



PSP[®] 프로그래밍 입문서

릴리스 1.0

2005 년 3 월

© 2005 Sony Computer Entertainment Inc.

Publication date: March 2005

Sony Computer Entertainment Inc.
2-6-21, Minami-Aoyama, Minato-ku
Tokyo 107-0062, Japan

Sony Computer Entertainment America
919 E. Hillsdale Blvd.
Foster City, CA 94404, U.S.A.

Sony Computer Entertainment Europe
30 Golden Square
London W1F 9LD, U.K.


The *Introduction to Programming the PSP® Release 1.0* is supplied pursuant to and subject to the terms of the Sony Computer Entertainment PlayStation® license agreements.

The *Introduction to Programming the PSP® Release 1.0* is intended for distribution to and use by only Sony Computer Entertainment licensed Developers and Publishers in accordance with the PlayStation® license agreements.

Unauthorized reproduction, distribution, lending, rental or disclosure of this book to any third party, in whole or in part, is expressly prohibited by law and by the terms of the Sony Computer Entertainment PlayStation® license agreements.

Ownership of the physical property of the book is retained by and reserved by Sony Computer Entertainment. Alteration to or deletion of the book, in whole or in part, its presentation, or its contents is prohibited.

The information in the *Introduction to Programming the PSP® Release 1.0* is subject to change without notice. The content of this book is Confidential Information of Sony Computer Entertainment.

“” and “PlayStation” are registered trademarks of Sony Computer Entertainment Inc. All other trademarks are property of their respective owners and/or their licensors.

목차

설명서 소개	v
제 1 장: 예비사항	1-1
서문	1-3
에뮬레이터	1-3
ALLEGREX CPU 에뮬레이션	1-3
libGu 및 libGum 그래픽 라이브러리의 기능	1-4
컨트롤러 지원	1-4
PSP DTP-T1000A 개발 하드웨어 (하드웨어 툴)	1-4
에뮬레이터와 하드웨어 툴의 차이점	1-4
하드웨어 툴을 위한 소프트웨어	1-5
이 문서의 예제 코드	1-5
에뮬레이터와 하드웨어 툴에 사용되는 실행가능 파일 포맷의 차이점	1-5
정보 및 지원	1-5
제 2 장: PSP 하드웨어: 시스템 투어	2-1
일반 개요	2-3
ALLEGREX CPU 코어	2-4
그래픽 엔진	2-4
그래픽 엔진의 내부	2-4
디스플레이 리스트	2-6
디스플레이 리스트의 유형	2-6
LibGu	2-7
libGu 수행 과정을 보여주는 예	2-7
디스플레이 리스트의 데이터	2-8
제 3 장: 간단한 프로그램의 컴파일	3-1
서문	3-3
데이터 구조	3-3
초기화	3-3
드로 및 디스플레이 버퍼	3-4
뷰포트 초기화	3-5
초기화 섹션의 최종화	3-6
메인 루프	3-6
점점 포맷과 그래픽 엔진에 의한 해석 방식	3-7
색인 대 비색인 프리미티브	3-7
요약	3-8

제 4 장: 간단한 3 차원 장면	4-1
서문	4-3
깊이 버퍼	4-3
3 차원 좌표 시스템 및 libGum	4-4
LibGum	4-4
조명	4-5
패치	4-7
컨트롤러 취급	4-7
에뮬레이터의 컨트롤러 입력	4-8
하드웨어의 컨트롤러 입력	4-8
시야 변환: 간단한 카메라	4-9
물 질	4-10
패치	4-11
행렬 변환	4-11
요약	4-12
제 5 장: VFPU (벡터 부동 소수점 유닛)	5-1
서문	5-3
어셈블리 언어	5-3
The VFPU	5-3
Libvfpv	5-3
행렬 레지스터	5-4
코드	5-6
기능	5-6
요약	5-11
결론	5-11
부록 A: gcc 인라인 어셈블리 구문	1
어셈블러 옵션의 설정	3
어셈블러 코드	4
출력 / 입력 변수	4
제약	4
클로버	5
부록 B: 에뮬레이터 및 하드웨어 툴 라이브러리의 차이점	1
에뮬레이터의 특성	3
하드웨어 툴의 특성	3
메모리 관리	4
참조 문헌	4

설명서 소개

이 문서의 제목은 *PSP® 프로그래밍 입문서 릴리스 1.0*입니다.

이 문서는 PSP 환경에 익숙하지 않는 프로그래머를 위해 마련되었습니다. 또한, 이 문서와 동일한 소스 디렉터리에 포함되어 있는 코드의 예제를 설명합니다.

이 문서는 3차원 컴퓨터 그래픽에 대한 기본적 이해는 물론 C 및 C++와 같은 컴퓨터 언어에 대한 지식을 갖춘 프로그래머를 위하여 마련되었습니다. 어셈블리 언어의 지식은 VFPU(벡터 부동 소수점 유닛) 코프로세서를 다루는 제 5 장의 내용을 이해하는 데에 도움이 됩니다. 부록 A는 이 장을 뒷받침하는 정보를 포함하고 있으며, 어셈블리 언어가 익숙하지 않은 초보자를 위한 인라인 어셈블러 구문에 대한 서문을 제공합니다.

이 문서는 런타임 라이브러리 문서(<https://psp.scedev.net>)에 나와 있는 기능이나 지침을 참고로 사용합니다. 본 설명서를 사용하기 전에 런타임 문서를 먼저 읽거나, 아니면 참고 사항도 함께 사용하는 것을 권장합니다.

이 문서는 런타임 라이브러리 문서를 대신하지 않지만, 런타임 문서에서 다루는 내용이 반복되거나 그 개념을 보다 자세히 설명합니다.

이 문서에 나와 있는 코드의 예나 예제에는 에뮬레이터 및 DTP-T1000A 하드웨어 환경에서 작동하는 기능이 포함됩니다. 그러나 제 5 장에 있는 내용의 일부는 DTP-T1000A 하드웨어 환경에만 관련이 있으며, 그러한 정보는 VFPU를 사용하는 코드 작성에 관한 것입니다.

부록 B에는 에뮬레이터(Emulator) 및 DTP-T1000A 하드웨어와 관련있는, 소스 코드의 기능 호출에 관한 정보가 나와있습니다.

최종 릴리스 이후의 변경 사항

없음: 신규 설명서.

예비 요구사항

PSP™ 개발에 관한 지식.

인쇄에 관한 규약

이 문서에서는 문장의 의미를 명확히 하기 위해 다음과 같은 인쇄에 관한 규약을 사용합니다:

규약	의미
커리어체	리터럴 프로그램 코드를 표시.
<i>이탤릭체</i>	인수 및 구조 멤버의 명칭을 표시 (구조/기능 정의에만 적용).
중간 굵기	데이터 유형 및 구조/기능의 명칭을 표시 (구조/기능 정의에만 적용).
청색	하이퍼링크를 표시.

개발자 지원

Sony Computer Entertainment America (SCEA)

SCEA 개발자 지원은 북미지역의 라이선스 소유자에게만 제공됩니다. 개발사 지원 서비스나 이 문서의 사본을 얻으려면 다음으로 연락하시면 됩니다:

주문 안내	개발자 지원
Attn: Developer Tools Coordinator Sony Computer Entertainment America 919 East Hillsdale Blvd. Foster City, CA 94404, U.S.A. Tel: (650) 655-8000	이메일: scea_support@psp.scedev.net 웹: https://psp.scedev.net 개발자 지원 핫라인: (650) 655-5566 (월 - 금요일, 오전 8 시 - 오후 5 시, PST/PDT)

Sony Computer Entertainment Europe (SCEE)

SCEE 개발자 지원은 PAL 텔레비전 지역(유럽, 오스트랄라시아 포함)에서의 라이선스 소유자에게만 제공됩니다. 개발사 지원 서비스나 이 문서의 사본을 얻으려면 다음으로 연락하시면 됩니다:

주문 안내	개발자 지원
Attn: Development Tools Manager Sony Computer Entertainment Europe 13 Great Marlborough Street London W1F 7HP, U.K. Tel: +44 (0) 20 7859-5000	이메일: scee_support@psp.scedev.net 웹: https://psp.scedev.net 개발자 지원 핫라인: +44 (0) 20 7911-7711 (월 - 금요일, 오전 9 시 - 오후 6 시, GMT/BST)

제1장:

예비사항

이 페이지는 공백으로 합니다.

서문

PSP에는 다음 두 가지의 개발 환경이 있습니다:

- 에뮬레이션 환경
- DTP-T1000A 하드웨어 툴을 사용하는 하드웨어 환경

이 장은 두 가지 환경의 차이점을 요약합니다. 또한 PSP를 위한 프로그램 작성에 대한 사전 요구사항도 정의합니다.

이 장은 이미 시스템을 설치하지 않은 경우 처음 시작하는 데 유용하며, PSP를 위한 프로그래밍 작업을 시작하는 데 필요한 자료의 목록을 제공합니다. 그러나 이 장이 설치 지침서는 아닙니다. DTP-T1000A 하드웨어 툴 및 에뮬레이터와 관련하여 두 환경을 위한 설치 과정을 설명하는 다수의 문서가 이미 있습니다.

에뮬레이터

PSP 에뮬레이터 환경이란 PC 즉 윈도 환경에서 PSP 프로그램의 컴파일, 실행, 보기 및 디버그를 할 수 있는 툴들을 말합니다. 에뮬레이터 코드의 컴파일 및 실행에 필요한 모든 파일은, <https://psp.scedev.net>의 다운로드 섹션에서 다운로드 할 수 있습니다.

에뮬레이터는 개발자가 런타임 라이브러리를 사용하여 코드의 개발을 시작하는 데 도움이 되는 유용한 방법임이 이미 입증되었습니다.

설치에 관한 정보는 다음 패키지와 함께 제공되는 문서를 참조하십시오:

- Win32 컴파일러 (gcc)
- Win32 디버그 환경
- Win32 PSP 에뮬레이터 프로그램
- 런타임 라이브러리 및 문서

에뮬레이터는 PSP CPU, 그래픽 엔진 및 핸드헬드 컨트롤러만을 지원합니다. DTP-T1000A 하드웨어 툴 대신 에뮬레이터를 사용하여 PSP 프로그램의 개발을 시작하는 경우, 나중에 하드웨어 툴에서 같은 코드를 실행할 때 코드에 대한 최소한의 변경이 필요합니다.

ALLEGREX CPU 에뮬레이션

이것이 에뮬레이터의 주 기능입니다. 모든 코드는 PSP의 ALLEGREX CPU 상에서 실행될 때 MIPS 4000 표준에 따라 컴파일 됩니다. 이것은 표준 MIPS 에뮬레이터이므로, 윈도 환경 하에서 MIPS 코드의 컴파일과 디버그를 수행할 수 있습니다.

libGu 및 libGum 그래픽 라이브러리의 기능

LibGu 는 Sony Computer Entertainment (SCE)가 공급하는 기능이 풍부한 그래픽 라이브러리로서, 에뮬레이터 및 개발자의 하드웨어 툴과 함께 공급됩니다. 이 그래픽 라이브러리의 기능들의 대부분은 에뮬레이터에서 OpenGL 을 사용하여 지원되므로, PSP DTP-T1000A 하드웨어 툴에 대한 접속을 위해 대기하는 동안 libGu 를 사용하여 PSP 고유의 엔진 코드를 작성할 수 있습니다.

또한 에뮬레이터 환경에서는 LibGum API 도 사용할 수 있습니다. 이것은 libGu 위에 위치한 또 하나의 그래픽 API 이며, 주로 행렬 수학의 기능을 제공합니다.

컨트롤러 지원

에뮬레이터는 일반 윈도 컨트롤러 드라이버를 통하여 컨트롤러 에뮬레이션을 지원합니다. 그렇게 하여 윈도 호환 컨트롤러는 에뮬레이션 환경에서 PSP 의 통제 기능과 같이 행동하도록 설정할 수 있습니다. 에뮬레이터는 아날로그 스틱은 물론 PSP 의 모든 버튼을 지원합니다.

PSP DTP-T1000A 개발 하드웨어 (하드웨어 툴)

하드웨어 툴이란 PSP 소프트웨어의 개발에 요구되는 장비를 말합니다. 이 툴에는 디버그 환경이 네트워크 상에서 툴과의 통신을 허용하는 PC 하드웨어뿐만 아니라 PSP 의 완벽한 기능도 포함됩니다.

하드웨어 툴은 내장 화면을 갖춘 PSP 컨트롤러와 함께 공급됩니다. 또한 별도의 모니터에서 출력을 볼 수 있도록 하는 VGA-out 커넥터도 갖추고 있습니다.

하드웨어 툴의 설치 방법에 관한 설명은 그와 함께 공급되는 문서를 참조하십시오.

작성된 코드를 시험하려면, 윈도 혹은 리눅스를 실행하는 호스트 PC 에서 코드를 컴파일한 다음, 네트워크를 통하여 프로그램을 하드웨어 툴에 보내면 그곳에서 소프트웨어가 실행됩니다.

에뮬레이터와 하드웨어 툴의 차이점

하드웨어 툴을 사용하여 코드를 개발하는 경우, 위에서 언급한 CPU 의 하드웨어 기능 및 그래픽 엔진 뿐만 아니라 제작용 PSP 콘솔의 일부인 다른 하드웨어 기능도 사용할 수 있습니다. 제작용 PSP 콘솔의 특징은 다음과 같습니다:

- VFPU (벡터 부동 소수점 유닛)
- 오디오 및 비디오
- UMD
- 네트워킹
- 메모리 스틱
- USB

위에 관한 정보는 다음 사이트에서 제공되는 문서에서 찾을 수 있습니다: <https://psp.scedev.net>.

하드웨어 툴을 위한 소프트웨어

SCE 는 리눅스 및 윈도 운영체제를 위한 각종 소프트웨어 툴을 제공합니다. 이 문서에서 설명하는 예제는 리눅스 메이크파일과 함께 제공됩니다. 그러나 이러한 예제는 네트워크 연결을 거쳐 하드웨어 툴에 연결되는 리눅스나 윈도 디버거에서 실행할 수 있습니다. 다음의 필수 패키지는 <https://psp.scedev.net> 을 통해 얻을 수 있습니다. 이 패키지들은 설치 방법을 설명하는 지침과 함께 제공됩니다.

- Linux 컴파일러 (gcc)
- Linux / Win32 디버거 환경
- 런타임 라이브러리 및 문서

이 문서의 예제 코드

본 자습서에 나와 있는 소스 코드는 다음의 두 디렉토리로 나누어집니다: 에뮬레이터에서 사용하는 코드 및 하드웨어 툴에서 사용하는 코드. 두 환경의 소스 코드는 서로 매우 유사하며, 앞으로 제공되는 예를 통하여 알 수 있습니다. 주요 차이점은 Library Linkage 옵션에 있으며, 메이크파일에 명시되어 있습니다.

에뮬레이터와 하드웨어 툴에 사용되는 실행가능 파일 포맷의 차이점

각 환경에서 사용하는 실행가능 파일의 포맷은 다음과 같이 다릅니다:

- 에뮬레이터에서는 .elf 파일을 실행
- 하드웨어에서는 .prx 파일을 실행

이에 대한 예를 읽기 전에, 사용하려는 플랫폼 즉, 윈도나 리눅스에서의 일반적인 컴파일 및 디버거 절차를 확실하게 알아두며 런타임 라이브러리에 포함된 예제 코드를 실행하십시오.

정보 및 지원

- 최신 라이브러리 릴리스에 관한 뉴스, 업데이트 및 정보 등 PSP 에 관한 정보는 다음 사이트를 참조하십시오:
<https://psp.scedev.net>
- 라이선스를 소지한 PSP 개발자를 위한 개발자 지원에 관한 정보는 다음을 참조하십시오:
<https://psp.scedev.net/support/about>
- PSP 개발의 전용 뉴스 그룹에 관한 정보는 다음을 참조하십시오:
<http://www.scedev.net/psp/connect.html>

이 페이지는 공백으로 둡니다.

