논문번호 08-02-08

A Design of 2D Vector Graphics Accelerator with a Modified Scan-line Edge flag Algorithms including Clipping and Super Sampling

클리핑과 슈퍼샘플링을 포함한 스캔라인 엣지 플래그 방식의 2D 벡터 그래픽 가속기 설계

Kwang-Yeob Lee

이광엽

Abstract

Vector Graphics describes an image with mathematical statements instead of pixel information. Which enables easy scalability without loss in image quality and usually results in a much smaller file size compared with bitmap images. In this paper, we propose Vector Graphics Accelerator for mobile device with scan-line edge flag algorithm to render vector image without sorting process of edge. Proposed Vector Graphics Accelerator was verified with OpenVG 2D Vector image. The estimated processing time of proposed Accelerator with Tiger image is 12ms on Tessellation process, and total rendering time is 208ms. Estimated rendering performance with Tiger image is about 5 frame per second

요 약

벡터 그래픽스는 좌표 정보를 이용하여 이미지를 표현하기 때문에 이미지 퀄리티의 손실 없이 쉽게 확대 축소가 가능하며, 일반적으로 래스터 그래픽스로 표현되는 이미지보다 더 작은 파일 크기를 가진다. 본 논문에서 제안하는 벡터 그래픽 가속기는 개선된 스캔라인 엣지 플래그 방식을 사용하여 엣지의 정렬과정을 수행하지 않고 렌더링을 수행할 수 있도록 설계되었으며 OpenVG 2D 벡터 이미지를 사용하여 검증되었다. 본 논문에서 제안하는 가속기는 Tiger image를 기준으로, 테셀레이션을 수행하는데 12ms, 전체 이미지 렌더링에 208ms의 시간이 소요되며, Tiger image 기준으로 약 초당 5 프레임의 성능을 가진다.

Keywords : *vector graphics, OpenVG, 2D graphics*

Ⅰ. 서 론

최근 모바일 기기의 성능이 향상되면서 작은 화면 내에서 고 화질의 유저 인터페이스 및 텍스트, 애니메이션과 같은 고 수준의 그래픽 환경의 필요성이 증대되고 있다.[1]

기존의 비트맵 그래픽스 방식에서 애니메이션을 표현할 경우 각 프레임 별로 별도의 이미지를 제작하여야 하는 반면, 벡터 그래픽스의 경우 하나의 이미지에서 수학적인 좌표값을 변화시킴으로서 애니메이션을 표현할 수 있으며 이미지의 확대/축소가 가능하고 이미지 품질의 손실이 발생하지 않아 해상도에 관계없이 동일한 이미지를 표현할 수 있으며, 각 픽셀의 값을 모두 가지고 있는 비트맵 방식에 비해, 정점의 위치정보와 색상 정보 등으로 구성되어 이미지 크기도 작아 모바일 기기에서 고 수준의 그래픽 환경을 구현하는데 적합한 방식이라 할 수 있다.

* 평생회원, 서경대학교 컴퓨터공학과
(Dept. of Computer Engineering Seokyeong Univ.)

※ 본 논문은 서울특별시 나노혁신 클러스터 사업의 지원과 한국전자통신연구원 SoC 산업진흥센터의 지원으로 제작되었습니다.

모바일 시스템에서 벡터 그래픽스에 대한 필요성이 증가함에 따라 그래픽스 표준 API들을 제정하는 Khronos Group에서 2005년 7월 벡터 그래픽스 렌더링 부분을 표준화하고 가속화하기 위한 표준안으로 OpenVG 1.0을 발표하였으며 2007년 1월 이를 확장시킨 OpenVG 1.0.1을 발표하였다. OpenVG는 Flash와 SVG같은 벡터 그래픽 라이브러리들을 위한 하드웨어 가속 인터페이스를 제공하는 공개된 플랫폼 독립적인 API이다.[2]

본 논문에서는 모바일 기기에서 OpenVG API를 지원하며 Vector Graphics 연산을 가속화 하기 위한 하드웨어 구조를 제안한다.

Ⅱ. OpenVG 파이프라인 구조

OpenVG Specification에서 제안하는 파이프라인은 그림 1과 같다. OpenVG Specification에서 제안하는 파이프라인은 OpenVG가 수행하는 기능들을 개념적으로 설명하기 위한 파이프라인이다. 몇몇 작업들은 동시에 수행 가능하며, 렌더링 환경 설정에 따라 수행하지 않는 파이프라인도 존재한다. 따라서 실제 구현에 있어서 파이프라인을 연산을 수행하는 단위별로 분류하여 그룹화 시킬 수 있다.

