



PSP[®] プログラミングの概要

リリース 1.0

2005 年 3 月

© 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.

目次

本ドキュメントについて	v
第 1 章: 予備知識	1-1
はじめに	1-3
エミュレータ	1-3
ALLEGREX CPU のエミュレーション	1-3
libGu および libGum グラフィックライブラリの機能	1-3
コントローラのサポート	1-4
PSP DTP-T1000A 開発用ハードウェア (ハードウェアツール)	1-4
エミュレータとハードウェアツールの違い	1-4
ハードウェアツール用のソフトウェア	1-4
本ドキュメントに付属のサンプルコード	1-4
エミュレータとハードウェアツールで使用する実行可能ファイル形式の違い	1-5
情報とサポート	1-5
第 2 章: PSP ハードウェア: システムの概要	2-1
一般的な概要	2-3
ALLEGREX CPU コア	2-4
グラフィックエンジン	2-4
グラフィックエンジンの内部	2-4
ディスプレイリスト	2-5
ディスプレイリストの種類	2-6
libGu	2-6
libGu の使用方法のデモ例	2-6
ディスプレイリスト内のデータ	2-7
第 3 章: 簡単なプログラムのコンパイル	3-1
はじめに	3-3
データ構造体	3-3
初期化	3-3
描画バッファとディスプレイバッファ	3-4
ビューポートの初期化	3-5
初期化セクションのファイナライズ	3-5
メインループ	3-5
頂点の形式と、グラフィックエンジンでの解釈	3-6
インデックス付きとインデックスなしのプリミティブ	3-6
まとめ	3-7
第 4 章: 簡単な 3 次元シーン	4-1
はじめに	4-3
深度バッファ	4-3
3 次元座標系と libGum	4-3
libGum	4-4
ライティング	4-5
パッチ	4-6
コントローラの処理	4-7
エミュレータ用のコントローラ入力	4-7
ハードウェア用のコントローラ入力	4-7
ビュー変換: 簡単なカメラ	4-8

マテリアル	4-9
パッチ	4-9
行列変換	4-10
まとめ	4-11
第 5 章: VFPU (Vector Floating Point Unit)	5-1
はじめに	5-3
アセンブリ言語	5-3
VFPU	5-3
libvfpv	5-3
行列レジスタ	5-4
コード	5-5
関数	5-5
まとめ	5-10
最後に	5-10
付録 A: gcc インラインアセンブリ構文	1
アセンブラのオプションの設定	3
アセンブラのコード	4
入出力変数	4
制約	4
クロバー	4
付録 B: エミュレータとハードウェアツールライブラリの違い	1
エミュレータ固有	3
ハードウェアツール固有	3
メモリ管理	4
参考文献	4

本ドキュメントについて

本ドキュメントは『PSP® プログラミングの概要リリース 1.0』です。

本ドキュメントは、PSP 環境を初めて使用するプログラマを対象としており、本ドキュメントと同じソースディレクトリに用意されているサンプルコードについて説明しています。

本ドキュメントは、C や C++などのプログラミング言語の知識があり、3次元コンピュータグラフィックの基礎知識のあるプログラマを対象としています。また、アセンブリ言語の概念について知識があれば、第 5 章で説明する VFPU (Vector Floating Point Unit) について理解しやすくなります。ただし付録 A で、第 5 章の説明を補足するための情報と、アセンブリ言語の知識のないプログラマ向けに、インラインアセンブラの構文について簡単に説明しています。

また本ドキュメントでは、ランタイムライブラリのドキュメント(<https://psp.scedev.net/>から入手可能)で説明している関数や命令を紹介しています。そのため、事前にランタイムライブラリのドキュメントをお読みになるか、少なくとも、本ドキュメントを参照する際にお手元に置いておくようにしてください。

なお、本ドキュメントは、ランタイムライブラリのドキュメントの代わりとなるものではありません。ただし、ランタイムライブラリのドキュメントで扱っている内容を詳しく説明している箇所もあり、内容が重複している場合もあります。

本ドキュメントで説明している例やサンプルコードには、エミュレータ環境と DTP-T1000A ハードウェア環境の両方で動作する関数が数多く含まれています。ただし第 5 章には、DTP-T1000A ハードウェア環境だけに関係した情報 VFPU を使用したコードの作成についての情報 を説明しています。

さらに付録 B では、ソースコード内のさまざまな関数呼び出しについて説明します。この情報は、エミュレータと DTP-T1000A ハードウェアの両方に関する説明です。

変更履歴

新規ドキュメント

前提要件

PSP™ の開発について知識があること。

表記上の規則

本ドキュメントでは、文章の意味をはっきりとさせるために、以下のような表記規則を用いています。

書体	意味
タイプライタ体	プログラムのコードを表します。
イタリック体	変数や引数の名前、構造体のメンバー名 (構造体/関数の定義時のみ) を表します。
ボールド体	データの型や構造体/関数の名前 (構造体/関数の定義時のみ) を表します。
青色	ハイパーリンクを表します。

ディベロッパサポート

Sony Computer Entertainment America (SCEA)

SCEA によるディベロッパサポートは、北米地域に在住のライセンスだけが利用できます。開発者向けのサポートや、本ドキュメントの追加コピーを入手するには、下記までご連絡してください。

連絡先	ディベロッパサポート
Attn: Development Tools Coordinator Sony Computer Entertainment America 919 East Hillsdale Blvd, Foster City, CA 94404, U.S.A.. Tel: (650) 655-8000	電子メール: scea_support@psp.scedev.net Web サイト: https://psp.scedev.net Developer Support Hotline: (650) 655-5566 (月～金の午前 8 時～午後 5 時、PST/PDT)

