工學 碩士學位 論文

임베디드 환경에서의 3차원 그래픽스 파이프라인의 설계 및 구현

(Design and Implementation of 3D Graphics Pipeline on Embedded Systems)

亞洲大學校 大學院

미디어 學科

金 玟 廷

임베디드 환경에서의 3차원 그래픽스 파이프라인의 설계 및 구현

指導 敎授 崔 廷 柱

이 論文을 工學 碩士 論文으로 提出 함

2005年 2月

亞洲大學校 大學院

미디어 學科

金 玟 廷

金 玟 廷의 工學 碩士學位 論文을 認准함.

審査委員長　최 정 주　인

審査委員　고　욱　인

審査委員　경 민 호　인

亞洲大學校 大學院

2005年 2月 14日

감사의 글

길었던 듯 짧았던 2 년이 이제는 정말 뒷걸음치는 중인 것 같습니다.

이 논문이 완성되기까지 도움을 주신 모든 분들에게 감사 드립니다. 먼저 저로 인해 마음 고생이 크셨을 존경하는 최 정주 교수님께 무한한 감사를 드립니다. 대학원에서 느끼고 배운 점을 뒤돌아 보며 사회에서도 열심히 살겠습니다.

같은 랩실의 현수, 순재, 현찬 오빠, 모두 2년간 고생 많았고, 앞으로도 하시는 일 모두 잘 되기를 바랄게요. 그리고 영식 오빠, 종혁이, 환직이,. 대학원의 모든 동기와 선배 후배 님들, 모두 기술하지는 못하지만 같은 연구실에서 함께 하여 즐거웠습니다. 여러분의 앞날에 무한한 성취와 기쁨이 함께 하기를 바랍니다.

항상 저를 믿어주신 사랑하는 할머니, 부모님, 동생 범선이, 너무나 사랑하는 가족에게 부족하지만 이 논문을 바칩니다.

요 약

본 논문에서는 임베디드 환경에 최적화된 3차원 그래픽스 파이프라인 라이브러리를 구현하고, 그 상위 레벨에서 3차원 게임 엔진을 구현한다. 이 그래픽스 파이프라인은 렌더링 파이프라인의 모든 단계들이 소프트웨어로 동작하고 있으므로, 그래픽 하드웨어 기반이 없는 임베디드 시스템에서도 3차원 렌더링을 가능하게 한다. 또한 크로노스 그룹이 제안한 국제 공개 표준 규격인 OpenGL-ES 의 표준 인터페이스를 따르기 때문에, 이 위에서 작업된 어플리케이션과 엔진들은 다른 시스템 간의 이식이 수월할 것이다.

목 차

그 림 목 차

표 목 차

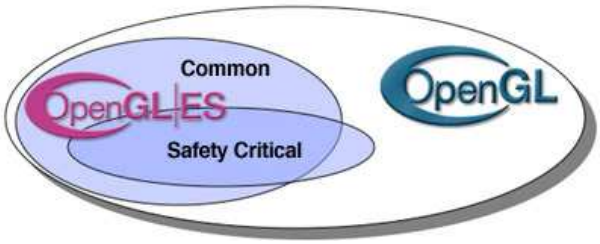
제 1 장 서론

고성능 휴대폰 용 CPU 인 'ARM 9' 이 탑재된 휴대폰이 국내외 시장에 지속적으로 등장하고 있다. 모바일 및 전자가전 등의 임베디드 시스템은 최근 몇 년간 기술과 수요면에서 큰 환경의 발전이 있었고, 이는 앞으로 더욱 가속화될 것이다. 이를 통해 '컴퓨터' 이외의 장비에서는 어렵다고 여겨지던 3차원 그래픽 환경도 다양한 시스템에서 가능해질 것으로 예측되고 있다.

제 1 절 OpenGL ES

2003년 7월, 그래픽 인터페이스 개발 동맹인 크로노스 그룹(Khronos Group)은 모바일 3D의 국제 공개 표준 규격인 OpenGL ES Ver.1.0을 공개하였다. OpenGL ES는 OpenGL의 임베디드 시스템 버전으로, 휴대용 단말기와 같은 임베디드 디스플레이에서 보다 향상된 2D/3D 그래픽 성능을 제공하기 위해 제안되었다. 즉, OpenGL ES는 OpenGL을 기반으로 하되 사용빈도가 낮거나 불필요한 부분을 제거하여 만들어진 OpenGL의 subset이다. <그림 1>.

OpenGL ES Ver.1.0에서는 software-only 구현을 명시하지만, 크로노스 그룹은 향후 그래픽 시스템으로의 이식을 고려한 유연한 Low-Level 인터페이스도 함께 제공하고 있다. 또한 Ver.1.0은 부동 소수점- 고정 소수점의 계산 방식에 따른 두 가지 버전을 포함하며, 임베디드 하드웨어 시스템과의 소통을 위한 EGL 계층을 가지고 있다.



About OpenGL ES: Royalty free 2D/3D API consisting of well-defined subset profiles of OpenGL

< 그림 1 > 크로노스 그룹의 OpenGL ES 정의와 개념도

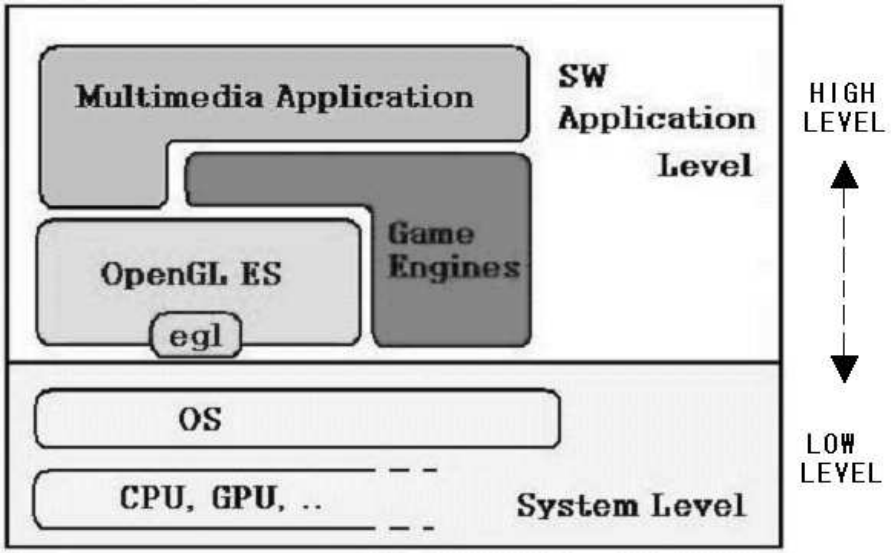
크로노스 그룹에 의하면, OpenGL-ES 표준 API는 어플리케이션의 개발에 있어 다음과 같은 다양한 편의성을 제공한다.

◆ 확장 가능 & 진화 발전
◆ 산업 표준 & 로열티 프리
◆ 사용 편리성
◆ 적은 메모리 요구량 & 저전력 소비
◆ 풍부한 문서 자료
◆ SW에서 HW 렌더링으로의 자연스러운 이동

제 2 절 연구 내용 및 기대 효과

본 논문에서는 OpenGL ES 표준을 따르는 그래픽스 라이브러리를 제작하고, 그 상위레벨에서 실행되는 간단한 게임 엔진과 실험용 게임 컨텐츠를 제작하였다. 구현은 Arm 9 프로세서와 국내 무선인터넷 개발 표준 플랫폼인 WIPI 시스템에서 테스트되었고, 기대 효과는 다음과 같다.