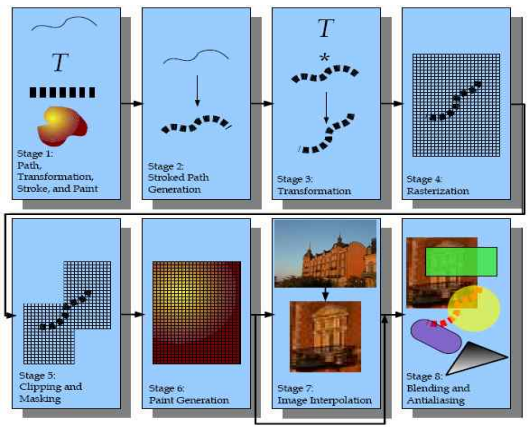

그림 1. OpenVG 파이프라인
Fig. 1. OpenVG Pipeline

OpenVG API 함수를 이용하여 그려질 Path에 대한 세그먼트 커맨드, 좌표이동, 색상등 특정 동작에 대한 제어좌표나 렌더링 조건 등을 설정하는 단계가 파라메터 설정 단계이다. 모든 파라메터들은 렌더링을 시작하는 함수가 호출되기 전에 설정이 완료되어야 하며 설정되지 않은 파라메터들은 초기값으로 설정된다.

패스 생성 단계는 어플리케이션 프로그래머에 의해 정의된 Path 정보를 이용하여 Path의 형태를 구성하는 과정이다. Path 내에 정의된 segment command와 coordinate data를 입력받아 Path가 형성하는 Shape를 구성하는 정점 (Vertex) 단위로 분할한다. 생성된 정점은 모델링을 위한 User Coordinates로 정의 되어 있다. 이를 화면에 출력하기 위해 Surface Coordinate로 좌표를 이동시키는 과정을 좌표이동 단계라 한다.

래스터라이제이션 단계는 좌표변환 과정을 수행한 정점에 의해 그려질 그리의 모양이 되는 Shape의 형태와 내부를 채울 색상의 Coverage 값을 계산하는 과정이다. 래스터라이제이션 과정에서 생성되는 Coverage 값은 추후 픽셀 오퍼레이션 단계에서 사용된다.

픽셀 오퍼레이션 단계는 실제 화면에 출력 될 이미지를 생성하기 위해, 각 픽셀 당 연산을 수행하는 과정으로 페인트 생성, 블렌딩 그리고 안티앨리어싱, 마스킹, 이미지 샘플링 동작을 수행한다.

Ⅲ. 제안하는 파이프라인 구조

본 논문에서는 OpenVG 모델에 대하여 그림 2와 같은 파이프라인을 제안한다.

Coverage 값과 관계되는 클리핑과 시저링 유닛을 래스터라이저에 포함하였다. 래스터라이제이션 과정에 클리핑을 포함함으로써 화면 영역 밖으로 나가는 엣지에 대하여 이후 파이프라인 단계에서 연산을 수행하지 않아 연산량을 줄였다.

또한 각 픽셀 당 연산을 수행해야 하는 페인트 생성 과정과 블렌딩, 마스킹, 안티 앨리어싱 과정을 픽셀 오퍼레이션 단계에서 수행한다.

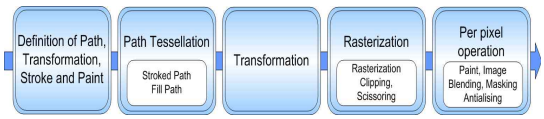

그림 2. 제안하는 파이프라인
Fig. 2. Proposed Pipeline

1. 좌표 변환

좌표 변환은 아핀 변환(Affine transformation)에 의해 수

행되며, 이동(Translation), 확대/축소(Scale), 회전(Rotation)을 행렬 연산으로 수행한다.

하나의 정점의 좌표를 변환하기 위해서는 총 4번의 부동소수점 곱셈과 덧셈 연산을 필요로 하며 그 수식은 다음과 같다.

$$NV.X = V.X * M[0][0] + V.Y * M[0][1] + M[0][2]$$
$$NV.Y = V.X * M[1][0] + V.Y * M[1][1] + M[1][2]$$

※ NV : 좌표변환 수행 후 정점 좌표
※ V : 좌표변환을 수행 할 정점 좌표
※ M : 좌표변환 수행을 위한 매트릭스

좌표이동을 위해 4개의 곱셈기와 4개의 덧셈기를 사용하여도, 연산 결과의 의존성(dependency) 때문에 최종 결과를 얻기 위해서는 최소 3Cycle이 소요된다.