Sony Computer Entertainment Europe (SCEE)

SCEE によるディベロッパサポートは、PAL 方式のテレビ放送地域 (ヨーロッパとオーストラレーシアを含む) に在住のライセンスだけが利用できます。開発者向けのサポートや、本ドキュメントの追加コピーを入手するには、下記までご連絡してください。

連絡先	ディベロッパサポート
Attn: Development Tools Manager Sony Computer Entertainment Europe 13 Great Marlborough Street London W1F 7HP, U.K. Tel: +44 (0) 20 7859-5000	電子メール: scee_support@psp.scedev.net Web サイト: https://psp.scedev.net Developer Support Hotline: +44 (0) 20 7911-7711 (月～金の午前 9 時～午後 6 時、GMT/BST)

第 1 章:

予備知識

このページは空白です。

はじめに

PSP には、以下の 2 種類の開発環境があります。

- エミュレーション環境
- DTP-T1000A ハードウェアツールを使用したハードウェア環境

この章では、上記 2 つの開発環境の違いについて概要を説明します。また、PSP 用のプログラムを作成するための前提要件についても定義します。

PSP 開発環境のシステムをまだセットアップしていなければ、この章の説明が役に立つはずですが、また、この章には、PSP のプログラミングを始めるのに必要な資料のリストも記載してあります。ただし、本ドキュメントはセットアップガイドではありません。エミュレータと DTP-T1000A ハードウェアツールの両方の開発環境のセットアップ手順を説明したドキュメントが数多く用意されていますので、詳しくはそれらのドキュメントを参照してください。

エミュレータ

PSP エミュレータ環境を使用すると、開発中の PSP プログラムを PC の Windows 環境でコンパイルしたり、実行したり、バグすることが可能です。エミュレータのコードをコンパイルして実行するのに必要なファイルは、すべて <https://psp.scedev.net/> のダウンロードセクションからダウンロードできます。

ランタイムライブラリを用いた PSP プログラムのコードを開発するには、エミュレータ環境を使用すると非常に便利です。

エミュレータのインストール方法について詳しくは、以下のパッケージに付属のドキュメントを参照してください。

- Win32 コンパイラ (gcc)
- Win32 デバッグ環境
- Win32 PSP エミュレータプログラム
- ランタイムライブラリとドキュメント

エミュレータは、PSP の CPU、グラフィックエンジン、およびハンドヘルド・コントローラだけをサポートしています。DTP-T1000A ハードウェアツールではなく、エミュレータを使用して PSP プログラムの開発を始めておけば、同じコードをハードウェアツールで実行させたい場合でも、コードへの変更が最小限で済みます。

ALLEGREX CPU のエミュレーション

エミュレータのメインとなる機能は、ALLEGREX CPU のエミュレーションです。すべてのコードは、ALLEGREX CPU で実行可能な標準の MIPS 4000 にコンパイルされます。つまり、PSP エミュレータ環境は標準的な MIPS エミュレータで、Windows 環境で MIPS コードのコンパイルとデバッグが行えます。

libGu および libGum グラフィックライブラリの機能

libGu は Sony Computer Entertainment (SCE) が提供する高性能のグラフィックライブラリで、エミュレータと開発者用ハードウェアツールの両方に付属しています。このグラフィックライブラリのほとんどの機能は、OpenGL を用いたエミュレータでサポートされています。そのため、PSP DTP-T1000A ハードウェアツールをまだ入手していなくても、libGu を使用して PSP 特有のエンジンコードが開発できます。

エミュレータ環境では libGum API も使用できます。この libGum は libGu の上位に位置する別のグラフィック API で、主に行列演算機能を提供します。

コントローラのサポート

エミュレータは、標準的な Windows コントローラドライバを使用してコントローラのエミュレーションをサポートしています。そのため、エミュレーション環境で、Windows 互換のコントローラを PSP のコントローラとして動作させるように設定できます。エミュレータは、PSP のボタンとアナログスティックをすべてサポートしています。

PSP DTP-T1000A 開発用ハードウェア(ハードウェアツール)

ハードウェアツールは、PSP ソフトウェアの開発に必須の機器です。PSP のすべての機能が組み込まれており、ネットワーク接続を介してデバッグ環境とハードウェアツールとのコミュニケーションを行うための PC ハードウェアで構成されています。

ハードウェアツールには、画面の内蔵された PSP コントローラが付属しているほか、別のモニタで出力を確認できるように VGA 出力コネクタも付いています。

ハードウェアツールのインストール方法については、ハードウェアツールに付属のドキュメントを参照してください。

開発したコードを実際にテストするには、Windows か Linux の動作しているホスト PC 上でコードをコンパイルしてから、ネットワークを介してハードウェアツールにプログラムを送信し、ハードウェアツールでプログラムを実行します。

エミュレータとハードウェアツールの違い

ハードウェアツールを使用してコードを開発する場合、先に説明した CPU やグラフィックエンジンなどのハードウェア機能だけでなく、実際の PSP コンソールに含まれているすべてのハードウェア機能が使用できます。実際の PSP コンソールには、以下の機能が含まれています。

- VFPU (Vector Floating Point Unit)
- オーディオとビデオ
- UMD
- ネットワーク機能
- メモリースティック
- USB

