임베디드 그래픽 프로세서 칩 개발

Ⅰ. 서 론

최근 임베디드 시스템에 고성능의 3D 그래픽 게임이나 그래픽 어플리케이션이 탑재되면서 고성능의 그래픽 연산이 요구된다. 이러한 연산은 기존의 Mobile CPU로 처리하기에 어렵기 때문에 점차 임베디드 환경에 특화된 Mobile GPU(Graphics Processing Unit)의 개발이 큰 관심사로 떠오르고 있다.

GPU는 병렬프로세서로써 부동소수점 연산과 프로그래밍 성능을 가지는 칩으로써 메모리 대역폭과 수치연산의 throughput 측면에서는 CPU를 능가하는 성능을 가지며 DirectX, OpenGL 등의 세이딩(Shading) 언어를 사용한다.

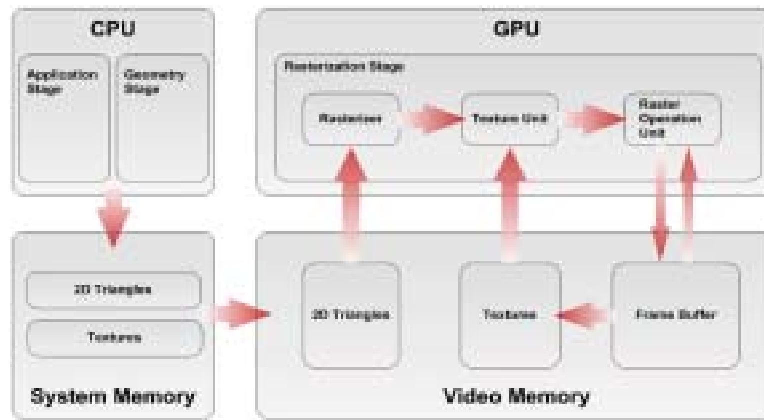
Ⅱ. GPU의 발전 과정

다양한 그래픽 효과의 처리와 정교한 모델링에 의한 처리 데이터의 증가로 인하여 그래픽 처리에 관한 CPU의 부담이 가중됨에 따라 이러한 CPU의 부담을 감소시키면서 그래픽 처리의 가속화를 위하여 GPU가 등장하였다.[1] 이러한 GPU는 3D 그래픽의 발전과 함께 다양한 기능을 지원하는 구조로 발전해 왔으며, 연산 중심적인 특성에 따라 처리속도의 발전은 CPU의 처리속도를 능가하며 매우 빠른 속도로 발전하고 있다.

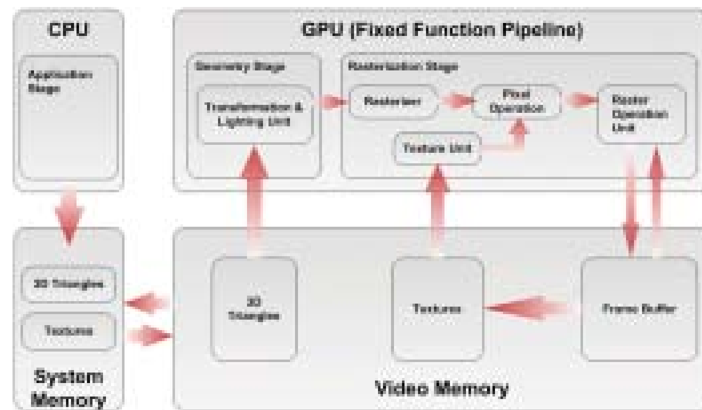
1. 래스터라이제이션 가속을 위한 GPU

가장 처음 등장한 GPU는 3D 그래픽 처리 과정 중 래스터라이제이션 단계를 가속하기 위한 목적으로 비교적 간단한 구조를 가지고 있었다. 이 때의 GPU는 3D 모델에 대하여 적은 연산만으로 사실

이 광 엽
서경대학교



<그림 1> 래스터라이제이션 가속을 위한 GPU 구조



<그림 2> 지오메트리 단계의 가속을 지원하는 GPU 구조

감을 더해줄 수 있도록 텍스처 맵핑(Texture-Mapping)을 지원하였으며 GPU의 발전에 따라 지원 가능한 텍스처 맵의 수가 증가하여 하나의 모델에 대하여 다수의 텍스처 맵을 맵핑 할 수 있게 되었다.

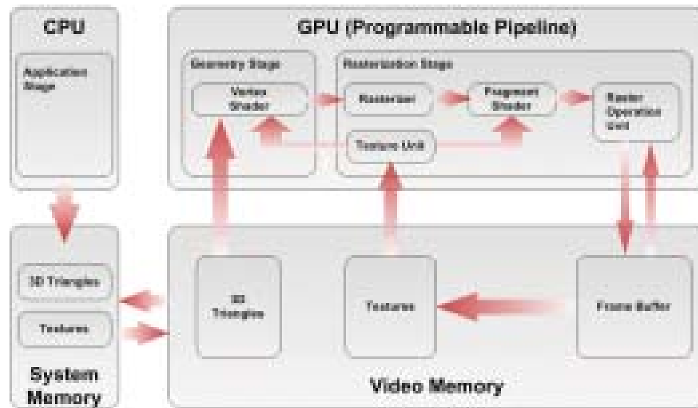
2. 지오메트리 단계의 가속을 지원하는 GPU

래스터라이제이션 단계의 가속을 위한 GPU 이후에 등장한 GPU는 지오메트리 단계의 가속을 지원하였다. 이 때부터 3D 그래픽 파이프라인의 전체 과정이 GPU를 통하여 처리가 가능해졌다. 이 때의 GPU에서 지원한 지오메트리 단계의 처리는 정점 변환과정을 위한 매트릭스의 연산과 광원에 의한 빛 계산에 대한 연산을 포함하고 있다. 이러한 과정들은 GPU내부 모듈로써 고정되어 있었으며, 이에 따라 지오메트리 처리 역시 고정된 처리과정을 통하여 처리되었다. 이런 GPU를 통한 3D 그래픽의 처리를 통하여 기존보다 짧은 시간 안에 3D 그래픽의 렌더링이 가능해 졌으며 기존보다 더 많은 텍스처 맵을 지원하게 되었다.