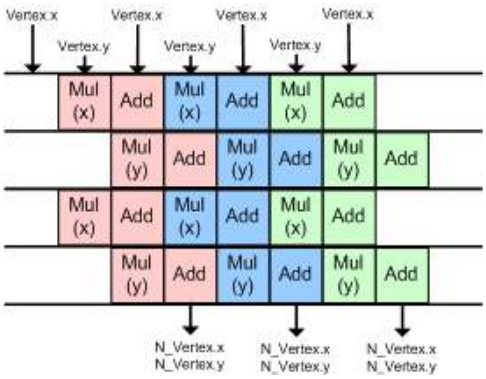

그림 3. 좌표 이동 유닛 구조
Fig. 3. Transformation Unit Architecture

본 논문에서 제안하는 좌표 이동 유닛은 매트릭스 연산의 의존성을 고려하여 연산 수행시간은 똑같이 3Cycle 의 연산 수행 시간을 필요로 하지만, 사용되는 연산기의 개수를 줄여, 2개의 곱셈기와 2개의 덧셈기를 이용하여 좌표 변환 유닛을 설계하였다.

첫 번째 Cycle에 정점의 X 좌표와 매트릭스의 [0][0], 그리고 매트릭스의 [1][0]의 곱셈이 수행된다. 두 번째 Cycle에 첫 번째 Cycle에서 곱셈이 수행되어진 X, Y좌표에 각각 매트릭스의 [0][2]와 매트릭스의 [1][2] 값이 더해진다. 동 Cycle에 정점의 Y좌표와 매트릭스의 [0][1], 그리고 매트릭스의 [1][1]의 곱셈이 수행된다. 세 번째 Cycle에 두 번째 Cycle의 결과 값의 덧셈이 수행됨으로써 좌표 변환 수행이 완료된다. 곱셈 연산과 덧셈 연산을 병렬로 수행함으로써 연산기의 개수를 줄였으며 3 Cycle의 연산 수행 시간을 유지할 수 있었다.

2. 래스터라이제이션

가. 클리핑

클리핑은 실제 렌더링이 수행되어질 영역 밖으로 나가는 엣지들을 제거하여, 이후 파이프라인에서는 렌더링 영역 외부에 해당하는 영역에 대한 연산을 수행하지 않음으로서 연산량을 줄이는 방법이다. 렌더링 영역의 위쪽과 아래쪽 그리고 오른쪽으로 완전히 벗어나는 엣지는 이후 렌더링에 영향을 미치지 않는다. 따라서 해당되는 영역의 엣지들은 무시한다. 렌더링 영역에 일부만 걸쳐지는 엣지들은 렌더링 영역에 맞추어 새로운 좌표를 할당한다. 또한 수평한 엣지도 렌더링에 영향을 미치지 않으므로 제거한다.

나. 엣지 플래그 알고리즘

래스터라이저는 스캔라인 단위로 연산 수행을 하기 위해 슈퍼샘플링을 포함하는 스캔라인 엣지 플래그 알고리즘[5] 을 적용하였다. 스캔라인과 엣지가 교차하는 위치에 플래그 값을 기록해 두고 플래그 사이를 색상으로 채운다. 스캔라인 엣지 플래그 알고리즘에 대한 기본적인 방법은 그림 4에 도시하였다.

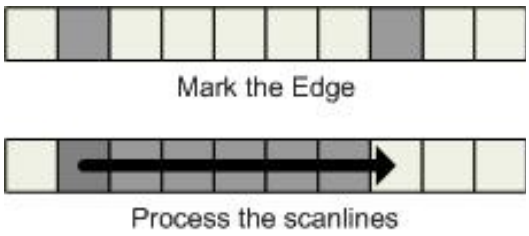

그림 4. 스캔라인 엣지 플래그 알고리즘
Fig. 4. Scanline Edge-Flag Algorithms

기본적인 엣지 플래그 알고리즘의 경우 OpenVG에서 지원해야 하는 2가지의 Fill-Rule 중 Even-Odd fillrule밖에 수행하지 못한다. Non-Zero fillrule을 수행하기 위해서는 엣지의 방향에 따라 Winding 회수를 세기 위한 추가 버퍼를 필요로 한다. 렌더링 과정은 Even-Odd fillrule 을 수행할 때와 동일하다. 차이점은 Even-Odd fillrule을 수행하기 위해선 플래그가 기록된 부분에서 비트를 반전시켜 채우기를 실행하는 반면, Non-Zero fillrule에선 엣지의 방향에 따라(+1 혹은 -1) Winding 회수를 더해주며 이때 Winding 회수가 0이 아닐때 채우기를 수행한다.