これらの情報は、<https://psp.scedev.net/>から入手可能なドキュメントに記載されているので参考にしてください。

ハードウェアツール用のソフトウェア

SCE では、Linux と Windows の両方のオペレーティングシステム用に、ソフトウェアツールを一式用意しています。本ドキュメントで説明するサンプルには Linux の makefile が付属していますが、ネットワークを介してハードウェアツールに接続した Linux や Windows のデバッグでも、これらのサンプルを実行することが可能です。以下の必須パッケージは、<https://psp.scedev.net/>から入手できます。それぞれのパッケージには、インストール方法を説明したドキュメントが付属しているので参考にしてください。

- Linux コンパイラ (gcc)
- Linux / Win32 デバッグ環境
- ランタイムライブラリとドキュメント

本ドキュメントに付属のサンプルコード

本ドキュメントに付属しているサンプルコードは、2 つのディレクトリに分かれています。1 つはエミュレータが使用するソースコード、もう 1 つはハードウェアツールが使用するソースコードです。それぞれのソースコードをご

覧になるとわかるように、どちらの開発環境のソースコードもよく似ています。大きな違いはライブラリのリンクオプションで、実際のオプションは makefile の中に指定されています。

エミュレータとハードウェアツールで使用される実行可能ファイル形式の違い

実行可能ファイルの形式は、それぞれの環境で以下のように異なります。

- エミュレータでは.elf ファイルを実行する
- ハードウェアツールでは.prx ファイルを実行する

本ドキュメントで説明しているサンプルを使用する前に、まずはランタイムライブラリに付属のサンプルコードをコンパイル・実行して、自分の開発プラットフォーム (Windows または Linux) でのコンパイルおよびデバッグのプロセスに慣れておくようにしてください。

情報とサポート

- ニュースやアップデート情報、また最新のライブラリリリースなど、PSP に関連した情報については、以下の Web サイトを参照してください。
<https://psp.scedev.net/>
- ライセンスを受けた PSP 開発者へのデベロッパサポートについては、以下の Web サイトを参照してください。
<https://psp.scedev.net/support/about/>
- PSP の開発に関する専用のニュースグループについては、以下の Web サイトを参照してください。
<http://www.scedev.net/psp/connect.html>

このページは空白です。

第 2 章:

PSP ハードウェア: システムの概要

このページは空白です。

本章では、エミュレータとハードウェアツールの両方の環境と互換性がある PSP システムのアーキテクチャ部分つまり、CPU とグラフィックシステム について、さまざまな側面を簡単に紹介しましょう。

PSP には、最先端のポータブルコンソールとしての機能に関わるさまざまな特徴があり、それらの機能の多くは、それ自体が大きなテーマです。知識を深め、最終的には開発するゲームを豊かなものにする、PSP 開発の多くの側面を理解してください。

全般的な概要

PSP は多機能のポータブルゲームシステムです。PSP の持つ機能は、大きく以下の 3 つに分類できます。

- CPU およびグラフィック
- メディア (オーディオとビデオ)
- 接続性 (WiFi、USB、メモリースティックなど)

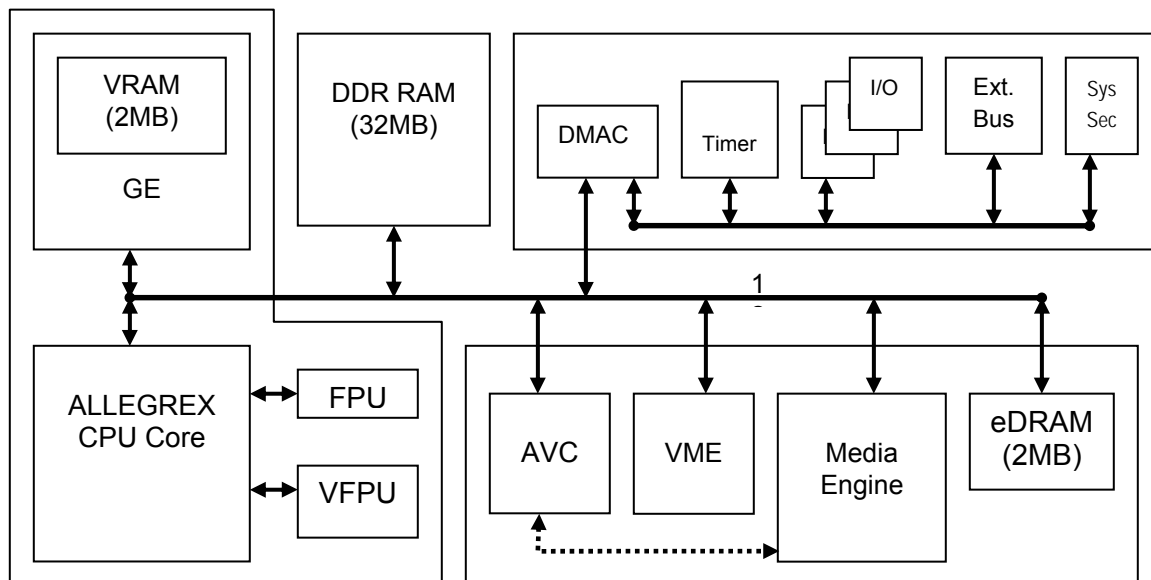
さらに図 2-1 に示したように、PSP ハードウェアは、大きく以下の 3 つのブロックに分けることが可能です。