3. 쉐이더를 포함하는 프로그래밍 가능한 GPU

과거의 GPU는 3D 그래픽 파이프라인의 가속을 통한 빠른 렌더링이 가능했으나 처리 방식이 고정되어 있어 표현의 한계를 가지고 있었다. 그러나 3D 그래픽의 발전에 따라서 더 많은 효과의 처리가 요구되었고, 이에 따라 정점 처리 과정 및 픽셀 처리 과정을 프로그래밍이 가능하도록 한 쉐이더를 지원하는 GPU가 등장하였다.

지오메트리 단계의 가속에는 정점 처리를 담당하는



〈그림 3〉 쉐이더를 지원하는 GPU 구조

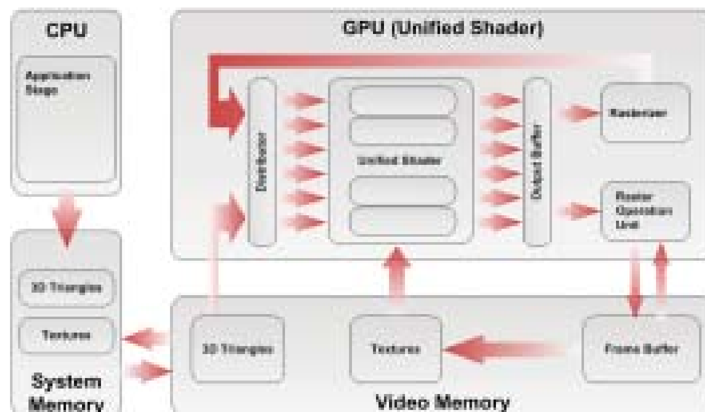
버텍스 쉐이더(Vertex Shader)가 포함되었으며 래스터라이제이션 단계에는 픽셀 처리를 담당하는 픽셀 쉐이더(Pixel Shader)가 포함되었다.[2] 이러한 쉐이더의 지원은 이전 고정된 처리방식에서 나타냈던 표현의 한계를 극복할 수 있었으며 이를 통하여 보다 더 실제에 가까운 렌더링 결과를 얻을 수 있게 되었다.

또한, 쉐이더의 구현을 위한 프로그래밍도 처음에는 어셈블리 언어와 같은 저 수준의 언어를 통하여 기술되었지만 점차 고수준의 언어인 Shading Language의 개발을 통하여 Cg, HLSL, GLSL, ESSL 등과 같이 C 언어와 유사한 형태로 기술이 가능해 졌다. 이러한 버텍스 쉐이더와 픽셀 쉐이더는 각각 지원하는 기능을 증가시키고 새로운 구조의 개발을 통하여 많은 발전을 거듭 해 왔다.

4. 통합 쉐이더를 포함한 GPU

GPU에 포함된 쉐이더의 경우 3D 그래픽 파이프라인의 가속을 위한 Unit으로서 프로그래밍이 가능 하며 더 많은 기능들을 지원함에 따라서 연산에 특화된 프로세서 형태로 발전되어 왔다. 3D 그래픽의 처리 과정에서 버텍스 쉐이더와 픽셀 쉐이더의 처리가 독립적으로 이루어 지며, 다수의 데이터를 동일한 연산을 통하여 처리하고 컴포넌트 별로 수행되는 연산이 동일한 경우가 많은 특성을 통하여 다수의 버텍스 쉐이더 및 픽셀 쉐이더를 포함하여 이 들을 병렬적으로 처리하는 구조로 발전하고 있다.

그러나 이러한 다수의 쉐이더 프로세서가 경우에 따라 지오메트리 연산단계에 집중되거나 래스터라이제이션 단계에 치중됨으로써 효율적으로 활용되지 못하고



〈그림 4〉 통합쉐이더를 포함한 GPU 구조

어느 한쪽만의 사용이 가중되는 경우가 많다. 이는 하드웨어의 낭비를 초래한다.

이러한 단점을 보완하기 위하여 버텍스 쉐이더와 픽셀 쉐이더를 통합한 통합 쉐이더가 등장하였다.[3] 버텍스 쉐이더와 픽셀 쉐이더가 점차 발전해 감에 따라 그 구조가 일반적인 프로세서의 형태로써 두 쉐이더간의 구조가 비슷해지고 이를 하나로 통합하여 응용 범위에 따라 버텍스쉐이더의 동작과 픽셀 쉐이더 역할을 선택적으로 할 수 있는 구조이다.