제2장:

PSP 하드웨어: 시스템 투어

이 페이지는 공백으로 둡니다.

이 장은 에뮬레이터와 하드웨어 톨 환경 즉, CPU 및 그래픽(Graphics) 시스템과 호환되는 PSP 시스템 아키텍처의 여러 측면을 간단히 설명합니다.

PSP에는 첨단 휴대용 콘솔로서의 기능을 위한 많은 특징이 있으며, 이를 이해하려면 상당한 설명이 요구됩니다. PSP 개발의 다양한 양상과 관련 지식을 더 많이 배양함으로써 게임 개발 능력을 향상시킬 수 있습니다.

일반 개요

PSP는 다기능의 휴대용 게임 시스템입니다. 그 기능은 크게 다음의 세 가지로 분류할 수 있습니다:

- CPU 및 그래픽
- 미디어 (오디오 및 비디오)
- 연결성 (WiFi, USB, 메모리 스틱 등)

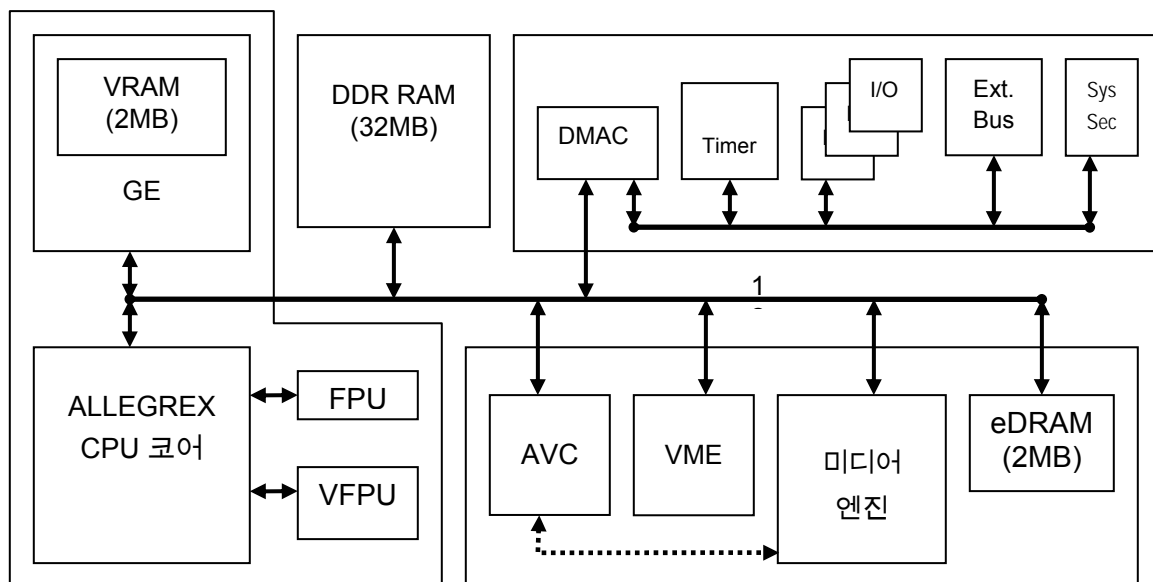
그림 2-1은 PSP 하드웨어의 세 가지 주요 블록의 기능성을 다음과 같이 분류한 것입니다:

- CPU/GE 블록
- 미디어 엔진 블록
- I/O 블록

위의 세 블록과 32MB DDR 주 메모리는 128 비트 버스에 의해 연결됩니다.

그림 2-1: PSP 시스템 블록의 구성

PSP 시스템 블록 선도



ALLEGREX CPU 코어

ALLEGREX는 PSP의 주 CPU입니다. 이것은 32 비트 MIPS RISC 프로세서이며, 표준 MIPS R4000과 동일한 명령 세트 및 예외 취급 기능을 갖습니다. 하지만 ALLEGREX는 별도의 FPU 및 강력한 벡터 부동 소수점 유닛(VFPU)을 코프로세서로 갖추고 있습니다. 명령 및 데이터를 위한 두 개의 캐시가 포함되며, 각각 16 KB입니다. 일반적으로 이것은 표준 MIPS CPU이며, 컴파일된 고유 C 또는 C++ 코드를 문제없이 수용합니다. MIPS 프로세서의 메모리에 있는 데이터는 그 크기의 배수에 따라 정렬시켜야 합니다. VFPU 또한 이것을 요구조건으로 규정합니다. 주 CPU에 관한 상세한 정보는 [1]에 나와 있습니다.

그래픽 엔진

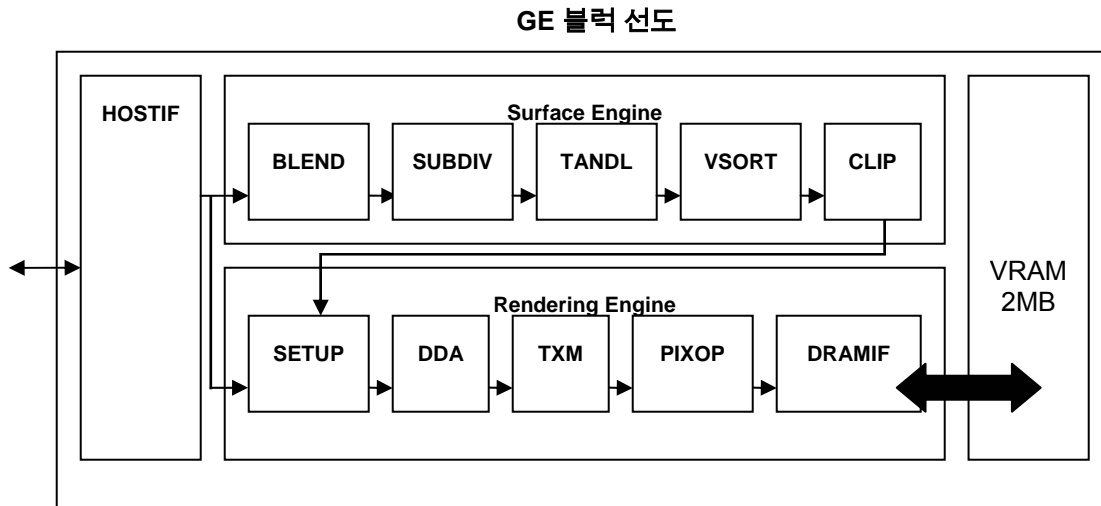
이 절은 PSP 그래픽 엔진(GE)에 관한 세부사항을 다룹니다. 이것은 선택사항으로서 구축된 GPU이므로, 보다 자세한 내용은 필요에 따라 제공될 것입니다.

그래픽 엔진이란 여러 가지의 뛰어난 하드웨어 동작을 수행하는 강력한 3 차원 및 2 차원 그래픽 칩을 말합니다. 이 엔진은 디스플레이 리스트에 의해 정보와 명령이 공급되며, 디스플레이 리스트들은 그래픽 엔진 내부에서 레지스터를 설정하고, 다양한 상태 변경을 그래픽 엔진에 알리며, 실행을 시작 또는 중단시킴으로써 프레임 버퍼에 대한 프리미티브를 제공합니다. 하드웨어를 위한 코드 작성 시, 자체의 디스플레이 리스트 매니저를 작성할 수 있지만, 이러한 기능은 이전 장에서 언급된 바와 같이 SCE 그래픽 라이브러리인 libGu에 의해 제공됩니다. libGu는 나중에 더 자세히 설명합니다.

그래픽 엔진의 내부

그래픽 엔진이란 고정 기능 그래픽 프로세서입니다. 여러 가지의 강력한 성능을 위하여 하드웨어 가속 기능을 가지고 있습니다. 그림 2-2에 나와 있는 다양한 명칭의 직사각형은 그래픽 파이프라인의 여러 부분을 수행하는 개별적인 기능 블록입니다.

그림 2-2: PSP GE 내부 블록 구성



이 도면은 다음의 두 주요 블록을 보여줍니다: '서피스 엔진' 및 '렌더링 엔진'. 이 두 엔진은 일반적으로 정점 및 픽셀 동작을 각각 수행합니다. 호스트 인터페이스인 HOSTIF 는 DMA 컨트롤러와 디스플레이 리스트 해석기(parser)를 포함합니다. 이것은 디스플레이 리스트의 명령을 해석하여 주 메모리로부터 정점 데이터를 끌어내어, GE 의 다양한 부분에 보내어 처리시킵니다. 따라서, HOSTIF 는 디스플레이 리스트 명령을 해석함으로써 다양한 기능 블록에 대해 렌더링 동작을 중재합니다.

그림 2-2 에서 보는 바와 같이 GE 내부에는 HOSTIF 에서 그래픽 파이프라인의 경로가 2 개 있습니다. 정점 데이터는 GE 의 '서피스 엔진' 부분을 우회함으로써 화면 공간에 대해 프리미티브를 렌더링합니다. 이 기능은 GE 가 지원하는 두 가지의 주요 정점 모드인 'NORMAL 모드' 및 'THROUGH 모드'와 관련이 있습니다:

- NORMAL 모드는 정점은 서피스 엔진에서 시작하여 GE 파이프라인 전체를 거쳐 처리되도록 지정합니다.
- THROUGH 모드의 정점은 서피스 엔진을 우회하며, 렌더링 엔진 블록에서 취급됩니다. 다시 말해서 NORMAL 모드의 정점은 서피스 엔진에 의해 처리된 다음, GE 파이프라인의 다음 단계를 위하여 THROUGH 모드 정점으로 변환됩니다.

위의 내용은 다음 장에서 더 자세히 설명합니다.

서피스 엔진 내부의 기능별 블록은 다음과 같습니다:

- BLEND 블록은 PSP 의 정점 블렌딩 동작을 취급합니다. 필요에 따라 스키닝 및 모핑 과정을 수행합니다.
- SUBDIV 블록은 Bezier 및 Spline 패치 데이터의 모든 테셀레이션을 관리합니다.
- TANDL 블록은 모든 정점을 다양한 좌표 시스템으로 변환시킵니다:
로컬→월드→눈→클립→화면→그리기 공간.

- VSORT 블록은 triangle-strip 정점 오더링 등의 동작을 수행하여 삼각형 정점을 정렬합니다.
- CLIP 블록은 삼각형 데이터에 대한 클리핑 동작을 수행합니다.

렌더링 엔진 내부의 기능별 블록은 다음과 같습니다:

- SETUP 은 일차 RGBA 및 이차 RGB 색상 값을 포함하는 프리미티브를 위한 델타 및 안개 델타를 계산하며, 원근이 제대로 이루어진 텍스처를 위한 UV 계산을 수행합니다.
- DDA 유닛은 이전 단계에서 얻은 프리미티브 정보를 보관하여 프래그먼트 정보를 생성합니다.
- TXM 유닛은 필터링 및 Mip-매핑을 포함하여 모든 텍스처 매핑 동작을 수행합니다.
- PIXOP 블록은 래스터라이저이며, 프레임버퍼에 픽셀을 기록합니다. 또한 2Mb 가 내장된 VRAM 에 대한 인터페이스인 DRAMIF 에 직접 연결됩니다. PIXOP 는 모든 알파 블렌딩 동작을 포함하여 시저링, 스텐실, 알파 테스트 등의 활동들도 취급합니다.

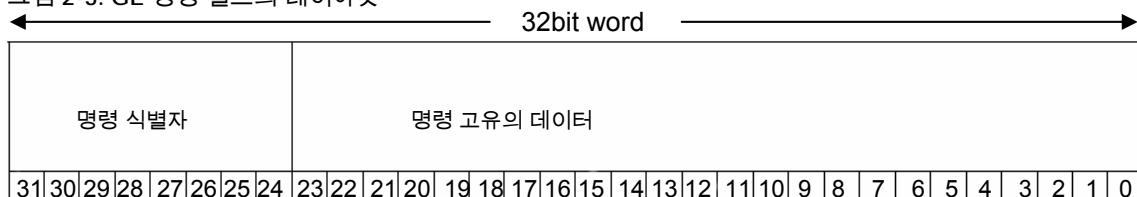
디스플레이 리스트

이 지침서는 디스플레이 리스트 프로그래밍에 관한 상세한 정보는 제공하지 않지만, GE 가 어떻게 작동하며 libGu 와의 관계를 이해하는 데 유용합니다.

디스플레이 리스트란 GE 를 위한 레지스터 명령이 포함된 메모리의 영역입니다. 이는 칩의 렌더링 기능을 제어하는 GE 를 위한 내부 명령 포맷입니다. 이러한 명령은 다양한 비트 세트를 갖춘 32 비트 필드입니다. 그래픽 라이브러리에서 기능을 호출하면, 관련 있는 명령이 디스플레이 리스트에 추가됩니다. 디스플레이 리스트가 GE 로 보내지면, HOSTIF 는 디스플레이 리스트를 해석하여 GE 내부의 관련 섹션으로 렌더링 명령을 중재합니다.

그림 2-3 은 GE 명령의 포맷을 보여줍니다.

그림 2-3: GE 명령 필드의 레이아웃



24 – 31 까지의 상위 8 비트는 명령 식별자로서 고정되며, 0x00 부터 0xFF 까지 범위의 16 진수입니다. 이 비트들은 헤더 파일 *gecmd.h* 에서 정의된, GE 내부의 명령을 식별합니다.

나머지 24 비트는 명령에 필요할 수 있는 데이터를 위해 유보되며, 이 부분의 배정은 명령에 따라 다릅니다.

디스플레이 리스트의 유형

디스플레이 리스트는 두 가지 유형이 있습니다:

- IMMEDIATE 리스트는 작성이 되면서 바로 GE 로 보내지는 것입니다.

- CALL 리스트는 언제든지 작성, 보관 및 보내질 수 있는 것입니다.

CALL 사용의 한 예로는, 디스플레이 리스트의 작성 및 렌더링에 대한 2 중 버퍼를 애플리케이션 렌더링 엔진 내부에서 렌더링하는 것입니다. 이것을 "자연 렌더링"이라고도 부르며, 리스트 하나가 렌더링되면서 또 다른 리스트가 작성되는 것을 말합니다. 따라서 이것은 보다 효율이 높은 렌더링 방법일 수 있습니다.

위와 같은 CALL 리스트는 최적화에 관한 자습서에서 설명하는 것이 더 적합하며, 이 문서에서는 다루지 않습니다.

LibGu

LibGu 는 PSP 를 위한 소프트웨어 개발에 사용하도록 SCE 가 제공하는 그래픽 API 입니다. 구문 스타일의 측면에서 OpenGL 과 유사합니다. OpenGL 은 반드시 알아야 하는 것은 아니지만, PSP 에서 그래픽을 프로그래밍을 보다 빨리 시작하는 데 도움이 될 것입니다. 시간이 지남에 따라 자신만의 디스플레이 리스트 매니저를 사용할 수도 있지만, 이 문서에서는 libGu 를 사용한 그래픽 프로그래밍의 예만을 다룹니다.

libGu 수행 과정을 보여주는 예

libGu 의 요점은 하나의 호출로써 다수의 명령을 설정할 수 있다는 것이며, 이는 개발 과정의 속도를 상당히 증가시킵니다. 그 한 예로서, [4]에 인용된 libGu 기능 `sceGuDrawArray(...)`를 살펴보겠습니다. 기능 자체의 매개변수와 Command Reference 설명서 [16]에 나와 있는 상호 인용 명령을 살펴보면, 이 기능이 디스플레이 리스트에 추가시키는 몇 가지 명령을 알 수 있습니다.

```
void sceGuDrawArray(    int prim,
                        int type,
                        int count,
                        const void *index,
                        const void *p
                        );
```

이 기능은 GE 로 하여금 정점 어레이를 점, 선, 삼각형 등 특정한 프리미티브 유형으로써 그리도록 지시하며, 다음 명령을 디스플레이 리스트에 설정합니다:

- `CMD_PRIM`
이 명령은 그려야 할 프리미티브 유형과 기대하는 정점 숫자를 GE 에게 알립니다. 이 명령은 *prim* 및 *count* 매개변수에 대해 생성될 것입니다.
- `CMD_VTYPE`
이 명령은 *type* 으로 명시되며 GE 가 기대하는 정점 유형을 지정합니다. 이것은 정점 포맷에 관한 많은 정보를 GE 에게 제공하는 데, 텍스처 좌표, 정점 데이터의 비트 깊이 등에 관한 것입니다.
- `CMD_BASE`
이 명령은 색인 모드에 있는 것과 관계 없이, 항상 정점의 주소에 대하여 설정됩니다.