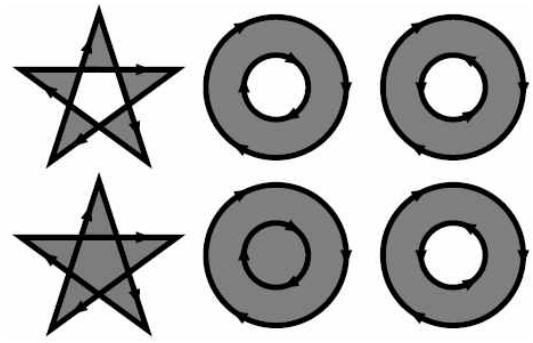

그림 5. Even-Odd fillrule vs Non-Zero fillrule
Fig. 5. Even-Odd fillrule vs Non-Zero fillrule

다. 스캔라인 엣지 플래그 알고리즘

스캔라인 단위의 렌더링을 수행하지 않을 경우, 렌더링을 수행 할 해상도 크기의 커다란 버퍼를 가져야 한다. 만약 4 샘플링 안티 앨리어싱을 수행한다면 버퍼의 크기는 렌더링을 수행 할 화면의 크기보다 4배 큰 버퍼를 가지게 된다. 하지만 스캔라인 단위로 렌더링을 수행할 경우, 렌더링을 수행할 해상도의 Width 크기의 버퍼만을 가지게 되므로 효율적이라 할 수 있다.

기본적인 스캔라인 알고리즘의 경우 스캔라인 단위의 연산 수행을 위해 AET(Active Edge Table)을 생성하고 이를 X좌표 순으로 정렬하는 과정을 필요로 하였다. 스캔라인 알고리즘을 수행하기 위해서 AET 를 정렬하는 과정은 복잡하고 추가적인 메모리 오퍼레이션으로 인한 오버헤드가 발생한다.

라. 제안하는 래스터라이저

본 논문에서 제안하는 래스터라이저는 AET를 정렬하는 과정 없이 렌더링을 수행하도록 설계되었으며 이를 위해 내부에 스캔라인 크기의 마스크 버퍼를 가진다. 또한 Non-Zero fillrule을 수행할 수 있도록 Winding 회수를 기록할 수 있는 Winding Buffer를 가진다. 엣지는 엣지가 시작되는 Y좌표와 종료되는 Y좌표, 그리고 해당 엣지의 시작 Y좌표에서 교차되는 X좌표와 기울기 값을 담고 있다. 만약 시작점 Y가 종료점 Y보다 작다면 엣지의 방향은 1이 되며, 시작점 Y가 종료점 Y보다 크다면 엣지의 방향은 -1이 된다. 이를 이용하여 Winding 회수를 산출할 수 있다.

마. 안티 앨리어싱

안티앨리어싱이 적용되지 않는 경우, 각 픽셀의 중심에서 한번 샘플링을 수행한다. 슈퍼샘플링은 하나의 픽셀에 여러 개의 샘플 포인트를 두어 여러 개의 샘플을 취해 coverage value를 계산하는 방법이다. 각 샘플 포인트는 샘플 가중치(sample weight)를 가지고 있으며, 샘플링 된 샘플 가중치를 이용하여 coverage value를 결정한다. 각 픽셀에서 모든 샘플 포인트는 shape의 외부인지 내부인지 판별한다. 만약 shape의 내부라면 coverage value에 샘플 가중치가 더해지게 된다.

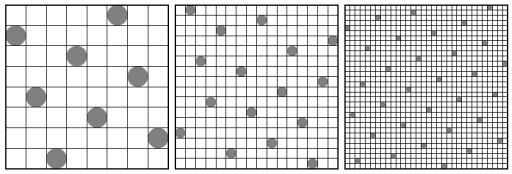

그림 6. 8, 16, 32 n-rooks 패턴
Fig. 6. n-rooks pattern with 8, 16 and 32 sample

안티 앨리어싱이 적용된 픽셀의 색상은 래스터라이제이션 과정에서 생성된 Coverage 값을 사용하여 픽셀 오퍼레이션 단계에서 최종적으로 결정된다.