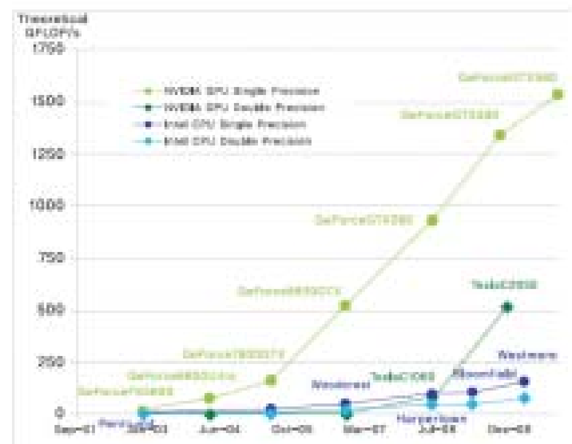
이렇게 통합된 구조를 통하여 버텍스 쉐이더 연산 및 픽셀 쉐이더 연산 의 프로그래밍에 동일한 명령어 셋을 이용하기 때문에 보다 용이하게 프로그래밍을 할 수 있다는 장점이 있다. 또한 다수의 통합 쉐이더의 집합체에서 쉐이더의 필요 연산에 따라 그 역할을 변경해 줌으로써 버텍스 쉐이더 와 픽셀 쉐이더간의 연산 편중에 따른 H/W 활용의 비효율성을 줄이고 쉐이더 프로세서의 H/W 자원을 최대한으로 활용할 수 있게 되었다.

5. 일반적인 연산에 활용 가능한 GP-GPU

통합 쉐이더의 등장을 통하여 쉐이더 프로세서는 연산 중심적 특성을 가지는 프로세서의 형태로 발전해 갔으며 그 처리 성능의 발전은 CPU의 처리 속도를 능가하여 더욱 빠르게 발전하고 있다. GPU의 부동 소수점 연산 능력은 이미 2002년 이후부터 CPU의 연산 능력을 능가하였으며 계속해서 빠른 발전속도를 보이고 있다. 이러한 연산 중심적 특성과 빠른 처리속도는 GPU를 그래픽 처리뿐만 아니라 많은 양의 연산이 필요한 우주항공분야, 복잡한 분자구조의 분석, 물리적 분석 등에 활용하여 기존보다 더욱 빠른 처리속도를 기대할 수 있으며, 그 활용 분야는 더욱 확대될 것이다.

6. 모바일 환경의 GPU

앞에서 본 바와 같이 3D 그래픽과 이를 지원하는 GPU는 막강한 연산능력과 연산 특성에 최적화 된 처리구조를 통하여 계속해서 진화해 왔다. 이를 통하여 모바일용 기기에도 이러한 GPU가 활용되고 있으며 컴



〈그림 5〉 Floating-Point operations per Second and Memory Bandwidth for the CPU and GPU(Reference. nVidia, NVIDIA Cuda C Programming Guide version 4.0, 5/6/2011)

퓨터 시스템이나 게임콘솔의 GPU의 발전과 비슷한 추세로 발전해 가고 있다. 아직까지는 성능 면에서는 컴퓨터 시스템이나 게임콘솔에 적용된 GPU와는 많은 차이를 보이지만, 현재 이미 모바일용 기기에도 버텍스 쉐이더 및 픽셀 쉐이더가 도입되었으며 앞으로 통합 쉐이더로의 발전 및 GP-GPU 구조로의 발전도 예측해 볼 수 있다.

Ⅲ. GPU를 위한 API

1. OpenGL

OpenGL은 Open Graphics Library의 약자로 1990년 SGI에 의해 고안된 3D 그래픽스 하드웨어를 위한 소프트웨어 인터페이스로써, 3D 그래픽스 응용 프로그램을 만들기 위한 API(Application Program Interface)이다.

OpenGL은 로열티가 없는 공개 표준으로 하드웨어 제작자와 이를 이용하는 응용프로그램 개발자가 서로 독립적으로 일하는 것을 가능하게 한다. 개발 초기부터 Sun, DEC, SGI 등의 다양한 플랫폼에서 Windows95, X Windows, Windows NT, OS/2 등 다양한 운영 시스템에 이식되어 성능이 검증된 OpenGL은 최근 크로노

스 그룹의 일부가 된 OpenGL ARB 워킹 그룹에 의해 제정과 관리가 이루어지고 있다. OpenGL은 주로 데스크톱 PC나 워크스테이션에서 사용된다.

OpenGL은 GPU의 발전과 밀접하게 관련이 있어 초기에는 고정된 기능의 파이프라인 형태를 유지하다가 프로그램 가능한 파이프라인의 형태를 갖추게 되었다. OpenGL 1.0, 1.1, 1.2, 1.3, 1.4, 1.5는 고정된 기능의 파이프라인에 해당하며 상위 버전으로 갈수록 기능이 추가되었고 1.5 버전 이후로는 고정된 파이프라인을 갖는 버전은 만들어지고 있지 않았다. 반면 OpenGL 2.0과 최근에 발표된 2.1 버전은 버텍스 프로세서와 프래그먼트 프로세서를 갖는 프로그램 가능한 파이프라인을 갖는다. 이들 프로세서를 프로그래밍 하기 위한 GLSL도 함께 발표되고 있다. OpenGL 2.x는 프로그램 가능하도록 파이프라인에 큰 변화를 주었지만 앞선 버전들에 대한 호환성을 유지하고 있다. 다시 말해 1.x에서 실행되던 응용프로그램이 2.x에서 수정 없이 실행된다. 한편, OpenGL ARB는 가장 최근에 OpenGL 3.0에 대해 공식적으로 발표하였다.