1. 그래픽 하드웨어 기반이 없는 임베디드 시스템에서 3D 렌더링이 가능하게 된다.
2. OPENGL ES의 표준 인터페이스를 따르기 때문에 어플리케이션의 제작이 쉽고, 시스템간의 이식이 용이하다.
3. 게임 엔진 위에서 개발자는 3D 게임/솔루션을 손쉽게 개발할 수 있다. 이 엔진은 게임을 위한 캐릭터 애니메이션 처리를 포함한다.



< 그림 2 > Embedded 3D Framework

제 2 장 Fixed Point Representation

제 1 절 고정 소수점 형식 사용 이유와 주의할 점

대다수의 임베디드 시스템은 부동 소수점 연산에 대한 하드웨어 지원을 제공하지 않고 이런 시스템에서 부동 소수점 연산을 이용하게 되면 그 계산 속도가 매우 느려져 프로그램 성능에 나쁜 영향을 미친다. 고정 소수점 계산이란 실수를 대략적으로 정수로 표현하여 계산하는 방법으로, 제한된 범위 내에서는 부동 소수점 계산에 근사한 정밀도의 계산이 가능하다. 즉 실수 레벨의 계산이 필요할 때, 부동 소수 값을 특수한 정수 값으로 변환하여 그것을 부동 소수인 것으로 가정하여 계산하는 것이다. 3D 그래픽스 분야의 경우, 많은 수학적 계산이 필요하고 부동 소수점 연산 또한 많이 이용하기 때문에, 고정 소수점 형식을 이용하여 성능을 향상시키는 것이 바람직하다.

OpenGL ES의 스펙은 16.16 고정 소수 연산을 지원한다.

본 엔진에서는 OpenGL ES 표준 규약에 따라 내부의 자료형을 16.16 고정 소수형으로 통일한다. 즉 integer변수(32bit)를 2부분으로 나누어서, 상위 16bit을 정수 부분, 하위 16bit을 소수 부분을 표현하는 데 사용한다. 고정 소수점은 부동 소수점과 달리 소수점의 위치가 이동하지 않기 때문에 가질 수 있는 값의 범위가 매우 한정되어 있다. 고정 소수로 표현할 수 있는 범위는 최대 32767(2^15)에서 최소 -32768(-2^15)이다. 그리고 정확도는 0.000015259(2^-16)이다.

하지만 어플리케이션에서 모델의 정점 위치 값으로 사용 가능한 범위는 이 범위

와 일치하지 않는다. 어플리케이션에서 입력된 값들이 3D 파이프라인을 통하면서 내부적으로 값의 변환이 일어나기 때문이다. 이 변환된 값의 범위도 고려해주어야 하므로, 어플리케이션 레벨의 3D 자료 값은 범위의 제약이 더 크다. 일반적으로 소수점 이하의 작은 수(+-2 범위)들이 위치 값으로 좋은 것으로 확인되었다. SW 개발자는 이런 오류범위를 숙지하여야 한다.

제 2 절 곱셈/나눗셈 정밀도

고정 소수점 형식의 두 수를 곱하거나 나누는 경우, 소수점의 위치가 각각 좌우로 16비트씩 옮겨지게 되므로 다시 16.16 표준으로 재 정비해 주어야 한다. 본 엔진에서는 정밀도와 속도의 반비례 관계를 가진 3 종류의 곱셈/나눗셈 방법을 제공한다. 고정 소수점 곱셈과 나눗셈 방법은 어플리케이션의 종류에 따라 제작자가 적절히 수정, 교체할 수 있다(표 1 참고). 레벨 1은 속도는 빠르지만 정밀도가 낮은 방법의 매크로이고, 레벨 2는 속도가 느리지만 정밀도가 높은 매크로이다. 레벨 3은 함수를 이용하는 것으로, 속도는 가장 느리고 정밀도는 높다. 레벨 2와 레벨 3의 정밀도는 큰 차이가 없으므로, 각 시스템에 맞추어 빠른 방법을 선택하면 된다.

레벨 1은 피연산자 두 수를 곱셈/나눗셈 계산 전후로 16bit를 분할하여 shift 해주는 방법이다. 분배되는 방식을 어플리케이션 특성에 맞추어 오버/언더플로우가 최소가 되도록 조정하는 것도 어플리케이션 제작자의 몫이다. 예를 들어 현재 나눗셈 매크로의 경우 나누어지는 수 a는 2의28승 이하의 수여야 하고, 나누는 수 b의 소수는 2의-12승까지만 유효하다. 특히 (b>>3) 이 0이 되지 않도록 주의한다.

또한 이 분배되는 방식에 따라서도 계산 속도가 달라진다.

레벨 1 매크로에서 부호 체크 부분을 삭제하면 약간의 시간 절약이 가능하지만, 그림이 손상될 수 있다. <그림 3>에서 호랑이의 발바닥 부분의 텍스쳐가 제대로 표현되지 않은 것을 확인할 수 있다.

레벨 2는 피연산자들을 64bit 정수 형으로 캐스팅하여 계산하는 방법으로, 결과값 역시 64bit 데이터 공간에 저장된 후 32bit로 캐스팅 되기 때문에 연산으로 인한 데이터의 손상이 거의 없다. 레벨 2의 방법은 부동 소수점 연산법과 거의 같은 매우 높은 정밀도를 표현하지만 64bit 캐스팅이 일어나는 과정에서 시간 소요가 크다. 속도의 차이는 어플리케이션에 따라 다르겠지만, 레벨 1에 비해 보통 2배 이상 느려지는 것으로 보인다. 레벨 3는 곱셈과 나눗셈을 위한 특별한 함수를 이용하는 것으로, 속도는 가장 느리다.

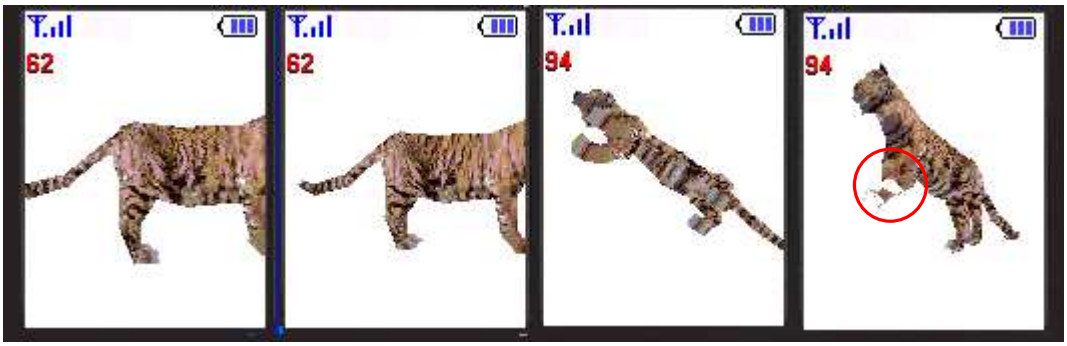
텍스쳐 효과는 정밀도의 영향을 특히 크게 받는다. 낮은 정밀도의 매크로를 사용하게 되면, 시선 방향에 따라 텍스쳐 무늬가 흔들리게 된다. (또한 나눗셈 정밀도의 영향력이 곱셈 정밀도에 비해 더욱 크다.) 텍스쳐를 이용하지 않는 시스템에서라면 레벨 1 정밀도를 사용하여 속도를 빠르게 해도 좋을 것이다. 레벨 1 정밀도의 이용시 발생할 수 있는 문제에 대해서는 다음 절에서 기술하겠다.

