OpenGL ES 2.0 기반 셰이더 설계

김종호[0], 최완, 김성진, 김태영
서경대학교 컴퓨터 그래픽스 연구실
{sebhi,wanne,likellihu,tykim}@skuniv.ac.kr

Design of a Shader Based on the OpenGL ES 2.0

Jong-Ho Kim[0], Wan Choi, Sung-Jin Kim, 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.x을 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. PC환경의 프로그래밍 가능한 파이프라인은 DirectX 와 OpenGL 그래픽 라이브러리에 의해 제공되고 있지만, 모바일 환경에서는 이를 지원하기 위한 관련 제품이 아직 출시되지 않고 있는 상태이다. 본 논문에서는 2005년 9월에 발표된 프로그래밍 가능한 그래픽스 파이프라인에 대한 표준인 OpenGL ES 2.0에 기반한 효율적인 셰이더 구조와 이의 구동방식을 제시한다. 본 연구는 PC상에서 소프트웨어로 개발되었고, 연구 결과는 그래픽스 하드웨어 설계를 위한 검증용으로 사용될 수 있을 뿐 아니라 응용 프로그래머의 모바일 콘텐츠 제작을 위하여 활용될 수 있다.

1. 서론

최근 모바일 폰 사용자가 기하급수적으로 늘어남에 따라 관련 모바일 시장은 매년 급속한 성장을 보이고 있다. 이에 따라 3D 게임, 동영상등 3D 기술의 효과적 표현을 위한 모바일에서의 3D 그래픽스 엔진의 필요성이 대두되었다. 모바일은 PC에 비하여 저사양이기 때문에 자원관리에 있어 고효율을 요구하며 휴대성에 의한 장기 사용시간을 위하여 저전력에 바탕하고 있다. 이런 모바일의 특성에 적합한 3D 그래픽스 라이브러리로 떠오른 것이 OpenGL ES 2.0 이다. 2005년 9월 발표된 OpenGL ES 2.0은 기존의 OpenGL ES 1.X가 그림 1과 같은 고정된 그래픽스 파이프라인을 지원하는 모바일 환경의 그래픽스 라이브러리인 것에 반해 그림 2와 같은 프로그래밍 가능한 그래픽스 파이프라인을 지원한다.

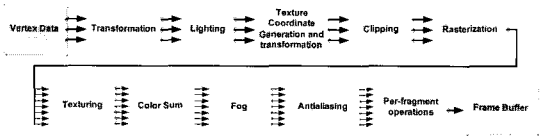


그림 1. 고정된 그래픽스 파이프 라인(OpenGL ES 1.X)

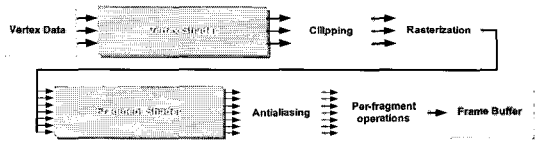


그림 2. 프로그래밍 가능한 그래픽스 파이프 라인(OpenGL ES 2.0)

프로그래밍 가능한 셰이더는 DirectX 8.0부터 도입된 기술로서 좌표 변환이나 조명 계산을 커스터마이징 할 수 있도록 한다. 따라서 영화에서나 볼 수 있었던 다양하며 풍부한 렌더링의 실현이 PC 환경에서도 가능하게 되어 3차원 그래픽스 기술의 많은 발전을 가져왔다. 이러한 셰이더 기능을 저사양 플랫폼인 모바일에서도 제공하고자 하는 시도가 버텍스 셰이더와 프래그먼트 셰이더를 포함하는 OpenGL ES 2.0의 표준안 정의이다.

버텍스 셰이더는 각 정점에 가해지는 연산을 정의한다. 기존 파이프라인 중 좌표 변환과 조명 계산 부분이 버텍스 셰이더로 대체가 되는 것이다. 버텍스 셰이더는 각 정점과 상수 요소들을 입력으로 받는다. 버텍스 셰이더의 출력은 좌표 변환과 조명이 완료된 정점 정보이다[1-3].