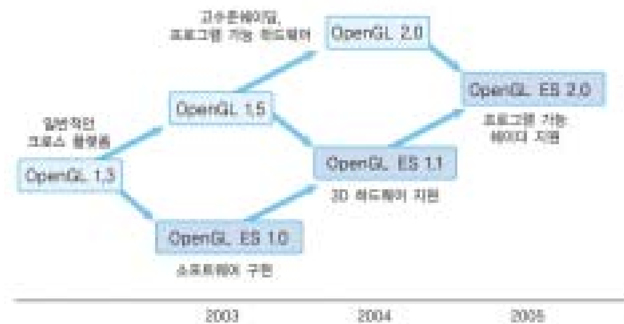
2. OpenGL ES

OpenGL ES[4]는 3D 그래픽을 지원하는 내장 시스템(embedded system)을 위한 API로 크로노스 그룹에 의해 제정과 관리가 이루어지고 있다. OpenGL ES는 OpenGL과 마찬가지로 로열티가 없는 공개 표준이며, OpenGL의 부분집합으로 구성된 프로파일이다. OpenGL ES는 소프트웨어와 GPU간의 유연하면서도 강력한 저 수준의 인터페이스를 제공하는 한편 메모리가 작은 환경과 저전력이 요구되는 환경에 적합하여 주요 모바일 및 내장 시스템 플랫폼 환경에서 3D 게임과 다양한 고급 3D 그래픽 기능을 제공하는 데 기여하고 있다. OpenGL ES는 OpenGL을 기반으로 하고 있으므로 특별히 새로운 기술을 배울 필요 없이 응용프로그램 개발의 시너지효과를 기대할 수 있다.

OpenGL ES 1.1은 고정된 파이프라인을 갖고 있으며 1.0과 완전한 호환이 가능하다. 반면에 2.0은 프로그램 가능, 즉 버텍스 및 프래그먼트 프로세서를 포함하며 1.x와의 완벽한 호환을 지원하지 않는다. OpenGL이 고정된 기능의 파이프라인을 갖는 버전을 더 이상 제정하고 있지 않는 것과 달리 OpenGL ES는 1.x와 2.x가 각각 성능 개선을 위한 개정이 진행 중에 있다.

3. DirectX Direct3D

마이크로소프트사의 DirectX는 Windows 운영체제 기반 컴퓨터를 멀티미디어 응용프로그램을 위한 이상적인 플랫폼으로 만들기 위해 기술된 기술 모음으로써, DirectX를 사용하면 2D 그래픽, 비디오, 3D 그래픽, 오디오 등 다양한 멀티미디어 구성요소를 이용하여 응용프로그램을 개발할 수 있다. 이 가운데 Direct3D[5]는 DirectX 중에서 가장 큰 부분을 차지하며 활발한 개정이 이루어지고 있다. Direct3D는 가장 보편적으로 많은 게임 개발업체들이 지원하는 API이다. Direct3D 역시 GPU의 발전 과정과 밀접하게 연관되어 있는데, 최근에는 Direct3D의 다음 버전이 발표되면 하드웨어 제작사들은 이를 지원하는 GPU를 경쟁적으로 내놓고 있는 추세이다. GPU가 나오기 시작하던 초기에는 OpenGL의 사용만이 3D 그래픽스 응용프로그램을 개발하기 위한 유일한 효율적인 방법이었지만 DirectX의 초기 버전이 발표되고 빠른 속도로 새로운 기법들을 추가해감으로써 인기를 얻기 시작하였다. 대부분의 사용자들이 Windows 운영체제를 사용하고, 따라서 PC용 3D 게임 역시 대부분 Windows를 위해 개발되기 때문에 DirectX의 입지는 더욱 굳건해



〈그림 6〉 OpenGL 발전 과정

지고 있다.

Direct3D와 OpenGL은 완전히 일치하지는 않지만 적어도 비슷한 기능을 지원하는 API들이 많으며 한 쪽에서 가능한 일은 대부분 다른 쪽에서도 가능하다. Direct3D는 새 버전을 발표하면서 꾸준히 새로운 버전의 '셰이더 모델'을 발표하고 있다. 셰이더 모델은 버텍스 셰이더, 픽셀 셰이더 등을 프로그래머 입장에서 바라본 모델로 입출력 레지스터, 사용 가능한 명령어 등을 정의한다. 현재 최신의 DirectX 10에서는 기존의 버텍스 셰이더와 픽셀 셰이더 이외에 새롭게 지오메트리 셰이더가 추가된 '셰이더 모델 4.0'을 지원하도록 하고 있다.

4. Direct3D Mobile

모바일 기기를 위한 마이크로소프트의 운영체제인 Windows Mobile 5.0에서 Direct3D Mobile을 사용하여 3D 그래픽 응용프로그램을 작성할 수 있다. Direct3D Mobile은 Direct3D 8에 가장 근접하며 Direct3D 9버전의 일부 요소도 포함하고 있다. Direct3D Mobile은 데스크톱용 버전들보다 작은 런타임을 갖도록 간단하게 설계되어 있다. OpenGL ES가 그러하듯, Direct3D Mobile 역시 데스크톱 버전에서 필수적인 부분만을 모아 놓은 프로파일이라 할 수 있다. 그러나 Direct3D Mobile에서는 아직 버텍스 및 픽셀 셰이더를 지원하지는 않고 있다.

Ⅳ. 모바일 GPU 파이프라인 구조

모바일 기기에서 대표적으로 사용되는 OpenGL ES API에 적합한 3차원 그래픽 파이프라인의 구조를 바탕으로 다양한 GPU구조를 알아보겠다.

