모바일 환경을 위한 정점 프로그램 가상머신 설계

김태영
tykim@skuniv.ac.kr

Design of a Vertex Program Virtual Machine on Mobile Platform

Tae-Young Kim
Dept. of Computer Engineering, Seokyeong University

요약

모바일 환경에서 고급 그래픽스 기술을 적용하고자 하는 시도로 최근 3D 그래픽 엔진을 탑재한 단말기가 출시되고 있다. 이 단말기는 OpenGL ES 1.x 을 기준으로 고정된 파이프라인을 통해 그래픽 연산을 처리하고 있으므로 사용자가 다양한 그래픽 표현을 수행하는데 제약이 따른다. 최근 PC 환경의 그래픽 엔진에서는 고정 기능의 파이프라인이 아닌 프로그래밍 가능한 파이프라인을 제공하여 기존 고정 파이프라인에서 불가능했던 유연한 그래픽스 기술을 제공하고 있다. PC 환경의 프로그래밍 가능한 파이프라인은 DirectX 와 OpenGL ARB Extension 그래픽 라이브러리에 의해 제공되고 있지만, 모바일 환경에서는 이를 지원하기 위한 관련 제품이 아직 출시되지 않고 있는 상태이다. 본 논문에서는 OpenGL ARB Extension 1.0 을 근거로 정점 프로세싱 과정을 프로그래밍 가능한 파이프라인 구조로 동작하도록 하는 모바일용 정점 프로그램 가상머신을 제시한다.

1. 서론

현재 3D 그래픽 엔진을 탑재하여 시장에 출시되고 있는 단말기는 OpenGL ES 1.x 표준을 기준으로 고정된 파이프라인을 통해 그래픽 연산을 처리하고 있다. 고정된 파이프라인의 그래픽 연산 과정은 표현하고자 하는 사물의 모양을 삼각형 형태의 폴리곤 집합으로 구분하고, 폴리곤을 구성하는 세 개의 정점 좌표와 각 정점의 색, 법선 벡터 등의 데이터를 응용 프로그램으로부터 입력 받아 그림1과 같은 고정된 파이프라인을 통해 최종적으로 렌더링 된 결과를 화면에 표현하는 일련의 과정을 말한다.

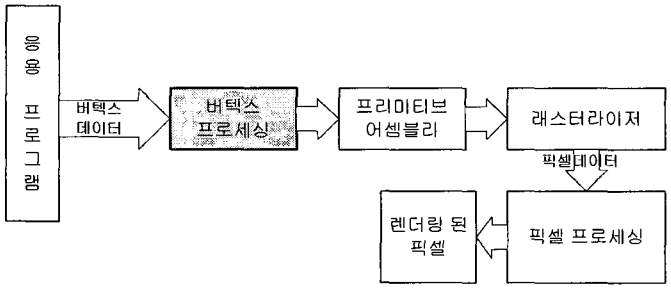


그림 1. 그래픽스 파이프라인

처리 과정 중 정점 프로세싱을 구체화 하면 그림 2 와 같이 모델 좌표계에서 스크린 좌표계로 변환하는 과정과 그림 3 과 같은 조명 계산 과정으로 구분된다. 좌표 변환 과정은 정점 좌표값을 행렬 계산식으로 변환하는 일련의 과정이며, 조명 처리 과정은 색에 관계된 정점 데이터를 조명 방정식에 따라 연산하는 과정이다.

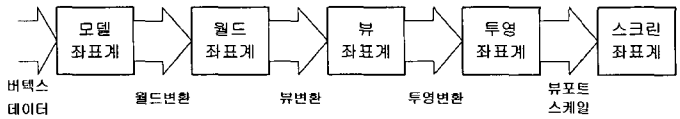


그림 2. 좌표 변환 과정

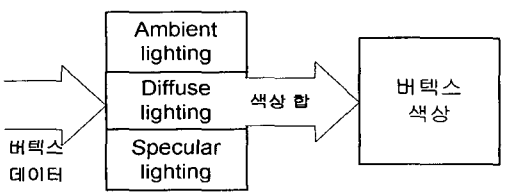


그림 3. 조명 계산 과정

OpenGL ES 1.x 표준은 위와 같은 좌표변환과 조명계산을 위한 정점 프로세싱을 정의하고 있으며, 고속 처리를 위하여 고정 기능의 파이프라인을 설계하여 연산하도록 하고 있다. 하지만 3D 그래픽 엔진을 설계하는 단계에서 기능을 고정하고 있으므로 고정 파이프라인 기반의 그래픽 응용 프로그램에서는 다양한 그래픽 표현에 제약이 따른다.