- CPU/GE ブロック
- メディアエンジンブロック
- I/O ブロック

これら 3 つのブロックと 32MB の DDR メインメモリは、128 ビットのバスで接続されています。

図 2-1: PSP システムブロックの構成

PSP システムブロック図



ALLEGREX CPU コア

ALLEGREX は PSP のメイン CPU です。32 ビット MIPS RISC プロセッサで、標準的な MIPS R4000 と同じ命令セットおよび例外処理を提供しています。また ALLEGREX には、コプロセッサとして独立した FPU と強力な VFPU (Vector Floating Point Unit) も備わっています。命令用とデータ用に 2 つのキャッシュがあり、それぞれのサイズは 16KB です。基本的に標準の MIPS CPU なので、コンパイルされたネイティブの C または C++ コードが問題なく動作します。なお MIPS プロセッサでは、メモリ内のデータをメモリサイズの倍数にアライメントさせる必要があります。これと同じ要件が VFPU にも該当します。VFPU と VFPU で可能な操作については第 5 章で説明しますが、メイン CPU について詳しくは、「参考文献」セクションの[1]に記載したドキュメントを参照してください。

グラフィックエンジン

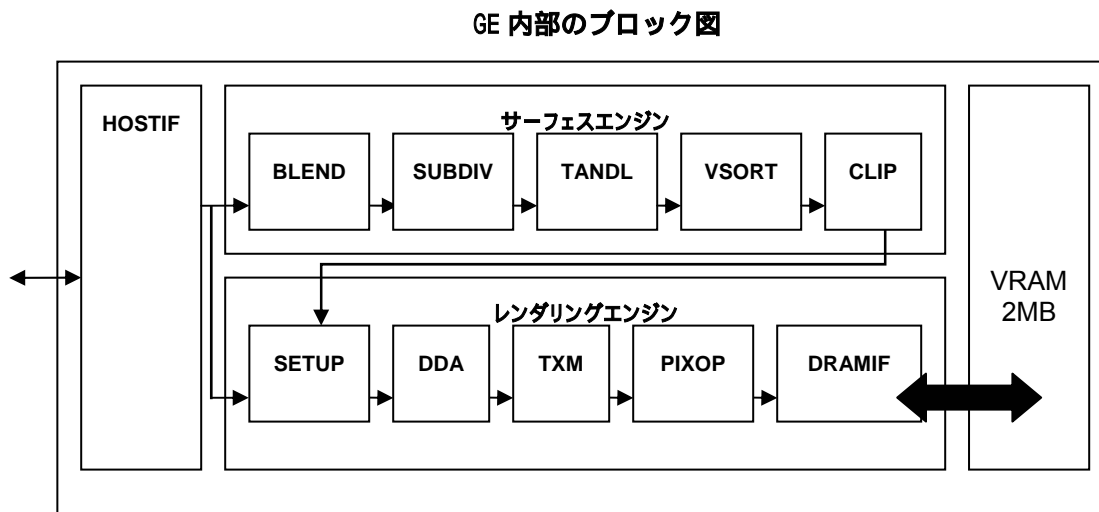
このセクションでは、PSP グラフィックエンジン (GE) について詳しく説明します。GE はカスタムビルトの GPU で、詳しくは本ドキュメントの各所で説明しましょう。

グラフィックエンジンは強力な 3 次元および 2 次元グラフィックチップで、さまざまな演算をハードウェアで実行します。このグラフィックエンジンは、ディスプレイリストからコマンドを受けます。ディスプレイリストは、グラフィックエンジン内のレジスタを設定し、さまざまな状態の変化や実行の開始または停止をグラフィックエンジンに通知して、プリミティブをフレームバッファに描画します。ハードウェアツール用のコードを作成しているときは、独自のディスプレイリストマネージャも作成できますが、前の章で述べたように、この機能は SCE のグラフィックライブラリ libGu に用意されています。libGu について詳しくは、本ドキュメントの後半で説明します。

グラフィックエンジンの内部

グラフィックエンジンは固定機能のグラフィックプロセッサで、数多くの強力なハードウェアアクセラレーション機能を備えています。図 2-2 に示した名前付きの四角形は、グラフィックパイプラインのさまざまな部分を実行する個別の機能ブロックです。

図 2-2: PSP GE 内部ブロックの構成



この図から、「サーフェスエンジン」と「レンダリングエンジン」という 2 つのメインブロックに分かれていることがおわかりだと思います。一般に、これらは頂点とピクセルの演算処理をそれぞれ担当します。ホストインターフェイス (HOSTIF) には、DMA コントローラとディスプレイリストのパarser 機能が含まれています。この HOSTIF

は、ディスプレイリスト内のコマンドを構文解析し、頂点データをメインメモリから取り出して GE のさまざまな部分に渡して処理させます。つまり、ディスプレイリストのコマンドを構文解析することで、レンダリング演算を内部の各機能ブロックに対して調整しています。

また図 2-2 に示されているように、HOSTIF からグラフィックパイプへのパスは、GE 内部に 2 つあります。頂点データは、GE の "サーフェスエンジン" をバイパスして、画面空間にプリミティブをレンダリングすることが可能です。この機能は、GE がサポートしている各種の頂点モードの中で、特に "NORMAL モード" と "THROUGH モード" の 2 つと関係しています。

- NORMAL モードは、サーフェスエンジンのブロックから始まって GE パイプラインの全体で頂点の処理が行われるように指定するためのものです。
- THROUGH モードでは、頂点はサーフェスエンジンをバイパスし、レンダリングエンジンのブロック内で処理されます。したがって、NORMAL モードの頂点は、サーフェスエンジンで処理された後、GE パイプラインの次のステージに備えて THROUGH モードの頂点に変換されます。