- `CMD_VADR`
이 명령은 정점 데이터의 메모리 주소를 GE 에게 알립니다.
- `CMD_IADR`
이 명령은 색인된 프리미티브가 사용되는 경우 설정되며, 색인 모드에서 그릴 때 색인 어레이의 주소를 GE 에게 제공합니다.

모든 libGu 기능에는 “sceGu”의 접두사가 사용됩니다. 이러한 기능들의 대부분은 디스플레이 리스트 시작 및 종료 블록 내부에서 호출해야 하며, `sceGuStart(...)` 및 `sceGuFinish()`로 불러냅니다. 그러나 이 규칙에 대한 몇 가지 예외가 존재합니다. 이러한 예외는 다음 장에서 자세히 설명됩니다.

디스플레이 리스트가 채워지면, DMA 를 거쳐 그래픽 엔진으로 보내지며 그곳에서 그래픽 엔진의 호스트 인터페이스에 의해 픽업됩니다.

디스플레이 리스트의 데이터

정점과 텍스처 데이터는 절대로 디스플레이 리스트에 복사되지 않습니다. 이 데이터는 항상 해당 데이터의 포인터로서 공급되며, 필요에 따라 그래픽 엔진이 가져올 수 있습니다. 디스플레이 리스트는 명령 그리고 색상 정보, 행렬 등의 작은 값만을 포함합니다.

이것이 OpenGL 등의 다른 API 와 상당히 다른 점입니다. OpenGL 에서는, 즉시 모드를 통하여 `glVertex3f(...)`와 같은 호출을 사용함으로써 기능 내에서 정점이 국부적으로 작성될 수 있습니다.

정점 데이터는 디스플레이 리스트 자체에 복사되지 않으므로 주소적으로 작성하는 것은 가능하지 않습니다. 로컬 정점 데이터는 기능이 복귀되면 범위를 벗어나게 됩니다.

제3장:

간단한 프로그램의 컴파일

이 페이지는 공백으로 둡니다.

서문

이 장은 화면에 삼각형을 그리는, 간단한 프로그램에서 사용되는 코드를 설명하고 libGu 를 사용하는 일부 텍스트를 표시합니다. 이미 다뤄진 내용의 일부를 검토하여, libGu 에 대한 기능 호출의 일반 포맷을 설명하고 그 기능의 일부를 살펴보겠습니다.

이 예에서 사용된 코드는 소스 파일 `hellotriangle.c` 에서 따온 것이며, 이 문서와 함께 제공되는 자습서의 소스 디렉터리에서 찾을 수 있습니다.

이 코드는 에뮬레이터 및 하드웨어 톨 환경 모두를 위해 제공되며, 그 차이점은 부록 B 에서 자세히 설명합니다.

데이터 구조

```
static char    disp_list[0x10000];
```

위의 어레이는 디스플레이 리스트를 위한 정적 보관 영역입니다. 각 `sceGu*` 기능이 호출될 때마다, 이 어레이 내부의 데이터가 각 프레임에 대해 설정됩니다.

```
typedef struct {
    SceUShort16 x, y, z;
} shortVector;
```

```
shortVector gVectors[3];
```

`gVectors` 는 삼각형의 정점이 보관되는 선택사양적으로 정의된 유형의 어레이입니다. 여기서 부호가 없는 짧은 정수를 사용하는 이유는 이 삼각형이 THROUGH 모드로서 그려지기 때문입니다. THROUGH 모드를 사용할 때에는, 정점 위치가 화면 공간 내에 있어야 하며 또한 짧은 정수 유형이어야 합니다.

초기화

이 프로그램에서 대부분의 코드는 그래픽 라이브러리를 초기화하며 메인 루프에서 사용되는 일부의 렌더링 상태를 설정하는, `Init()` 기능 내부에 위치합니다.

```
sceGuInit();
```

이것은 다른 것보다 먼저 호출되어야 하는 첫째 `sceGu*` 기능입니다. 이 기능은 사용 대상의 그래픽 라이브러리를 초기화합니다.

```
sceGuStart(SCEGU_IMMEDIATE, disp_list, sizeof(disp_list));
```

`sceGuStart(...)` 는 디스플레이 리스트에 대한 진입 지점을 정의합니다. 또한 하드웨어 톨을 사용하여 코드를 작성할 때 특히 중요한 몇 가지 디스플레이 리스트 기능들의 일부입니다.

호출되는 `sceGu*(...)` 기능들은 모두 시작 및 종료 디스플레이 리스트 기능인 `sceGuStart(...)`와 `sceGuFinish()` 사이에 위치해야 합니다.

이에 대한 유일한 예외는 다음과 같습니다:

- `sceGuSync(...);`
- `sceGuDisplay(...);`
- `sceGuSwapBuffers();`

이러한 예외가 있는 이유는, `sceGuStart(...)` 호출 전에 연속 명령이 추가되는 디스플레이 리스트의 시작에 대해 `sceGuFinish()`가 포인터를 정의하기 때문입니다.

위의 예외와는 별도로, 모든 `sceGu*(...)` 기능들은 명령을 디스플레이 리스트에 쓴 다음, 디스플레이 리스트 포인터를 충분히 줍니다. 그러므로 작성한 코드가 `sceGuStart(...)`, `sceGuFinish()` 쌍 외부에 있는 `sceGu*(...)` 기능을 호출하는 경우, 그 코드는 관리되지 않는 주소에 명령을 쓰게 됩니다.

에뮬레이터는 위에서 설명한 방식으로 `sceGuStart(...)`, `sceGuFinish()` 쌍 외부에 쓰는 것을 방지하지 않음을 유의하십시오. 만약 하드웨어 툴 사용 시 외부에 쓰는 경우, 예기치 않은 결과를 초래할 수 있습니다.

```
sceGuStart(      int mode,
void *p,
int size
);
```

이 기능은 그 크기나 사용되는 모드와 관계 없이 디스플레이 리스트의 시작을 정의합니다. 이미 언급한 바와 같이, 나중에 사용되는 리스트나 즉시 사용될 리스트를 작성할 수 있습니다. 이 예제에서의 리스트는 즉시 사용될 것이므로, *모드* 매개변수가 `SCEGU_IMMEDIATE`로 설정됩니다.

소스 파일의 위 부분에서 선언된 정적 어레이는 디스플레이 리스트 데이터 포인터로서 보내지며, `sizeof`를 사용하여 디스플레이 리스트 크기를 보냅니다. 이 크기는 `libGu`에 의해 경계의 확인에 사용되지 않으며, 내부에서 링 버퍼로서 취급되지 않음을 유의하십시오. 그러므로 `libGu`는 디스플레이 리스트의 끝이 지나더라도 쓰는 것을 허용합니다. 따라서 예기치 못한 결과를 초래하게 되므로, 반드시 충분한 크기의 디스플레이 리스트를 사용하십시오.

드로 및 디스플레이 버퍼

다음의 두 기능은 PSP 드로 및 디스플레이 버퍼의 설정에 필요한 변수들을 초기화합니다. 기능 안으로 보내지는 인수들은 SCE에 의해 정의되는 매크로입니다. 이것은 색상 깊이, 너비 및 높이 그리고 버퍼 포인터 시작 주소로서 지정되는 VRAM의 메모리 주소와 같은 프레임 버퍼 정보를 설명합니다.

이 기능들은 표준 너비 및 높이(480 * 272)의 32 비트 프레임 버퍼를 설정합니다:

```
sceGuDrawBuffer( SCEGU_PF8888,
SCEGU_VRAM_BP32_0,
SCEGU_VRAM_WIDTH
);

sceGuDispBuffer( SCEGU_SCR_WIDTH,
SCEGU_SCR_HEIGHT,
```



```
SCEGU_VRAM_BP32_1,
SCEGU_VRAM_WIDTH
);
```

libgu.h 에서 있는 SCEGU_VRAM_BP*** 매크로들은 프레임 버퍼 픽셀 포맷에 대한 것입니다. 이 매크로들은 드로, 디스플레이 및 깊이 버퍼를 위한 사전 정의된 VRAM 주소입니다. 하지만 첫째 프로그램에서는 깊이 버퍼를 포함시키지 않아도 됩니다. 이에 대한 내용은 3 차원 그래픽 프로그래밍을 설명하는 다음 장에서 다룹니다.

이 예에서는, 32 비트 프레임 버퍼가 사용됩니다. SCEGU_VRAM_BP32_0 은 무효 포인터에 지정되는 SCEGU_VRAM_TOP (0x00000000)로서 정의됩니다. 이것은 VRAM 의 시작을 위한 메모리-매핑 주소입니다. 이것은 드로 버퍼 시작 주소로서 넘겨집니다.

디스플레이 버퍼에서는, 화면 높이(272), SCEGU_VRAM_WIDTH (512)와 바이트 단위의 픽셀 크기(4)를 곱한 다음, SCEGU_VRAM_TOP 에 더하여 포인터를 계산합니다. 이것은 드로 버퍼 후에, VRAM 에서 다음으로 가용한 영역을 제공합니다. 화면 너비는 480 픽셀이지만, 버퍼들은 VRAM 너비를 512 로 하여 계산됩니다. 그 이유는 픽셀 색상의 깊이가 16 혹은 32 인 것에 따라, 버퍼 너비를 128 혹은 265 바이트의 단위로 지정해야 하기 때문입니다. 두 색상 깊이 모두에 대해 디스플레이 버퍼 너비는 512 로 설정해야 합니다.

뷰포트 초기화

```
sceGuViewport(2048, 2048, SCEGU_SCR_WIDTH, SCEGU_SCR_HEIGHT);
```

sceGuViewPort(int x, int y, int width, int height)는 뷰포트, 즉 클립 좌표 공간에서 화면 공간으로의 뷰포트 변환을 정의합니다. 클립 좌표 시스템의 N/H , 높이, x 및 y 는 화면 좌표에서의 동등한 것으로 됩니다. 예를 들어, 여기서 처음 두 개의 매개변수인 x 및 y 는, 화면 공간에서는 클립 좌표 시스템의 원점이 될 위치를 정의합니다. 그러나 클립 공간에서의 원점은 0,0 이므로, 다음의 질문을 하게 됩니다: 원점이 코드에서 2048, 2048 로 정의되는 이유는 무엇인가?

그 이유는 화면 공간 좌표 시스템의 전체 범위가 x 및 y 모두에 대하여 0 - ~4096 이기 때문입니다. 그러므로 2048, 2048 이 화면 공간의 중심이며, 클립 공간의 객체가 화면 공간으로 올바르게 변환됩니다.

sceGuOffset(int x, int y)는 드로 공간 즉, 0 - 1023 의 xy 범위에 존재하는 화면 공간의 하위 섹션으로의 최종 변환입니다. 다음의 예는 x 좌표에 대한 화면 공간에서 드로 공간으로의 변환을 보여줍니다.

```
Xd = Xs - OFFSETX
```

화면 공간으로의 뷰포트 변환 이후 x 원점이 2048 로서 정의되는 것과 같은 방식으로, 화면 공간의 x 의 중심(2048)이 드로 공간의 x 의 중심(240)으로 어떻게 변환되는지를 알 수 있습니다:

```
Xd = 2048 - OFFSETX
Xd = 2048 - ((4096 - SCEGU_SCR_WIDTH) / 2)
Xd = 2048 - 1808
Xd = 240
```

`sceGuScissor(int x, int y, int width, int height)`는 시저 원점을 정의합니다. 시저 시험은 프래그먼트가 래스터라이징 되는지를 결정하는 최초의 픽셀 시험입니다.

초기화 섹션의 최종화

`sceGuColor(unsigned int col)`는 그리는 색상을 설정합니다. 이것은 정점마다 명백히 정해진 색상이 없는 기하학 도형에 적용할 수 있는, 일정한 정점 색상으로 생각할 수 있습니다. 따라서 커다란 모델에 대한 렌더링을 수행할 때, 정점 색상이 객체에 걸쳐 변경되지 않는다면, 이 기능을 사용할 수 있으며 색상 데이터를 제거할 수 있습니다. 그러나 이 색상 값 보다는 명백한 정점 색상이 우선하게 됩니다.

`sceGuFinish()`는 리스트에 명령을 추가하여, IMMEDIATE 모드에서는 그리기의 끝을 알리며, CALL 모드에서는 리턴 명령을 추가합니다.

`sceGuSync(int mode, int block)` 이 기능은 넘겨지는 매개변수에 따라 그리기를 동기화합니다. 이 경우 GE 가 렌더링을 마칠 때까지 대기할 것을 요청하는 것입니다.

초기화 단계의 마지막 기능은 V-Blank 와 관련이 있습니다. 이것은 이 프로그램에서 사용되는 몇 가지의 기능 가운데, 하드웨어 톨에서 사용되는 방식과 에뮬레이터에서 사용되는 방식이 다른 하나입니다. 이 기능은 단순히 계속하기 전에 다음의 수직 블랭크의 시작을 대기합니다. 에뮬레이터와 하드웨어 톨 사이의 차이점은 부록 B 에서 보다 상세히 설명됩니다.

메인 루프

초기화 이후의 프로그램의 주 기능은 다음 세 기능을 단순히 순환하는 것입니다:

```
StartFrame();
Render();
EndFrame();
```

이미 설명한 바와 같이, 두 가지의 예외를 제외하고는 모든 `sceGu*` 기능들은 `sceGuStart`, `sceGuFinish` 쌍 내에 나타나야 합니다. 이 경우 이와 같은 '포함시키는' 기능들은 `StartFrame` 및 `EndFrame` 에 나타납니다.

`sceGuClear(int mask)`는 VRAM 에서 관련 있는 버퍼를 지웁니다. 이 기능은 넘겨진 마스크에 따라 지워야 할 색상, 깊이, 스텐실 또는 세가지 모두를 중재합니다. 이들은 `libgu.h` 에서 정의됩니다. 여기서는 색상 버퍼만을 지우므로, 예에서는 `mask` 를 `SCEGU_CLEAR_COLOR` 으로 설정합니다.

`sceGuDebugPrint(int x, int y, unsigned int col, const char* str)`는 메시지를 화면에 인쇄합니다. `x` 및 `y` 인수는 텍스트의 시작을 위한 그리기 공간에서의 좌표들입니다. `col` 은 색상이며, 32 비트 필드 그리고 구성요소 당 8 비트로 정의됩니다. `str` 은 화면에 인쇄하려는 문자열이며, 여기서는 "Hello triangle"이 됩니다.

`sceGuDrawArray(int prim, int type, int count, const void *index, const void *p)`는 이전에 GE 명령으로 해체되었던 기능입니다. 이것은 GE 로 하여금 `libGu` 에서 대부분의 프리미티브를 렌더링시키는, 표준 인터페이스입니다. 이 기능은 그려야 할 프리미티브의 유형, 정점 포맷에서 기대하는 데이터, 렌더링 대상의 정점 숫자, 그리고 색인된 프리미티브에 대한 렌더링의 실행 유무를 GE 에게 알려줍니다. 이 기능은 GE 가 인출하도록 포인터를 정점에게 보내며, 이 모드가 사용되는 도중 포인터를 색인으로 보냅니다.

이 예와 같이 디스플레이 리스트가 IMMEDIATE 모드인 경우, GE 는 디스플레이 리스트를 직접 해석하기 시작하며 프리미티브에 대한 렌더링을 실행합니다.