프래그먼트 셰이더는 각 프래그먼트에 가해지는 연산을 정의한다. 기존 파이프 라인에서는 각 프래그먼트에 대하여 '다중텍스쳐' 기능을 처리하기 위한 직접적인 명령어를 지원하는 것이 아니라 고정된 구조에 상태 값들을 조정하는 방법으로 여러개의 텍스쳐 값을 읽어서 연속된 이진 연산의 제한된 형태의 연산만을 제공한다. 프래그먼트 셰이더는 이 부분을 대체하여 프로그램밍을 통하여 보다 일반적이고 복잡한 연산을 가능하게 한다[1-3]. 즉 기존 파이프 라인 중 텍스쳐 값 읽기, 색상 계산 등의 연산을 이 프래그먼트 셰이더로 대체 가능하다.

본 논문에서는 OpenGL ES 2.0에 기반한 셰이더 구조를 제시하고 셰이더 프로그래밍을 위한 명령어 집합 및 머신 코드 형식을 제안한다. OpenGL ES 2.0 표준은 고수준 셰이딩 언어만 정의[4]하고 있으므로 이를 효율적으로 처리하기 위한 저수준 언어(Low-level language) 및 머신 코드 형식의 정의가 필요하다. 본 논문에서 제시된 명령어 집합 및 머신코드 형식은 향후 하드웨어 제작업체들을 위한 표준 인터페이스로 활용될 수 있다.

2. 셰이더 구조

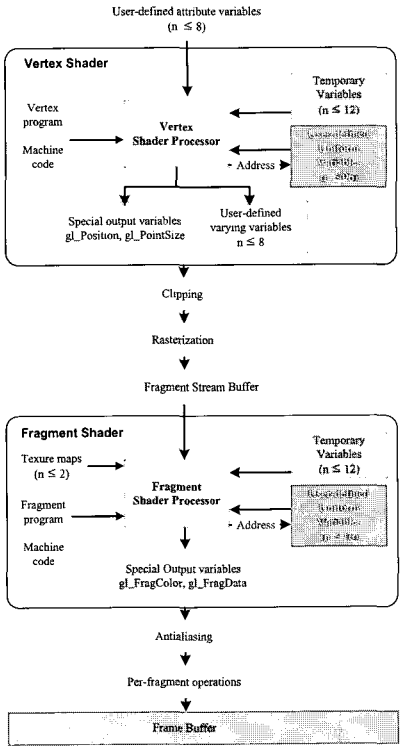


그림 3. 그래픽스 파이프 라인의 상세 구조도

그림 3은 OpenGL ES 2.0 Spec 및 Shading Language의 요구사항을 나타낸 그래픽스 파이프 라인 구조이다.[1,5] 버텍스 셰이더는 각 정점 및 정점 계산에 필요한 상수 요소들을 입력으로 받아 좌표 변환 및 조명을 계산하여 변환 된 값을 출력하며 이러한 출력 값들은 Clipping과 Rasterization 과정을 마친 후 프래그먼트 셰이더에서 입력 값으로 쓰이기 위하여 프래그먼트 버퍼에 담긴다[2]. 프래그먼트 셰이더는 버퍼의 값을 입력으로 받아 텍스쳐 연산과 색상 계산 및 처리를 마친 후 안티앨리어싱과 알파 테스트, 깊이 테스트, 스텐실 테스트 등의 프래그먼트 대상 작업을 통해 최종적인 디스플레이를 위한 프레임 버퍼로의 처리과정을 가진다.[2] 그림에서 보는 바와 같이 각 셰이더의 내부구조를 보면 버텍스 셰이더와 프래그먼트 셰이더는 역할은 구분되나 기능적으로 매우 유사함을 알 수 있다. 따라서 본 논문에서는 버텍스 셰이더와 프래그먼트 셰이더를 통합하는 Unified 셰이더 개념을 도입하였다. Unified 셰이더는 버텍스 셰이더와 프래그먼트 셰이더를 구분하지 않고 하나의 셰이더가 처리과정에 따라 두 셰이더의 역할을 수행하기 때문에 셰이더 내부처리에 따른 자원(Temporary, Uniform 등)을 공유한다[6].

