모바일 기기에서의 방사형 그라디언트 페인트 가속

김진우*, 박진홍**, 한탁돈*
*연세대학교 컴퓨터과학과
**LG전자 SIC센터
e-mail: jwkim@msl.yonsei.ac.kr

Acceleration of Radial Gradient Paint Processor for Mobile Device

Jin-Woo Kim*, Jinhong Park**, Tack-Don Han*
*Dept. of Computer Science, Yonsei Univ.
**SIC Center, LG Electronics Inc.

요 약

방사형 그라디언트 페인트(radial gradient paint)는 벡터 그래픽스(vector graphics)에서 적은 정보로 다양한 효과를 적용시킬 수 있는 방법이다. 기본적으로 이 방법은 곱하기, 나누기, 제곱근 등의 복잡한 연산이 필요하기 때문에 모바일 같은 저성능 환경에 적합하지 않았다. 하지만 최근 모바일 기기들은 SIMD 연산 지원 및 고성능의 GPU 탑재 등으로 성능이 향상됨에 따라 이러한 문제를 해결할 수 있게 되었다. 본 논문은 ARM의 SIMD연산인 NEON을 이용하여 최대 2.6배의 성능을 가속시켰으며 GPU의 쉐이더를 이용하여 4.9배의 성능을 가속하였다.

1. 서론

출력 장치의 크기에 종속적인 픽셀(pixel) 단위로 저장하는 그래픽스를 비트맵(bitmap) 방식이라고 한다. 벡터 그래픽스는 이와 달리 도형의 좌표, 선의 특성 및 채색 방법을 저장하는 방식으로 출력 장치의 크기에 독립적으로 이미지를 확대하여도 계단현상 같은 이미지 손상이 발생하지 않는다.

출력 장치에 독립적인 벡터 그래픽스의 특성은 모바일 환경에서 다음과 같은 장점을 갖는다.

비트맵 방식은 출력 장치에 종속적이므로 출력 장치의 크기가 증가하면 이미지를 저장하기 위한 크기도 증가한다. 반면 출력 장치에 독립적인 벡터 방식은 영향을 받지 않으므로 비트맵 방식보다 저장공간에 대해서 효율적이다. 또한 모바일 환경에서는 비트맵 방식을 이용하여 컨텐츠를 제작하면 다양한 출력 장치 크기 마다 재작업이 필요하지만, 출력 장치에 독립적인 벡터 방식은 영향을 받지 않으므로 콘텐츠의 추가적인 작업이 필요하지 않다. <표 1>은 비트맵 방식과 벡터 방식의 특징을 비교한 것이다 [1].

<표 1> 비트맵 방식과 벡터 방식의 비교

비교 항목	비트맵	벡터
재생 복잡도	작다	크다
디바이스 독립성	없다	있다
데이터 크기	크다	작다
적합한 컨텐츠	사진, 동영상	만화, 지도

<표 1>에서 재생 복잡도는 벡터 그래픽스가 크기 때문에 성능이 낮은 모바일 환경에는 적용하기 부적합하였다. 하지만 최근의 모바일 기기들은 SIMD 연산 지원 및 고성능의 GPU 탑재 등으로 성능이 향상됨에 따라 벡터 그래픽스 처리에 요구되는 많은 연산량을 충분히 처리할 수 있다.

본 논문은 벡터 그래픽스에 요구되는 많은 연산량을 모바일 기기에서 효율적으로 가속하기 위한 방법에 대한 것이다. 논문의 구성은 다음과 같다. 2절에서는 벡터 그래픽스의 표준인 OpenVG[2]의 페인트와 이를 가속하기 위해 필요한 관련 기술들을 간단히 살펴보고 3절에서는 효과적인 가속 기법들을 제안할 것이다. 4절에서는 제안한 가속 기법들을 스마트폰 환경에서 구현하여 성능을 검증 및 평가하고 마지막으로 5절에서 결론을 맺는다.