이 경우에는, 정점의 유형이 `short` 이므로 `SCEGU_VERTEX_SHORT`, 혹은 `SCEGU_THROUGH` 로서 `THROUGH` 모드에서 렌더링이 실행되어야 함을 GE 에게 알립니다. 이 매개변수는 `type` 으로서 넘겨집니다. 이 예는 삼각형을 그리므로, `prim` 유형을 `SCEGU_PRIM_TRIANGLES` 로서 명시하고, 그려야 할 정점의 숫자를 셋으로 정의합니다.

정점 포맷과 그래픽 엔진에 의한 해석 방식

[3]에서 설명하는 바와 같이, GE 는 고정 순서형 정점 포맷을 수용합니다. 정점 포맷 내에서 요소들의 순서도 [3]에 문서화 되어 있습니다. 그래픽 엔진이 정점에 대해 최소한으로 정의할 내용은 위치 벡터입니다. 이미 언급된 바와 같이, `sceGuDrawArray(...)` 의 `type` 필드는 그래픽 엔진에게 기대할 내용을 알리는 데 사용됩니다. 이것은 그래픽 엔진이 해석하여 정점 데이터 내부에서 오프셋을 중재할 OR-ed 비트필드입니다. 이는 다양한 정점 요소들의 순서가 고정되어 있다 하더라도, 정점으로 정의한 것은 융통성이 있음을 의미합니다.

마지막으로 설명할 기능 호출은 `sceGuSwapBuffers()` 입니다. 이 호출은 드로와 디스플레이 버퍼를 교환함으로써, 이미 렌더링된 프레임을 화면에 표시합니다.

색인 대 비색인 프리미티브

이 예제와 같이 `sceGuDrawArray(...)` 를 호출할 때, 아직까지 설명하지 않은, 넘겨줄 수 있는 또 하나의 포인터 인수가 있는데 바로 `const void *index` 입니다. 이 매개변수는 색인된 프리미티브에 의한 렌더링에 사용되는 것으로 색인 리스트에 대한 포인터입니다.

색인된 프리미티브는 어레이에서 그에 해당하는 특정 색인에 근거하여 렌더링 대상의 정점을 인출하는 방법입니다. 이 방법은 정점 캐시를 갖춘 하드웨어에서 대역폭을 절약할 수 있지만, GE 에는 정점 캐시가 없으므로, 색인된 프리미티브의 사용을 권장하지 않습니다. 그 이유는 GE 가 '풀' 스타일 칩이므로, 색인 포인터에서 색인을 인출한 다음 각 해당 정점이 정점 포인터로부터 인출하기 때문입니다. 그 결과 버스 활동이 증가하게 되는데, 이러한 활동은 최저 수준으로 유지시키는 것이 최상이며 그렇지 않으면 성능에 영향이 있습니다.

요약

이 장에서는 시작하기 위해 사용할 수 있는 간단한 프로그램을 설명했으며, 구체적으로 그리기에 필수적인 렌더링 문맥의 초기화 방식, 간단한 렌더링 루프를 설정하여 텍스트를 표시하고 삼각형 그리는 방법 그리고 이러한 작업에 사용된 모든 libGu 명령을 다루었습니다.

이 장에서 사용된 예를 통하여 2 차원 그래픽 프로그래밍을 보여주었습니다. 다음 장에서는 3 차원 그래픽 프로그래밍을 설명하고 또 하나의 라이브러리인 libGum 을 소개합니다. 다음에 나오는 예는 이 장의 예제 코드를 바탕으로 구축하여 조명이 있는 간단한 장면을 작성합니다. 또한 사용자가 예의 프로그램을 사용하여 PSP 와 대화할 수 있도록 하는 컨트롤러 입력도 포함됩니다.

제4장:

간단한 3 차원 장면

이 페이지는 공백으로 둡니다.

서문

이 장에서는 이 문서를 동반하는 3 차원 장면 예를 작성하는 코드를 설명하며, 그 코드는 다음 파일에 있습니다: 3dscene.c.

이 예제는 바닥 면, 2 개의 도너스 및 램프로 구성된 간단한 3 차원 장면을 생성합니다. 이 장면의 객체들은 GE 하드웨어 패치 텔레스테이션을 사용하여 렌더링됩니다. 이 예제는 3 차원 효과를 생성하므로, libGu 의 설정 호출에 대한 추가 사항도 포함이 됩니다.

또한 이 예제는 서문에서 언급한 libGum 이라 부르는 추가의 라이브러리도 사용하며, 컨트롤러 라이브러리를 사용하여 컨트롤러 입력의 형태로서 사용자 상호활동을 추가합니다.

이 장의 목적은 위의 예에서 사용되는 기능들의 기반이 되는 이론의 일부를 설명하는 것입니다. 예를 들어, 일부의 PSP 조명 기능에 대한 설명에는 기능과 하드웨어 사이의 관계에 대한 이해를 제공하기 위해 GE 가 고수하는 조명 모델의 간단한 설명이 동반됩니다. 이 정보를 통하여 독자들이 일부의 PSP 그래픽 기능들에 대한 개요와 그 기능의 구현의 쉽다는 점을 인식하기를 바랍니다.

깊이 버퍼

PSP GE 에는 16 비트로서 고정된 하드웨어 깊이 버퍼 또는 Z-버퍼가 있습니다. 사용자는 깊이 버퍼가 차지하는 VRAM 의 영역을 명백히 정의해야 하며, 이는 자동으로 실행되지 않습니다. 이것은 드로 및 디스플레이 버퍼가 차지하는 VRAM 의 영역의 정의 방법과 비슷하게 할 수 있으며,

sceGuDepthBuffer(void *zbp, int zbw)를 사용하며 여기서 *zbp* 는 SCEGU_VRAM_BP32_2 입니다. 이것은 32 비트로 설정된 경우 드로 및 디스플레이 버퍼 다음, 사용가능한 메모리의 다음 영역의 시작으로 *libgu.h* 에 정의됩니다.

깊이 버퍼의 너비는 16-비트이므로, 128 의 배수로서 정의되어야 하며, 따라서 *zbw* 는 매크로 SCEGU_VRAM_WIDTH 을 사용하여 512 로서 정의됩니다.

Z-버퍼는 픽셀을 시험하여 이미 존재하는 것들의 뒤에 있으면 거부하는, 주로 숨겨진 표면 제거에 사용됩니다. Z-버퍼는 다양한 시험 하에서 픽셀의 채택 혹은 거부를 선택할 수 있습니다. 이 예에서는 깊이 값이, *sceGuDepthFunc* (SCEGU_GEQUAL)로써 설정된 깊이 버퍼의 값 이상이면 픽셀을 채택하게 됩니다.

다음으로 필요한 초기화는 깊이 범위의 설정입니다. 여기서 설정되는 값들은 뷰포트 변환에 의해 매핑되는 깊이 범위가 됩니다. 이 범위 외부의 픽셀은 모두 래스터라이징이 되지 못합니다. Z-버퍼는 각 픽셀의 Z 값을 비교한 결과에 근거하여 프래그먼트를 입력시키거나 거부합니다. 깊이 범위를 매우 작게 설정하면, 픽셀 깊이가 비교적 멀리 떨어지기 때문에, 픽셀의 래스터라이징 시 Z-시험에 의한 인위적 현상이 발생합니다. 이 예에서는 *sceGuDepthRange*(int nearz, int farz)를 사용하여, 가까운 z 값의 경우 깊이 범위를 65535 의 기본 설정값으로, 먼 z 값의 경우 0 으로 각각 설정합니다.

`sceGuClearDepth(int depth)`는 어떤 값을 지울 때 GE가 Z-버퍼 안으로 설정하는 그 값을 설정합니다.

마지막으로 깊이 시험은 `sceGuEnable(SCEGU_DEPTH_TEST)`로써 사용가능케 합니다.

3 차원 좌표 시스템 및 libGum

[3]에서 지적하는 바와 같이, GE는 파이프라인에서 여섯 개의 좌표 시스템을 사용하여 프리미티브에 대한 렌더링을 화면에 실행합니다. 이전 장에서는 예를 통하여 THROUGH 모드 프리미티브에 대한 렌더링이 어떻게 이루어지는가를 설명했으며, 이는 드로 공간에 명시된 정점들이 요구됩니다.

이 장의 3 차원 장면 예제는 3 차원 변환 세트를 모두 사용하므로, 결과적으로 6 개 좌표 시스템 모두를 사용하여 화면에 렌더링 된 3 차원 프리미티브를 변환시킵니다. PSP 프로그래밍 시, 변환 파이프라인을 통하여 수동으로 프리미티브를 변환시킬 필요가 없습니다. GE는 하드웨어에서 이러한 기능 모두를 취급하며, 이는 libGu 및 libGum을 통하여 노출됩니다.

사용가능한 행렬은 4 개가 있으며, 원근, 보기, 월드 및 텍스처 행렬입니다.

그래픽 엔진 사용자 설명서에서 설명한 바와 같이, 위의 네 행렬은 파이프라인을 형성하여 3 차원 공간의 객체를 변환시켜 2 차원 화면에 대해 렌더링을 실행하며, 그 작동 방식은 다음과 같습니다:

- **월드 행렬**은 지역 공간에서 월드 공간으로 객체를 변환시킵니다.
- **보기 행렬**은 월드 공간에서 보기 공간으로 객체를 변환시킵니다.
- **원근 행렬**은 아이 공간 객체를 클립 공간으로 변환시킵니다.
- **텍스처 행렬**은 투영된 텍스처 등의 효과를 위해 사용되지만, 이 지침서에서는 다루지 않습니다.

LibGum

LibGum [4]은 근본적으로 행렬 매니저 그리고 행렬 변환 및 스택과 같은 유용한 기능의 일부를 추가시키는 수학 라이브러리입니다. OpenGL에 익숙한 사람은, OpenGL과 LibGum 구문의 유사성을 알 수 있을 것입니다. 행렬 변환 기능은 다음과 같이 요약할 수 있습니다:

- `sceGumTranslate(...)` 번역 연산에 적용된다
- `sceGumScale(...)` 크기 조정 연산에 적용된다
- `sceGumRotate*(...)` x, y, 및/또는 z 축에서의 회전에 적용된다
- `sceGumLoadIdentity(...)` 행렬을 항등으로 리셋한다.

이러한 기능들은 기존 행렬에서 연속 변환으로 작동하므로, 기능이 호출되면 다음 변환이 기존 행렬 안으로 곱해집니다. 이러한 행렬들은 열 순서이므로, 그 연산의 나중 곱하기를 의미하며 변환의 순서를 생각할 때 뒤를 앞으로 간주해야 합니다.

스택 기능성은 변환을 유지하여 나중에 복원하는 것이 필요할 때 유용할 수 있습니다. 일부 객체들은 기반 변환을 공유할 수 있으므로, 추가의 개별적 변환이 요구됩니다. 예를 들어, 기반 변환은 스택으로 밀 수 있으며, 각 객체는 고유한 변환을 추가하여 다음 객체를 위해 기반 변환을 복원할 수 있습니다. 그리하여 각 객체에 대한 기반 변환의 처리를 절약할 수 있습니다. 이것에 대한 간단한 예를 3 차원 장면 예를 통해 나중에 설명합니다.

PSP 상에서 행렬 스택은 4x4 행렬의 어레이를 먼저 선언함으로써 초기화합니다. 여기서 선언할 수 있는 숫자에 대한 이론적 제한은 없습니다. 스택을 위해 행렬을 설정하는 기능은 다음과 같습니다:

```
sceGumSetMatrixStack(  ScePspFMatrix *m,
                      int proj,
                      int view,
                      int world,
                      int tex
                      );
```

대개 원근 행렬은 애플리케이션 당 한 번 설정되며, 보기 행렬은 프레임 당 한 번 설정될 확률이 큼니다. 이 예제는 애플리케이션 동안 원근 행렬을 수정하지 않기 때문에, 초기화 단계에서 한 번 설정됩니다.

`sceGumMatrixMode(int mode)` 기존 행렬 스택을 설정합니다. 연속 libGum 행렬의 모든 기능들은 스택 상에서 최고 수준의 행렬에 대해 변환을 적용하며, 이것은 기존의 작업 행렬이 됩니다.

이 예제는 기존 행렬을 투영 행렬로 설정한 다음, `sceGumPerspective(float fovy, float aspect, float near, float far)`를 호출합니다. 이 기능은 매개변수가 있는 원근 행렬을 작성하여, 이를 기존의 작동 행렬로 곱합니다.

GE 에서 단순(flat) 및 고라드(Gouraud) (삼각형에 걸쳐 보간된 정점 기준) 셰이딩이 모두 가능합니다. 평면 혹은 매끄러운(smooth) 셰이딩의 사용의 지정 방법은 `sceGuShadeModel(int model)` 기능을 통하여, `model` 매개변수로서 `SCEGU_FLAT`, 혹은 `SCEGU_SMOOTH` 를 지정하면 됩니다.

PSP 에서는 조명이 하드웨어에서 수행됩니다. libGu 를 통하여 노출되는 빛의 숫자는 최대 4 개의 하드웨어 빛이 있는 데, 다음 절에서는 이러한 빛의 설정 방법을 다루고 그 특성과 물질에 관한 설명을 하겠습니다.

조명

모든 셰이딩 계산에는 단순이거나 고라드이든지 조명 처리식의 일부로서 법선이 필요합니다. 평면의 경우에는 표면 법선이 필요하고, 고라드 셰이딩의 경우에는 정점 기준(per-vertex) 법선이 필요합니다. 이 예제에서는 기하학적 도형의 렌더링을 위해 패치를 사용합니다. 이 기능을 사용함으로써, GE 는 모든 법선을 자동으로 생성합니다. 조명이 per-vertex 기반으로 이루어지기 때문에, 적절한 결과를 얻으려면 비교적 높은 수준의 텔레스테이션 메시가 필요합니다.

GE 조명 모델은 다음 4 가지 매개변수를 기반으로 조명을 계산합니다: 주변광(ambient), 난반사(diffuse), 정반사(specular), 방사성(emissive).

- 주변광에 의한 기여는 기본적 환경 조명의 기여로서 간주됩니다. 이것은 매우 퍼져 있으며 장면으로부터 반사된 빛으로 다방향적이므로 평평하게 보입니다.
- 난반사 조명은 그 기여도가 특정 광원으로 계산된다는 점에서 방향성입니다. 난반사 조명은 매트로 나타나며, 시야 각도에 관계 없이 객체 전체에 걸쳐 동일하게 나타납니다. 난반사 조명은 *시야에 독립적*으로 정의됩니다.
- 정반사 조명은 난반사 조명과 같이 방향성이며, 객체에서 특정 방향으로 반사되는 것처럼 나타납니다. 정반사 조명은 반짝이는 표면의 모습을 제공하는 한 방법입니다. 정반사 조명은 빛의 위치와 표면 법선 및 카메라 뷰포인트에 의해 계산됩니다. 이 조명은 객체나 카메라의 이동에 따라 표면에 걸쳐 이동하는 것처럼 보이며, “의존적(view-dependent)”입니다.
- 처리식의 *방사성* 요인은 내부에 광원이 있는 것처럼 객체 자체에서 나오는 것으로 생각하면 됩니다. 이 조명은 표면 조명의 전체적 기여에 추가되지만, 다른 표면에 제공되는 빛을 실제로 방사하지는 않습니다.

위의 모든 요인들은 표면의 재질적 특성과 합쳐지며, 이에 관한 설명은 나중에 제공됩니다.

GE 는 하드웨어에서 최고 4 개의 정점 빛을 계산하며, 이 빛에는 방향 광원, 점 광원 및 집중 조명 광원이 있습니다. 모든 조명 기능성은 libGu 에 의해 노출됩니다. 그러므로 조명 효과의 부착은 매우 쉬우며, 이는 리얼리즘을 강화시키고 렌더링된 장면에 대해 깊이를 더해줍니다.