OpenGL ES 1.x 버전은 고정 소수점 데이터를 사용하여 고정기능 파이프라인으로 구성되며, OpenGL ES 2.x는 고정된 연산의 일부분을 셰이더로 대체하여 프로그램 가능한 파이프라인을 지원하게 구성되어 있다.

1. 3차원 그래픽스 파이프라인

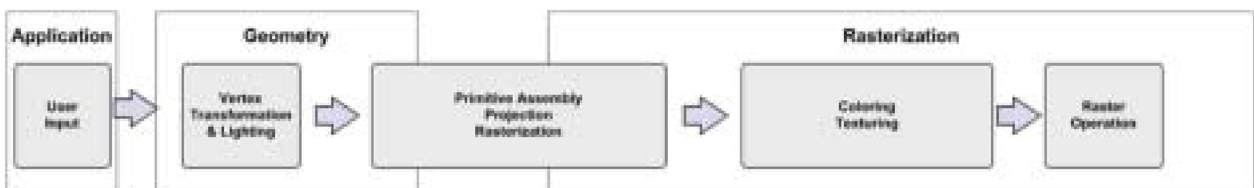
3D 그래픽 파이프라인은 <그림 7>과 같이 모델을 구성하는 각 정점(Vertex)들의 정보들을 입력으로 받아 이를 이용하여 다양한 처리 과정을 거쳐 렌더링 된 이미지를 출력한다. 처리 과정을 살펴보면 크게 응용프로그램 단계(Application Stage), 기하변환 단계(Geometry Stage), 래스터라이제이션 단계(Rasterization Stage) 세 단계로 구분할 수 있다.

1.1 응용 프로그램 단계

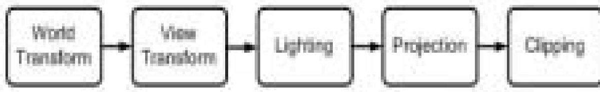
응용 프로그램 단계는 CPU에서 수행되며 이 단계에서는 3D 그래픽을 처리하기 위한 준비단계로 응용프로그램을 통한 시스템 메모리의 할당 및 3D 그래픽을 처리하기 위한 상수데이터와 정점데이터를 전달하는 역할을 한다. 이러한 정점 데이터는 모델을 구성하는 정점으로써 정점의 좌표, 색상, 텍스처 좌표, 법선 벡터등을 나타낸다.

1.2 기하(Geometry)변환 단계

기하변환 단계에서는 전달 받은 정점 좌표들을 이용하여 기하 변환과정과 광원에 의한 빛 계산을 한다. 기하 변환 과정은 모델을 구성하는 정점들을 이동 및



〈그림 7〉 3D 그래픽 처리 과정



〈그림 8〉 기하 변환 과정

회전, 크기변환 등의 연산을 수행하며 변환행렬과 정점 데이터의 매트릭스 곱셈을 통하여 연산되고 광원에 의한 빛 계산은 광원의 종류 및 개수에 따라 그 연산 방법이 달라진다.

월드 변환(World Transform)은 모델 좌표계를 월드 좌표계로 변환시키는 과정이다. 모델 좌표계는 3차원 모델을 생성할 때 사용되는 좌표계로, 모델의 한 점이 좌표계의 중심이 된다. 모델의 변환은 Translate, Scaling, Rotation 과정으로 구성된다.

뷰 변환이 적용된 후에는 시점(View Point)까지 적용되어 구성된 3차원 좌표계를 2차원의 디스플레이 장치에 영상으로 표현하기 위해 2차원의 좌표계로 변환하는 과정을 수행한다. 이 과정은 투영 변환(Projection) 이라고 하며, 이를 통해 3차원 공간의 객체들이 2차원 공간에 투영되어 있는 화면을 얻을 수 있다.

라이팅(Lighting) 과정은 월드 변환과 뷰 변환이 수행된 후에 처리된다. 좌표계 내의 객체들과 광원간의 조명효과를 계산하여 각 정점의 색상을 결정한다. 조명효과는 광원의 위치와 종류, 정점의 위치, 색상과 노말 벡터 등을 이용하여 계산된다.

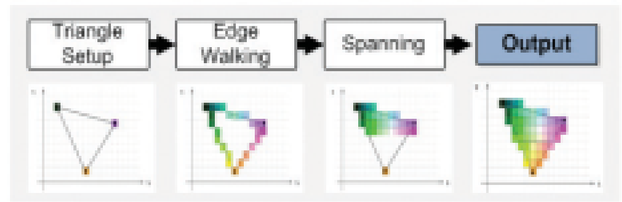
1.3 래스터라이제이션 단계(Rasterization)

좌표계 변환과 조명효과를 처리하는 기하 변환 과정이 끝나면 변환된 정점들을 이용하여 래스터라이제이션(Rasterization) 을 한다. 래스터라이제이션은 〈그림 9〉와 같이 트라이앵글 셋업(Triangle Setup), 에지워킹(Edge Walking), 스패닝(Spanning) 의 세 단계로 구성된다.