2. 관련기술

2.1 OpenVG의 페인트

벡터 그래픽스의 표준인 OpenVG에서 패스(path)의 내부를 채우기 위해 정의된 3가지 페인트 방법이 있다. 첫 번째 방법은 하나의 색으로 동일하게 채우는 컬러 페인트(color paint)이며, 두 번째 방법은 비트맵(bitmap) 형식으로 저장된 이미지를 패스의 내부에 매핑(mapping)하는 패턴 페인트(pattern paint)이고, 마지막 방법은 픽셀의 위치를 이용하여 색상을 결정하는 그라디언트 페인트(gradient paint)이며 이 방법은 다시 선형(linear)과 방사형(radial)으

로 분류된다. (그림 1)은 위의 3가지 방법을 나타낸 것이다.[2]

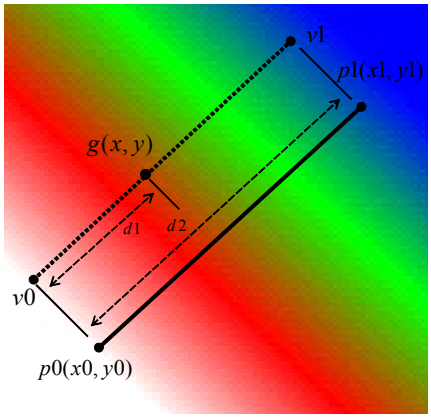
(그림 1) OpenVG에 정의된 페인터의 종류

위에서 설명한 채우기 방법 중 패턴과 그라디언트 방법은 픽셀마다 변화된 색상을 적용할 수 있다. 패턴 방식을 적용하기위해 필요한 이미지 정보는 많은 저장 공간을 요구는 비트맵 형식이지만, 그라디언트는 적은 정보로 다양한 표현이 가능하므로, 필요한 정보의 관점에서 그라디언트는 가장 효과적인 채우기 방법이라고 할 수 있다.

하지만 방사형 그라디언트를 처리하기 위해서 곱하기, 나누기, 제곱근 등의 복잡한 연산이 필요하므로 모바일과 같은 저성능 환경에는 적합하지 않다. 따라서 모바일 환경에서는 그라디언트 효과를 적용하기 위해 미리 계산된 그라디언트 정보를 이미지의 형태로 저장하여 패턴 페인트로 대체하여 표현하는 것이 일반적이다.

2.1.1 선형 그라디언트(Linear Gradient)

(그림 2)는 선형 그라디언트를 계산하는 방법이다. 픽셀 g(x,y)의 색상을 결정하기 위해 사용되는 선형 그라디언트의 설정 값은 두 점 p0, p1이며, 이 두 점 p0, p1를 지나는 선분(line segment)과 평행하고 점 g(x,y)를 지나는 가상의 선분 v0v1를 생성한다. 이때, 점 v0와 v1의 거리를 d2라 하고, 점 g와 점 v0 사이의 거리를 d1으로 할당하고 이 d1과 d2의 비율로 색상을 결정할 수 있다.



(그림 2) 선형 그라디언트

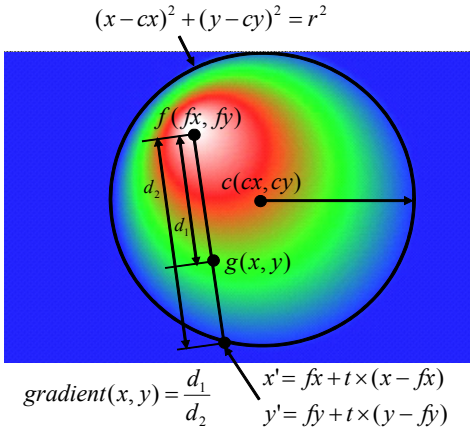
(그림 2)에서 비율 d1/d2는 [수식 1]을 이용해 계산할 수 있다. [수식 1]에서 △x와 △y는 매개변수인 y0, y1, x0, x1으로 계산되기 때문에 △x, △y, △x2, △y2은 최초 한 번만 계산을 하게 되기 때문에 실제 한 픽셀을 계산하기 위해서는 곱하기 2번, 더하기 1번, 나누기 1번이 필요하다.