이 예제에서는 장면의 조사를 위하여 하나의 난반사 및 정반사 광원점을 설정합니다. 빛 색상은 `sceGuLightColor(int n, int type, unsigned int col)`에 의해 설정됩니다. 여기서 *n* 은 0 – 3 의 값을 가지는 빛 식별자 번호이고, *type* 은 조명 처리식의 성분 즉, 주변광, 난반사 또는 정반사를 말하며, *col* 은 각 성분에 대해 정하는 색상을 의미합니다.

`sceGuSpecular(float power)`는 빛 정반사 지수를 설정합니다. 이것은 렌더링 대상 표면의 물질적 특성에 대한 광택 값을 설정하는 것과 동등합니다. 이 값이 높을수록 표면이 더 반짝이게 나타나며, 정반사 광이 더 작고 집중적으로 나타납니다. 이 샘플에서는 모든 물질이 동일한 광택 값을 갖는 것으로 가정합니다.

난반사 및 정반사 조명의 두 가지 요구조건은 빛 위치 및 시야 위치입니다. 시야 위치는 per-frame 기반으로 설정되므로 이 문서 뒷부분에서 다루어집니다. 이 예제에서는 빛을 이동할 필요가 없으므로, 그 위치는 `sceGuLight(int n, int type, int comp, const scePSpsFVector3 *vec)` 기능을 사용하여 초기화 단계에서 설정됩니다. 이 예제에서는 빛의 *type* 을 `SCEGU_LIGHT_POINT` 로써 정의합니다. 제공되는 조명의 유형은 *comp* 매개변수를 사용하여 `SCEGU_DIFFUSE_AND_SPECULAR` 에 대해 설정됩니다. 이 기능으로 보내지는 위치 벡터는 월드 공간에서 정의되어야만 조명이 제대로 작동할 수 있으며, 이것은 **vec* 매개변수로 보내집니다.

마지막으로, 빛 ‘0’은 `sceGuEnable (SCEGU_LIGHT0)`로써 사용가능하게 됩니다. 조명 자체는 `sceGuEnable(SCEGU_LIGHTING)`을 호출함으로써 모든 연차적 렌더링에 대해 사용가능하게 됩니다. 이것으로 이 예제의 초기화를 마칩니다.

패치

GE 의 강력한 기능인 하드웨어 패치 텔레스테이션이 이 예제에서 사용됩니다. 다음 절에서는 해당되는 Bezier(베이지어) 및 Spline(스플라인) 곡면과, 이 곡면의 사용이 제공하는 이점을 간단히 설명합니다. 그러나 이것도 스플라인과 그 사용에 대한 수학적 자습서를 제공하기 위한 것은 아닙니다. 이와 관련된 수학은 이 문서의 끝 부분에 있는 참조 문헌의 [5]에 나와 있으며, [3]에서도 패치 표면 계산을 위해 GE 가 사용하는 방법에 관한 내용이 많이 취급됩니다.

GE 가 지원하는 베이지어 및 스플라인 곡면은 매개변수적 표면의 유형입니다. 매개변수적 표면은 일반적으로 일련의 제어 점들로 정의되며, 이 점들이 표면 계산의 기본을 형성합니다. 그러나 표면 계산에 사용되는 방정식에 따라서 표면 자체가 이러한 제어 점을 반드시 통과할 필요는 없습니다.

표면 생성에 사용되는 제어 점은, 생성되는 표면에 대한 외부의 낮은 해상도로써 표시가 되는 크레이들 혹은 메시로서 생각할 수 있습니다. GE 는 한 세트의 제어 점을 취하여 이들을 보관함으로써, 렌더링 대상의 삼각형 표면 표시를 생성합니다. 이렇게 함으로써 각 삼각형의 정점 데이터가 GE 에게 주어질 필요가 없으며, 이는 버스를 거쳐 이전되는 데이터의 양을 상당히 감소시킵니다. GE 는 패치 조명에 요구되는 법선을 계산하므로, 정점 포맷에 법선을 포함시키는 것이 요구되지 않습니다. 삼각형의 텍스처 좌표는 제어 점마다 제공되는 텍스처 좌표를 거쳐 표면 전체에 대해 보관됩니다.

이 예제는 제어 점의 일부 위치 벡터를 초기화합니다. 평면, 구 및 원환면을 묘사하는 위치들이 생성됩니다.

하드웨어 코드에서는 스플라인 초기화 기능의 끝에서 `sceKerneDcacheWritebackAll()`에 대한 호출이 이루어집니다. 이 기능은 Data Cache (D-Cache)의 모든 것이 주 RAM 에 다시 쓰여진 다음 계속되도록 보장합니다. 이러한 기능은 때에 따라 필요하며, 일반적으로 동적 데이터의 경우 더욱 그러합니다. 그 이유는 `sceGuDrawArray(...)` 등의 libGu 기능이 주 CPU 와 비동기 관계에 있는 GE 로의 DMA 이전을 초기화하기 때문입니다. GE 가 명령을 받는 즉시 정점 데이터의 인출을 시작하게 되면, 아직 준비가 안 된 정점의 그리기를 시작하게 되는 데, 그 이유는 초기화된 데이터가 아직 D-Cache 에 머무를 수 있기 때문입니다. 이러한 경우 위치가 잘못된 정점과 같은 이상한 현상을 초래할 수 있습니다. 심지어 화면에 나타나지 않을 수도 있습니다.

모든 데이터를 다시 쓰지 않아도 되도록 D-Cache 를 관리할 수 있습니다. 그러나 D-Cache 에 대한 추적은 복잡한 절차이기 때문에 여기서는 다루지 않겠습니다. 캐시 기능에 관한 자세한 내용은 [17]을 참조하십시오.

컨트롤러 취급

이 절에서는 컨트롤러 입력 데이터를 접속하는 기능에 관해 자세히 설명합니다. 하드웨어 톨과 에뮬레이터 사이의 중요한 차이점은 같은 API 를 공유하지 않는 것입니다. 이 예제와 다음 예제에서는 사용자 입력의 취급하는 모든 코드가 `ReadPad()` 기능에 포함되며, `ReadPad()`에 포함된 플랫폼 의존 코드를 다음에 설명합니다.

에뮬레이터의 컨트롤러 입력

에뮬레이터의 컨트롤러를 위한 코드 취급 방식은 매우 분명하며, 에뮬레이터 라이브러리의 일부입니다. 그러므로 특정 헤더를 포함시킬 필요가 없습니다. 이 예제에서 사용하는 기능은 다음과 같습니다:

```
unsigned int sceEmuAnalogRead ( unsigned char* buffer );
```

위의 기능은 버튼 상태 및 아날로그 스틱 이동에 관한 정보를 수집합니다. 그리고 어떤 버튼이 눌러졌는지에 대한 정보를 포함하는 비트필드로서 부호 없는 정수를 보냅니다. `unsigned char* buffer` 매개변수는 4 개의 부호 없는 8 비트 정수들로 이루어진 어레이에 대한 포인터입니다. `sceEmuAnalogRead` 는 아날로그 스틱에 관한 정보로써 어레이를 채웁니다. 아날로그 스틱은 2 개까지 지원됩니다. 이 예제에서는 왼손 아날로그 스틱을 입력으로 사용하는 데, 이는 `sceEmuAnalogRead` 가 리턴한 다음, `x` 및 `y` 어레이의 마지막 두 요소에 보관되는 데이터를 말합니다.

하드웨어의 컨트롤러 입력

하드웨어를 위한 컨트롤러 취급 코드는 별개의 API 이므로, 반드시 다른 헤더를 포함시켜야 합니다:

```
#include <ctrlsvc.h>
```

사용되는 디지털 버튼만을 구별하도록 혹은 아날로그 스틱도 서비스에 포함되도록 컨트롤러 라이브러리를 설정할 수 있습니다. 그 이유는 애플리케이션에서 아날로그 스틱이 필요 없는 경우, 이에 대한 처리를 피하기 위해서입니다. 이 예제에서는 다음에 대한 호출로써 컨트롤러 서비스 라이브러리를 초기화합니다:

```
sceCtrlSetSamplingMode( unsigned int uiMode );
```

여기서 예제는 아날로그 및 디지털 제어 기능 모두의 사용을 보여주므로, 기능에 보내지는 매개변수는 `SCE_CTRL_DIGITALANALOG` 입니다. 하드웨어 소스 코드의 `ReadPad()` 내에서, 한 기능이 컨트롤러 상태에 관해 필요한 정보를 수집합니다:

```
sceCtrlReadBufferPositive( SceCtrlData *pData,
int nBufs
);
```

`SceCtrlData` 데이터 구조는 컨트롤러 입력 매개변수들을 결정하는 데 필요한 정보를 포함하도록 설계됩니다:

```
typedef struct SceCtrlData {
    unsigned int TimeStamp;
    unsigned int Buttons;
    unsigned char Lx;
    unsigned char Ly;
    unsigned char Rsrv[6];
} SceCtrlData;
```

`sceCtrlReadBufferPositive(...)`에 대한 호출은 위의 데이터 구조를 채워줍니다. 그 다음은, 버튼 정보의 추출을 위하여, 관련 비트 마스크에 대한 결과에서 AND 를 사용합니다.

아날로그 스틱의 데이터는 x 및 y 축을 위한 구조의 L_x 및 L_y 요소에서 각각 8 비트 이내로 인코딩됩니다. 이 데이터는 양 및 음 정보를 모두 포함하며, 측당 0-127 은 음값, 128-255 는 양값이 되도록 unsigned char 로 인코딩됩니다. 이 예제에서는 이 데이터를 -1.0 에서 +1.0 의 범위로 변환시키는 데, 이는 컨트롤러 입력을 판독하는 표준 방법입니다.

데드 존(dead-zone)도 설정되는 데, 여기서는 중심으로부터 30 퍼센트의 이동에 해당되며, 이는 아날로그 컨트롤러가 대개 어느 정도의 부정확성을 나타내기 때문입니다. 이러한 부정확성은 컨트롤러를 놓은 다음에도 어떤 방향으로 고정된 것처럼 보이는 '부유 이동'을 초래할 수 있습니다.

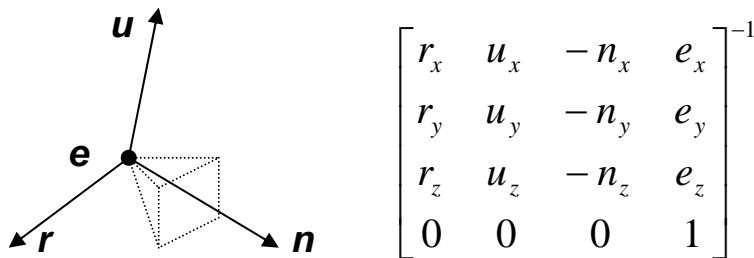
이러한 경우 사용자의 작용은 카메라가 사용하는 회전 각도를 증분시켜 장면에 대해 회전시킬 뿐입니다. 이에 대한 내용은 렌더링 루프 내에서 시야 변환을 취급하는 다음 절에서 설명합니다.

시야 변환: 간단한 카메라

각 프레임에 대해 먼저 해야 할 것은 시야 변환의 설정입니다. 이미 설명한 바와 같이, 정반사 조명은 시야 위치의 설정을 요구합니다. 이러한 설정은 시야 행렬 내에서 인코딩됩니다. 이 예제에서는 시야가 이동되므로, 시야를 매 프레임마다 설정해야 합니다. 시야 행렬은 카메라를 위한 정보를 나타내는 4 개의 벡터로 구성됩니다. 첫 3 개의 벡터는 카메라 '기본'의 것으로서, 단위 길이를 가진 3 개의 직교 벡터로 이루어진 집합으로 정의됩니다. 이것은 '직교 기저'로도 알려져 있습니다. 이 기반이 행렬의 회전 부분을 형성합니다. 네 번째 벡터는 카메라의 월드 좌표에서의 위치입니다.

시야 행렬을 정의하는 간단한 방법의 하나는 회전 및 평행 이동으로만 구성된 월드 행렬을 역변환시키는 것입니다. 여기에 크기 조정 연산을 포함시키게 되면, 행렬이 더 이상 정규직교가 아니므로 그 연산이 실패하게 됩니다.

그림 4-1: 카메라 및 시야 행렬의 관계



첫째, `sceGumMatrixMode(...)`를 호출하여 `SCEGU_MATRIX_VIEW`를 기존의 작동 행렬로서 설정한 다음 항등 행렬로 설정합니다. 다음, `sceGumTranslate(...)` 및 `sceGumRotate(...)`를 각각 사용하여 평행 이동 및 회전 기능을 장면으로 멀리 그리고 장면 주위에 필요한 수량에 대해 적용합니다. 보내지는 회전 값들은 컨트롤러를 통한 사용자 입력에 의해 수정되는 변수들입니다.

이 행렬을 `sceGumStoreMatrix(ScePspMatrix4 *m)`로써 검색하여 역변환합니다. `libGu`나 `libGum`에는 고유의 행렬 역변환 기능이 없으므로 여기서 제공됩니다. 행렬 역변환 기능은 4 * 4 행렬에

있는 16 개 요소 가운데 12 개만을 역변환시키면 되며, 그 이유는 GE 에서 시야 행렬에 사용되는 내부 포맷이 4x3 이기 때문입니다.

이 행렬이 역변환을 거치게 되면, 원점으로부터 일정한 거리의 위치에서 그를 중심으로 회전하며 원점을 보는 카메라를 나타냅니다. 다음, `sceGumLoadMatrix(ScePspMatrix4 *m)` 는 이를 시야 스택에서 최고 수준의 행렬로서 다시 로딩합니다. 이것이 시야 행렬로서의 카메라를 정의하는 간단한 방법입니다.

이 프레임의 나머지 부분에서는, 모델 행렬과의 연산이 사용됩니다. 이것이 기존 행렬로서 설정되며 또한 항등 행렬에 대해 설정됩니다. 이 예제의 렌더링 루프 끝 부분에서 빛 표시를 그릴 목적으로 이 행렬을 사용불능 시키면, 조명이 사용가능케 됩니다.

이 장면에서 렌더링되는 첫째 객체는 바닥 면입니다.

물질

물질 색상은 조명이 사용가능케 될 때 정점의 최종 색상을 결정하는 전체 조명 처리식의 다른 부분을 형성합니다. 조명에 대한 완전한 처리식은 그래픽 엔진 사용자 설명서 [3] (p.56)에 나와있습니다. 이 정보를 여기서 다시 반복하지 않으나, 다만 물질 색상들은 각 수준마다 처리식으로 곱해진다는 것 즉, 난반사 조명 성분은 물질의 난반사 성분과 곱해지고 정반사 조명 성분은 물질의 정반사 성분과 곱해지는 식으로 된다는 점만을 설명합니다. 일반적으로, 물질은 렌더링 대상의 객체의 색상으로 생각할 수 있습니다. 그러나 이 장에서 나중에 설명하는 바와 같이, 물질 매개변수를 사용하여 미묘한 효과를 얻을 수 있습니다.

방정식 4-1: 정점 색상에 대한 빛 및 물질의 기여

$$\begin{aligned}
 \text{Colour} &= \text{Emissive}_{mat} + \text{Ambient} + \text{Diffuse} + \text{Specular} \\
 &= \text{Emissive}_{mat} \\
 &\quad + \text{Ambient} \times \text{Ambient}_{mat} \\
 &\quad + \text{Ambient}_{light} \times \text{Ambient}_{mat} \\
 &\quad + \text{Diffuse}_{light} \times \text{Diffuse}_{mat} \times \left(\frac{\max(L \cdot N, 0)}{|L||N|} \right) \\
 &\quad + \text{Specular}_{light} \times \text{Specular}_{mat} \times \left(\frac{\max(H \cdot N, 0)}{|H||N|} \right)
 \end{aligned}$$

식 4-1 은 정점 색상의 빛과 물질 요소로 이루어진 완벽한 방정식입니다. 여기서 이 식은 하나의 빛으로 정의됩니다. 복수의 빛인 경우, 이 식은 각 빛에 대한 빛과 물질의 기여를 합한 것으로 됩니다.