これらの形式について詳しくは、次の章で説明します。

サーフェスエンジン内の機能ブロックは、次のとおりです。

- BLEND ブロックは、PSP での頂点のブレンド操作を処理します。必要に応じて、スキニングおよびモーフィングの処理を実行します。
- SUBDIV ブロックは、ベジェおよびスプラインのパッチデータのテッセレーションをすべて管理します。
- TANDL ブロックは、すべての頂点をローカル → ワールド → 視点 → クリップ → 画面 → 描画というように、さまざまな座標系に変換します。
- VSORT ブロックは、三角形の頂点を並べ替えて、三角形の一片の頂点を順序付けするといった操作を実行します。
- CLIP ブロックは、三角形データに対して、クリッピング操作を実行します。

また、レンダリングエンジン内の機能ブロックは、次のとおりです。

- SETUP は、プリミティブについて、すべてのデルタを計算します。これには、パースペクティブ補正テクスチャに対するプライマリ RGBA とセカンダリ RGB のカラー値、フォグデルタ、および UV 計算などが含まれます。
- DDA ユニットは、前のステージからのプリミティブの情報を補間することで、フラグメントの情報を生成します。
- TXM ユニットは、すべてのテクスチャマッピング操作(フィルタリングや Mip マッピングなど)を実行します。
- PIXOP ブロックはラスターライザで、ピクセルをフレームバッファへ書き込みます。2MB の内蔵 VRAM へのインターフェイスである DRAMIF に直接接続しています。PIXOP では、すべてのアルファブレンド操作と、シザリング、ステンシル、アルファテストなどを処理します。

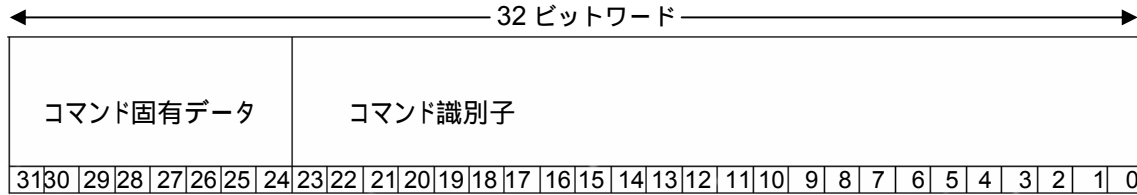
ディスプレイリスト

ここではディスプレイリストのプログラミングについて詳しく説明しませんが、概要だけでも理解しておく、GE の動作や GE と libGu との関係を理解する上で役立ちます。

ディスプレイリストとは、GE のレジスタコマンドを含むメモリの領域のことです。レジスタコマンドは GE の内部命令形式で、チップのレンダリング機能を制御します。これらのコマンドは 32 ビットのフィールドで、状況に応じて各種のビットがセットされています。グラフィックライブラリから関数を呼び出すと、関連するコマンドがディスプレイリストに追加され、そのディスプレイリストが GE に送信されます。HOSTIF は、そのディスプレイリストを構文解析し、レンダリングコマンドを GE 内の関連するセクションに対して調整します。

図 2-3 に、GE コマンドの形式を示します。

図 2-3: GE コマンドのフィールドレイアウト



上位 8 ビット (24 ~ 31) は、コマンド識別子として固定されています。コマンド識別子は、16 進数で 0x00 ~ 0xFF の範囲です。これらのビットで、ヘッダーファイルの *gecmd.h* に定義されている GE 内のコマンドを指定します。

残りの 24 ビットは各コマンドで必要となるデータ用に予約されており、この部分のレイアウトはコマンドごとに異なります。

ディスプレイリストの種類

ディスプレイリストには、以下の 2 つの種類あります。

- IMMEDIATE リストは、作成されると GE に送信されるリストです。
- CALL リストは、いつでも作成、保管、送信することが可能です。

CALL の使用例としては、アプリケーションのレンダリングエンジン内のディスプレイリストの作成と、レンダリングを二重にバッファする場合とがあります。これは "遅延レンダリング" とも呼ばれます。遅延レンダリングでは、あるリストがレンダリングされている間に別のリストが作成されるため、状況によっては非常に効率の良いレンダリング方法となります。

このような CALL リストの使い方は、最適化に関するチュートリアルなどで説明する方が適切ななので、本ドキュメントでは取り上げません。

libGu

libGu は SCE が提供するグラフィック API で、PSP 用のソフトウェアを開発するために使用できます。構文の形式は OpenGL に似ています。OpenGL の知識は必要はありませんが、OpenGL を理解していれば、PSP でのグラフィックのプログラミングが短期間で行えます。将来的には独自のディスプレイリストマネージャを作成することがあるかもしれませんが、本ドキュメントでは、libGu を使用したグラフィックプログラミングの例だけを説明しましょう。

libGu の使用方法のデモ例

libGu の特徴は、1 つの関数呼び出しで数多くのコマンドが設定できるという点です。そのため、開発プロセスを大幅にスピードアップすることが可能です。次に簡単な例として、「参考文献」セクションの[4] で説明されている libGu の `sceGuDrawArray(...)` という関数を考えてみましょう。コマンドリファレンスのマニュアル[16]で、この関数と各コマンドのパラメータを調べてみると、この関数でディスプレイリストに追加されるコマンドがわかります。

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

この関数では、GE に対して頂点の配列を特定の種類のプリミティブ(点、線、三角形など)として描画するように命令し、ディスプレイリストに以下のコマンドを設定します。