3. 픽셀 오퍼레이션

가. Gradient 페인트 생성

페인트 생성 과정의 경우 페인트 모드에 따라 Gradient 페인트를 생성하는 과정이다.

기존의 방식에서는 각 픽셀의 Gradient 색상을 계산하기 위해 픽셀마다 Gradient offset 값을 산출하고 이를 이용하여 두 색상을 보간(Interpolation) 하여 최종 픽셀의 색상을 산출하였다. 기존의 방식의 경우 각각의 픽셀의 색상을 계산하기 위해 좌표변환 단계에서 사용하는 매트릭스 곱셈을 필요로 하며 각 색상마다 색상의 보간 작업을 요구한다.

나. 제안하는 페인트 생성 유닛

본 논문에서 제안하는 페인트 생성 유닛은 LUT (Look up table) 방식을 사용하여 매 픽셀 연산수행마다 Gradient 색상을 계산하기 위해 색상 보간 작업을 수행하지 않으며 최초 Gradient 색상을 설정하기 위해 색상의 범위 값을 입력받을 때 선행하여 색상 보간 작업을 수행하여 32bit X 256의 크기(1024Byte)를 가지는 LUT를 생성하고, 추후 Gradient 페인트 생성 과정에서는 생성된 LUT를 이용하여 색상을 산출한다. 또한 스캔라인 단위로 Gradient 색상을 생성하여 매 픽셀의 Gradient 색상을 산출하기 위해 매트릭스 곱셈을 수행하지 않고 스캔라인 당 한 번의 매트릭스 곱셈을 수행하여 Gradient 색상을 생성하기 위해 수행되는 연산량을 줄였다.

Ⅳ. 검증 및 결과

이미지를 렌더링 하는 과정에서 사용되는 연산의 종류와 복합 연산의 회수, 그리고 기능 검증을 수행하여 분석하였다.

Tiger Sample Image의 연산 수행 분석 결과 Floating point의 Addition과 Multiplication 및 Square root, Division이 집중적으로 사용됨을 알 수 있다. 이는 Tessellation 과 Paint 단계에서 수학적 연산이 많이 사용되기 때문이며, 이러한 연산들로 인해 임베디드 보드 상에서 OpenVG를 Floating Point로 구현하는데 H/W 구현을 통한 속도의 향상을 필요로 한다.

표 1은 Tiger Image를 그리기 위해 수행되는 연산의 회수를 분석하여 도시하였다.

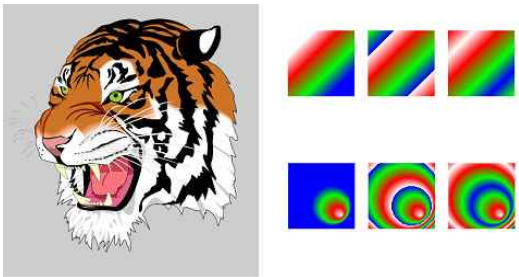

그림 7. Tiger & Gradient Image
Fig. 7. Tiger & Gradient Image

Operation	Count
addition	3,313,847,888
multiplication	181,431,340
square root	2,269,697
division	346,706
sign	409

표 1. Tiger Image 에 사용된 연산 횟수
Tab. 1. Number of Arithmetic operation for drawing Tiger image

Operation	Count
Matrix Inverse	305
Matrix Multiplication	307
Matrix Affine Transformation	1,260,003
Vector Dot product	62,239,632
Vector normalization	1,701,718
Vector length	569,236

표 2 Tiger Image 에 사용된 복합 연산 횟수
Tab. 2. Number of Complex Arithmetic operation for drawing Tiger image

Matrix를 이용한 좌표 이동인 Affine Transformation, 내적을 구하는 Vector Dot product, Vector의 방향을 위해 정규화 시키는 과정인 Vector normalization과정이 다른 과정에 비해 상대적으로 많이 수행됨을 확인할 수 있다. 해당 연산들은 주로 Floating point의 Addition과 Multiplication에 구성되어 있으므로 이들 연산을 H/W로 구현하여 가속화 하였다.

Gradient는 Linear과 Radial 2가지의 타입을 제공하며 각각의 타입마다 SPREAD PAD, SPREAD REPEAT, SPREAD REFLECT의 3가지 모드가 존재하여 총 6가지의 Gradient를 생성하도록 제안하고 있다.

그림 5의 Gradient 이미지는 순서대로

Linear - Pad, Repeat, Reflect

Radial - Pad, Reflect, Reflect

의 이미지를 생성하여 출력된 결과를 보여주고 있다.