최근 PC 환경의 그래픽 엔진에서는 고정 기능의 파이프라인이 아닌 프로그래밍 가능한 파이프라인을 제공하여 기존 고정 파이프라인에서 불가능했던 유연한 그래픽스 기술을 제공하고 있다. PC 환경의 프로그래밍 가능한 파이프라인은 DirectX 와 OpenGL ARB Extension 그래픽 라이브러리에 의해 제공되고 있지만 [1-7], 모바일 환경에서는 2005년 9월 프로그래밍 가능한 파이프라인 구조의 OpenGL ES 2.0 표준이 발표되었지만 고급 언어에 관한 스펙만 정의되어 있고 이를 기반한 관련 제품도 출시되고 있지 않은 상태이다.

본 논문에서는 OpenGL ARB Extension 1.0 을 근거로 OpenGL ES 2.0 스펙과 부합하는 프로그래밍이 가능한 정점 프로그램 가상머신을 제시한다. 2 장에서는 본 연구에서 제안하는 정점 프로그램 가상머신 구조를 소개하고 3 장에서는 명령어 셋과 머신 코드 인코딩 방법을 설명한다. 4 장에서는 가상머신에서 수행하는 핵심적 기능인 머신코드 디코딩 방법을 제시한 다음, 5 장에서 몇가지 실험을 통해 본 방법의 성능을 평가한 후 6 장에서 결론을 맺는다.

2. 가상머신 구조

OpenGL ARB Extension 에서는 프로그래밍 가능한 정점 프로세싱 과정을 정점 프로그램이라 하고, 일련의 프로세싱 과정을 CPU 와 유사한 연산 장치를 통해 프로그래밍 가능하도록 한다. 이러한 처리 구조는 연산에 대한 명령어 코드와 연산 과정에 필요한 데이터를 저장하는 레지스터 장치를 필요로 한다.

가상머신에서는 정점 데이터와 연산에 필요한 상수 값을 입력 받아 명령어 코드에 따른 연산을 수행하여 결과를 출력한다. 출력된 결과는 정점 프로그램에 의한 연산 과정을 통한 결과이다. 이는 같은 정점 데이터라 할지라도 작성된 정점 프로그램에 따라 다양한 연출이 가능함을 말한다.

그림 4 중 상단 그림은 OpenGL ARB extension 1.0 구조의 정점 프로그램 처리 과정을 블럭화 한 것이며, 하단 그림은 위의 과정을 에뮬레이트 하기 위해 필요한 연산 장치와 저장 장치를 소프트웨어로 구현한 본 논문에서 제시하는 정점 프로그램 가상머신(VM: Virtual Machine)의 구조도이다. 정점 프로그램을 구성하는 각 장치에 대한 제한 크기는 가상머신이 모바일 환경에서 구동되는 그래픽 엔진임을 고려하여 ARB v1.0 에서 지원하고 있는 최소 기준을 적용하였다. 각 장치의 역할과 구조는 다음과 같다.