- `CMD_PRIM`
GE に対して、描画するプリミティブの種類と、予期される頂点の数を通知します。このコマンドは、*prim* および *count* パラメータに関連して生成されます。
- `CMD_VTYPE`
GE が予期する頂点の種類(*type* で指定)を指定します。テクスチャの座標を予期するかどうかや、頂点のデータのビット深度など、頂点の形式について多くの情報を GE に提供します。
- `CMD_BASE`
常に頂点のアドレスに関連して設定されます。インデックスモードかどうかは関係ありません。
- `CMD_VADR`
GE に対して、頂点のデータのメモリアドレスを通知します。
- `CMD_IADR`
インデックス付きプリミティブが使用される場合に設定されるコマンドです。インデックス付きモードで描画されるときに、GE に対して、インデックス配列のアドレスを提供します。

すべての `libGu` 関数には、"sceGu" というプレフィックスが付けられています。これらの関数のほとんどは、ディスプレイリストの開始から終了までのブロック内で呼び出す必要があります。このブロックは、`sceGuStart(...)` ~ `sceGuFinish()` として呼び出します。詳しくは次の章で説明します。

ディスプレイリストがいっぱいになると、DMA によってグラフィックエンジンに送信され、グラフィックエンジンのホストインターフェイスによってピックアップされます。

ディスプレイリスト内のデータ

頂点またはテクスチャのデータがディスプレイリストにコピーされることはなく、常に、グラフィックエンジンが必要に応じてフェッチするデータへのポインタとして渡されます。ディスプレイリストには、コマンドと小さな値(カラー情報やマトリックスなど)だけが含まれています。

この点は、OpenGL などの他の API とは大きく異なった部分です。OpenGL の IMMEDIATE モードでは、`glVertex3f(...)` などの呼び出しを使用して、頂点を関数内部でローカルに作成することができます。

頂点データはディスプレイリスト自体にはコピーされないため、頂点をローカルに作成することは適切ではありません。ローカルの頂点データは、関数から戻るとスコープ外になります。

このページは空白です。

第 3 章:

簡単なプログラムのコンパイル

このページは空白です。

はじめに

この章では、画面に三角形を 1 つ描画し、さらにテキストをいくつか表示する簡単なプログラムコードを見て行きましょう。この処理には libGu を使用します。libGu の関数呼び出しの一般的な形式について説明しながら、すでに説明した内容を復習し、libGu の機能の一部をもう少し詳しく説明します。

ここに示したコードは、ソースファイルの `hellotriangle.c` から引用したものです。このソースファイルは、本ドキュメントに付属しているサンプルのソースディレクトリにあります。

また、コードそのものは、エミュレータとハードウェアツールの両方の環境で使用できます。違いについては、付録 B を参照してください。

データ構造体

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

この配列は、ディスプレイリスト用の静的記憶域です。それぞれの `sceGu*` 関数が呼び出されるたびに、この配列内のデータが各フレームに対して設定されます。

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

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

`gVectors` はカスタム定義の配列で、三角形の頂点の座標を保持します。この配列では符号なしの単精度整数が使用されていますが、これは、三角形が THROUGH モードで描画されるためです。THROUGH モードの場合、頂点の位置は画面空間内に存在していて、単精度整数型でなければなりません。

初期化

このプログラムのコードの大部分は、`Init()` 関数内にあります。この関数は、グラフィックライブラリを初期化し、メインループで使われるレンダリング状態をいくつか設定します。

```
sceGuInit();
```

この関数は最初に呼び出さなければならない `sceGu*` 関数で、グラフィックライブラリを初期化して、利用可能な状態にします。

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

`sceGuStart(...)` は、ディスプレイリストのエントリポイントを定義します。この関数は、特にハードウェアツールを使用してコードを作成している場合には、数少ない非常に重要なディスプレイリスト関数の一部です。呼び出す `sceGu*(...)` 関数は、いずれもディスプレイリスト開始用の関数 `sceGuStart(...)` とディスプレイリスト終了用の関数 `sceGuFinish()` の間に位置させておく必要があります。

ただし、以下は例外です。

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

これらの例外がある理由は、`sceGuStart(...)` は、`sceGuFinish()` の呼び出し前に、連続したコマンドが追加されるディスプレイリストの開始位置へのポインタを定義するためです。

いま述べた例外を除くすべての `sceGu*(...)` 関数は、ディスプレイリストに対するコマンドを記述し、その後、ディスプレイリストポインタを増加させます。そのため、`sceGuStart(...)` ~ `sceGuFinish()` ペアの外部で `sceGu*(...)` 関数を呼び出すと、管理対象でないアドレスにコマンドが書き込まれてしまいます。

エミュレータでは、上記のように `sceGuStart(...)` ~ `sceGuFinish()` ペアの外部に記述できます。しかし、ハードウェアツールを使用しているときに、このペアの外部にコードを記述すると、予期しない結果になることがあります。

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

この関数は、ディスプレイリストの開始位置、サイズ、および使用するモードを定義します。先に説明したように、後で使用するリストだけでなく、すぐに使用するリストを作成することもできます。サンプルではすぐにリストを使用するため、`mode` パラメータは `SCEGU_IMMEDIATE` に設定されています。

ソースファイルの先頭で宣言されている静的配列は、ディスプレイリストデータのポインタとして渡され、`sizeof` は、ディスプレイリストのサイズを渡すために使用されます。このサイズは、`libGu` による境界確認では使用されず、内部的にはリングバッファとして扱われません。そのため、`libGu` では、ディスプレイリストの終了位置を超えて記述することができます。このように記述すると予期しない結果が発生するため、かならずディスプレイリストのサイズが十分であることを確認してください。