나눗셈의 경우 0으로 나누어 지게 되면 치명적인 결과를 맞을 수 있으므로 어느 경우에라도 레벨 2의 높은 정밀도 사용을 권장한다. 레벨 3 곱셈/나눗셈 함수의 사용은 추천하지 않는다.

- 현재 구현에서는 레벨 2의 정밀도를 사용하고 있으나, 래스터라이즈 단계 최말단의 함수 안에선 레벨 1 방식의 나눗셈(텍스쳐 계산을 위해 화면의 픽셀 개수만큼 나누기 호출)을 이용하고 있다. 이를 통해 매 프레임 40ms의 시간을 절약할 수 있었다. 이때 16 shift의 분배 방식에 의해서도 +-10ms 정도의 시간이 차이남을 볼 수 있었다.

< 표 1 > 곱셈/나눗셈 매크로

레벨 1 빠른 시간, 낮은 정밀도	속도는 빠르지만, 정밀도가 낮아 에러가 발생할 수 있다.
곱셈: ((((a)^(b))<0) ? ((ABS(a)>>8)*(ABS(b)>>8)) : ((a)>>8)*((b)>>8)) 나눗셈: ((((a)^(b))<0) ? -(((ABS(a)<<4)/(ABS(b)>>3))<<9) : ((((a)<<4)/((b)>>3))<<9))	
레벨 2 높은 정밀도, 느린 시간	매우 깨끗한 화면을 그려주지만, 속도가 다소 느리다.
((__gl_x) (((__gl_ll) (a)*(b)))>>__GL_X_FRAC_BITS)) ((__gl_x) ((((__gl_ll)(a))<<__GL_X_FRAC_BITS)/(b)))	
레벨 3 높은 정밀도, 느린 시간	함수를 이용한다. 속도가 가장 느리다.
FixedMul(a,b) FixedDiv(a,b)	



< 그림 3 > 나눗셈/곱셈 매크로 정밀도

<표 2>는 프로젝트에서 제공하는 고정 소수점 수학 함수의 리스트이다.

< 표 2 > 프로젝트에서 제공되는 고정 소수점 수학 함수

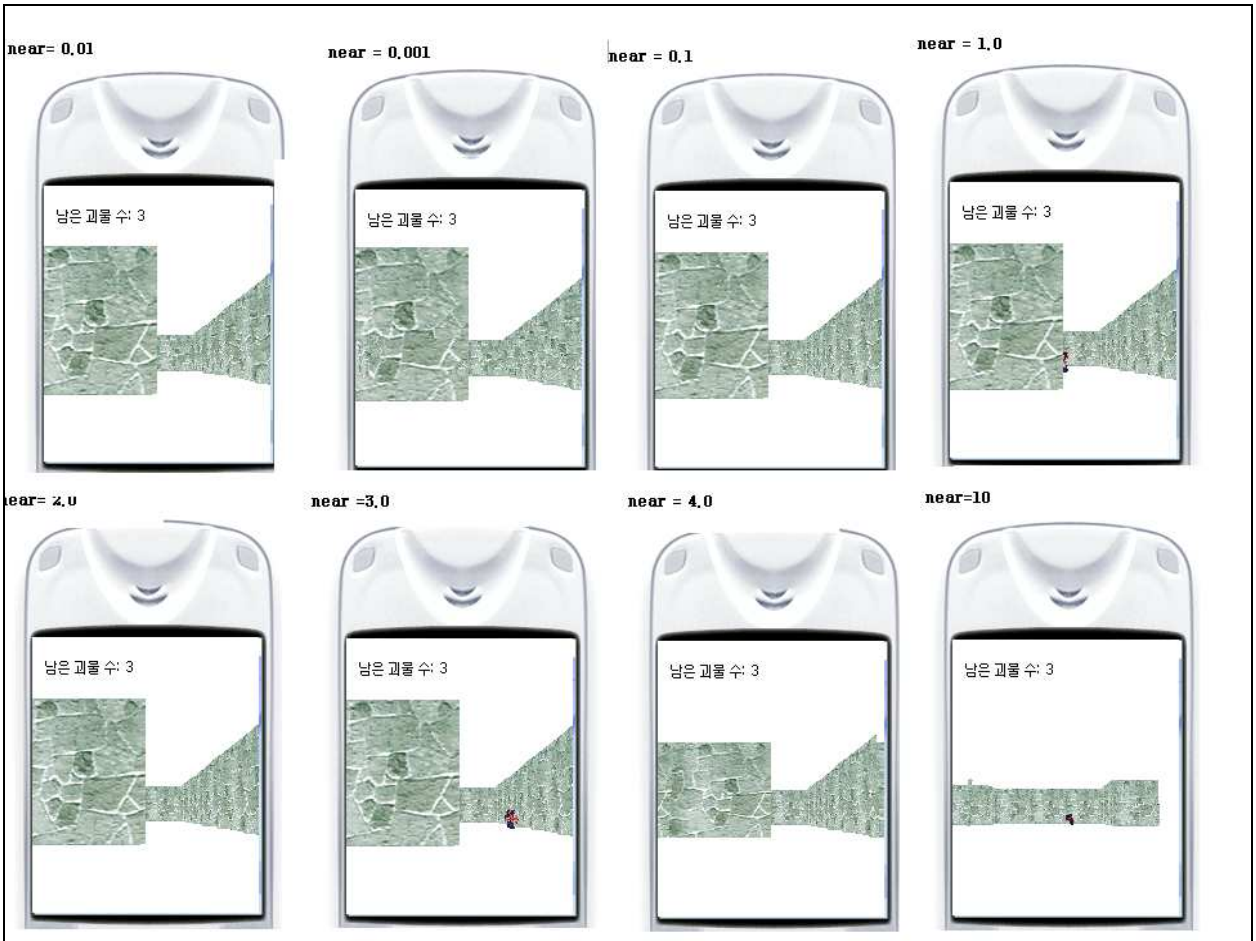
__gl_f_cos(__gl_f a)	Cosign (radian)
__gl_f_sin(__gl_f a)	Sign (radian)
__gl_f_sqrt()	Squrted root
__gl_f_pow(__gl_f a, __gl_f n)	a의 n승
__gl_f_recip(__gl_f a)	