그림 5에서 보여준 이미지를 생성하기 위해 수행한 연산을 분석하여 표 3과 4에 나타내었다.

Operation	Linear Gradient	Radial Gradient
Addition	812,787	1,110,439
Multiplication	570,507	1,210,219
Square root	7	19,207
Division	27,791	66,191

표 3. Gradient Image 에 사용된 연산 횟수
Tab. 3. Number of Arithmetic operation for drawing Gradient image

그림 5에 보여진 이미지를 렌더링 하는데 수행되는 시간을 Khronos Group에서 제공하는 레퍼런스와 비교 측정하여 표 5, 6과 7에 도시하였다. 본 논문에서 제안하는 Khronos Group에서 제공하는 레퍼런스는 OpenVG를 개발

할 개발자들이 개발 시 필요한 기능 및 연산의 구현에 초점을 두고 있으며, 속도는 고려하지 않고 구현되었다. 본 논문에서 제안하는 OpenVG는 스펙에서 제안하는 기능 구현 및 속도에 중점을 두고 설계하였다.

Operation	Linear	Radial
Matrix Inverse	3	3
Matrix Multiplication	7	7
Matrix Affine Transformation	19,244	19,244
Vector Dot product	56,166	152,166
Vector normalization	4	4
Vector length	3	19,203

표 4. Gradient Image 에 사용된 복합 연산 횟수
Tab. 4. Number of Complex Arithmetic operation for drawing Gradient image

	Reference	Proposed
Tiger	167ms	12ms
Radial Gradient	2ms	1ms
Linear Gradient	2ms	1ms

표 5. 테셀레이션 수행 시간
Tab. 5. Tessellation Time

		Reference	Proposed
Radial Gradient	Pad	36ms	3ms
	Repeat	37ms	4ms
	Reflect	37ms	3ms
Linear Gradient	Pad	21ms	3ms
	Repeat	22ms	4ms
	Reflect	21ms	3ms

표 6. Gradient 페인트 생성 수행 시간
Tab. 6. Gradient Paint Generation Time

	Reference	Proposed
Tiger	593ms	208ms
Radial Gradient	63ms	21ms
Linear Gradient	58ms	27ms

표 7. 렌더링 연산 수행 시간
Tab. 7. Rendering Time

V. 결 론

본 연구에서는 OpenVG 하드웨어 가속을 위한 파이프라인 분석 및 2D 벡터 그래픽스 파이프라인을 구성하는 알고리즘을 연구하여, OpenVG 파이프라인 아키텍처를 구성하였다.

모바일 환경에 적합하도록, 적은 자원을 사용하여 소프트웨어와 하드웨어 구현에 있어 추가비용을 줄일 수 있는 24비트 부동 소수점 데이터 형식을 사용하였으며 OpenVG Specification에서 제안한 파이프라인을 기능별, 혹은 연산별로 그룹화 하여 하드웨어 구현에 적합한 새로운 파이프라인을 제안하였다.

본 연구에서 설계된 OpenVG는 크로노스 그룹에서 제공하는 Tiger Sample Image와의 결과 이미지 비교를 수행하여 동작 및 기능이 정확하게 동작함을 검증하였고, 검증 프로그램 수행을 통하여 OpenVG에서 제공해야 하는 여러 가지 기능들의 구현 및 검증을 수행하였다.

참 고 문 헌

[1] Kari Pulli, *"New APIs for Mobile Graphics"*, Proceedings of SPIE - The International Soci ety for Optical Engineering Vol. 6074, art, no. 607401, 2006
[2] Khronos Group Inc. *"OpenVG Specification Version 1.0.1"* http://www.khronos.org/openv g/, January 2007
[3] Gene Frantz, Ray Simar *"Comparing Fixed and Floating Point DSPs*, Texas Instruments 2004
[4] ARM *"Fixed Point Arithmetic on the ARM"* , Application Note 33, ARM, September 1996
[5] Kiia Killio *"Scanline edge-flag algorithm for antialiasing"* EG UK Theory and Practice of Computer Graphics 2007

저 자 소 개

이 광 엽 (정회원)

1985년 8월 서강대학교전자공학과 학사.
1987년 8월 연세대학교전자공학과 석사.
1994년 2월 연세대학교전자공학과 박사.
1989~1995년 현대전자 선임연구원.
1995년~현재 서경대학교 컴퓨터공학과 부교수.
<주관심분야 : 마이크로 프로세서, Embedded System, 3D Graphics System>