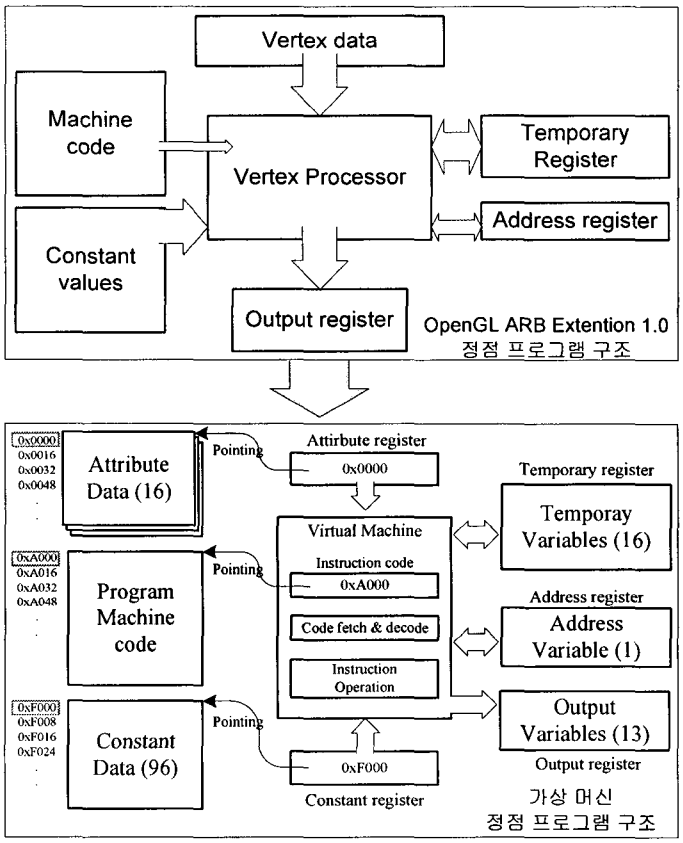


그림 4. 정점 프로그램 구조

- Machine code : 정점 프로그램에서 수행할 연산 과정을 정의하는 명령어 코드이다.
- Vertex Processor : Machine code를 해석하여 주변의 저장 장치와 내부의 연산장치를 사용하여 정점 프로세싱 과정을 처리하는 장치이다. 정점 프로그램의 핵심 장치로 명령어를 하나씩 읽고(fetch), 해독(decoding)한 후 수행(operate)하는 일련의 과정을 처리한다.
- Vertex data : 물체를 구성하는 각 정점들의 stream data이며 각 정점에 대한 좌표, 색상, 법선 벡터, 텍스쳐 좌표 등과 같은 속성 정보를 담고 있다. 각 속성 정보는 4개의 실수 집합으로 구성되는 정보로 ARB v 1.0에서는 Vertex data에 대해 최소 16개 속성을 지원하도록 하고 있다. Vertex processor에서는 읽기만 가능하다.
- Constant Registers : 정점 프로그램에서 사용되는 상수들의 저장

장치이다. 행렬 계산을 위한 값이나 특정 색상 값, 조명 계산을 위한 계수 값 등을 저장하는 장치이며, 각 상수 값은 4개 실수 집합으로 구성된다. ARB v1.0에서는 최소 96개 상수를 사용 할 수 있다. Vertex processor에서는 읽기만 가능하다.

• Temporary Registers : 정점 프로그램이 수행되는 동안 사용되는 임시 값을 저장하는 장치이다. ARB v1.0에서는 최소 12개를 지원하도록 하고 있으나, 본 가상머신에서는 기본 12개 이외에 특수한 용도로 사용할 수 있는 4개의 확장 공간을 포함하여 총 16개를 지원한다. 정점 프로그램이 수행되는 동안 값을 읽고 쓸 수 있으며, 각 레지스터 공간은 4개 실수 집합으로 구성된다.

• Address Register : Constant Registers에 저장된 상수를 읽고자 할 때, Address Register에 저장된 기본 주소(base address) 값에 대한 상대적인 위치(offset)를 참조할 때 사용된다.

• Output Register : 정점 프로그램에 의해 연산 처리된 결과 값을 저장하는 장치이다. 정점 프로세싱 처리된 결과는 그래픽 파이프라인의 나머지 처리 과정을 거치게 된다. ARB v1.0는 최소 13개 Output Register를 지원하도록 하고 있으며, 각 레지스터 공간은 4개 실수 집합으로 구성된다. Vertex processor에서는 쓰기만 가능하다.

Machine code, Constant data, Vertex data에 해당되는 Attribute data는 응용 프로그램에서 제공되는 데이터로 본 가상머신에서는 이들 데이터에 대해 별도의 저장 장치를 구현하여 접근하지 않고, 메모리 상에 저장되어 있는 데이터에 대한 포인터를 통해 직접 참조하는 방식을 사용한다. 이는 별도의 저장 공간을 할당하여 복사하는 과정에서 발생하는 시간 지연을 줄이기 위함이며, Machine code, Attribute data, constant data가 가상머신에서는 읽는 용도로만 사용되기 때문이기도 하다.

Temporary registers, Address register, Output registers는 가상머신의 수행 과정에서 데이터의 쓰기 동작이 이루어지는 장치로 가상머신의 환경을 초기화 하는 과정에서 장치의 크기 만큼 메모리를 할당하여 사용한다.

가상머신은 명령어를 읽고(fetch) 해독하여(decoding) 수행하는(operate) 과정을 통해 정점 프로그램을 처리한다.

3. 명령어 형식

정점 프로그램은 vertex processor 에서 명령어 코드를 해석하여 연산함으로써 동작한다. 명령어 코드는 정점 프로그램에서 어떠한 연산을 수행할지를 나타내는 코드이다. 정점 프로그램은 프로그래머의 편의를 위해 어셈블리 언어와 유사한 저수준 언어(low-level programing language)를 통해 프로그래밍이 가능하며, 어셈블러를 통해 가상머신에서 실행 가능한 기계 코드로 번역된다. 기본 문법 구조는 다음과 같이 명령어(opcode), 대상 오퍼런드(destination operand), 소스 오퍼런드(source operand)로 구성된다.

```
opcode  destination operand, source operand0 [, source operand2, source operand3];
```

그림 5. 기본 문법구조

Vertex Program ARB v1.0 의 기본 문법 구조에서 오퍼런드는 각 레지스터와 바인드된 변수 또는 레지스터의 이름이 직접 쓰이고 있으며, 본 가상머신에서는 이들의 참조를 위해 각 레지스터의 Index 주소를 사용한다.

