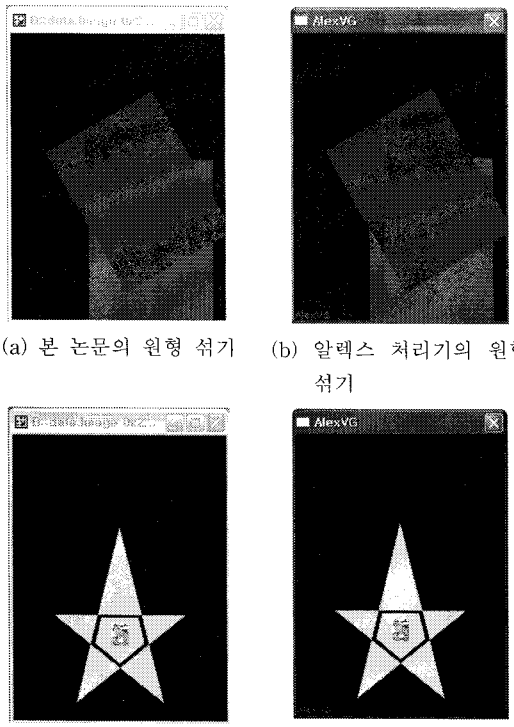


(c) 본 논문의 영상 곱하기



(d) 알렉스 처리기의 영상 곱하기

그림 15. 설계된 하드웨어 가속기의 검증 실험 (I)



(a) 본 논문의 원형 섞기 (b) 알렉스 처리기의 원형 섞기

(c) 본 논문의 바닥 채우기 (d) 알렉스 처리기의 바닥 채우기

그림 16. 설계된 하드웨어 가속기의 검증 실험 (II)

으로 두 가지 처리기에서 얻은 영상이 동일함을 볼 수 있다.

그림 16(a)와 그림 16(b)는 섞기 형태 중 원형 섞기의 처리 결과를 보기 위한 영상이고, 그림 16(c)와 그림 16(d)는 영상의 여러 형태 중 바닥(tile) 채우기 기반의 단색 채우기에 대한 결과를 보기 위한 영상이다. 역시 알렉스 처리기와 본 논문에서 제안한 하드웨어 가속기의 출력이 동일함을 확인 할 수 있다.

마지막으로 본 논문에서 구현한 OpenVG의 하드웨어 가속기의 가속 성능을 검증하기 위하여 (주)휴원의 알렉스 처리기와 본 논문에서 제안한 가속기의 처리 속도를 비교했다[12]. 표 7은 참고문헌[12]의 결과와 비교하기 위하여 하나의 프레임으로 구성된 그림 15의 호랑이 영상을 100회 반복 출력하면서 처리 시간을 측정한 결과이다.

표 7에서 보면 소프트웨어 렌더링을 사용한 알렉스 처리기는 하나의 프레임을 처리하는데 0.6초가 소요된 반면 본 논문의 가속기는 3.17초가 걸렸다. 하지만 본 논문의 가속기가 구동된 환경은 알렉스 처리기

표 7. 제안된 가속기의 처리 속도 비교

항 목	알렉스 처리기[12]	본 논문의 가속기
CPU(ARM)의 동작속도	210MHz	10MHz
메모리(RAM)의 양	128MB	128MB
호랑이 영상의 해상도	320*240	320*240
처리속도(초/프레임)	0.6초	3.17초

가 구동된 환경에 비하여 CPU와 시스템의 동작 속도가 1/21로 낮다. 따라서 본 논문의 측정 결과를 알렉스 처리기의 환경과 동일하도록 설계된 하드웨어 가속기의 동작속도를 210MHz로 올린다고 가정하면, 본 논문의 하드웨어 가속기는 한 프레임의 처리에 3.17초의 1/21인 0.15초가 소요되어 약 4배 빠르다.