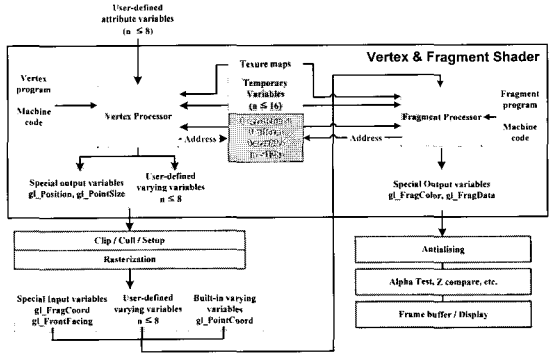


그림 4. 본 연구에서 제시하는 그래픽스 파이프 라인의 설계도

그림 4는 본 연구에서 설계한 그래픽스 파이프 라인이다. Unified 셰이더는 버텍스 셰이더와 프래그먼트 셰이더를 구분하지 않으나 각 셰이더의 동작 흐름을 표현하기 위하여 구분하여 제시하였다. 또한 버텍스 셰이더와 프래그먼트 셰이더가 내부 자원을 공유하기 때문에 Uniform 및 Temporary 의 개수를 각 128, 16개로 확장하였고, 두 셰이더 모두 텍스쳐 모듈의 접근이 가능토록 하였다.

다음은 세이더 내부자원에 대한 설명이다[5].

• Attribute : 모델의 정점 구성 요소들로 Position, Normal, Color 등의 값을 가지는 입력 레지스터이다.

• Uniform : 정점 세이더 및 프래그먼트 세이더 처리를 위해 사용되는 상수들로 Matrix, Light, 픽셀 가중치, Shiness 등과 같은 세이더 연산에 공통적으로 사용될 상수와 사용자 정의에 따른 상수로 구성된다. Uniform은 버텍스 세이더와 프래그먼트 세이더가 공유하는 레지스터로 최대 128개 요소를 가진다.

• Address : 버텍스 세이더와 프래그먼트 세이더에서 사용할 Uniform의 구성이 배열과 같은 구성일 시 Base 주소값을 저장하여 Uniform에 접근할 수 있도록 한다.

• Temporary : 입/출력이 가능한 요소로 버텍스 세이더와 프래그먼트 세이더를 처리하는 과정에서 임시로 사용할 수 있는 레지스터이다.

• Position, PointSize : 버텍스 세이더 처리 과정 후 좌표 변환된, 즉 Model 좌표로부터 View Plane으로의 Projection된 좌표 값이 Position 요소에 출력되며 Primitive가 점 일때 한 점의 크기가 PointSize 요소로 출력된다.

• Varying : 버텍스 세이더의 출력 레지스터로 프래그먼트 세이더에서 사용하기 위한 값들이 출력된다. Varying은 사용자에 의해서 정의되며 래스터라이저를 거쳐 보간된 값은 프래그먼트 세이더의 입력 요소로 사용된다.

• FragCoord, PointCoord, FrontFacing : 래스터라이저를 통해 결정되는 값들로 각 프래그먼트의 좌표 값(FragCoord), Primitive가 point일시 FragCoord를 기준으로 한 프래그먼트의 x, y좌표 (PointCoord)와 FrontFace 여부 등의 정보를 포함한다.

• FragColor, FragData : 프래그먼트 세이더의 처리 결과로 출력되는 정보이다. FragColor는 Single Render Target의 경우 사용되는 출력정보이고 FragData는 Multi Render Target을 사용할 경우의 출력정보이다. FragColor와 FragData는 동시에 사용될 수 없다.

각 세이더 프로세서는 머신 코드를 읽고(fetch) 해독하여 (decoding) 수행하는(operate) 과정을 통해 세이더 프로그램을 처리한다.

3. 세이더 프로그램 명령어

3.1 명령어 형식

세이더 프로그램은 세이더 프로세서에서 명령어 코드를 해석하여 연산함으로써 동작한다. 명령어 코드는 세이더에서 어떠한 연산을 수행할지를 나타내는 코드이다. 세이더 프로그램은 프로그래머의 편의를 위해 어셈블리 언어와 유사한 저수준 언어 (low-level programing language)를 통해 프로그래밍이 가능하며, 어셈블러를 통해 세이더에서 실행 가능한 머신코드로 번역된다.