2. OpenGL ES 1.x 고정 기능 파이프라인

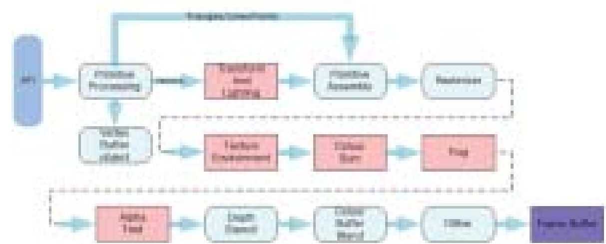
〈그림 10〉은 OpenGL ES 1.x를 기반으로 하는 파이프라인이다.[6] 고정 기능 파이프라인은 단계별 기능이 고정된 연산만 처리하는 하드웨어로 고정되어 있기 때문에 사용자는 하드웨어로 구현되어 있는 일부 모드 설정과 파라미터 세팅으로 그래픽스 파이프라인 내부의 동작을 제어할 수 있다. 이러한 방법은 이미 정해진 연산만을 처리하기 때문에 다양한 그래픽스 효과를 나타내는 알고리즘을 처리하기는 적합하지 않으나, 고정된 파이프라인을 따라 반복적인 연산을 수행하기 때문에 고속으로 그래픽스 처리를 할 수 있다는 장점이 있다.

〈그림 10〉에서 보여 지는 파이프라인 단계에서 Transform and Lighting은 기하 변화단계에 해당되며 하드웨어 적으로 고정된 Transform and Lighting 기능만 처리하는 역할을 한다. 그 다음 Rasterizer 프로세서를 따로 구현하여 Rasteriztion 단계를 처리하고 다음 픽셀 오퍼레이션 단계로 넘어간다. 그림에서 보듯 픽셀 오퍼레이션은 텍스처를 입히거나 각 픽셀에 안개(Fog), 투명도 테스트(Alpha Test), 블렌딩(Blending)처리에 해당하는 연산을 수행하고, 그 후에 프레임 버퍼에 픽셀을 찍어 화면에 나타낸다.

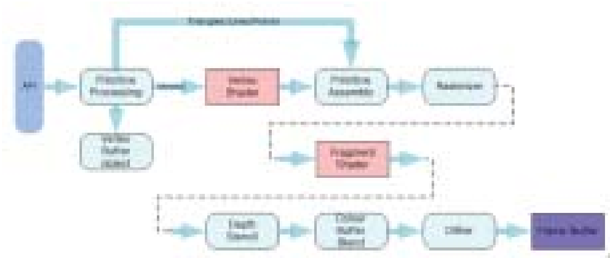


〈그림 9〉 래스터라이제이션 단계

3. OpenGL ES 2.x 프로그램 가능형 파이프라인



〈그림 10〉 고정 기능 파이프라인



〈그림 11〉 프로그램 가능한 파이프라인

〈그림 11〉은 고정 기능 파이프라인의 일부 기능을 정점 셰이더(Vertex Shader) 와 픽셀 셰이더(Pixel Shader)를 이용하여 처리하도록 구성된 프로그램 가능한 파이프라인[7]이다. 고정 기능 파이프라인과 비교해 보자면 정점 셰이더가 Transform and Lighting의 기능을 수행하며 래스터라이제이션 이후의 픽셀 별 색상 처리 과정을 픽셀 셰이더를 이용하여 처리한다.

프로그램 가능한 파이프라인의 가장 큰 특징은 프로그램이 가능한 셰이더 프로세서를 이용하여 파이프라인을 구성하기 때문에 사용자들이 그래픽스 파이프라인 과정 중 해당 부분 연산 처리과정을 직접 프로그램 할 수 있다는 것이다. 셰이더는 명령어 기반으로 그래픽스 연산을 처리하는 프로세서로 사용자가 고급 언어의 형태로 제공되는 셰이더 프로그래밍 언어[8]를 이용하여 작성한 프로그램을 처리한다. 고성능의 효과를 셰이더 프로세서를 이용해서 표현하거나 동일한 효과를 고정 기능 파이프라인에 비해 적은 연산 량으로 처리하는 것이 가능하다.

파이프라인의 기능을 사용자가 직접 프로그램 할 수 있다는 특성은 카툰 렌더링(Catoon Rendering), 범프 맵핑(Bump mapping), 정점 블렌딩(Vertex Blending), 모션 블러(Motion Blur) 등의 특수한 효과들을 가능하게 한다.[9][10] 또한 프로그램이 가능한 프로세서로 처리하면 고정 기능 파이프라인에서 특정한 연산을 하는 하드웨어를 줄일 수 있었기 때문에 프로세서 전체 크기가 줄어들고 언제든지 사용자에 의해 프로그램을 통한 최적화가 가능하여 휴대 기기 환경에서 특히 유용하게 사용될 수 있다.

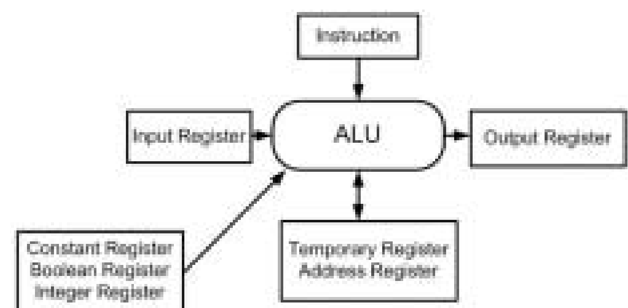
4. 프로그래밍이 가능한 셰이더 프로세서 구조

셰이더(shader) 프로세서는 프로그래머블 GPU의 코어 역할을 한다. 기하 변환 단계의 정점 처리를 담당하는 버텍스 셰이더와 래스터라이제이션로부터 출력된 픽셀을 처리하는 픽셀 셰이더로 구성된다.