描画バッファとディスプレイバッファ

次の 2 つの関数は、PSP の描画バッファとディスプレイバッファを設定するのに必要な変数を初期化します。関数に渡される引数は、SCE で定義されているマクロです。カラー深度、幅、高さなどのフレームバッファ情報のほかに、バッファポインタの開始アドレスとして指定された VRAM 内のメモリアドレスを記述します。

次の関数は、32 ビットフレームバッファを標準的な幅と高さ(480×272)に設定します。

```
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` に定義されているマクロ `SCEGU_VRAM_BP***` は、フレームバッファのピクセル形式に関するマクロです。これらのマクロは、描画、ディスプレイ、および深度バッファの事前定義済みの VRAM アドレスです。しかし、最初のプログラムでは、深度バッファを含める必要はありません。これについては、3 次元グラフィックプログラミングについて述べた次の章で説明しましょう。

サンプルでは、32 ビットフレームバッファが使用されています。`SCEGU_VRAM_BP32_0` は、`void` ポインタへの `SCEGU_VRAM_TOP` (0x00000000) キャストとして定義されています。このアドレスは、VRAM の開始位置を示すメモリマップされたアドレスで、描画バッファの開始アドレスとして渡されます。

ディスプレイバッファの場合、ポインタの計算は、画面の高さ(272)、`SCEGU_VRAM_WIDTH`(512)、およびピクセルサイズ(バイト単位、4)を乗算して、その値を `SCEGU_VRAM_TOP` に加算します。これで、描画バッファの次に利用可能な領域がわかります。画面の幅は 480 ピクセルありますが、バッファは VRAM の幅である 512 で計算されます。これは、バッファの幅は 128 または 256 バイト単位(ピクセルのカラー深度が 16 か 32 かによって異なる)で指定しなければならないためです。どちらのカラー深度の場合も、ディスプレイバッファの幅は 512 に設定する必要があります。

ビューポートの初期化

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

`sceGuViewPort(int x, int y, int width, int height)` は、ビューポートを定義するための関数です。これは、クリップ座標空間から画面空間へのビューポート変換です。クリップ座標系の *width*、*height*、*x*、および *y* は、画面座標系での各値と等しくなります。たとえば、最初の 2 つのパラメータ(*x* と *y*) は、クリップ座標系の原点になる画面空間内の位置を定義します。しかし、クリップ空間の原点は 0, 0 です。ここで、疑問が生じます。なぜ、コード内では 2048, 2048 と定義されているのでしょうか。

理由は、画面空間の座標系全体が、*x*、*y* とともに 0~4096 の範囲になるからです。したがって、2048, 2048 は画面空間の中心になります。つまり、クリップ空間内のオブジェクトは、画面座標に正しく変換されます。

`sceGuOffset(int x, int y)` は、"描画空間" への最後の変換です。描画空間は画面空間の一部で、*x* と *y* が 0~1023 の範囲に当たります。次の例は、*x* 座標についての画面空間から描画空間への変換を示したものです。