기본 문법 구조는 다음과 같이 명령어(opcode), 대상 오퍼런드 (destination operand), 소스 오퍼런드 (source operand)로 구성된다.

```
opcode destination operand, source operand0 [, source operand1, source operand2];
```

그림 5. 기본 문법구조

표 1. Primitive Instruction

Primitive Instruction	Description
NOP	No operation External operation
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*

세이더 프로그램의 명령어 집합은 ARB v1.0[7-8]에서 제시된 명령어 집합을 근거로 연산 종류 및 처리 방식에 따라 표 1과 같이 29개 Primitive Instruction으로 분류하였다. Macro Instruction은 POW와 같은 일련의 Primitive Instruction으로 처리 가능한 명령어들을 말한다. 표 1의 음영 처리된 Instruction은 ARB v1.0 명령어 집합에는 포함되지 않으나 Macro Instruction의 처리를 위해 본 연구에서 추가한 명령어이다.

3.2 머신 코드 구조

프로그래머가 작성한 세이더 프로그램은 어셈블러를 통해 세이더 가상머신에서 실행 가능한 머신 코드로 번역된다. 머신 코드는 위에서 설명한 명령어 형식을 지원하며, 그림 6과 같이 총 64bits구조를 가진다. MAD 명령을 사용하는 문법의 경우 MAD 명령어 형식으로 인코딩되며 Texture load 명령과 같은 외부 모듈을 사용하는 문법의 경우 NOP 확장 명령어 형식을 따른다. 그 외 기타 명령어 형식은 모두 Standard 명령어 형식으로 인코딩 된다.(그림 6, 7 참조)

각 머신 코드 구조에 따른 세부 설명은 아래와 같다.

(1) Opcode field : opcode

명령어(opcode)를 구분하기 위한 비트 필드로 6bit의 공간을 할당한다. (2^6 = 64, 최대 64개 Opcode 지정가능)

(2) Destination operand field

• MAD 명령어 및 Standard 명령어의 경우

명령어 수행 결과를 저장할 레지스터의 종류(type)와 위치(index), mask 정보를 구분하기 위한 비트필드로 9bit의 공간이 할당된다.

• Texture Load와 같은 NOP 확장 명령어의 경우

외부 모듈 사용여부(External flag)와 외부모듈의 종류(External Type), 값을 저장할 위치(index), Texture 모듈 사용 시 접근할 Texture의 Sampler 구분(Sampler_id) 정보를 위한 비트 필드로 9bit의 공간이 할당된다.

(3) Source operand(n) field : Src(n), $0 \leq n \leq 2$

명령어 수행에 필요한 n+1 번째 소스 레지스터의 종류와 위치, 스위즐 정보를 구분하기 위한 비트 필드로 15bit의 공간을 할당하며 소스 오퍼런드의 구성은 명령어와 오퍼런드 종류에 따라 Extended Swizzle bit field 및 Extended Constant bit field의 정보를 확장하여 사용하기도 한다.

• MAD 명령어의 경우

MAD 명령어의 경우 3개의 소스 오퍼런드를 요구하며 그림 8과 같은 구조의 머신 코드로 인코딩한다. Source Operand0, Source Operand1, Source Operand2 field 구성은 각 15bit로 동일한 구조를 가진다.

Temporary, Attribute/Varying의 경우0~15의 범위 내로 표현 가능하므로 4bit Index로 충분하나 Uniform의 경우 0~127의 범위를 가지므로 4bit만으로 구분이 불가능하다. 명령어에서 Uniform을 항상 사용하는 것이 아니므로 Index에 7bit를 할당하는 것은 전체 명령어 크기를 늘리는 결과를 가진다. 따라서 Type에 따라 Uniform을 사용하는지 여부를 확인하고 Extended Const field 3bit를 Source Operand field의 Index에 확장하여 0~127의 범위 표현이 가능하도록 하였다.