제 3 절 사칙 연산 정밀도에 의한 품질 차이

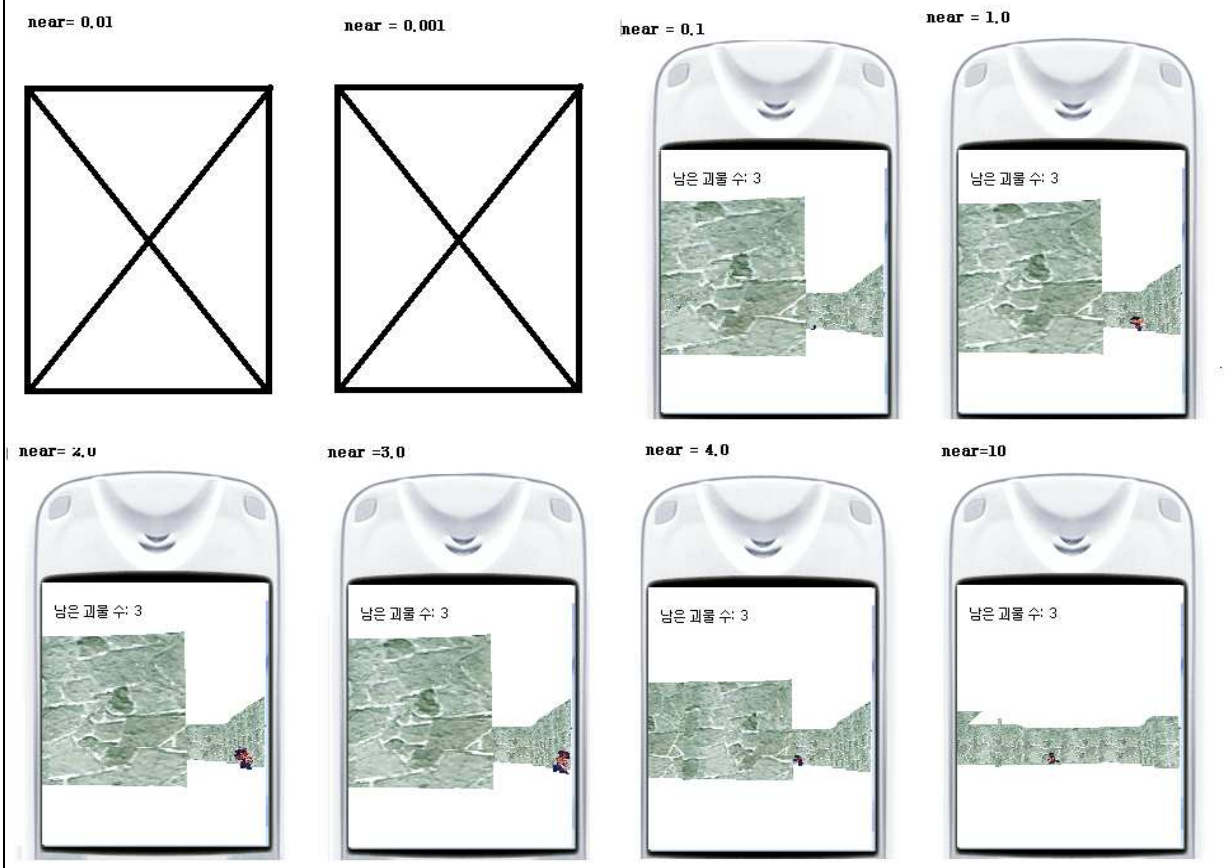
<그림 4>와 <그림 5>에서 곱셈/나눗셈 연산 정밀도에 의한 품질 차이를 확인할 수 있다. <그림 4> 은 고정 소수점 정밀도에 따른 near plane의 범위 제한에 대한 것이다. 높은 정밀도의 레벨 2 매크로를 사용한 경우(그림 4-1), near plane은 0.001~10 사이의 모든 범위에서 세팅이 가능하였다. 하지만 낮은 정밀도의 레벨 1 매크로를 사용한 경우(그림 4-2) 0.01 이하의 near plane 세팅은 런-타임 에러를 유발시키며 프로그램을 종료시켰다. 이는 매크로 연산 내부의 쉬프트 이동과 관련

하여 자료 값의 손실이 발생하기 때문이다.

<그림 5>는 고정 소수점 정밀도에 따른 Z-buffer 에러에 대한 화면이다. 낮은 정밀도의 매크로를 이용할 경우 화면에서 멀리 떨어질수록 z-depth의 오류가 커진다. 높은 정밀도를 사용할 경우 깨끗한 화면을 확인할 수 있으나(그림 5-3), 낮은 정밀도를 사용한 경우에는 멀리 있는 물체간에는 z-depth 비교에 실패한 것을 <그림 5-2>에서 확인할 수 있다. (벽 뒤쪽에 있는 괴물의 형상이 벽 앞에 아른거린다.)



4-1. 레벨2의 MUL/DIV 매크로: Near plane은 0.001~ 10 모든 범위 세팅 가능



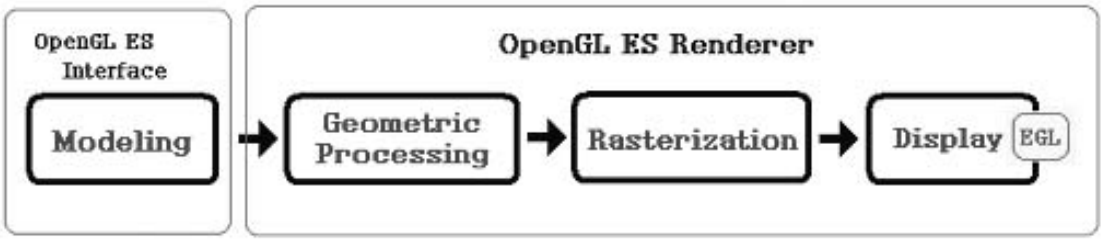
4-2. 레벨1의 MUL/DIV 매크로: Near plane 0.01 이하 세팅 불가능.

< 그림 4 > Fixed 정밀도에 따른 Near Plane 의 범위 제한

< 그림 5 > Fixed 정밀도에 따른 Z-buffer 에러

제 3 장 OpenGL ES Renderer

본 렌더러는 렌더링 파이프라인의 모든 단계가 소프트웨어로 동작하고 있다.



< 그림 6 > OpenGL ES 렌더링 파이프라인

제 1 절 구 조

3차원 그래픽스 파이프라인의 기본 구조는 대부분 비슷하다(그림 6 참조). 우선 제일 왼쪽의 Modeling 단계는 응용 단계로서, 어플리케이션 제작자가 렌더링 될 기하 구조를 구성하고 OpenGL ES 인터페이스를 이용해 파이프라인의 다음 단계로 넘기는 부분이다. 이렇게 입력된 기하 정보는 기하연산(Geometric Processing) 단계에서 모델 변환과 시야변환, 투영과 클리핑의 과정을 거치면서 화면 크기에 맞게 다듬어진다. 렌더링 파이프라인의 마지막 단계인 Rasterization 단계에서는 전 단계에서 넘어온 정점과 색상,텍스쳐 좌표 등을 참고로 하여 모든 기하 요소를 윈도우 픽셀들로 주사한다. 이렇게 모든 화면 버퍼가 채워지게 되면, 마지막 Display 단계에서 하드웨어 Display로 최종 출력된다.

OpenGL ES 모듈의 경우, Khronos 홈페이지에서 공개된 인터페이스 원천 소스를 바탕으로 구현작업을 진행했다. 원천 소스에는 래스터라이즈 모듈이 Desktop

OpenGL을 이용하는 형태로 존재하고 있었다. Desktop GL을 사용하는 루틴 부분을 제거하고, 거기에 해당하는 루틴을 새로 구현해 주는 작업이 가장 핵심이 되는 내용이다. 원천 소스에는 현재 빛과 텍스쳐를 적용한 삼각형 기반 메쉬를 돌릴 수 있는 정도의 내용만 구현되었다. Fog/Blending/ stencil buffer 부분이 아직 미 구현되어있다.

차후 전역변수 사용이 불가능한 시스템으로의 이식을 고려해 전역변수 구조체를 두었다. 현재는 속도를 고려해 전역변수 구조체를 '전역'으로 두었으나, 이 엔진을 전역변수 사용이 불가능한 환경으로 이식할 시엔 전역 구조체의 주소를 내부 API들의 인자로 전달하게 수정해 주면 된다.

제 2 절 구 현

기존 소스에 코헨-서덜랜드 클리핑 기법에 따라 구현된 클리퍼가 제공되고 있으나, 이것은 매우 불완전하게 작동하고 있어서 새로 구현하였다. 가장 단순한 개념의 뷰 프러스텀 컬링 클리퍼로서, 각각의 삼각형을 뷰 프러스텀 6개 면과 순차적으로 교차 계산하여 여러 개의 삼각형으로 분할하여 저장한다. (삼각형과 뷰 프러스텀의 면이 교차하게 되면, 면의 안쪽 부분에 걸치도록 새로운 삼각형들로 분할된다.)