$$\triangle x = y1 - y0$$
$$\triangle y = x1 - x0$$
$$g(x,y) = \frac{\triangle x(x - x0) + \triangle y(y - y0)}{\triangle x^2 + \triangle y^2}$$

[수식 1] 선형 그라디언트 계산식

2.1.2 방사형 그라디언트(Radial Gradient)

(그림 3)은 방사형 그라디언트에 필요한 매개변수들과 그들의 관계를 나타낸다. 색상을 구하고자 하는 점 g(x,y)와 f(fx,fy)의 거리를 d1이라 정의하고, 이 두 점을 연결하여 연장한 선이 점 c(cx,cy)를 중심으로 한 반지름이 r인 원에 접하는 점과 f(fx,fy)점과의 거리를 d2라고 정의하면, g(x,y)의 색상은 d1/d2로 결정된다. (그림 3)에서 각 매개변수들의 관계를 식으로 나타내면 [수식 2]와 같다[2].



(그림 3) 방사형 그라디언트

$$fx' = fx - cx,\quad fy' = fy - cy$$
$$dx = x - fx,\quad dy = y = fy$$

$$g(x,y) = \frac{dx^2 + dy^2}{\sqrt{r^2(dx^2+dy^2) - (dx\,fy' - dy\,fx')^2} - (dx\,fx') + (dy\,fy')}$$

[수식 2] 방사형 그라디언트 계산식

2.2 ARM NEON

ARM NEON[3]은 CortexTM-A series에서 제공하는 64/128bit hybrid SIMD 아키텍처 기술로 오디오, 비디오 및 3D 그래픽을 가속하는 데 사용된다. NEON은 메인 ARM 정수 파이프라인(integer pipeline)과는 별도의 실행 파이프라인(execution pipeline)과 자체 레지스터 파일을 갖고 있으며 단수(integer)와 단정도 플로팅-포인트(single precision floating-point) 값을 모두 처리할 수 있다. 또한

비정렬 데이터 액세스(unaligned data accesses)를 지원하고 구조 형태(structure form)로 저장된 인터리브 된 데이터(interleaved data)의 간편한 로딩을 지원한다. 본 논문에서는 ARM NEON 기술을 이용하여 4개 픽셀의 그라디언트 페인트 계산을 동시에 처리하도록 가속하였다.

2.3 OpenGL/GLSL

GLSL(OpenGL Shading Language)[4]은 C 언어를 기초로 한 OpenGL의 쉐이딩 언어이다. 이와 유사한 쉐이딩 언어로 마이크로소프트사(Micro Soft)의 HLSL[5]이 있으며 엔비디아(NVIDIA)사의 cg가 있다. 최근 CUDA와 OpenCL등의 고수준의 GPGPU 개발환경이 제공되지만 모바일에는 아직 제공되고 있지 않기 때문에 본 논문에서는 GLSL을 이용하여 GPU기반의 그라디언트 페인트를 구현하였다.

3. 방사형 그라디언트 가속 기법

그라디언트 페인트의 계산 과정은 크게 두 가지로 구분할 수 있다. 첫 번째 과정은 그라디언트 공간에서의 거리 값을 구하는 것으로 [수식1,2]를 계산한 결과이다. 두 번째 과정은 거리 값을 색상으로 변환하는 것으로 스칼라의 입력이기 때문에 중복된 연산이 많이 발생한다. 따라서 참조 테이블(LUT: Lookup Table)을 사용하는 것이 일반적이다. 그라디언트 페인트 처리 과정에서 두 번째 과정은 LUT로 처리할 수 있기 때문에 첫 번째 과정이 대부분의 연산시간을 차지한다. 본 절에서는 모바일 환경에서 방사형 그라디언트 페인트를 가속하기위한 다양한 방법을 제시한다.

3.1 기본 방법

기본 방법은 실제 가속을 하지 않는 방법으로 다른 가속 기법들의 성능을 비교하기 위한 기준으로만 사용된다.