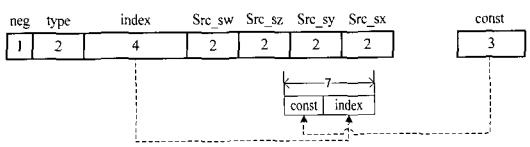


그림 9 Source Operand bit field 및 Extended Constant bit field

• Standard 명령어 및 NOP 확장 명령어의 경우

Standard 명령어의 경우 1 개 또는 2 개 소스 오퍼런드를 요구하며 그림 10(위)와 같은 형식의 머신 코드로 인코딩된다. 확장 명령어의 경우는 1 개의 소스 오퍼런드를 요구하며 그림 10(아래)와 같은 형식의 머신 코드로 인코딩된다. 위의 두 가지 명령어 형식에서 Source operand0 field는 Extended Swizzle field와 함께 Source operand0 의 정보를 구성한다. 확장 스위즐은 Source Operand0 에 대하여 component 별로 부호를 지정하고 x,y,z,w외 0 또는 1 을 선택할 수 있도록 한다. 이러한 구조는 셰이더 프로그램 작성 시 명령어 개수를 줄여주어 연산 과정을 단순화 시킨다.

3.3 머신 코드 디코딩

• Opcode decoding

머신 코드로부터 명령어의 종류를 해석하는 과정으로 정의한 머신 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 component의 mask 정보에 따라 (mask bit가 1이면 업데이트) 값을 저장시킨다.

type정보를 통해 대상 레지스터의 종류를 결정하며 4bit Index 정보를 offset 값으로 하여 대상 레지스터를 인덱싱한다.

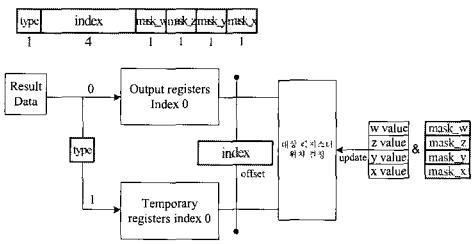


그림11. Destination operand의 디코딩 및 업데이트

• Source operand decoding

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 그림12와 같은 과정으로 디코딩 한다. 소스 오퍼런드 필드로부터 type 정보를 통해 소스 레지스터의 종류를 결정하고 확장 인덱스 필드와 조합하여 소스 레지스터의 위치를 인덱싱한다. 소스 레지스터의 위치가 결정되면 스위즐 정보를 통해 component를 선택하고 부호정보를 반영하여 최종 소스 오퍼런드 값을 결정할 수 있다. 그림12에서는 첫번째 component 스위즐 과정(b)을 보인다.

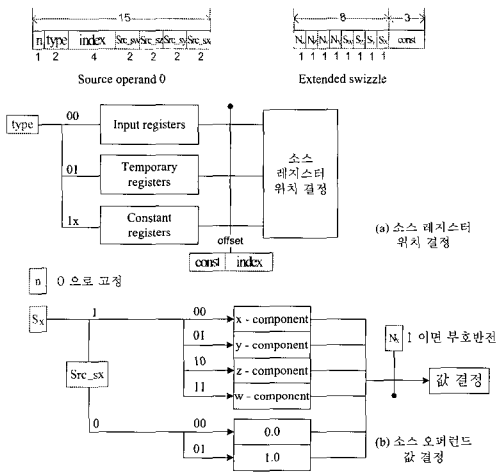


그림12. Source operand 0의 디코딩 과정 (확장 스위즐 적용 시)

확장 스위즐 필드가 적용되지 않는 경우의 소스 오퍼런드 디코딩은 그림12의 (a) 과정은 동일하나 (b) 과정은 다음 그림13와 같이 동작한다.