표 1. Primitive Instruction

Primitive Instruction	description
ARL	Address register load
MOV	move
ABS	Absolute
FLR	floor
FRC	fraction
SWZ	Extended swizzle
ADD	addtion
MUL	multiply
DST	Distance vector
XPD	Cross product
MAX	maximum
MIN	minimum
SGE	Set on greater or equal than
SLT	Set on less than
DPH	Homogeneous dot product
DP3	3-component dot product
DP4	4-component dot product
clamp	Clamp
mulz	Multiply on z
MAD	Multiply and add
EXP	Exponential base 2 (approximate)
LOG	Logarithm base 2(approximate)
EX2	Exponential base 2
LG2	Logarithm base 2
RCP	Reciprocal
RSQ	Reciprocal square root
rEX2	Exponential base 2(rough)
rLG2	Logarithm base 2(rough)

표 2. Macro Instruction

Macro Instruction	description
LIT	: Light coefficients LIT f, a, b *clamp tmp, a.0, b* *rLG2 tmp.w, tmp.w* *MUL tmp.w, tmp.w, tmp.y* *rEX2 tmp.w, tmp.w* *mulz f, tmp.1xz1, tmp.w*
POW	POW f, a, b ($f = a^b$) *LG2 tmp, a* *MUL tmp, tmp, b* *EX2 f, tmp*
SUB	SUB f, a, b (f = a − b) *ADD f, a, -b*

정점 프로그램의 명령어 Set 은 ARB v1.0 에서 제시된 27 개를 근거로 연산 종류 및 처리 방식에 따라 표 1, 2 와 같이 28 개 Primitive Instruction 과 3 개 Macro Instruction 으로 분류하였다. Macro Instruction 은 일련의 Primitive Instruction 으로 처리가 가능하므로 가상머신에서 해석하여 처리하는 Instruction 은 Primitive Instruction 이다. 표 1 의 음영 처리된 Instruction 은 ARB v1.0 명령어 Set 에는 포함되지 않으나 Macro Instruction 으로 분류한 명령어의 처리를 위해 정의하여 추가된 명령어이다.

프로그래머가 작성한 정점 프로그램은 어셈블러를 통해 가상머신에서 실행 가능한 기계 코드로 번역된다. 기계 코드는 위에서 설명한 명령어 형식을 지원하며, 그림 6 과 같이 총 64bits 구조를 가진다. 하위 [17:3] 비트 영역은 오퍼런드(Src2) 필드와 확장 스위즐 필드가 중첩되는 영역으로 3 개의 소스 오퍼런드를 가지는 MAD 명령어일 경우는 오퍼런드 필드로 사용하고 그 밖의 경우에서는 확장 스위즐 필드로 사용된다.

기계 코드의 각 필드는 다음과 같이 구성된다.

• Opcode field : opcode

명령어(opcode)를 구분한기 위한 bit field 로 6bit 의 공간을 할당한다. (2^6 = 64, 최대 64 개 Opcode 지정가능)

• Destination operand field : Dest

명령어 수행 결과를 저장할 레지스터의 종류(type)와 위치(index), mask 정보를 구분하기 위한 비트필드로 9 비트의 공간이 할당되며 각 비트는 다음과 같이 해석된다.

T (1bit) : destination operand 의 종류 결정	/ 0(Temporary) / 1(Result)
index (4bit) : operand 의 index 값	/ 범위 : 0~15
mask (4bit) : operand 각 component 에 대한 mask flag	

• Source operand(n) field : Src(n), 0≤ n ≤2

명령어 수행에 필요한 n+1 번째 소스 레지스터의 종류와 위치, 스위즐 정보를 구분하기 위한 비트 필드로 15 비트의 공간을 할당하며 소스 오퍼런드의 해석은 명령어와 오퍼런드 종류에 따라 확장 스위즐이 적용되는 경우와 적용되지 않는 경우로 구분하여 처리된다.

• Exteneded Swizzle bit field

Extended Swizzle 정보는 Source Operand 0의 확장 정보로 사용되며, 각각의 component에 대한 negation이 가능하고 swizzle의 선택에 x, y, z, w component 외에 1, 0을 선택할 수 있게 되어 다음과 같은 사용이 가능하다.

Colr.{-}[01xyzw] {-}[01xyzw] {-}[01xyzw] {-}[01xyzw]

• Extended constant index field

Constant register의 인덱스 정보를 위한 확장된 상위 인덱스 정보이다. source operand의 인덱스(index : 4bit)정보와 확장된 상수 인덱스(extended constant index(const : 3bit))를 통해 7 비트 인덱싱 이 가능하다.

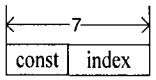


그림 11. 확장 인덱스

* 2^7(128) : Constant register indexing (확장 + 기본 index 조합)

4. 디코딩 방법

가상머신이 기계 코드를 해석하는 동작은 정의된 명령어 형식을 바탕으로 하며, 각 필드에 대한 해석 동작은 다음과 같다.

• Opcode decoding

기계 코드로부터 명령어의 종류를 해석하는 과정으로 정의한 기계 코드의 [63:58] 비트 영역에 위치하는 정보를 논리 연산을 통하여 mask 한다.