5. 결 론

휴대용 기기의 벡터 그래픽 처리기는 과도한 연산량 때문에 그 활용범위가 매우 제한적이다. 따라서 제한된 연산 능력을 보유한 휴대용 기기의 벡터 그래픽 처리에서 가속기는 필수 요소이다. 본 논문에서는 휴대용 기기에서 이차원 벡터 그래픽의 처리에 효율적으로 사용될 수 있는 OpenVG의 알고리즘을 설계하고, FPGA에 이식한 하드웨어 가속기를 제안했다. 설계된 알고리즘은 OpenVG 1.0에 대한 검사용 콘텐츠를 사용한 OpenVG의 명세서에 대한 적합성 시험에서 적합함을 확인했다. 제안된 하드웨어 가속기는 내부 메모리를 제외하고 40만 게이트가 소요되었는데, 방사형(radial)과 휘감기(convolve) 항목의 영상 처리에서 약 60배 이상 빠른 처리 속도를 보였고, 호랑이 영상의 경우 기존 소프트웨어 렌더링 방식인 알렉스 처리기에 비하여 약 4배 이상 빠르게 처리됨을 확인하였다. 또한, OpenVG API 1.0의 표준을 검증하기 위한 여러 가지 콘텐츠를 사용하여 제안된 가속기에서 처리된 영상이 기준으로 제시된 영상과 동일함을 보였다.

참 고 문 헌

[1] W3C, *Scalable Vector Graphics (SVG) Tiny 1.2 Specification Draft 13*, W3C SVG

Workgroup, Massachusetts Institute of Technology, April 2005.

[2] K. Pulli, "New APIs for Mobile Graphics," *Proceedings of SPIE - The International Society for Optical Engineering*, Vol.6074, art. No.607401, 2006.

[3] Khronos Group, *OpenVG Specification 1.0*, Khronos Group, Oregon, Aug. 2005.

[4] S. Harrington, *Computer Graphics a Programming Approach 2nd edition*, McGraw Hill, New York, 2006.

[5] A. Watt, *3D Computer Graphics 3rd edition*, Addison-Wesley, Boston, 1999.

[6] Hybrid Graphics Forum, *OpenVG Reference Implementation*, http://forum.hybrid.fi, 2005.

[7] G. He, Z. Pan, C. Quarre, M. Zhang, and H. Xu, "Multi-stroke freehand text entry method using OpenVG and its application on mobile devices," *Technologies for E-Learning and Digital Entertainment*, LNCS, Vol.3942, pp. 791-796, 2006.

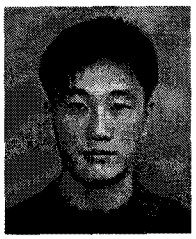
[8] R.C. Gonzalez and R.E. Woods, *Digital Image Processing 2nd edition*, Addison-Wesley, Boston, 1992.

[9] H. Keding, "FRIDGE: A Fixed-Point Design and Simulation Environment," *Design, Automation and Test in Europe*, Vol.23-26, pp. 429-435, Feb. 1998.

[10] Huone, http://www.hul.com.

[11] Nvidia, http://developer.nvidia.com.

[12] 이환용, 박기현, 우종정, "모바일 통신 단말기를 위한 벡터 그래픽스 커널 개발," 한국해양정보통신학회논문지, Vol.10, No.6, pp. 1011-1018, 2006.



김 영 옥

2000년 2월 한국방송통신대학교 (전자계산학 이학사)
2008년 8월 서울벤처정보대학원 대학교 임베디드시스템 학과(공학석사)
1999년 10월~2001년 1월 위넷컴 대표이사
2001년 1월~2002년 5월 (주)지엔비커뮤니케이션 연구1팀 개발실장
2002년 12월~2004년 2월 IMT SOFT 선임연구원
2004년 3월~2008년 7월 (주)코어로직 책임연구원
2008년 8월~현재 NSA 모바일 그래픽 자문위원
관심분야 : 임베디드시스템, 이차원/삼차원 그래픽스, 모바일 자바

노 영 섭

1988년 2월 인하대학교 전자공학과(공학사)
1996년 8월 한국과학기술원 정보및통신공학과(공학석사)
2005년 2월 고려대학교 전기, 전자, 전파공학과(공학박사)
1987년 11월~1998년 2월 LG전자 미디어통신연구소 선임연구원
1998년 3월~2001년 2월 청강문화산업대학교 이동통신과 교수
2001년 3월~2005년 2월 주식회사 싸이버뱅크 연구개발부문 상무이사
2005년 3월~현재 서울벤처정보대학원대학교 임베디드시스템학과 교수
관심분야 : 임베디드시스템, 모바일 컴퓨팅, 이동통신, 유비쿼터스 네트워크