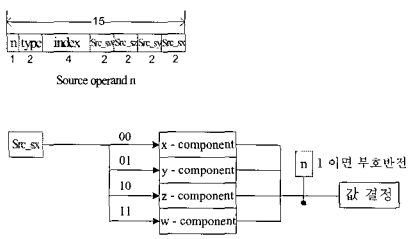


그림13. Source operand 디코딩 과정 (일반)

4. 실험

본 논문에서 제시된 셰이더를 기설계된 고정된 그래픽스 파이프라인에 포팅시키고, 펜티엄4.3Ghz, ATI Radeon 9800xt 그래픽 카드의 환경에서 성능 실험을 수행하였다. 화질상 비교를 위하여 동일한 셰이더 프로그램 샘플에 대하여 PC 환경의 그래픽 하드웨어를 통한 렌더링 이미지와 제시된 셰이더를 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 대상 환경이 모바일임을 감안한 24bit 부동소수점 연산을 수행) 때문에 약간의 색상차이를 볼 수 있다. 샘플 3의 환경 매핑 방법은 법선벡터 값이 조금만 변해도 텍스쳐 좌표가 크게 변하는 경우가 발생하여 이로 인한 계산된 텍스쳐 좌표값에 오류가 있을 수 있다. 텍스쳐 좌표의 오류가 있는 경우 다른 위치의 색상을 가져오게 되어 원래 색상과 차이가 크게 발생한다.

속도상 성능비교는 표 3과 같다. 수행 속도는 모델의 폴리곤 개수와 셰이더에서 처리할 명령어 수행 사이클에 비례함을 알 수 있으나 성능면에 있어서 기존의 모바일 환경에서 개발된 셰이더가 없으므로 성능을 비교할 대상이 없고 셰이더는 전체 파이프라인 일부에 포함되므로 속도에 대한 성능평가가 어렵다.

표 3. 성능 비교(그림 15 참조)

구분	폴리곤 수 / 명령어 개수			FPS
	Vertex Shader		Fragment Shader	
Gloss mapping	960 faces			8
	15개		25개	
Image Processing	32 faces			
	brightness	5개	9개	28
	contrast	12개	5개	25
	saturation	7개	13개	26
Parallax Mapping	32 faces			5
	14개		33개	

향후 연구에서 전체 파이프라인 성능 평가를 거친 후 하드웨어로 구현 될 예정이다.

5. 결론

본 논문은 OpenGL ES 2.0표준에 기반한 모바일 용 셰이더를 설계하고, 표준에 부합하는 명령어 셋을 정의하고 효과적인 머신 코드 구성 방법을 제시하였다. 본 연구에서 제안한 셰이더 구조는 하드웨어 제작 업체의 표준 인터페이스로서 사용될 수 있으며, 셰이더 모듈 칩 생산을 위한 성능 실험 및 안정성 검사용으로 활용 할 수 있다. 실제로 본 구조에 따르는 모바일용 3D그래픽 칩을 제작중에 있으며 이를 통해 모바일에서의 3D 게임 및 애니메이션, 사진 편집 등이 가능하다. 또한, 셰이더를 확장시 MPEG동영상 플레이어, MP3 플레이어, 전자사전 등의 용도로 활용 할 수 있다. 향후 High-Level Shading Language를 지원하고 셰이더 프로그램의 동적 흐름 제어를 위한 셰이더 구조를 연구하고자 한다.

6. 참고문헌

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

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

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

[4] http://www.khronos.org/

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

[6] Victor Moya, Carlos Gonzalez, A Single (Unified) shader GPU Microarchitecture for Embedded Systems, 2005

[7] Charles river media, Eric Lengyel, The OpenGL Extensions guide,

[8] http://oss.sgi.com/projects/ogl-sample/registry/ARB/

그림 15. 결과 이미지

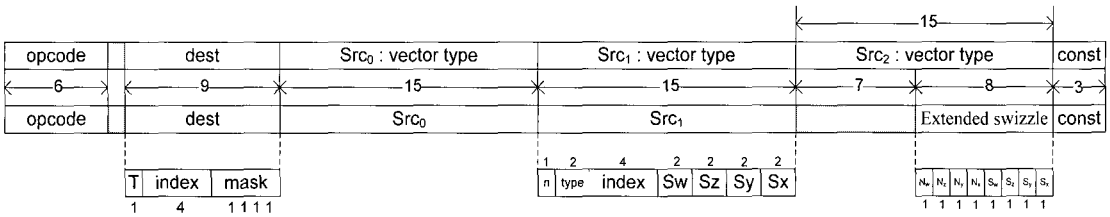


그림 6. 머신 코드 형식 (위: MAD 명령어 형식 / 아래: Standard 명령어 형식)