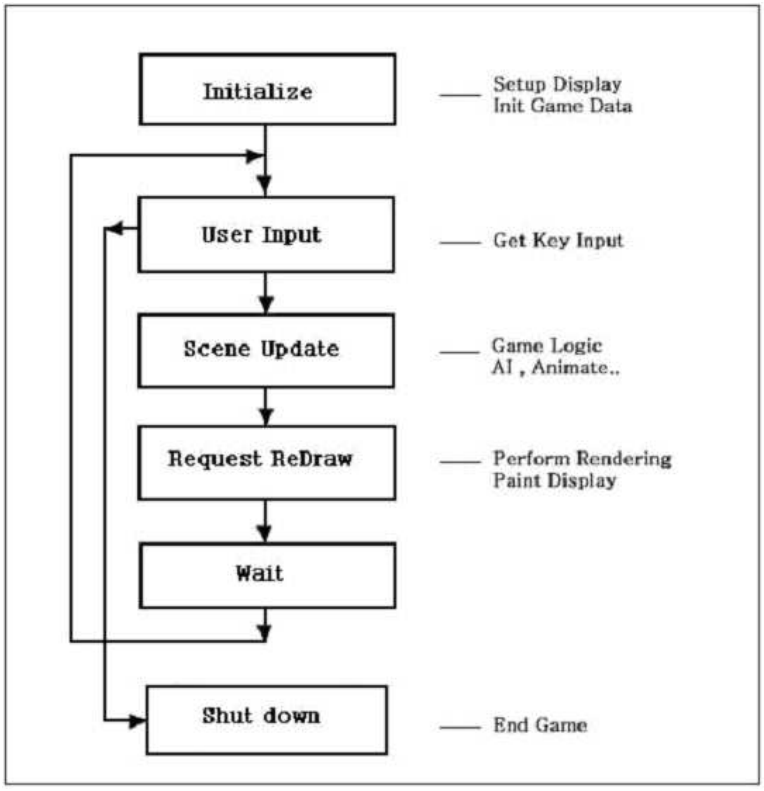
이렇게 분할된 삼각형들은 Rasterization 단계로 넘겨진다. 이 부분에선 각 삼각형의 2차원 화면상에서의 각 변의 픽셀 스팬을 계산한 후, 삼각형 내부를 주사하여 렌더링을 완성한다. 래스터라이저 역시 클리퍼와 마찬가지로 널리 알려진 이론을 이용하여 구현하였다. 모든 삼각형들에 대해 이 작업이 끝나면, Display단계로 넘어가 egl에 등록된 display로 화면 칼라 버퍼가 복사되고 출력된다.

제 4 장 게임 엔진

제 1 절 엔 진

상위 3D 엔진 모듈에서는, md2 모델의 애니메이션, 캐릭터 핸들링, 충돌 검사 등의 기능을 포함한다. 작업 과정에서는 하나의 게임을 만들어 가면서 3d 게임 엔진에서 제공해야 하는 간단한 기능들을 갖추어 나갔다. 제작한 게임을 바탕으로 설명하겠다. <그림 7> **일반적인 게임의 플로우 차트**: 게임에서 사용될 데이터를 초기화 한 후, 외부 키 입력에 따라 캐릭터들의 AI등 게임 로직을 계산한다. 렌더링 파이프라인 호출하여 한 장면을 완성/ 업데이트 된 화면을 출력한다.

< 그림 7 > 일반적인 게임의 플로우 차트



]제 2 절 게임 구현

이것은 둠을 모방한 게임으로 주인공이 미로를 다니면서 적 3마리를 없애는 게임이다. 주인공은 총과 칼 등의 무기를 이용해 적에게 타격을 입혀 적을 없앨 수 있다. 총알이 벽과 충돌할 경우 벽에 총알 자국의 decal이 생성된다. 적은 .md2 파일로 그려지고 있으며, 각자의 상태와 인공 지능으로 움직인다. 캐릭터들은 큐브의 집합으로 이루어진 미로 세계를 이동하며 다른 오브젝트(주인공, 캐릭터,벽, 총알, 칼)와의 충돌 여부를 감지하며 상태를 변화, 유지한다.

제 1 항 큐브 기반의 미로

게임의 미로 월드는 방, 문, 벽의 세 타입의 정육면체 오브젝트들의 집합으로 표현된다. 이렇게 단순화된 월드의 표현으로 오브젝트-월드 간 충돌 계산이 간단해져(곱셈과 나눗셈의 사용이 없다) 성능을 향상시키는데 도움이 된다.

미로는 방들의 집합이고, 방은 벽과 문과 공간의 집합이다. 주인공이 문을 통하여 방을 이동할 때마다 주인공의 위치와 시선 방향에 따라 특정 방이 활성화 되고, 이렇게 활성화 된 방만 렌더링 파이프라인에 들어가게 된다.

제 2 항 캐릭터 렌더링

.md2 파일처럼 많은 수의 삼각형으로 이루어진 오브젝트는 렌더링에 큰 시간을 소요한다. 이와 같은 커다란 오브젝트가 화면에 보이지 않는다고 판단되면 렌더링 파이프라인을 거치기 전에 계산에서 제외시켜 주는 것이 좋다.

* ChkDrawElements 함수: 입력되는 메쉬가 렌더링 될 것인지의 여부를 판단해 주는 gl 레벨의 함수이다. OpenGL ES의 Draw함수인 glDrawElements 함수의 쌍둥이 버전으로, 화면에 하나의 픽셀이라도 그려진다고 판단되면 1을 리턴하며 종료된다. 사용자는 heavy한 메쉬를 그리기 전에 임의의 바운딩 오브젝트와 ChkDrawElements 함수로 시야에 보이는지의 여부를 테스트할 수 있다. 이 방법은 어플리케이션의 성능을 향상시키는데 매우 유용하다. 하지만 Z-buffer가 불완전한 환경에서는 이 방법은 효과가 없을 것이다.