`sceGuMaterial(int type, unsigned int col)` 기능은 따라 나오는 정점들의 물질 색상을 설정하는 데 사용되며, 여기서 `type` 은 `SCEGU_AMBIENT`, `SCEGU_DIFFUSE` 혹은 `SCEGU_SPECULAR` 이고 `col` 은 32 비트 정수 색상 필드입니다.

패치

바닥 면은 GE 의 하드웨어 스플라인 곡면 기능성에 의해 렌더링이 이루어집니다. 렌더링 기능이 libGum 에 있는 올바른 변환 행렬을 사용하도록 하기 위해, libGum 에는 libGu 의 `sceGuDraw*()` 기능들과 대응되는 것이 있습니다:

```
sceGumDrawSpline( int type,
                  int ucount,
                  int vcount,
                  int uflag,
                  int vflag,
                  const void* index,
                  const void *p
                );
```

이 기능의 일부 매개변수는 `sceGuDrawArray(...)`와 공통되며, `type`, `*index` 및 `*p` 인수가 그것입니다. 매개변수적 표면은 일반적으로 2 차원 기호로 정의됩니다. 2 차원 텍스처 좌표에 대한 규약과 마찬가지로, `u` 및 `v` 의 범위로서 정의됩니다. `ucount` 및 `vcount` 는 각 방향의 제어 점들의 숫자에 해당됩니다. 이 예제에서는 $10 * 10$ 벡터들로 이루어진 어레이를 제어 점의 그리드로서 설정했으며, 따라서 `ucount` 및 `vcount` 는 각각 10 입니다.

`uflag` 및 `vflag` 매개변수는 패치의 구축 방식과 관련이 있습니다. 이 값들은 B-Spline 곡면의 계산에서 중요한, 매듭과 관련이 있습니다. 매듭은 `u` 및 `v` 의 시작과 끝에서 열림 또는 닫힘으로 설정할 수 있습니다. 만약 매듭이 열리도록 설정하면 패치는 베이지어 패치와 같이 거동합니다. B-Spline 곡면은 점유하는 면적은 작지만 제어 점 당 더 많은 삼각형을 생성하게 되는 데, 이는 생성되는 곡면의 밀도가 더 높다는 것을 의미합니다. 넓은 면적의 B-Spline 곡면을 생성할 때 텔레스테이션을 적절히 관리해야 하며 그렇지 않으면 성능이 저하될 수 있으므로, 유의해야 합니다.

패치 렌더링 하드웨어는 `u` 또는 `v` 방향으로 1 에서 64 사이의 하위분할 수준을 수용합니다. 높은 숫자는 매끄러운 기하학적 도형을 생성하므로 렌더링이 더 오래 걸립니다. 따라서 질과 속도의 균형을 스스로 결정해야 합니다. 이 예제에서는 `sceGuPatchDivide(int udiv, int vdiv)`를 사용하여 바닥면과 원환면에 대해 각각의 하위분할 수준을 설정합니다.

베이지어 대 B-스플라인 패치의 수학 기호, 장점, 단점 및 특이사항의 상세한 비교는 그래픽 엔진 사용자 설명서 [3] 및 참조 문헌 [5]에 나와있습니다.

행렬 변환

다음의 월드 변환의 조합은 이 장에서 이미 언급된 행렬 스택을 간단히 보여줍니다.

원환면 설정을 위한 제어 점들은 모델 공간에 있으며 원점에 대해 위치합니다. 이 예제에서는 두 가지 모두 기본 평행 이동 행렬을 공유합니다. 이것은 `sceGumTranslate(&offset)` 호출로써 설정됩니다. 첫째로 그려지는 원환면은 모델 공간에 있는 동안, 회전을 위해 나중에 곱해지는 회전을 제공합니다. 이 회전으로 인해 원환면이 수평으로 위치하게 나타납니다. 이러한 회전은

원환면 #2 에 의해 사용되는 기본 변환에 방해가 되므로, 그 행렬을 먼저 스택에 보관하여 보호합니다. 다음, 다음을 호출하여 원환면에 대한 렌더링이 이루어집니다:

```
sceGumDrawBezier( int type,
int ucount,
int vcount,
const void *index
const void *p
);
```

이 기능은 패치의 열림이나 닫힘을 지정할 필요가 없는 것을 제외하고는 구문상으로 스플라인의 것과 동일합니다. 다음 sceGuPopMatrix()의 호출은 저장된 행렬을 스택의 맨 위로 복원시킵니다.

둘째 행렬은 첫째 행렬의 고리 주위의 궤도를 회전합니다. 이에 대한 변환은 다른 것 주위에 느슨히 끼워지도록 원점으로부터 일정 거리를 평행 이동하는 것입니다. 이렇게 함으로써 원점으로부터 오프셋이 이루어지며, 회전을 적용하면 변환이 완료됩니다. 이러한 변환은 나중에 곱해지므로 변환을 역 순서로 선언해야 함에 유의하십시오. 다음 원환면에 대한 렌더링을 수행합니다.

이미 언급한 바와 같이, 객체 물질은 객체마다 나타나는 방식에 영향을 주도록 사용할 수 있으며, 정교한 사용으로 좋은 결과를 얻을 수 있습니다. 원환면의 경우, 정반사 값은 진한 회색으로 선언되었습니다. 이 값이 빛의 정반사 값과 곱해지므로, 최종 정반사 성분의 강도는 감소되어 보기 좋은 밀납 표면의 효과가 생성됩니다.

이 장면의 최종 객체는 빛의 표시입니다. 복잡하지 않도록 간단한 흰색 공이 사용됩니다. 이를 위해서, 조명을 사용불능 시킨 다음 그리기 색상을 흰색으로 설정합니다. 이제 GE 는 조명을 통한 정점 색상의 계산을 중단하게 되며, 일정한 그리기 색상을 사용합니다. 모델 행렬이 항등 행렬로 지워지며, 빛 위치가 평행 이동으로서 적용된 다음 렌더링이 실행됩니다. 이것이 3 차원 장면 예제의 끝입니다.

요약

이 장에서는 예제 코드에 3 차원을 개입시키는 과정을 설명했습니다. 깊이 버퍼와 시야 행렬을 지정하고, 다른 라이브러리인 libGum 도 소개했습니다. PSP 하드웨어의 기능인 조명 및 패치 텔레스테이션을 소개했으며, 예제에서는 단 몇 줄의 코드로 간단한 3 차원 장면을 표시했습니다. 다음 장에서는 VFPU 에 대해 소개하며, VFPU 어셈블리 코드 예제를 사용하여 기존 예제를 향상시키는 방법을 설명합니다.

제5장:

VFPU (벡터 부동 소수점 유닛)

이 페이지는 공백으로 둡니다.

서문

이전 장에서 사용된 소스 코드는 몇 가지 예외를 제외하고는 하드웨어 툴과 에뮬레이터 모두와 호환되었습니다. 이것은 본 문서 작성을 위한 의도적인 결정이었습니다. 그러나 VFPU 는 에뮬레이션의 대상이 아니므로, 이 장에서 소개되는 소스 코드는 에뮬레이터에서 지원되지 않습니다.

이 장에서는 VFPU 를 소개하고 시스템의 나머지 부분과의 관계를 설명합니다. 또한 프로세서의 일부 애플리케이션과 애플리케이션에서의 초기화 방법도 설명합니다. 그 다음에는 이전 예제를 향상시키는 간단한 수학 기능을 위해 VFPU 를 어떻게 사용하는지 설명합니다. 이 예제는 VFPU 코드를 쓰는 데 많은 도움이 될 것입니다.

어셈블리 언어

PSP VFPU 는 C 나 C++와 같은 고급 언어로는 프로그램 할 수 없습니다. 즉, 프로그래머가 VFPU 어셈블리 언어 명령 세트를 사용해야만 한다는 것을 의미합니다. 이 문서는 어셈블리 언어에 대한 초급 정도의 지식을 가진 독자를 위해 마련되었습니다. 어셈블리 언어에 대한 좋은 자습서나 설명서는 다음의 참고 문헌에서 찾을 수 있습니다: [6], [7]. 이 장에서는 각 기능에 존재하는 특정 어셈블러 연산 코드만을 다루며 gcc 인라인 어셈블러 구문 등의 주제에 관한 자세한 설명은 피합니다. 이러한 정보는 부록 A 에 나와있습니다.

The VFPU

VFPU 는 강력한 128 비트 벡터 부동 소수점 유닛입니다. 이는 ALLEGREX CPU 의 코프로세서 역할을 하며, 행렬과 행렬 연산과 같은 수학 기능에 매우 적합합니다. 그리고 사용자를 위한 128 개의 32 비트 레지스터를 포함합니다. 이를 '**행렬 레지스터 파일**'이라 부르며, 다음의 **행렬 레지스터** 섹션에서 설명합니다.

VFPU 는 코프로세서 모드에서만 실행할 수 있으므로 ALLEGREX 와 병행하여 실행할 수 없습니다. 그러나 주 CPU 에 비하여 벡터 연산 속도가 훨씬 빠르기 때문에 성능을 상당히 개선시킬 수 있습니다. 본 입문서의 마지막 예제는 VFPU 를 사용하여 광원을 장면 주위로 회전시키는 것으로 이전의 예제를 바탕으로 구축됩니다.

Libvfpv

SCE 는 툴 체인의 소스 코드를 갖춘, "libvfpv"라 부르는 C 라이브러리를 제공합니다. 이 라이브러리는 2, 3 및 4 요소의 벡터, 인라인 VFPU 어셈블리 코드를 사용하는 행렬 연산을 포함하여 종합적인 수학 기능을 포함하고 있습니다. 소스 코드는 다음에 나와있습니다: `/usr/local/psp/devkit/src/vfpv`. 다음 절에서는 예제에서 사용되는 기능을 설명합니다.

프로그램에서 `libvfpv` 기능을 사용하려면, `#include libvfpv.h` 를 실행하고, 연계된 라이브러리 목록에 `“-lvfpv”`를 추가하여 메이크파일에서 애플리케이션과 `libvfpv` 를 연계시켜야 합니다.

PSP 커널 소프트웨어는 방대한 복수 스레드 기능을 갖고 있으며, 우선순위에 근거한 체계를 통해 스레드를 관리합니다. 이 소프트웨어는 스레드 문맥 내에서 VFPU 레지스터를 관리하므로, 스레드가 변경되면 VFPU 레지스터는 저장되어 복원됩니다. VFPU 문맥에는 128 개의 레지스터에 관한 정보 등이 포함되므로, VFPU 문맥의 저장과 복원에 의해 성능에 영향을 줄 수 있습니다. 이를 해결하기 위해, PSP 커널에는 스레드의 생존 기간 동안 VFPU 를 사용할 것인지를 지정하는 기능이 포함됩니다. 이에 대한 기본값은 “off”이며, 따라서 어떤 스레드가 사용가능의 실행 없이 VFPU 에 대한 사용을 시도하는 경우 코프로세서 예외가 발생하여 프로그램이 종료됩니다. 그러나 스레드에서 VFPU 를 사용가능케 하는 것은 매우 간단한 절차입니다. 다음은 이에 대한 설명입니다.

실행은 `main()`에서 시작되며, 이것은 이 예제에서 관리되는 단 하나의 스레드입니다. 그러므로 유일한 요구조건은 그 스레드를 사용하기 전 어떤 시점에서 VFPU 를 실행가능케 하는 것입니다. 사용되는 기능은 `main()` 내에서 처음으로 호출되는 것입니다.

`sceKernelChangeCurrentThreadAttr(SceUInt clearAttr, SceUInt setAttr)`은 호출하는 스레드의 속성을 변경시킵니다. 이 경우 어떠한 특정 속성을 지울 필요가 없으므로, 첫째 매개변수로서 0 이 보내집니다.

둘째 인수로서 보내지는 `SCE_KERNEL_TH_USE_VFPU` 가 이 스레드에 대한 속성을 설정하도록 커널에 알립니다. 이로써 이 예제의 스레드 관리가 완료됩니다. PSP 커널 및 스레드 라이브러리에 관한 보다 상세한 정보를 원하는 경우, 참조 문헌 [9] 및 [10]을 참조할 것을 권장합니다.

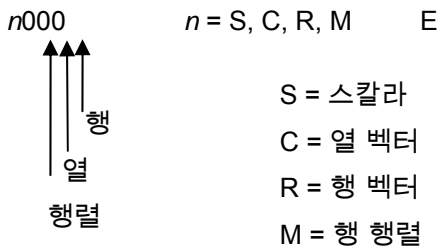
행렬 레지스터

VFPU 코드를 쓰기 위해서는 VFPU 칩 자체에 있는 행렬 레지스터에 관해 알아두면 도움이 됩니다. VFPU 의 행렬 레지스터 파일은 다양한 방식으로 접근할 수 있는 128 개의 32 비트 레지스터로 구성됩니다. 단일 스칼라 및 요소가 2, 3 및 4 인 벡터가 지원됩니다. 또한 2x2, 3x3 및 4x4 행렬로서도 레지스터에 접근할 수 있습니다. 벡터 및 행렬에 대한 모든 사용 옵션은 열 및 행 포맷으로 가능합니다. 따라서 이는 당황할 정도로 많은 옵션처럼 보일 수도 있으며, VFPU 사용자 설명서[8]에서 처음 보는 경우 더욱 그러할 것입니다.

행렬 레지스터는 \$0 - \$127 의 명칭을 가진 레지스터의 순차적 목록입니다. 각 레지스터는 32 비트 워드입니다.

레지스터 파일은 한 줄로 늘어서 있는 8 개의 4x4 행렬로 생각해 볼 수 있습니다. 이 경우 행렬 8 개 모두의 최상위 행은 \$0 - \$31 범위의 레지스터의 연속 집합이며, 둘째 행은 \$32 - \$63 등등으로 계속됩니다. 이는 VFPU 어셈블리 언어가 지정하고 있는, 다음 특성을 가진 4 자리 영숫자 문자열의 액세스 명명 규약과 매우 잘 대응합니다:

그림 5-1: VFPU 행렬 레지스터 명명 규약



예를 들어, S000 이라는 행렬 레지스터 액세스 코드는 첫째 4x4 행렬에서 첫째 열 및 첫째 행의 스칼라를 의미합니다.

벡터와 행렬의 정의 작업은 약간 더 복잡한 데, 이는 레지스터 파일에서 벡터나 행렬이 시작되는 곳으로 숫자가 번역되며 또한 요소가 있는 경우 순환되어 다른 성분을 정의하기 때문입니다.

그림 5-2: M000 액세스 코드

\$0	\$1	\$2	\$3	\$4	\$...	\$31
\$32	\$33	\$34	\$35	\$36	\$...	\$63
\$64	\$65	\$66	\$67	\$68	\$...	\$95
\$96	\$97	\$98	\$99	\$100	\$...	\$127

\$0	\$1	\$2	\$3
\$32	\$33	\$34	\$35
\$64	\$65	\$66	\$67
\$96	\$97	\$98	\$99

그림 5-2 는 M000 액세스 코드를 보여줍니다. 이것은 행렬 레지스터 파일에서 첫째 행렬의 첫째 열 및 첫째 행에서 시작되는 행 행렬입니다. 오른쪽에 있는 행렬은 M000 에 따른 행렬 레지스터 레이아웃입니다.

그림 5-3: M012 액세스 코드

\$0	\$1	\$2	\$3	\$4	\$...	\$31
\$32	\$33	\$34	\$35	\$36	\$...	\$63
\$64	\$65	\$66	\$67	\$68	\$...	\$95
\$96	\$97	\$98	\$99	\$100	\$...	\$127

\$65	\$66	\$67	\$64
\$97	\$98	\$99	\$96
\$1	\$2	\$3	\$0
\$33	\$34	\$35	\$32

그림 5-3 은 첫째 행렬에서 둘째 열의 셋째 행에서 시작하는 것으로 정의되는 행 행렬입니다. 그림에서 보는 바와 같이, 이 행렬의 시작 위치는 행렬 레지스터 파일로 오프셋 되며, 밝은 음영 부분은 순환되는 값을 보여줍니다. 오른쪽에 있는 행렬은 M012 에 따른 행렬 요소 레이아웃을 보여줍니다. 그러나 모든

구성이 전부 가능한 것은 아니며, 액세스 코드의 완벽한 목록 및 그에 상응하는 요소에 대한 설명은 참조 문헌 [8]에 나와있습니다.