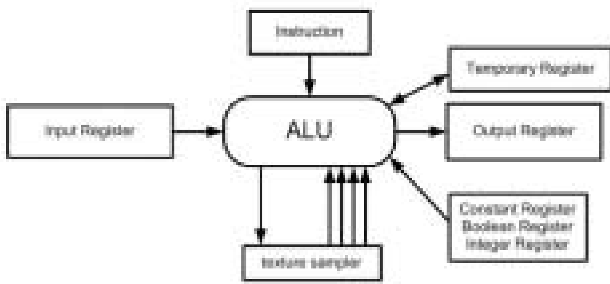
버텍스 셰이더는 정점과 관련된 좌표, 법선, 색상 등을 입력으로 셰이더 프로그램에 따라 일련의 처리과정을 거쳐 변환된 좌표와 색상을 출력한다. 고정 기능 하드웨어서는 표현할 수 없었던 모핑(Morphing), 모션 블러(Motion Blur), 렌즈 효과(Lens Effect)등의 다양한 그래픽스 효과를 표현할 수 있다.

버텍스 셰이더는 레지스터 파일과 연산 유닛으로 구성되어 있다. 레지스터 파일은 기본적으로 입력(Input) 레지스터, 출력(Output) 레지스터, 상수(Constant)레지스터, 임시(Temporary)레지스터, 주소(Address)레지스터가 있으며 셰이더의 버전 및 종류에 따라 다양한 레지스터가 사용된다. 입력 레지스터는 읽기 전용의 속성을 가지며 x ,y, z, w 또는 r, g, b, a 와 같은 3차원 그래픽스 성분을 표현하기 위해 4개의 컴포넌트로 구성된다. 출력 레지스터는 쓰기 전용의 속성을 가지며 입력 레지스터와 같이 4개의 컴포넌트로 구성된다. 상수 레지스터는 연산에 사용하기 위한 상수 값들을 저장하기 위해 사용하며, 주소 레지스터를 이용해 간접 주소 방식으로 접근할 수 있다. 임시 레지스터는 연산 결과를 임시로 보존하기 위하여 사용하며 읽기 쓰기 모두 가능하다.

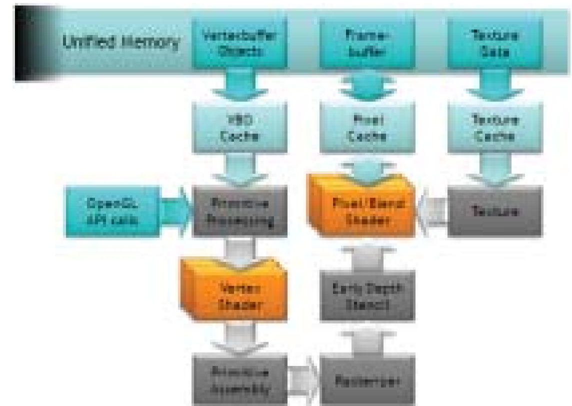
픽셀 셰이더는 픽셀 색상의 변경을 통해 픽셀 라이



〈그림 12〉 버텍스 셰이더 구조



〈그림 13〉 픽셀 셰이더 구조



〈그림 14〉 nVidia ULP Geforce의 3차원 그래픽 파이프라인

팅(Pixel Lighting), 퐁 셰이딩(Phong Shading), NPR(Non-Photo realistic Rendering), 범프 맵핑(Bump Mapping) 등의 다양한 그래픽스 효과를 표현 할 수 있다.

초기의 셰이더 모델 1.x 버전에서는 8비트의 낮은 정확도를 갖는 연산이 지원되었으며 사용 가능한 명령어의 종류와 슬롯이 매우 적다. 셰이더 모델 2.x 버전에서는 연산 능력이 향상되어 정점 셰이더와 비슷한 수준의 연산을 할 수 있게 되었다. 또한 텍스처 기능도 향상되어 기존의 픽셀 셰이더에 비해 많은 양의 텍스처를 이용하여 정확한 연산을 할 수 있게 되었다. 셰이더 모델 3.0 버전에서는 동적 흐름 제어 기능을 지원하면서 일부 명령어를 제외하고는 정점 셰이더와 동일한 구조를 갖게 된다. 픽셀 셰이더는 정점 셰이더의 출력 데이터와 텍스처 데이터를 이용하여 연산을 수행한다. 픽셀 셰이더도 용도별 레지스터 파일과 연산 유닛으로 구성되어 있다.

V. GPU 칩 개발 사례

GPU의 대표적인 설계사례로 nVidia칩을 소개한다. nVidia는 모바일 환경에서 3D Graphics 처리를 위해 초절전(Ultar lLow Power) GeForce로 불리는 GPU를 개발 하였다.

이렇게 설계된 GPU는 모바일 디바이스를 위한 nVidia의 SoC인 Tegra에서 사용 중이며 nVidia의 기술인 CUDA Architecture를 모바일 환경을 위해 초절전을 최우선으로 고려하여 커스터마이징 하였다.

〈그림 14〉는 모바일 환경을 위해 ULP GeForce에서 사용되는 그래픽 파이프라인이다. 특징으로는 Early-Z test인데 이는 Depth Test를 먼저 수행함으로써 추후에 있을 라스터 연산 과정 중 텍스처링을 위한 텍스처 데이터와 픽셀 데이터들의 대역폭을 줄일 수 있어 대역폭의 제한이 있는 모바일 환경에 커스터마이징된 3D 파이프라인이다.