제 3 절 주의 사항

제 1 항 MD2 모델의 위치 값 제한

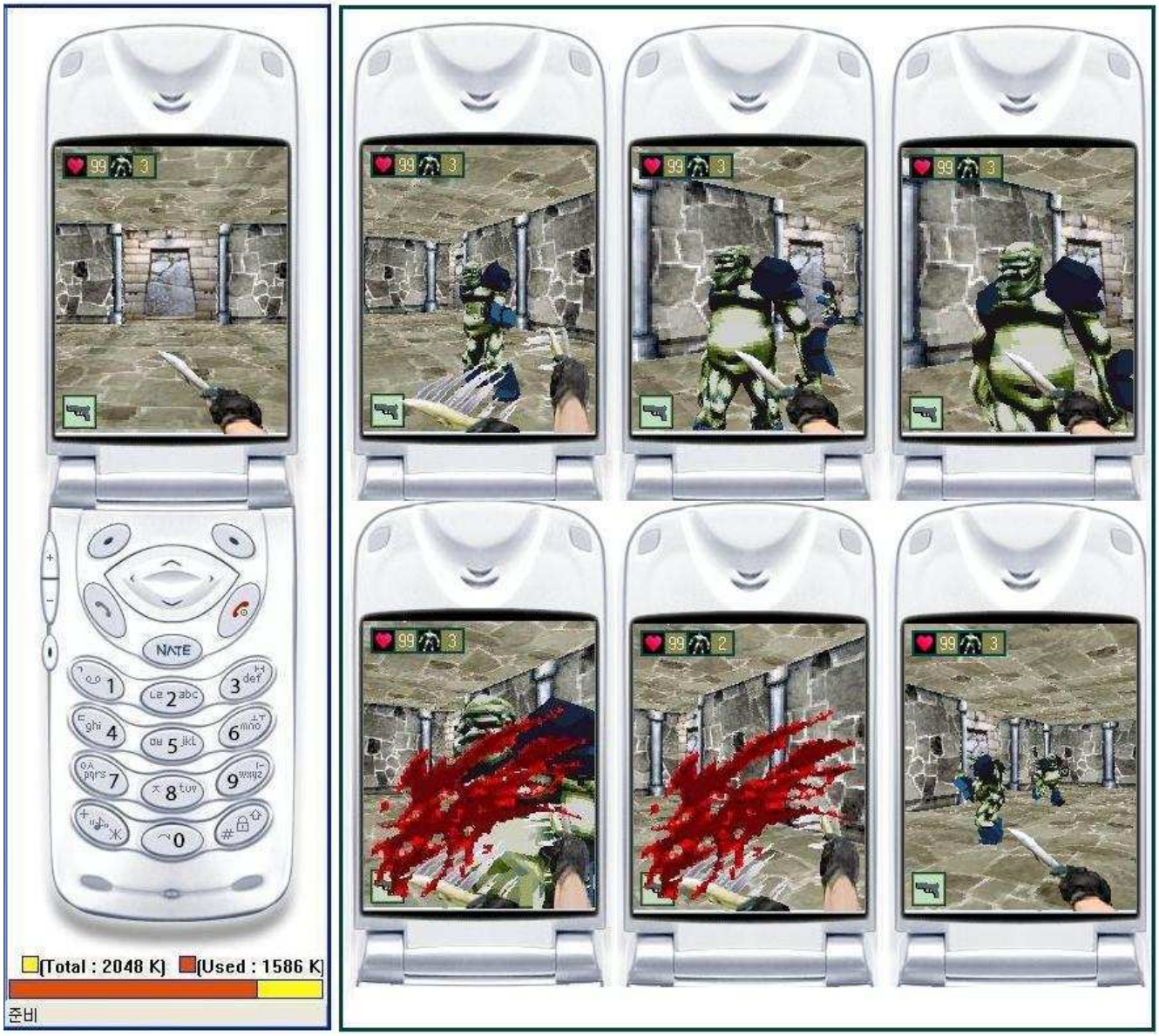
16.16 fixed point를 사용한 본 라이브러리의 경우, 표현할 수 있는 수의 범위는 최대 32767(2^15)에서 최소 -32768(-2^15)이다. 그리고 소수점 정확도는 0.000015259(2^-16)이다. 하지만 어플리케이션에서 모델의 위치 값으로 사용 가능한 범위는 이 범위와 일치하지 않는다. 어플리케이션에서 입력된 값들이 3D 파이프라인을 통하면서 내부적으로 값의 변환이 일어나기 때문이다. 이 변환된 값의 범위도 고려해주어야 하므로, 어플리케이션 레벨의 3D 자료값은 범위의 제약이 더 크다. 소수점 이하의 작은 수(+-2 범위)가 위치 값으로 좋은 것으로 확인되었다.

Scaling factor를 두어 md2 모델의 크기를 제어한다.

하지만 쉽게 구할 수 있는 MD2 파일 모델의 경우 그 사이즈 값은 천차만별이다. 따라서 어떤 MD2모델을 입력 받더라도 문제없도록 MD2 로딩 루틴에서 +-1의 범위에 들어오도록 스케일링하여 저장한다. 만약 어플리케이션 레벨에서 glScale()을 이용해 모델의 사이즈를 매번 변환해 주고 있는 프로그램이라면, 로딩 레벨에서 해당 부분의 scale 값을 조정해 주는 편이 성능을 향상시켜줄 것이다. 또한 MD2 값의 범위에 따라서 프로그램의 성능이 변하는 것으로 보이므로, 숙지하기 바란다. (Md2.c파일의 CMD2_Load 함수에 이 내용이 포함되어있다.)