코드

이 절에서는 예제에서 사용되는 libvfpv의 VFPU 기능들을 설명합니다. 그리고 어셈블리 코드 명령도 자세히 설명합니다.

기능

빛 위치는 월드 좌표에서 하나의 벡터입니다. 이것은 기하학적 도형과 같은 방식으로 GE 변환에 구속되지 않습니다. 따라서 빛 위치 벡터는 수동으로 이동해야 하며, 그 다음 `sceGuLight(int n, int type, int comp, const scePSpsFVector3 *vec)`를 호출하여 빛 위치를 재정의해야 합니다. 그 목적은 부동의 원환면 주위로 빛을 회전시키는 것입니다. 이를 위해서는 Y 축 주위에 3x3 회전 행렬을 작성하고, 이를 사용하여 빛 위치 벡터를 곱해야 합니다. libvfpv의 기능 3 가지가 이 작업에 도움이 됩니다.

```
sceVfpvMatrix3Unit( scePspFMatrix3 *pm0 );
```

이 기능은 3x3 행렬 포인터를 취하며 항등으로 설정합니다. 여기서 중요한 VFPU 명령은 단 2 가지이며 다음에 설명됩니다:

```
__asm__ (
    ".set      push\n"
    ".set      noreorder\n"
    "vmidt.t    e000\n"
    "sv.s       s000, 0 + %0\n"
    "sv.s       s001, 4 + %0\n"
    "sv.s       s002, 8 + %0\n"
    "sv.s       s010, 12 + %0\n"
    "sv.s       s011, 16 + %0\n"
    "sv.s       s012, 20 + %0\n"
    "sv.s       s020, 24 + %0\n"
    "sv.s       s021, 28 + %0\n"
    "sv.s       s022, 32 + %0\n"
    ".set      pop\n"
    : "=m" ( *pm )
);
```

위의 예제는 이 기능으로부터 VFPU 코드를 재생한 것입니다. 흐릿하게 표시된 영역은 표준 gcc 인라인 어셈블리 구문이며 모든 예제에서 사용됩니다. 이 구문에 관한 특징의 일부는 이 장 전체에서 필요에 따라 설명되지만, 보다 자세한 내용은 부록 A에서 다루고 있습니다.

대부분의 VFPU 명령은 ALLEGREX 나 FPU 명령과 차별하기 위해 "벡터"의 "v"로 시작하거나 이를 포함합니다. 필요에 따라서, 해당 명령이 유효한 벡터 요소들의 수량을 구별하는, 명령에 대한 접미사도 있습니다. 다음과 같은 접미사가 있습니다:

s	스칼라 요소와 관련
p	2 배 요소와 관련
t	3 배 요소와 관련
q	4 배 워드 요소와 관련

명령의 명칭은 대개 수행하는 연산에 대한 약자입니다.

```
"vmidt.t          e000\n"
```

vmidt.t 명령은 3x3 항등 행렬을 작성하여 행렬 레지스터의 지정한 위치에서 시작하여 이를 보관합니다. 여기서 이 명령의 "midt" 위치는 "행렬 항등"과 관계가 있으며, 접미사 "t"로부터 3 배 즉 3x3 행렬임을 알 수 있습니다. 약자에 따라 파악하기 쉬운 정도가 다를 수 있습니다. 그렇지만 곧 익숙해지게 됩니다.

```
"sv.s              s000, 0 + %0\n"
```

이 명령은 단일 워드를 지정된 메모리 주소 및 오프셋에 보관합니다. 이 경우 메모리 주소는 오프셋(0) + 주소(%0)로서 지정됩니다. 이 코드의 오프셋 부분은 쓰는 대상의 주소로부터 오프셋 되는 바이트 숫자를 결정합니다. 주소 부분은 gcc 인라인 어셈블리 구문을 사용하여 어셈블러 부위로 보내지는 주소 변수입니다. 위의 코드에서 오른쪽 괄호 이전의 마지막 라인은 다음을 포함합니다:

```
: "=m" (*pm)
```

이것은 write-to 메모리 변수로 보내지는 gcc 어셈블러 지시어입니다. 그리고 어셈블러 내부에서 "%0"로서 인용됩니다. "%" 부호는 코드 아래에 있는 목록에서 선언된 변수 중의 하나임을 지적하며, "0"는 처음으로 선언되었음을 의미합니다. 선언된 것이 또 있다면, 선언된 순서에 따라서 0 부터 차례로 접근할 것입니다. 자세한 내용은 부록 A 를 참조하십시오.

특정한 2 배 또는 3 배 지시어가 없으므로, "sv.s"를 9 번 호출하고 오프셋을 올바르게 증분시켜서 행렬 요소를 모두 올바른 장소에 보관해야 합니다. 그렇지만 4 요소의 벡터 및 행렬을 사용할 때는, sv.q 를 사용하여 1 개의 4 배 요소를 메모리에 한꺼번에 쓸 수 있습니다.

```
sceVfpuMatrix3RotY( scePspFMatrix3 *pm0,
                    const scePspFMatrix3 *pm1,
                    float ry
);
```

위의 기능은 Y 축이 대한 회전 행렬인 ry 라디언을 생성하여 pm0 행렬에 보관합니다. pm1 행렬이 NULL 이 아니라면, 생성된 행렬을 곱한 다음 보관하게 됩니다. 이 기능에 대한 VFPU 코드는 다음과 같습니다.

```
__asm__ (
".set          push\n"
".set          noreorder\n"
"mfcl         $8,  %2\n"
"mtv          $8,  s100\n"
"vrot.t       c000, s100, [c, 0, -s]\n"
"vidt.q       c010\n"
"vrot.t       c020, s100, [s, 0, c]\n"
```

```

"beql          %1, $0, 0f\n"
"vmmov.t      e200, e000\n"
"lvr.q        c100, 0(%1)\n"
"lvl.q        c100, 12(%1)\n"
"lvr.q        c110, 12(%1)\n"
"lvl.q        c110, 24(%1)\n"
"lvr.q        c120, 24(%1)\n"
"lvl.q        c120, 36(%1)\n"
"vmmul.t      e200, e000, e100\n"
"0:\n"
"sv.s         s200, 0 + %0\n"
"sv.s         s201, 4 + %0\n"
"sv.s         s202, 8 + %0\n"
"sv.s         s210, 12 + %0\n"
"sv.s         s211, 16 + %0\n"
"sv.s         s212, 20 + %0\n"
"sv.s         s220, 24 + %0\n"
"sv.s         s221, 28 + %0\n"
"sv.s         s222, 32 + %0\n"
".set         pop\n"
: "=m" (*pm0)
: "r"(pml), "f"(ry * (2.0f/3.14159265358979323846f))
: "$8"
);

```

VFPU 어셈블리 기능으로 보내지는 부동 소수점 변수들은, 일단 FPU 에 범용 레지스터로서 보관됩니다. 이 변수들은 VFPU 행렬 레지스터 파일로 이동되어야만 VFPU 가 사용할 수 있습니다. 이 기능의 처음 두 지시어에 의해 이 작업이 수행됩니다:

```

"mfc1          $8, %2\n"
"mtv          $8, s100\n"

```

ALLEGREX 지시어인 “mfc1”는 “코프로세서 1로부터 이동”을 의미합니다. 이 지시어는 그 변수를 FPU 범용 레지스터로부터 CPU 범용 레지스터로 이동시킵니다. “mtv” 지시어는 “워드를 VFPU 로 이동”의 의미이며, 그 변수를 CPU 레지스터로부터 VFPU 레지스터 파일에 있는 영역으로 이동하여 추가의 연산을 가능케 합니다.

그 다음의 3 라인은 회전 행렬의 생성을 다룹니다. Y 축에 대한 열 회전 행렬은 다음과 같이 정의할 수 있습니다:

그림 5-4

$$\begin{bmatrix} \cos \phi & 0 & \sin \phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \phi & 0 & \cos \phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

다음의 VFPU 지시어 3 개는 행렬 요소를 적절한 변수로 채워줍니다:

```

"vrot.t        c000, s100, [c, 0, -s]      \n"
"vidt.q        c010                          \n"
"vrot.t        c020, s100, [s, 0, c]       \n"

```


“vrot.t”는 이 작업에 필요한 완벽한 지시어입니다. 이는 스칼라를 취하여 그 변수의 코사인 및 사인을 계산한 다음, 행렬의 3 배 워드 영역 내에 그 결과를 배치합니다. 또한 양 혹은 음 변수와 연산이 되도록 지시할 수도 있습니다.

여기서 유의할 것은 VFPU 의 포맷입니다. 이 기능을 위한 어셈블리 매개변수 목록에서 *ry* 변수가 수정되었습니다:

```
"f"(ry * (2.0f/3.14159265358979323846f))
```

이것은 단위환산 절차에 해당합니다. VFPU 의 안각 포맷은 라디언이나 각도가 아니라 4 분면입니다. 4 분면이란 원의 4 분의 일을 말하며, 원에 대한 1 회 회전은 4 개의 4 분면과 동등합니다. 여기서 환산은 라디언 단위의 각도를 4 분면으로 환산하는 방법입니다. “vrot.t” 지시어의 포맷은 다음과 같습니다:

```
vrot.t          vDst, vSrc, Writemask
```

vDst 및 *vSrc* 는 각각 연산의 목적지와 출처에 해당합니다. *Writemask* 는 VFPU 가 *vDst* 레지스터에 변수를 쓰는 v 패턴의 결정에 사용하는 5 비트 숫자로 변환되는 니모닉입니다. 이 니모닉은 일부 제한이 있지만 변동 목록을 수용합니다. “c” 및 “s” 요소는 코사인 혹은 사인을 사용해야 하는가를 나타내며, “-” 기호는 음수에 대해 유효합니다. 이것들이 괄호 안에서 설정되는 순서에 따라 목적지 3 배 내에서 쓰기 위치를 결정합니다. “0”는 관련요소를 영으로 설정하게 됩니다. 이상의 것만이 사용할 수 있는 유효한 문자입니다.

“vidt.q” 지시어는 지정된 벡터를 항등 행렬의 해당 부분으로서 설정합니다. 여기서는 둘째 열이 회전 행렬 요구조건과 대응합니다.

```
"beql          %1, $0, 0f\n"
```

이 ALLEGREX 지시어는 “동등한 분기 가능”을 의미합니다. 여기서는 항상 영인 레지스터 “\$0”에 대해 둘째 입력 매개변수를 확인합니다. 입력 매개변수는 행렬 포인터 *pm1* 입니다. 입력 포인터가 NULL 인 경우, 프로그램 계수는 지정된 주소에 설정됩니다. 이 경우에는 “0:\n”에 의해 지정된 서브루틴이 되며, 코드에서는 “0f”로서 인용됩니다. 이 지시어는 분기 시험이 성공하는 경우에만 지연 슬롯의 지시어(분기 이후에 나오는 지시어)를 실행합니다.

```
"vmmov.t       e200, e000\n"
```

이 지시어는 3 배 워드 벡터를 “e000”로부터 “e200”에 복사합니다. 따라서 “e200”은 스칼라 형태이지만 이 부분의 행렬 레지스터로부터 모든 값이 주 메모리에 다시 쓰여진다는 것을 알 수 있습니다. 그러므로 행렬 포인터의 상태가 NULL 이며, 회전 행렬은 새 위치에 복사된 다음, 실행은 데이터를 주 메모리에 다시 쓰는 부분으로 분기됩니다. 그러므로 이 기능으로 NULL 상태를 보냄으로써, `sceVfpuMatrix3Unit(...)`을 요구조건으로 호출할 필요가 없게 됨을 알 수 있습니다. 그러나 이것은 예제로서 남겨둡니다. 분기가 실행하지 않으면, 행렬 데이터는 복사되지 않습니다. 대신 일련의 지시어들이 실행되며 다음에서 설명합니다:

```
"lvr.q         c100, 0(%1)\n"
"lvl.q         c100, 12(%1)\n"
```

일반적으로 주 메모리로부터 VFPU 로 변수가 로딩되면, 그 변수는 그와 동등한 크기에 대해 정렬되어야 합니다. 즉, 워드 주소는 워드 경계에 대해 정렬되고 4 배 경계에 대해 정렬되어야 합니다. 만약 메모리

주소가 이러한 사전 요구 조건을 만족시키지 못하면, CPU 가 예외를 생성하여 실행이 종료됩니다.

“lvr.q” 및 “lvl.q” 지시어는 이러한 문제가 발생되지 않도록 도와줍니다. 이 지시어들은 코드에 나와 있는 것과 같이 순차적으로 사용될 때 가장 유효합니다. 이들은 지정된 주소의 쿼드워드를 VFPU 행렬 레지스터 파일에 보관하지만, 순서 및 보관의 위치 조정이 중요하며, 그 이유는 그러한 위치 조정으로 일반 로드-워드/로드-벡터 지시어와 차별되기 때문입니다.

그림 5-5: lvr.q 및 lvl.q 소개

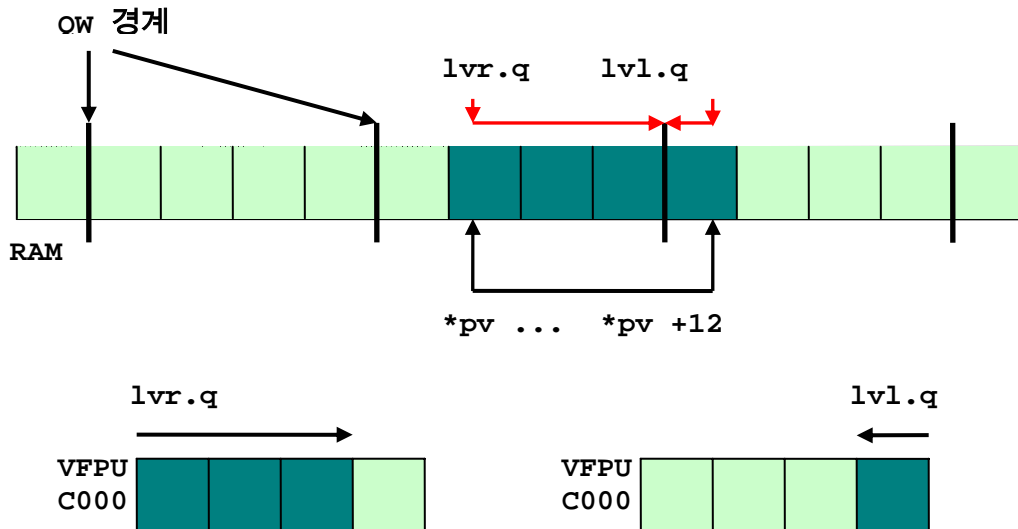


그림 5-5는 이 예제에서 “lvr.q” 및 “lvl.q” 지시어가 메모리 및 VFPU 행렬 레지스터에서 어떻게 작동하는지 보여줍니다. 그림의 위 부분은 주 RAM의 일부 즉, 어두운 음영에서 “pv”에 의해 지적되는 쿼드워드를 보여줍니다. 보는 바와 같이, 벡터는 쿼드워드 경계에 걸쳐 있습니다. “lvr.q” 지시어는 “pv”에서 시작하여 오른쪽으로 쿼드워드 경계의 끝까지 효과적으로 읽게 됩니다. 그리고 워드를 행렬 레지스터 파일에 왼쪽에서 오른쪽 방향으로 보관하게 됩니다. “lvl.q” 지시어는 “pv+12”에서 시작하여 왼쪽으로 쿼드워드 경계까지 읽은 다음, 그 워드를 행렬 레지스터 파일에서 오른쪽에서 왼쪽 방향으로 보관하게 됩니다. 이렇게 함으로써 두 기능을 모두 사용하여 쿼드워드 요소들을 행렬 레지스터 파일에 로딩하게 되며, 이는 요소들이 쿼드워드 경계에 대해 정렬되지 않더라도 상관없습니다.

```
"vmmul.t          e200, e000, e100\n"
```