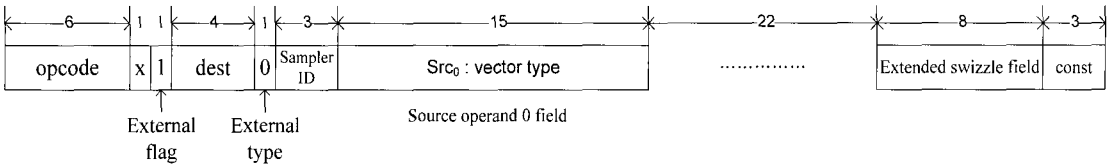


그림 7. 머신 코드 형식(NOP 확장 명령어 형식)

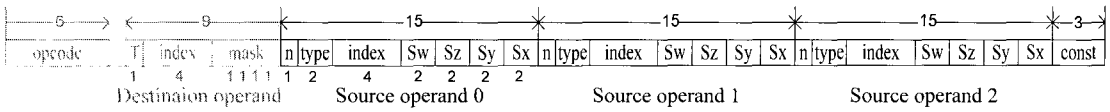


그림 8. 머신 코드 형식(MAD 명령어 형식)

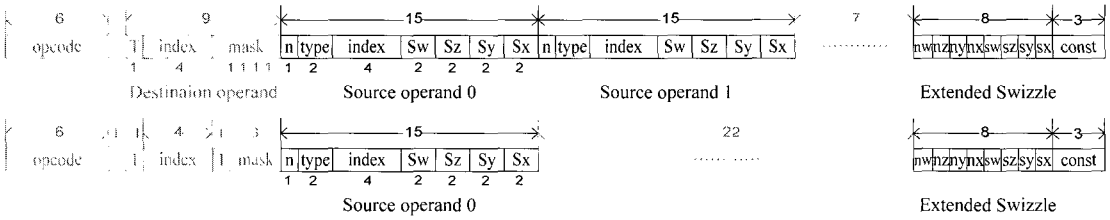
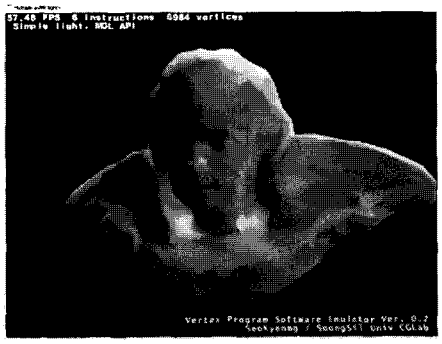
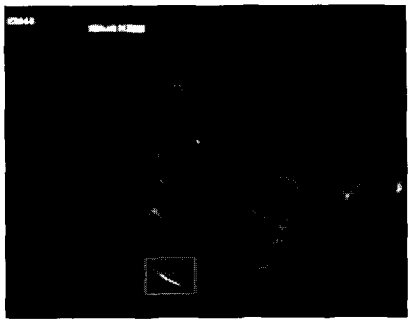


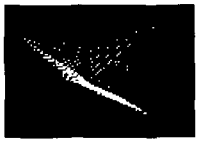
그림 10. 머신 코드 형식(Standard 명령어 및 NOP 확장 명령어 형식)



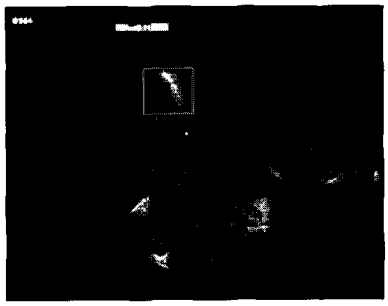
(a) 샘플1 (노말 값 출력)



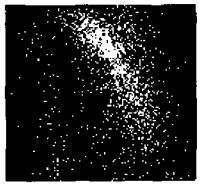
(b) 샘플1 오류표현이미지



(c) 샘플2 (쿡-토런스)



(d) 샘플2 오류표현이미지



(e) 샘플3 (환경 매핑)

(f) 샘플3 오류표현이미지

그림 14. 화질비교 ((b) (d) (f)는 색상차가 보이는 픽셀을 하얀색으로 표현한 이미지)