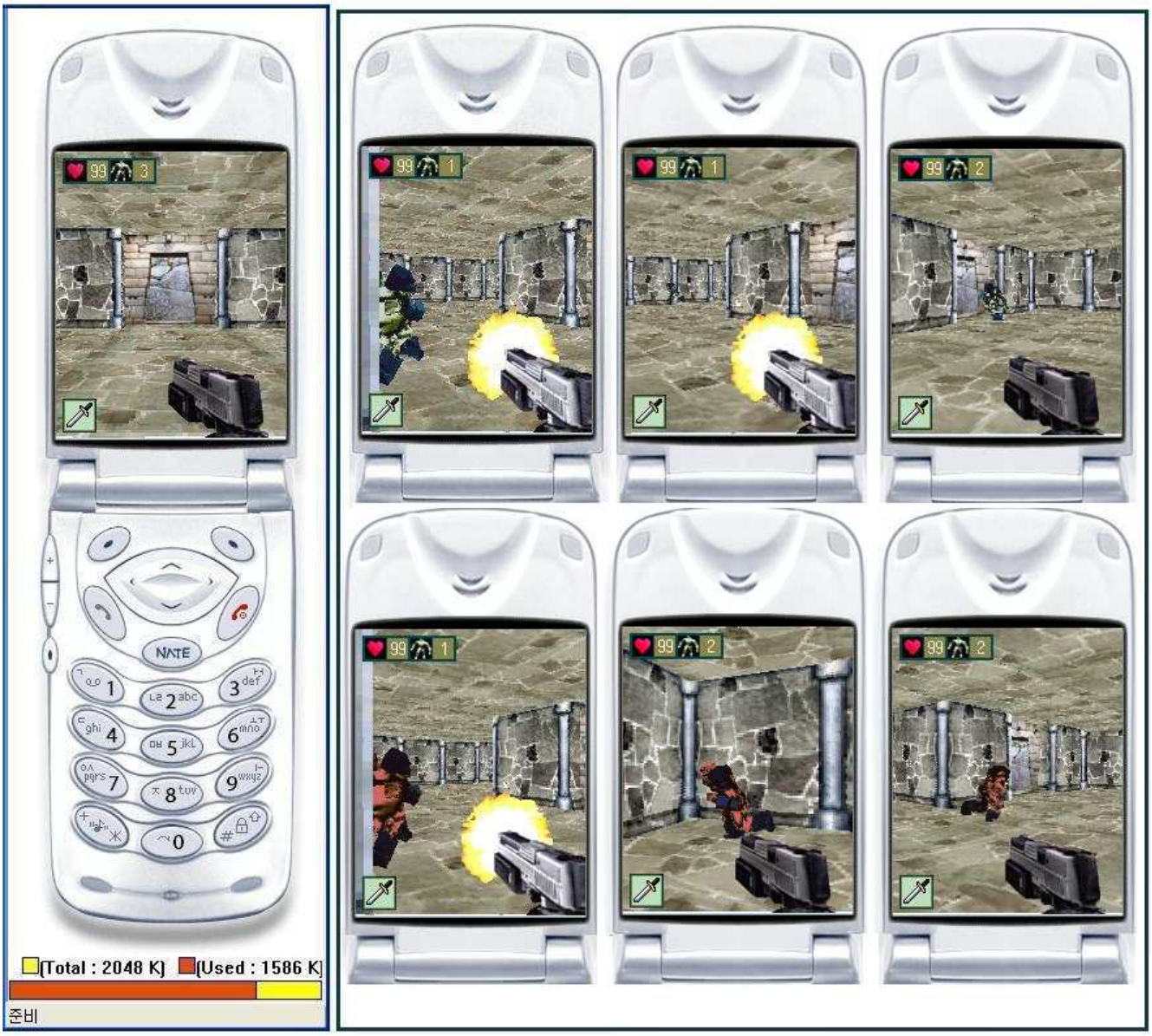
제 5 장 결과 및 결론

모바일 시스템에서 3차원 렌더링이 가능한 환경을 구현한 결과를 제시하였다. 3차원 환경을 표현하는 것은 문제가 없었으나, 많은 계산량으로 인해 만족할만한 성능을 얻지는 못하였다. 렌더링되는 개체가 화면에 차지하는 픽셀 크기와 클리핑 영역에 걸쳐 있는지 여부에 따라 성능은 달라지겠지만, 레벨 2의 매크로를 이용하여 그림 8과 같은 화면을 그릴 경우 4~5 프레임률을 보였다. 자연스러운 게임을 하기 위해서는 아직 2~3배의 성능 개선이 필요한 것으로 보인다. 특히 670개의 삼각형으로 이루어진 캐릭터가 화면에 비춰질 경우엔, 속도가 더욱 저하되었다. 캐릭터의 경우 로우 폴리곤을 이용하거나 빌보드를 이용하는 것이 프레임율을 균일하게 유지하는데 좋을 것이다. 배경 없이 렌더링을 하거나 텍스쳐를 이용하지 않는다면, 혹은 모든 오브젝트가 화면 안에 들어오는 어플리케이션이라면 (클리퍼 제거) 게임에 가능한 성능을 얻을 수 있을 것이다.



< 그림 8 > 결과 게임 화면 1

성능 개선을 위해 곱하기와 나누기 횟수를 줄이는 것도 중요하지만, MEMCPY 같은 메모리에 접근하는 표준 함수의 이용을 줄이는 것도 매우 중요하다. 현재 렌더링 파이프라인의 클리퍼 부분에서 이 같은 메모리 접근이 많이 시행되고 있는데 이 부분을 고치면 성능이 크게 개선될 것으로 보인다.



< 그림 9 > 결과 게임 화면 2

또한 같은 횟수의 곱하기와 나누기를 이용하더라도, 계산되는 두 수의 크기에 의해서도 성능이 달라지는 것을 볼 수 있었다. 예를 들어 너무 큰 수나 너무 작은 수를 서로 곱하는 것은 속도를 저하시키는 것이다. 나누기의 경우에도, 나누는 수가 너무 작으면 속도가 떨어졌다. 이 것은 cpu 성능의 문제이다.

이 라이브러리는 전역 변수를 허용하지 않는 시스템과의 호환을 위해 전역 변수 구조체를 두어 이용하고 있으며, 시스템 의존적인 Memory, I/O 관련 표준 함수, 수학 함수의 매핑 부분을 따로 관리하고 있다. 이 라이브러리들의 이용법과 주의사항이 부록에 기술되어 있다.

참고 문헌

[1]Tomas Akenine-Moller, Eric Haines저/ 신병석 오경수 공역, Real-Time Rendering 2판, 정보문화사, 2003

[2] BRIAN HOOK 저/ 김상호 역, C++로 구현한 3D 게임 엔진, 성안당, 2002

[3] Eric Lengyel 저/ 류광 역, 3D 게임 프로그래밍& 컴퓨터 그래픽을 위한 수학, 정보문화사, 2004

[4] Richard S. Wright, Jr.Michael Sweet 저/ 남기혁 역, OpenGL SuperBible 2nd Edition ,인포북, 2000

참고 링크

http://www.wipi.or.kr/

http://www.khronosorg/opengles/

http://www.gomid.com/2004/korean/ms/ms_engine_3.htm

http://www.imaso.co.kr/lecture/mobile/article.html?id=4494&forum=0

부록.

1.프로젝트의 파일들

Mapping.h	system standard 함수 매핑
SGlobalData.h	전역 변수 구조체. (gl 내부에서 사용하는 전역변수 포함)
g_sintab.h	fixed 형태의 sin table 초기화, 프로그램 초기화 루틴 안에서 한번 호출
CletModule.c	wipi main source
Bmp.c, Bmp.h	비트맵 로드
Md2.c, Md2.h	md2 로드/애니메이션. (Bmp.h include)
Character.h, Character.c	캐릭터 (md2.h include)
Cube.c, Cube.h	월드

- Main API

/* WIPI 레벨*/

void startClet	프로그램이 시작될 때 자동 호출.
void destroyClet	프로그램이 끝날 때 자동 호출.
void paintClet	화면이 업데이트 될 때
void handleCletEvent	이벤트 핸들러
void timerCallback	타이머 콜백 함수, 이상 wipi 레퍼런스 참조.

/* EGL 레벨*/

void createEGLWindow()	EGL 초기화, startClet 첫부분에서 호출한다.
void DestroyEGLWindow()	EGL 리소스 정리, destroyClet에서 호출한다.
void postEGLWindow()	하드웨어 버퍼 출력 함수, Render()에서 호출

/* 어플리케이션 레벨 */

boolean InitAppData();	어플리케이션 데이터 초기화, startClet에서 호출한다.
boolean CloseAppData();	어플리케이션 데이터 정리, destroyClet에서 호출한다.
void Render()	한 프레임의 완성+ 렌더링, 타이머 콜백 함수에서 호출

2. OpenGL ES 라이브러리 사용법

A. 시스템 이식 시 주의할 사항

시스템에서 사용하는 운영체제 환경에 따라 라이브러리 내에서 사용되고 있는 표준 함수들을 재정의 해 주어야 한다. 만약 메모리 제어 방식 자체가 일반 제어 방식과 다르다면, 렌더링 파이프라인의 해당 부분을 수정해 줄 필요가 있다. memory, I/O 관련 표준 함수와 수학 함수들의 재정의를 위해서는 'mapping.h' 파일을 수정한다. <그림 10>의 예시를 참고한다.

```
*현재 엔진의 내부 standard 함수들은 대문자 이름으로 사용되고 있다

#ifndef abs
#define ABS(x)      ((x)>0?(x):-(x))
#else
#define ABS          abs
#endif

#define MALLOC(X)           MC_knlAlloc(X)
#define FREE(X)             MC_knlFree((M_Uint32)X)
#define MEMCPY              memcpy
#define MEMSET              memset
.....
```

< 그림 10 > "Mapping.h" – Wipi의 경우

B. OpenGL ES 함수 이용 관련

본 라이브러리는 OpenGL ES 표준을 따르므로, 타 OpenGL ES 프로그램 사용법

과 동일하다. 단 fixed 전용 GL함수를 사용할 것을 권장한다. 아래와 같이 define 을 이용해 실수하는 일 없이 이용하자.

```
예) 어플리케이션 레벨. (wipi의 CletModule.c,)

#include "opengles/gl.h"          <- OpenGLES를 사용하기 위해 포함해야 한다.
#include "opengles/es/gl_imp.h"   <- Fixed 매크로를 사용하기 위해 포함해야 한다.
#include "SGlobalData.h"          <- 전역 구조체. mapping.h 선언.

#define glClearColor     glClearColorx
#define glTranslatef     glTranslatex
#define glRotatef        glRotatex        <- fixed 전용 GL함수만을 이용하자.
#define glMaterialfv     glMaterialxv
```