방사형 그라디언트 공간에서의 거리를 계산하는 [수식 2]는 구현을 위해 최적화가 필요하다. [수식 2]에 사용된 변수 중 x, y는 모든 픽셀마다 계산을 해줘야 하며, x, y에 종속적인 dx, dy 역시 모든 픽셀마다 계산이 필요하므로 그라디언트 설정 변수인 r, fx, fy 만을 나누기에 사용하는 것이 효율적이다. 따라서 [수식 3]와 같이 표현할 수 있다 [2].

$$g(x,y)=\frac{(dx\,fx'+dy\,fy')+\sqrt{r^2(dx^2+dy^2)-(dx\,fy'-dy\,fx')^2}}{r^2-(fx'^2+fy'^2)}$$

[수식 3] 방사형 그라디언트 계산식2

[수식 3]에서 나누기 연산은 모든 픽셀에 필요하며, 나누기 연산은 곱하기 연산보다 부담이 크기 때문에 곱셈으로 변환하는 것이 효율적이다. 따라서 [수식 3]의 제수(divisor)의 역수(inverse number)를 I라고 하면 [수식 4]와 같이 표현 가능하며, 역수 I를 만들기 위한 나누기 연산은 그라디언트 설정 변수가 입력될 때 한 번만 수행되어진다[2].

$$I=\frac{1}{r^2-(fx'^2+fy'^2)}$$
$$g(x,y)=I((dx\,fx'+dy\,fy')+\sqrt{r^2(dx^2+dy^2)-(dx\,fy'-dy\,fx')^2})$$

[수식 4] 방사형 그라디언트 계산식3

[수식 4]를 이용하여 화면상의 전체 픽셀을 계산하는 경우 좌표 x와 y를 이용한 이중 반복문 형태로 구현될 수 있다. 이때 반복문 전체에 공통으로 사용될 수 있는 x, y에 독립적인 변수는 한 번만 계산하도록 구현하며, 내부 반복문에서 공통으로 사용될 수 있는 x에 독립적인 변수는 외부 반복문에서 한 번만 계산하도록 구현하였다. 또한 계산된 거리를 픽셀로 변환하는 과정은 참조테이블을 이용하여 성능을 최적화시켰다. <표 2>는 기본 방법의 의사코드이다.

<표 2> 기본 방법의 의사코드

```
//x,y에 독립적인 변수(r², fx′, fy′, I) 계산
for(y=0;y<height;y++){
  //x에 독립적인 변수(dy) 계산
  for(x=0;x<width;x++){
    //x,y에 종속적인 변수(dx) 계산
    distance=g(x,y);
    pixel = LUT[distance]; //색상값으로 변환
    //계산된 픽셀 저장
  }
}
```

3.2 ARM NEON을 이용한 가속

ARM NEON을 이용한 가속 방법은 2.2절에서 설명한 SIMD연산을 이용하여 동시에 4개의 픽셀을 처리하는 방법이다. 이러한 가속 방법은 기본 방법에서 SIMD를 이용하여 동시에 4개의 픽셀을 처리하도록 변형한 형태이다. <표 3>은 ARM NEON을 이용한 의사 코드이다.

<표 3> ARM NEON을 이용한 의사코드

```
//x,y에 독립적인 변수(r², fx′, fy′, I) 계산
for(y=0;y<height;y++){
  //x에 독립적인 변수(dy) 계산
  for(x=0;x<width;x+=4){
    //SIMD 자료형으로 변환
    //x,y에 종속적인 변수(dx) SIMD 적용 계산
    distance=g(x,y);//SIMD 적용 distance 계산
    pixel = LUT[distance]; //색상값으로 변환
    //일반 자료형으로 변환
    //계산된 픽셀 저장
  }
}
```

3.3 GPU를 이용한 가속

GPU를 이용한 가속 방법은 2.3절에서 설명한 GLSL을 이용하는 방법으로 GPU의 프레그먼트 쉐이더(fragment shader)를 이용하여 그라디언트 페인트를 가속하는 방법이다. 프레그먼트 쉐이더의 경우 픽셀단위의 계산이기 때문에 x,y에 독립적인 변수를 CPU에서 계산하여 GPU에 전달하며 같은 y좌표를 갖는 x에 종속적인 변수는 중복되어 계산된다. <표 4>는 OpenGL GLSL을 이용한 의사코드이다.