• Destination operand decoding (& updating)

명령어와 소스 오퍼런드를 통해 연산을 수행한 후 대상 오퍼런드의 디코딩을 통해 결과 값을 저장할 위치를 결정한다. 대상 레지스터의 위치가 결정되면 각 component 의 mask 정보에 따라 (mask bit 가 1 이면 업데이트) 값을 저장시킨다.

1 비트 type 를 통해 대상 레지스터의 종류를 결정하며 4 비트 인덱스 정보를 offset 값으로 하여 대상 레지스터를 인덱싱한다.

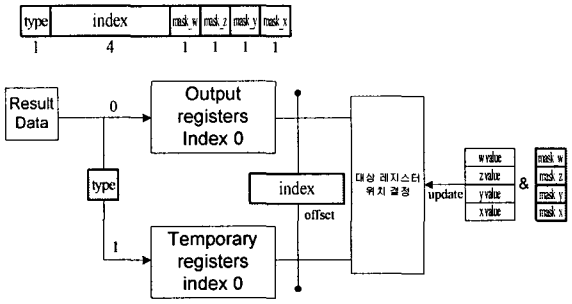


그림 12. Destination operand 의 디코딩 및 업데이트

• Source operand decording

소스 오퍼런드 필드의 해석은 확장 스위즐 필드를 적용하여 해석할 경우와 그렇지 않은 경우의 두 가지 경우에 따라 해석 방법을 달리한다. MAD 명령어의 경우를 제외한 명령어에 대해서 첫 번째 소스 오퍼런드는 확장 스위즐 필드와 조합하여 해석함을 기본으로 한다.

확장 스위즐 필드 적용의 경우는 그림 13 와 같은 과정으로 디코딩 한다. 소스 오퍼런드 필드로부터 type 정보를 통해 소스 레지스터의 종류를 결정하고 확장 인덱스 필드와 조합하여 소스 레지스터의 위치를 인덱싱한다. (확장 인덱스를 사용하지 않는 레지스터일 경우 const 필드의 값은 0 이므로 모든 레지스터의 인덱싱에서 확장 인덱스 필드를 사용하여도 문제가 발생하지 않는다.)

소스 레지스터의 위치가 결정되면 스위즐 정보를 통해 component 를 선택하고 부호정보를 반영하여 최종 소스 오퍼런드 값을 결정할 수 있다. 그림 13 에서는 첫번 째 component 스위즐 과정(b)을 보인다.

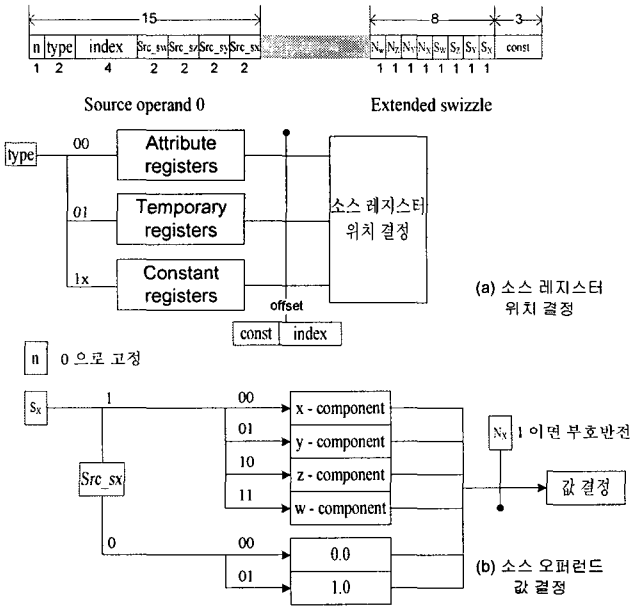


그림 13. Source operand 0 의 디코딩 과정 (확장 스위즐 적용)

확장 스위즐 필드가 적용되지 않는 경우의 소스 오퍼런드 디코딩은 그림 13 의 (a) 과정은 동일하나 (b) 과정은 다음 그림 14 와 같이 동작한다.

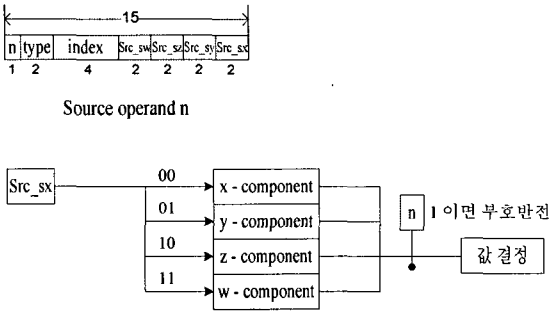


그림 14. Source operand 디코딩 과정 (일반)

5. 실험

본 실험은 펜티엄4 3Ghz, ATI Radeon 9800xt 그래픽 카드 환경에서 수행되었다. 본 연구방법을 실험하기 위하여 OpenGL ARB extension 1.0 을 기준으로 하여 정점 프로그램을 작성하고 정점 프로세싱을 가상 머신을 통해 수행함으로써 결과를 확인하도록 하였다. 본 가상머신의 연산기는 ES 2.0에 근거하여 24bits 부동소수점 처리를 지원한다. 실험에 사용한 데이터는 표 3과 같은 3가지 샘플로 구성되었다.