< 그림 11 > Fixed 전용 GL함수 사용

C. EGL

OpenGL ES 표준은 각기 다른 Embeded System으로의 원활한 이식을 위해, 하드웨어 장치 관련 API만을 모아 EGL API라 명명했다. 즉 하드웨어의 종류에 따라 EGL API를 재작성 해주어야 한다. 본 프로젝트에서는 WIPI 시스템에 맞추어 재정의하였다.

```
/* 하드웨어 및 GL 초기화, 프로그램의 시작 지점에서 호출. */
void createEGLWindow()
/* 화면 출력 , 하드웨어의 출력 버퍼를 얻어서 결과 이미지 출력*/
void postEGLWindow()
/* egl 리소르 정리해 주는 함수, 프로그램이 종료될 때 호출 */
void DestroyEGLWindow()
```

< 그림 12 > 프로젝트에서 호출하고 있는 EGL 함수

3. MD2 모델 사용법 (md2.h,.c)

md2 구조체 및 함수

CMD2Model	md2 구조체
init_CMD2Model	md2 구조체 초기화
CMD2_Load	md2, 텍스쳐 로드 : 현재 텍스쳐는 8bit 비트맵 형식만 지원
CMD2_Animate	애니메이션 출력
CMD2_SetupSkin	오픈지엘 텍스쳐 세팅 (CMD2_Load에 포함)
CMD2_Unload	리소스 정리 및 delete
CMD2_SetState	모델의 애니메이션 상태 Set
CMD2_GetState	모델의 애니메이션 상태 Get

사용 예)

```
(1) CMD2Model   model;                          <-구조체 변수 선언
(2) init_CMD2Model(&model);                     <-초기화
(3) CMD2_Load(&model,"md2/model.md2","md2/model.bmp");
        // 모델, 텍스쳐 파일의 경로를 입력
(4) CMD2_Animate(&model,0, 39, 100);            <- Draw
        // 내부에서 GL Draw 함수인 DrawElements 호출.
```

4. Bitmap 사용법 (Bmp.h, .c)

LoadBitmapFile	Bitmap 로드
LoadBitmapFileWithAlpha	Color Key를 가진 Bitmap 로드

현재는 8bit 비트맵 형식만 지원한다. 메모리의 ID를 관리하는 Wipi 스펙에 맞추어, 비트맵 로딩 성공 시 비트맵 버퍼의 포인터가 아닌 버퍼의 ID를 리턴한다.

LoadBitmapFile**WithAlpha**는 블렌딩을 위한 알파값을 지정해준다. 현재는 특정 색상(까만색)의 알파 값을 0으로 세팅한다. 이 색상은 함수 내부에서 수정하면 된다.

비트맵 텍스쳐의 알파값이 0으로 세팅된 모델의 부분은 래스터라이즈 할 때, 투명하게 나타난다. Decal이나 빌보드를 그리고자 할 때 이 함수를 이용한다.

5. 캐릭터 사용 관련 (Character.h, .c)

3D 게임 월드의 캐릭터는 일반적으로 위치, 방향, 속도, 크기, 모델 데이터, 각종 상태 변수들을 변수로 갖는다. Character.c/h 파일에 캐릭터를 관리하는 구조체와 함수를 포함하고 있다.

```
#include "md2/md2.h"          <- md2.h
/* 캐릭터들의 AI emum */
enum Enemy_status             // 적의 AI 상태, 주인공의 행동에 영향을 받는다.
{       AI_DEAD,
        AI_UNCARING,
        AI_SCARED,
        AI_ANGRY      };
enum Player_status            // 주인공의 상태, 핸들러 입력 이벤트에 영향을 받는다.
{       isDEAD,
        UP,
        DOWN,
        RIGHT,
        LEFT,
        SHOOTING,
        NOTHING
};
/* 적의 캐릭터 구조체 */
typedef struct
{       GLfixed radian;               // 방향의 radian
        GLfixed direction;            // 방향.
        GLfixed posX,posY,posZ;       // 위치
        int speed;                    // 속도
        int size;                     // 충돌 체크를 위한 모델의 반지름
        int sizeSqure;                // 반지름의 제곱
        int IsShowing;                //chkBox로 모델이 보이는지 여부.
//for AI
        int hitCount;                 // 총알 맞은 수.
        enum Enemy_status       status;   //모델 현재 상태.
```

```
        CMD2Model model;            // md2 모델 구조체.
}Enemy ;
extern Enemy    enemys[3];      // 적 3마리
extern Player    player;                //주인공
extern CMD2Model model; //복사용 전역 md2 모델 한 개.
.....
```

적 캐릭터는 md2 모델을 이용, 구조체 내부에 CMD2Model type의 structure를 포함한다. 여러 마리의 적 캐릭터를 이용할 때, 동일한 md2를 같은 수만큼 로딩한다면 그것은 메모리를 낭비하는 일이 될 것이다. 이 문제는 동일한 md2 모델은 한번만 로딩하여 여러 캐릭터에서 접속하는 방식을 이용하면 해결할 수 있을 것이다.

Character 함수

void initPlayer()	주인공 초기화
void initEnemy()	적 초기화, md2 로딩
void PlayerCollision()	**주인공- 적** 충돌체크, 주인공 위치 수정, 적의 AI에 영향
void EnemysCollision(int i, GLfixed prevX,GLfixed prevZ)	**i 번의 적 – 나머지 캐릭터, 월드**와의 충돌체크 충돌 시 해당 적의 위치 수정
void CaculatePlayer()	버튼 입력 상태에 따라 주인공 업데이트
void CaculateEnemy()	AI 상태에 따라 적들의 변수 업데이트
void PlayerSooting() void PlayerHitting()	**총 발사 시 계산 루틴, 검 공격 시 계산 루틴** 총알- 적 충돌 시 적의 AI 업데이트 총알- 월드의 충돌 시 decal 생성
Void GetBulletcollPos	**총알- 월드 충돌 시 호출** 정확한 충돌 면과 충돌 위치을 찾는다.
int drawchkbox()	해당 위치에 임의의 box를 가상으로 렌더하여, 그것이 화면에 보이는지 여부를 리턴.
int drawPlayers()	Draw

6. 월드 관련 (Cube.h, .c)

int **maze**[10][10]	미로를 표현하는 전역 배열
void drawMaze()	미로를 그린다.
void drawCube (int i,int j, int type)	정육면체를 그린다.

cube.c 에는 큐브를 기초로 한 월드 전체를 그리는 루틴이 포함된다.
현재는 -1~1 size의 Unit Cube 를 이용해 10* 10 크기의 미로를 그리고 있다. maze[][]전역 배열은 캐릭터와의 충돌체크를 위해 Character.h에도 extern으로 선언한다. Unit Cube의 vertex/texture coordinate, Index buffer는 **unitCube, CubeTex, CubeIndex**란 이름으로 제공된다. Decal을 그리는 루틴도 현재 Cube.c에 포함되어 있다.

7. Decal 사용 샘플 (Cube.h, .c)

Decal의 텍스쳐는 **LoadBitmapFileWithAlpha**로 로딩된다. Unit Decal의 vertex/texture coordinate, Index buffer는 cube.h에 **unitBDecal, BDecalTex, BDecalIndex** 이란 이름으로 제공된다. 샘플 게임에서는 총알과 벽의 충돌 면에 총알 자국의 Decal을 그리고 있다. BDecal의 생성(active)과 변수 세팅은 Character.c의 void PlayerSooting() 함수 참고한다.

< 그림 13 > 벽에 생성된 총알 자국

Abstract

This is about design and implementation of 3D Graphics Pipeline and 3D Game Engine on Embedded Systems. This graphics pipeline enables emebedded systems without graphic hardware to render 3-dimensional environment. And because it follows OpenGL-ES standard, Applications in different system can be easily ported in each other systems with these engines.