<표 4> OpenGL GLSL을 이용한 의사코드

```
Draw(){ //CPU OpenGL코드
  //x,y에 독립적인 변수(r², fx′, fy′, I) 계산
  //Shader에 x,y에 독립적인 변수 값 전달
  //Fragment_Shader() //쉐이더 수행
}

Fragment_Shader(x,y, texture){ //GPU GLSL코드
    distance=g(x,y);
    pixel = texture(distance); //색상 값으로 변환
    //일반 자료형으로 변환
    //계산된 픽셀 저장
}
```

4. 검증 및 성능 실험

본 논문에서 제시한 다양한 방법을 모바일 환경에 적용하기 위해 안드로이드 기반의 스마트폰을 사용하였다. 실험에 사용된 스마트폰은 삼성전자의 '갤럭시S'이며 <표 5>는 자세한 사양을 나타낸 표이다. (그림 4)는 실제 구현하여 동작한 결과의 사진이다.

<표 5> 실험 환경

제 품 명	삼성전자 갤럭시S
프로세서 (AP)	Hummingbird(S5PC100) - CPU: ARM Cortex A8 1 GHz - GPU: PowerVR SGX 540
해 상 도	WVGA(480 x 800)
운영체제	Android 2.2

(그림 4) 방사형 그라디언트 구현 결과

<표 6>은 본 논문에서 제시한 가속 방법을 스마트폰 환경에서 실험한 결과이다. <표 6>에서 계산시간은 그라디언트 페인트 연산 과정만을 기준으로 측정하였으며 재생률은 실제 화면에 재생되는 것을 기준으로 하였다.

<표 6> 성능 실험 결과

가속기법	계산시간		재생률(FPS)	
기본	136.44 ms		6.3 fps	
NEON	33.83 ms	4 x	16.2 fps	2.6 x
GPU	0.06 ms	2274 x	31.2 fps	4.9 x

순수한 연산시간 만을 측정하는 계산시간을 기준으로 하면 NEON을 이용한 SIMD 가속은 4배 정도 성능향상이 있었으며, GLSL을 이용한 GPU 가속은 2274배의 성능향상이 있었다. 하지만 실제 시스템 전반의 성능인 재생률을 기준으로 하면 NEON을 이용한 SIMD 가속은 2.6배 정도 성능향상이 있었으며, GLSL을 이용한 GPU 가속은 4.9배의 성능향상을 보였다.

5. 결론 및 향후 연구

본 논문은 ARM의 SIMD 기술인 NEON과 GPU 쉐이더 언어인 GLSL을 이용하여 방사형 그라디언트 페인트를 스마트폰 환경에서 가속하였고, 실험을 통해 GPU를 이용하는 경우 최대 4.9배의 성능향상이 있음을 확인하였다. 이러한 실험결과 그라디언트 페인트는 제곱근과 곱셈 등의 고급 연산을 많이 포함하지만 계산과정의 제어가 간단하여 연산위주로 최적화된 GPU에 적합함을 알 수 있다. 하지만 CPU를 이용한 가속에서 고정소수점을 이용한 최적화 등 다양한 기법들이 적용될 수 있기 때문에 향후 이런 부분에 대한 실험이 필요하다.

참고문헌

[1] 이환용, 박기현, 우종정, "모바일 통신 단말기를 위한 벡터 그래픽스 커널 개발", 한국해양정보통신학회논문지, 10권, 6호, pp1011-1018, 2006.

[2] Daniel Rice, "OpenVG Specification 1.0", Khronos Group, December 2008.

[3] ARM, "ARM® NEON™ ", http://www.arm.com/products/processors/technologies/neon.php.

[4] John Kessenich, "The OpenGL® Shading Language 4.10", Khronos Group, July 2010.