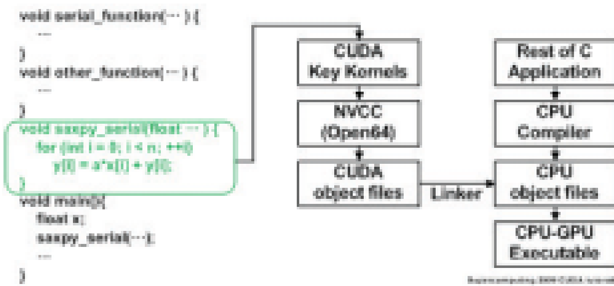
최근 모바일 디바이스인 스마트 폰은 기존의 휴대전화에서 벗어나 다양한 컨텐츠 소비를 위한 범용 플랫폼으로 자리를 잡고 있다. 그래서 모바일 GPU들 역시 PC 환경의 GPU처럼 3D 가속뿐만 아니라 범용 목적의 다양한 가속을 위한 GPGPU(General Purpose computing on GPU)로 변모하고 있다.

ULP GeForce도 이미 데스크 탑 환경에서 익숙한 CUDA Architecture를 사용함으로써 3D Graphics 뿐만 아니라 Adobe Flash의 가속 기능으로 플래시가 다수 사용되는 웹 환경에서 뛰어난 성능을 보인다.

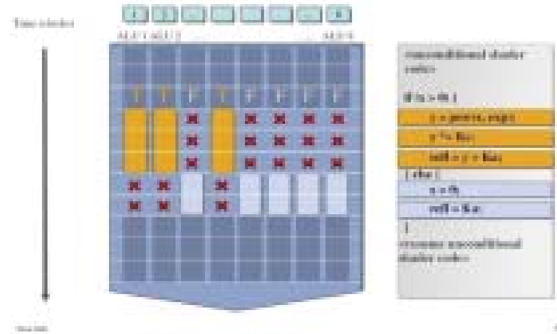
nVidia의 CUDA[11]는 호스트 프로세서와 GPGPU의 이원화된 프로그래밍 방법을 〈그림 15〉와 같이 추상화 시켜 단 하나의 프로그래밍으로 CPU와 GPGPU를 제어할 수 있도록 유도하고 있어 다른 GPGPU에 비해 비교적 쉬운 프로그래밍 환경을 조성하였다.

최근에는 페르미(Ferimi) Architecture에서는 C++를 GPGPU에서 지원함으로써 진정한 슈퍼컴퓨팅 GPGPU로 평가 받고 있다.

nVidia GPGPU는 〈그림 16〉과 같이 다 배열의



〈그림 15〉 nVidia CUDA의 소프트웨어 구조



〈그림 16〉 nVidia의 SIMD 실행 구조

SIMD구조로 되어 있다. 여러 스레드가 동시에 같은 명령어를 통해 실행되는 구조를 취하고 있다. 이는 분기와 같은 명령에서는 FLASE 경우와 TRUE 경우를 모두 실행되어야 하며, 각 스레드는 해당 코드만 실행되고 그렇지 않는 코드는 실행되지 않고 쉬게 된다.

이러한 방법은 프로세서를 최대한으로 활용할 수 있도록 하기 때문에 전체적 성능 향상에 도움을 주지만 스레드 관리에 별도의 프로세싱이 소요되기 때문에 싱글스레드 실행에는 적합하지 못하고 호스트 프로세서의 성능에 의해 영향을 받으므로 호스트 프로세서의 성능도 높아야 한다.

따라서 대부분의 GPGPU들은 높은 프로세서 활용을 위해 별도의 멀티 스레드 관리 모듈이 존재하고 이 모듈은 호스트 프로세서에 의해 제어된다. IBM의 CELL은 PPE를 통해 실시간 멀티 스레드 관리를 하며, nVidia의 CUDA의 경우 스레드 메니저를 별도로 두어 호스트 프로세서에 의해 제어된다.

참고문헌

[1] Israel Koren,"Computer Arithmetic Algorithms," Prentice Hall,1993

[2] J. S. Ha, H. G. Jeong, S. Y. kim, K. Y. lee, Design of a 3D Graphics Geometry Accelerator using the Programmable Vertex Shader, Journal of the Institute of Electronics Engineers of Korea v.43, no.9, 2006, pp.53-58

[3] Victor Moya, Carlos Gonzalez, Jordi Roca, Agustin Fernandez, amd Roger Espasa, "Shader Performance Analysis on a Modern GPU Architecture," Microarchitecture, 2005. MICRO-38.Proc.38th Annual

[4] OpenGL ES-The Standard for Embedded Accelerated 3D Graphics (http://www.khronos.org/opengles)

[5] Direct3D, (http://en.wikipedia.org/wiki/Microsoft_Direct3D)

[6] Microsoft Shader3.0, (http://msdn.microsoft.com)

[7] Mark Segal, Kurt Akeley. The OpenGL Graphics System : A Specification(Version 2.0 - October 22, 2004)

[8] Mauricio Breternitz, Jr, "Compilation, Architectural Support, and Evaluation of SIMD Graphics Pipeline Programs on a General-Purpose CPU" Proceedings of the 12th international conference on parallel architectures and compilaton techniques.

[9] "Real-Time Rendering", A K Peters, 2002

[10] Wolfgang Engel, "shader Programming Tips and Tricks with DirectX9", 2004

[11] NVIDIA CUDA, (http://developer.nvidia.com/object/cuda.html)

이 광 엽

1985년 8월 서강대학교 전자공학과 학사
1987년 8월 연세대학교 전자공학과 석사
1994년 2월 연세대학교 전자공학과 박사
1989년 5월~1995년 2월 현대전자 시스템IC연구소 선임연구원
1995년 3월~현재 서경대학교 컴퓨터공학과 교수

〈관심 분야〉
마이크로프로세서, 임베디드 시스템, 그래픽 시스템, SoC