행렬이 “e100”로 로딩된 상태에서, “vmmul.t” 지시어가 실행됩니다. 이 지시어는 3x3 행렬 곱하기 지시어이며, 이 경우 그 결과를 “e200”에 보관합니다. 다음, 목적지 주소에 다시 쓰여집니다.

```
sceVfpuMatrix3Apply(  scePspFVector3 *pv0,
                      scePspFMatrix3 *pm0,
                      scePspFVector3 *pv1
                    );
```

마지막으로 설명할 기능이 위에 나와있습니다. 이 기능은 3 요소 벡터에 의해 3x3 행렬을 변환시킵니다. 이 기능이 Y 축에 대해 벡터를 실제로 회전시키는 역할을 합니다:

```

__asm__ (
    ".set          push\n"
    ".set          noreorder\n"
    "lvr.q         c100, 0 + %1\n"
    "lvl.q         c100, 12 + %1\n"
    "lvr.q         c110, 12 + %1\n"
    "lvl.q         c110, 24 + %1\n"
    "lvr.q         c120, 24 + %1\n"
    "lvl.q         c120, 36 + %1\n"
    "lvr.q         c200, 0 + %2\n"
    "lvl.q         c200, 12 + %2\n"
    "vtfm3.t       c000, e100, c200\n"
    "sv.s          s000, 0 + %0\n"
    "sv.s          s001, 4 + %0\n"
    "sv.s          s002, 8 + %0\n"
    ".set          pop\n"
    : "=m" (*pv0)
    : "m" (*pm0), "m" (*pv1)
);

```

이제까지 이 예제에 사용된 지시어의 대부분을 설명했습니다. 이전 예와 마찬가지로 행렬 및 벡터는 “lvr.q” 및 “lvl.q” 지시어으로써 조립됩니다. 새로 추가되는 내용은 다음과 같습니다:

```
"vtfm3.t       c000, e100, c200\n"
```

이 하나의 지시어에 의해 행렬 곱하기가 이루어집니다. 그에 따른 벡터는 열 포맷으로 보관된 다음 “sv.s”로써 메모리에 다시 쓰여집니다.

요약

이 장에서는 VFPU 코프로세서를 소개했으며, SCE 가 제공하는 VFPU 수학 라이브러리인 libvfpu 와 관련 있는 기능들을 설명했습니다. 이 장에서는 제 4 장의 예제를 발전시켜 VFPU 로써 빛을 회전시켰으며, 코드의 설명을 통하여 지시어 별로 각종 기능들에 관해 알아보았습니다.

VFPU 는 PSP 시스템의 강력한 부분이며 하드웨어로부터 최대의 성능을 얻으려면 그 사용이 필수적입니다. VFPU 는 제공된 수학 기능은 물론 소립자 체계와 물리학 시뮬레이션과 같은 다른 용도에도 매우 적합합니다. 개발자들은 이미 이를 잘 활용하고 있으며, 앞으로도 새로운 용도를 계속 찾아낼 것입니다.

결론

이제까지의 설명이 PSP 프로그래밍을 깊이 이해하는 데 도움이 되었을 것입니다. 설명한 내용을 간단히 요약하면, 기본적인 2 차원 및 3 차원 그래픽을 말했고, 컨트롤러 출력 그리고 애플리케이션 내에서 VFPU 를 사용하여 시작하는 방법을 소개했습니다.

이 문서가 인쇄되는 현재, PSP는 가장 강력하고 흥미로운 휴대용 게임 콘솔이며, 이 지침서에서 요약한 내용 외에도 다른 많은 기능을 가지고 있습니다. 독자들이 애플리케이션에서 이러한 훌륭한 기능을 구현하는 것이 매우 쉽다는 사실을 인식하기를 바라며, 또한 핸드헬드 엔터테인먼트의 새로운 시대를 여는 시스템을 연구하는 데 도움이 되었으면 합니다.

부록 A:

gcc 인라인 어셈블리 구문

이 페이지는 공백으로 둡니다.

인라인 어셈블리는 어셈블리 코드가 C 또는 C++ 소스 파일 내에 삽입되는 것을 허용하는 매우 유용한 기능입니다. 한 예로서 기능 내에 최적화 VFPU 코드를 쓰는 데 편리합니다. Gcc 는 이 옵션을 고유하게 제공하지만, 사용하려면 일부 규칙을 따라야 합니다. 이 부록에서는 일반적인 용도를 위한 gcc 인라인 어셈블리 구문의 개요를 제공합니다. 전형적인 VFPU 인라인 어셈블리 기능이 다음에 나와있습니다:

```
ScePspFVector4 *sceVfpuVector4Scale( ScePspFVector4 *pv0, const ScePspFVector4
*pv1, float t)
{
    __asm__ (
        ".set          push\n"
        ".set          noreorder\n"
        "mfc1          $8,  %2\n"
        "mtv           $8,  s010\n"
        "lv.q          c000, %1\n"
        "vscl.q        c000, c000, s010\n"
        "sv.q          c000, %0\n"
        ".set          pop\n"
        : "=m"(*pv0)
        : "m"(*pv1), "f"(t)
        : "$8"
    );
    return (pv0);
}
```

이 기능은 스칼라에 의해 4 요소 벡터의 크기를 조정합니다. 여기서는 인라인 어셈블리 구문만을 설명하므로, 실제의 VFPU 코드를 흐릿하게 표시했으며 설명하지 않습니다.

```
__asm__ ( ... );
```

인라인 어셈블리 코드는 모두 위의 문 내에 배치되어야 합니다. 이것은 이 코드가 어셈블리 코드이며 어셈블러를 위하여 유보된 것이므로 일반 소스 코드와 마찬가지로 컴파일 하지 말 것을 gcc 에게 알리는 컴파일러 지시입니다.

이 부록에서의 설명을 위하여 위의 코드를 적절한 명명 규약으로 번역한 것은 다음과 같습니다:

```
__asm__ (
    "set          assembler option          \n"
    "set          assembler option          \n"
    "opcode       register,    variable      \n"
    "opcode       register,    register      \n"
    "opcode       register,    variable      \n"
    "opcode       register,    register, register \n"
    "opcode       register,    variable      \n"
    "set          assembler option          \n"
    : "constraint" (output variable)
    : "constraint" (input variable), "constraint" (input variable)
    : "clobber"
);
```

어셈블러 옵션의 설정

어셈블리 코드는 3 개의 ".set ... \n" 문에 의해 둘러싸입니다. 이러한 세트 어셈블러 옵션들은 코드에 매우 중요합니다. 다음의 3 가지가 가장 흔히 사용되는 것들입니다:

".set push\n"은 기존 어셈블러 옵션을 스택에 저장하여 필요에 따라 기존 상태를 복원시킬 수 있습니다.

".set noreorder\n"은 지시어에 대한 재명령을 하지 않도록 어셈블러에게 알린다는 점에서 중요하며, 어셈블러는 코드 최적화가 필요하다고 생각하는 경우 그렇게 할 수 있습니다. 수동으로 최적화된 코드가 언제든지 어셈블러에 의한 해결안보다 낫기 때문에 이 옵션을 최대한으로 제어하는 것이 중요합니다.

".set pop\n"은 이전의 어셈블러 옵션을 ".set noreorder\n" 이전의 상태로 복원시킵니다.

어셈블러 코드

어셈블러 코드 자체는 읽혀지거나 쓰여진 데이터, 입력 및 출력 변수 그리고 레지스터에 대한 연산을 수행하는 일련의 지시어를 말합니다. 입력 및 출력 변수는 "%" 기호 및 숫자로써 어셈블리에서 선언됩니다. 이러한 변수는 처음 두 개의 섹션에서 정의되며, 메인 코드 블록 이후 클론에 의해 분리됩니다.

출력 / 입력 변수

이것은 어셈블러 코드가 읽거나 쓰는 대상의 변수입니다. 인라인 어셈블러의 변수와 C 및 C++ 변수의 첫째 중요한 차이는 그 순서에 있습니다. C/C++에서는 변수가 사용되기 전에 선언됩니다. 인라인 어셈블리에서 변수를 사용하려면, 입출력 변수 목록에서 어셈블러 코드 아래에서 선언되며, 그 구문은 다음과 같습니다:

```
: "constraint" ( output variable 0 ) // referenced with %0
: "constraint" ( input variable 1 ) // referenced with %1, etc
```

출력 및 입력 변수는 위에 나와 있는 순서에 따라 %0 에서 시작하여 큰 숫자로 선언됩니다. 예에 있는 코드의 출력 변수는 벡터 포인터 pv0 이며 매개변수로서 기능으로 보내집니다. 입력 변수도 동일한 구문 규칙을 따르며 코드에서와 같은 방식으로 사용됩니다.

제약

제약은 변수를 어떻게 사용하는 지를 정의하는, 어셈블러를 위한 추가 정보입니다. 그 예는 다음과 같습니다:

```
"r", "f"
```

이것은 변수가 "r"에 대한 정수 레지스터 및 "f"에 대한 부동 소수점 레지스터에 배치되어야 함을 지적합니다.

```
"m"
```

이것은 변수가 주소임을 지적합니다.

그 밖에, 제약에 추가시킬 수 있는 변경자들이 있으며, 그 예는 다음과 같습니다:

```
"="
```

이것은 변수가 쓰기 전용이어야 함을 규정합니다.

"+"

이것은 판독 / 기록 변수입니다.

클로버

블럭에서 마지막 클론에 의해 분리된 부분이 클로버 목록입니다. 이것은 어셈블리 코드가 사용하는 레지스터의 목록입니다. 코드가 CPU 레지스터를 사용하면, 여기에 *반드시* 포함되어야 합니다. 그러나 VFPU 레지스터는 포함시키지 않아도 됩니다.

추가 참조 문헌:

[12], [13], [14]

이 페이지는 공백으로 둡니다.

부록 B:

에뮬레이터 및 하드웨어 툴 라이브러리의 차이점

이 페이지는 공백으로 둡니다.

이 부록에서는 에뮬레이터와 하드웨어 툴 사이의 코드를 차별화하는, 예제에서 사용된 여러 기능을 다룹니다.

에뮬레이터의 특성

에뮬레이터 `#includes` 및 기능

```
#include <libemu.h>
```

이 파일은 에뮬레이터 기능의 호출을 위해 반드시 포함시켜야 합니다.

`Init()` 기능에서, `libGu` 가 초기화되기 이전에 에뮬레이터가 먼저 초기화되어야 하며, 다음을 호출하여 수행합니다:

```
sceEmuInit()
```

마찬가지로 프로그램 종료 시 다음의 호출이 있습니다:

```
sceEmuTerm()
```

이것은 에뮬레이터 라이브러리를 종료시킵니다.

에뮬레이터 상의 수직 동기(sync)는 다음과 같습니다:

```
sceEmuVSync(SCEEMU_SYNC_WAIT)
```

이상은 제 4 장에서 이미 다뤘던 컨트롤러 차이점과는 별도로 호출해야 하는 에뮬레이터 고유의 기능들입니다.

하드웨어 툴의 특성

먼저, 에뮬레이터를 위해 작성된 코드가 하드웨어 툴에서 작동하도록 하려면, 모든 에뮬레이터 기능에 대해 이미 정의된 호출들을 모두 제거하십시오. 그 밖에 코드에 대한 중요한 추가사항이 몇 가지 있으며, 다음에 설명합니다.

다음 파일을 포함시켜야 합니다:

```
#include <moduleexport.h>
```

PSP 애플리케이션 즉 "모듈"을 생성하려면 하드웨어 툴에서 다음 매크로가 필요합니다:

```
SCE_MODULE_INFO( hellotriangle, 0, 1, 1 );
```

위의 매크로는 모듈에 필요한 정보를 생성합니다. 한꺼번에 여러 개의 모듈을 로딩할 수 있습니다.

모듈은 PSP 커널에 의해 관리됩니다. 커널이 모듈을 파악하려면 위의 매크로가 필요합니다.

모듈 이름은 실행 가능한 것의 이름과 동일해야 합니다.

```
#include <displaysvc.h>
```

이것은 다음과 같은 디스플레이 기능 호출에 필요합니다:

```
sceDisplayWaitVblankStart()
```

이 기능은 하드웨어 수직 동기(sync) 기능이며, 에뮬레이터 코드에서 사용되는

`sceEmuVSync(SCEEMU_SYNC_WAIT)` 를 대신합니다.

다음은 또 다른 하드웨어 고유의 디스플레이 기능으로, `InitGFX()` 기능에서 반드시 호출해야 합니다:

```
sceGuDisplay(SCEGU_DISPLAY_ON);
```

이 기능은 PSP 하드웨어 화면을 명백히 활성화시킵니다. 만약 이 기능이 사용되지 않으면, 화면에 아무것도 표시되지 않습니다.

본 자습서에서 사용된 예제들은 메모리를 동적으로 할당하지 않지만, 이 기능은 일정한 크기 이상의 거의 모든 애플리케이션에서 필요합니다. 그러므로 다음에서 설명하는 메모리 관련 사안을 알아두어야 합니다.

메모리 관리

PSP 하드웨어는 주 메모리로서 32MB의 DDR RAM을 갖추고 있습니다. 그러나 이 가운데 8Mb가 커널로서 유보됩니다. 디버거 정보를 제외한 실행가능한 정보는 주 메모리의 나머지 부분에 상주해야 합니다. 그러므로 예를 들어, 실행가능한 정보가 3MB이라면, 애플리케이션에 사용할 수 있는 RAM은 21MB입니다. `malloc()`을 사용하여 나머지 21MB 블록에서 메모리를 할당하기 위해서는, 코드가 다음 라인을 포함해야 합니다:

```
int sce_newlib_heap_kb_size = <size of heap in kilobytes>;
```

이것이 애플리케이션 내에서 `malloc()`에 사용할 수 있는 메모리의 양입니다. 개별적인 메모리 관리 방식을 가지고 있는 경우, `malloc()`에 관계 없이 이 블록을 관리하게 되지만, 이상은 일차적으로 메모리를 할당하는 일반적인 방식입니다.

참조 문헌

1. *ALLEGREX 설명서 – SCE*
<https://pspdev.net/projects/hwmanuals>
2. *LibGu 참조 설명서 – SCE*
<https://pspdev.net/projects/library>
3. *그래픽 엔진 사용자 설명서 – SCE*
<https://pspdev.net/projects/hwmanuals>
4. *컴퓨터 그래픽, C 버전 – 제2판*
Donald Hearn and N. Pauline Baker, Prentice Hall
5. *RISC 어셈블리 언어 프로그래밍 입문*
John Waldron, Addison Wesley
6. *See Mips run*
Dominic Sweetman, Elsevier, Morgan Kaufmann
7. *Vfpu 사용자 설명서 – SCE*
<https://pspdev.net/projects/hwmanuals>
8. *커널 개요 – SCE*
<https://pspdev.net/projects/library>

9. *PSP 스레드 매니저 참고서 – SCE*
<https://psp.scedev.net/projects/library>
10. *Vfpu 명령 참고 설명서 – SCE*
<https://psp.scedev.net/projects/library>
11. *Dylan Cuthberts Asm Webpage*
http://www.geocities.com/dylan_cuthbert/asmrules.html
12. *Gcc Extended Asm Page*
<http://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html#Extended-Asm>
13. *인라인 어셈블러 제약 사항*
http://gcc.gnu.org/onlinedocs/gcc-2.95.3/gcc_16.html#SEC175
14. *OpenGL 프로그래밍 지침서*
Fourth Edition, Shreiner, Woo et al Addison Wesley