표 3. 실험에 사용된 데이터

	샘플 1	샘플 2	샘플 3
모델	angel (6984 vertices)		
프로그램 구분	노말값 출력	쿡-토런스 조명 모델	환경 맵
명령문장 수	6	31	21

화질상 비교를 위하여 그림 15와 같이 동일한 정점 프로그램 샘플에 대하여 PC 환경의 그래픽 하드웨어를 통한 렌더링 이미지와 가상 머신을 통한 렌더링 이미지를 비교하였다. 실험결과 세가지 샘플 모두 육안으로 식별할 수 없는 음영차를 보였다. 단 윤곽 부분과 음영의 차가 심하게 변하는 부분에서 정점의 위치 정밀도 차이(PC 환경은 32bit 부동소수점 연산처리인 반면 본 방법은 24bit 부동소수점 연산을 하므로) 때문에 약간의 색상차이를 볼 수 있다. 샘플 3은 다른 두 샘플에 비해 심한편이다. 환경 매핑 방법은 법선 벡터 값이 조금만 변해도 텍스쳐좌표가 크게 변하는 경우가 있다. 이로 인해서 계산된 텍스쳐좌표값에 오류가 있을 수 있다. 텍스쳐 좌표의 오류가 있는 경우 다른 위치의 색상을 가져오게 되서 원래 색상과 차이가 크게 발생한다.

속도상 성능비교는 표4와 같다. 수행 속도는 정점 프로그램에서 처리할 명령어 수행 사이클에 비례함을 알 수 있다.

표 4. 속도 비교

	샘플 1	샘플 2	샘플 3
명령문장 수	6	31	21
FPS	61.79	17.54	26.2

6. 결론

프로그래밍 가능한 파이프라인 구조를 지원하는 모바일용 제품은 아직 출시되지 않았다. 본 논문은 OpenGL ARB Extension 표준을 근거로 모바일 환경에 맞게 정점 프로그램 가상머신을 설계하였고, 표준 스펙에 부합하는 명령어 셋을 정의하고 효과적인 머신코드 인코딩과 디코딩 방법을 제시하였다. 본 연구에서 제안한 가상머신의 성능을 비교한 결과 연산가상의 비트정의가 PC 와 상이함에 따른 것을 제외하면 화질상의 오류가 없이 대화형으로 구동이 가능함을 알 수 있었다. 향후 연구로 프래그먼트 프로그램 가상머신을 개발하여 고정된 파이프라인의 정점과 프래그먼트 처리를 프로그래밍 가능하도록 대치하여 전반적인 파이프라인상에서 본 시스템의 성능을 평가해 보고자 한다.

참고문헌

[1] Randi J. Rost, OpenGL Shading Language, Addison Wesley, 2004.

[2] Kris Gray, DIRECTX 9 PROGRAMMABLE GRAPHICS PIPELINE, 정보문화사, 2004.

[3] 타카시 이마기레, DirextX 9 셰이더 프로그래밍, 한빛 미디어 & MYCOM, 2004.

[4] Richard S. Wright Jr. & Benjamin Lipchak, OpenGL SUPER BIBLE Third edition, SAMS, 2005.

[5] http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_program.txt

[6] http://www.lighthouse3d.com/opengl/glsl/

[7] http://www.cgshaders.org/

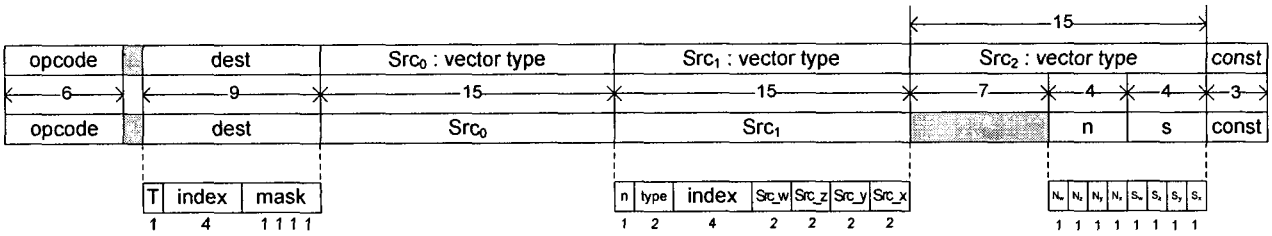


그림 6. 머신 코드 형식 (: unused bit)

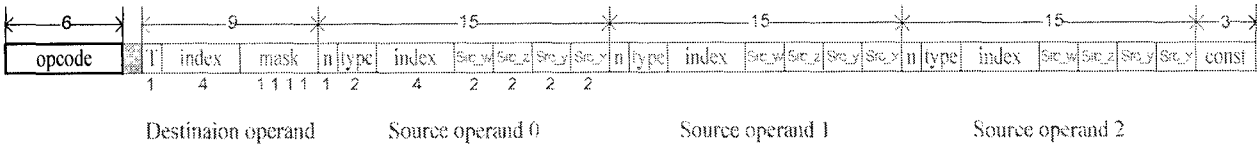


그림 7. 머신 코드 형식 (opcode)

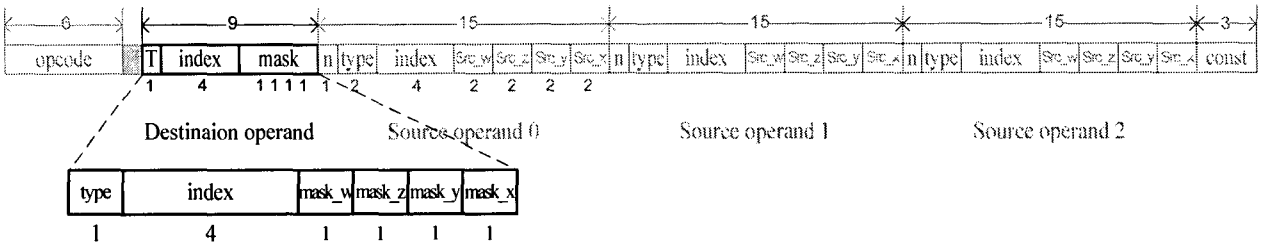


그림 8. 머신 코드 형식(Destination operand)

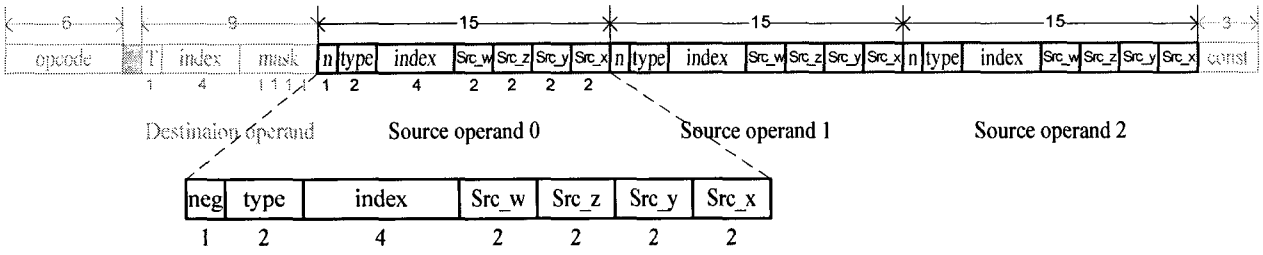


그림 9. 머신 코드 형식 (Source operand)

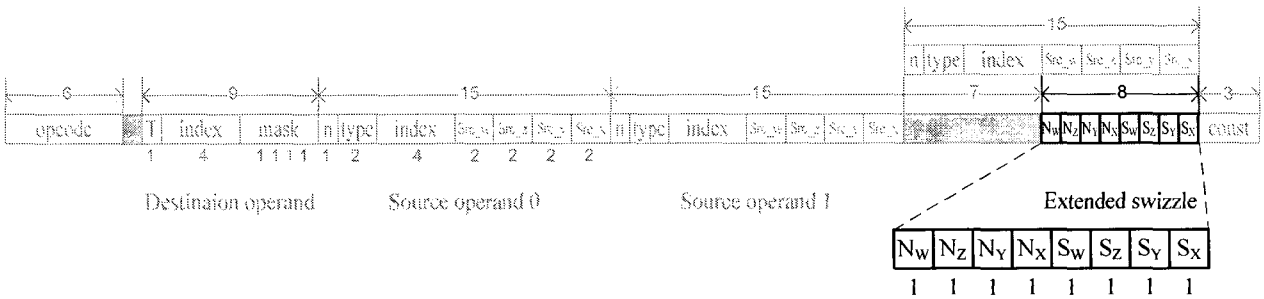
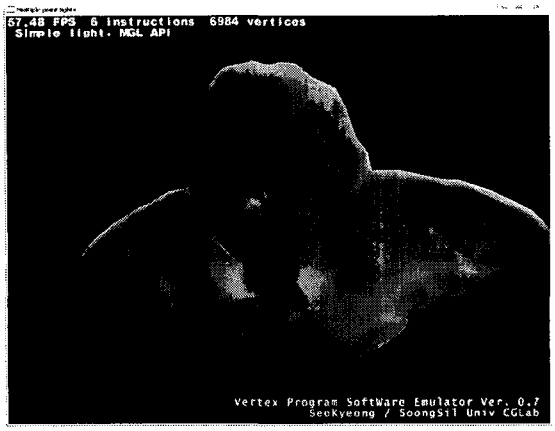
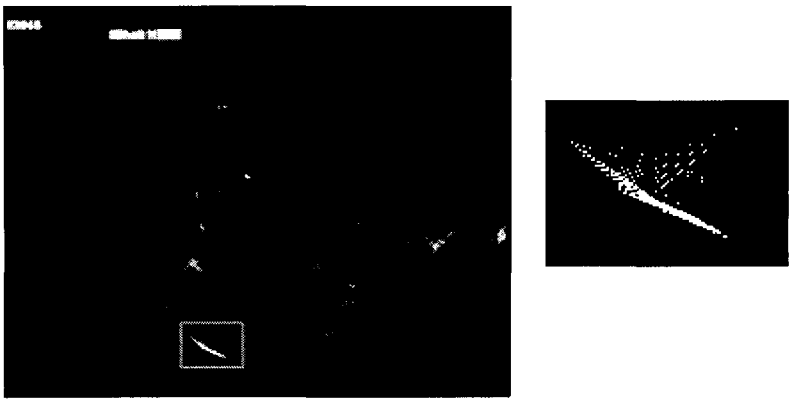


그림 10. 머신 코드 형식 (Extended swizzle)

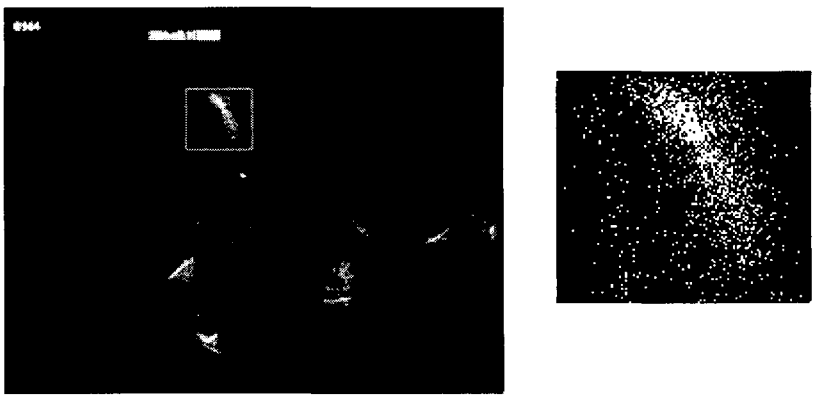


(a) 샘플1 가상 머신 렌더링 이미지

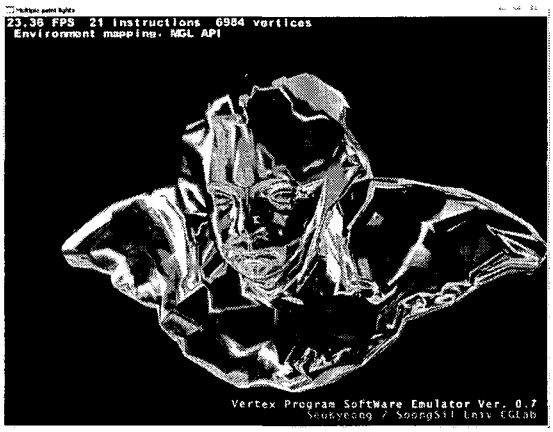


(b) 샘플1 오류표현이미지

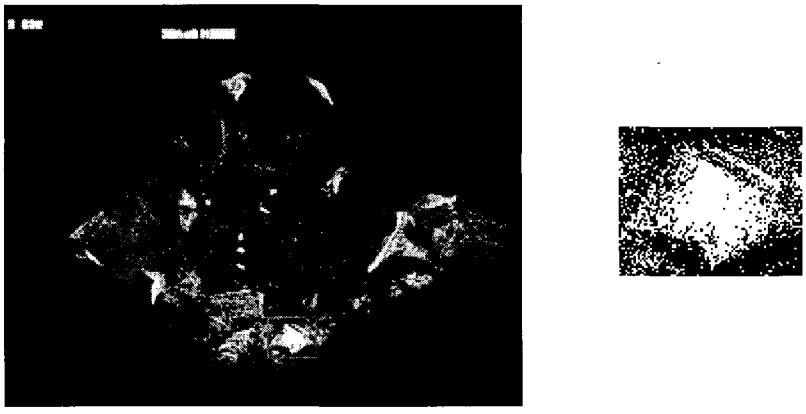
(c) 샘플2 가상 머신 렌더링 이미지



(d) 샘플2 오류표현이미지



(e) 샘플3 가상 머신 렌더링 이미지



(f) 샘플3 오류표현이미지

그림 15. 화질비교 ((b) (d) (f)는 색상차가 보이는 픽셀을 하얀색으로 표현한 이미지이다.)