```
Xd = Xs - OFFSETX
```

x の原点は、画面空間へのビューポート変換後に 2048 で定義されますが、これと同様の方法で、画面空間での *x* の中心(2048)は、描画空間での *x* の中心になります。

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

また、`sceGuScissor(int x, int y, int width, int height)` は、シザー領域を定義します。シザーテストとは、あるフラグメントがラスターライズされるかどうかを判断する最初のピクセルテストです。この領域内のピクセルだけが描画されます。

初期化セクションのファイナライズ

`sceGuColor(unsigned int col)` は、描画するカラーを設定します。これは、頂点ごとの明示的なカラーを持たないジオメトリに適用可能な、頂点の定カラーと見なすこともできます。そのため、大きいモデルをレンダリングしていて、オブジェクト全体で頂点のカラーが変化しない場合、この関数を使用して、カラーデータを取り除くことができます。ただし、このカラー値は、明示的に指定された頂点のカラーで上書きされます。

`sceGuFinish()` は、IMMEDIATE モードの場合は描画の終わりを通知するコマンドをリストに追加し、CALL モードの場合は return コマンドを追加します。

`sceGuSync(int mode, int block)` 関数は、渡されたパラメータに応じて描画処理を同期化します。サンプルでは、GE がレンダリングを終えるまで待つように指示しています。

初期化処理の最後の関数は、V-Blank に関する関数です。これは、エミュレータとハードウェアツールで使い方が異なる数少ない関数の 1 つです。この関数は、処理を続ける前に、次の垂直ブランクの開始まで待機するだけです。エミュレータとハードウェアツールの違いについて詳しくは、付録 B を参照してください。

メインループ

初期化後、このプログラムのメイン機能として、以下の 3 つの関数を単純にループ処理しています。

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

先に説明したように、すべての `sceGu*` 関数は、一部の例外を除いて、`sceGuStart` ~ `sceGuFinish` のペアの間に記述しなければなりません。サンプルでは、これらの関数を `StartFrame` と `EndFrame` の間に記述しています。

`sceGuClear(int mask)` は、VRAM 内の関連バッファをクリアします。この関数は、渡されるマスクに応じて、カラー、深度、ステンシル、もしくは、これら 3 つ全部でクリア対象を調整します。これらは `libgu.h` に定義されています。サンプルの場合にはカラーバッファだけをクリアするので、`mask` は `SCEGU_CLEAR_COLOR` に設定されています。

`sceGuDebugPrint(int x, int y, unsigned int col, const char* str)` は、メッセージを画面に出力します。引数 `x` と `y` は、テキストの開始位置を表す描画空間の座標です。`col` は 32 ビットフィールド(8 ビット/コンポーネント)で定義されたカラーで、`str` は画面に出力する文字列です。サンプルの場合には "Hello triangle" になります。

`sceGuDrawArray(int prim, int type, int count, const void *index, const void *p)` は、先に GE コマンドにデコンストラクトされた関数です。この関数は、`libGu` 内のほとんどのプリミティブを GE にレンダリングさせる標準インターフェイスです。GE に対して、描画するプリミティブの種類、頂点の形式で想定されるデータ、レンダリングされる頂点の数、およびインデックス付きプリミティブをレンダリングするかどうかを通知します。GE がフェッチする頂点自身へのポインタと、このモードが使用されている場合は、インデックスへのポインタを渡します。

サンプルのようにディスプレイリストが IMMEDIATE モードの場合、GE はディスプレイリストを直接解析して、プリミティブをレンダリングします。

サンプルでは、頂点が `short` 型であること、および `SCEGU_VERTEX_SHORT` の場合は THROUGH モードで、`SCEGU_THROUGH` の場合は論理和を取って頂点をレンダリングしなければならないことを GE に命令しています。このパラメータは `type` として渡されます。また、三角形を描画するサンプルなので、`prim` の種類としては `SCEGU_PRIM_TRIANGLES` を指定し、描画する頂点の数を 3 と定義します。

頂点の形式と、グラフィックエンジンでの解釈

「参考文献」セクションの [3] に説明されているように、GE では、順序固定の頂点の形式を受け入れます。頂点の形式内の要素の順序についても、同じ [3] で説明しています。グラフィックエンジンで頂点を定義するために最低限必要なのが位置ベクトルです。先に説明したように、`sceGuDrawArray(...)` の `type` フィールドは、グラフィックエンジンに何を予期するかを通知するために使用します。これはビットフィールドの論理和でグラフィックエンジンによって解釈されて、頂点データ内のオフセットが調整されます。つまり、頂点のさまざまな要素の順序が固定であっても、何を頂点として定義するかについては柔軟に指示できることになります。

記述しなければならない最後の関数呼び出しは `sceGuSwapBuffers()` です。この呼び出しで、描画バッファとディスプレイバッファをスワップし、画面にレンダリングしたフレームを表示します。

インデックス付きとインデックスなしのプリミティブ

サンプルのように `sceGuDrawArray(...)` を呼び出すときは、ここで説明した以外の別のポインタ引数 `const void *index` を渡すことができます。このパラメータはインデックスのリストへのポインタで、インデックス付きプリミティブでレンダリングするのに使用します。

インデックス付きプリミティブは、配列内の特定のインデックスを基にして、レンダリングする頂点をフェッチする方法です。この方法だと、頂点キャッシュを持つハードウェアで帯域幅を減らすことが可能ですが、GE には頂点キャッシュがないため、インデックス付きプリミティブを使用することはお勧めしません。大きな理由は、GE は "プル" する種類のチップであり、インデックスをインデックスポインタからフェッチして、対応する各頂点を頂点ポインタからフェッチするのが特徴だからです。パスのアクティビティが増えてしましますが、パフォーマンスに影響が出ないように最小限に抑えられています。

まとめ

この章では、簡単なグラフィックのプログラムを説明しました。描画に不可欠であるレンダリングコンテキストの初期化方法や、テキストを表示して三角形を表示するための単純なレンダリングループを設定する方法についても説明しました。また、これらを実現するために使用している libGu コマンドについてもすべて説明しました。

この章では 2 次元グラフィックのプログラミングを紹介したので、次の章では 3 次元グラフィックのプログラミングを説明し、さらに、もう 1 つのライブラリである libGum も紹介しましょう。次の章のサンプルでは、本章で説明したサンプルコードを基して、ライティングを使った単純なシーンを作成します。また、コントローラ入力についても説明し、サンプルプログラムを使用してユーザーが PSP をインタラクティブに操作できるようにしてみましょう。

このページは空白です。

第 4 章:

簡単な 3 次元 シーン

このページは空白です。

はじめに

この章では、本ドキュメントに付属する 3 次元シーンのサンプルを作成するコードについて説明します。このコードは、3dscene.c というファイルに用意されています。

このサンプルでは、フロア平面、2 個のドーナツ、およびライトがある単純な 3 次元シーンを作成します。シーン内のオブジェクトは、GE ハードウェアのパッチテッセレーションでレンダリングされます。またサンプルでは 3 次元効果を生成するため、libGu のセットアップ呼び出しへの追加が含まれます。

さらに、本ドキュメントの「はじめに」で述べたように、libGum と呼ばれる追加ライブラリを使用して、コントローラライブラリを利用したコントローラ入力というかたちで、ユーザーとのインタラクティブ性を追加しましょう。

この章の目的は、サンプルで使用している関数の背後にある理論を説明することです。たとえば、PSP のライティング関数の一部についての説明では、関数とハードウェアの関係を深く理解できるように、GE が遵守しているライティングモデルについて簡単に説明しています。これらの情報を知ることによって、PSP のグラフィック機能とその実装方法の概要が理解できます。

深度バッファ

PSP GE には、16 ビット固定のハードウェア深度バッファ (Z バッファ) があります。深度バッファが占める VRAM 領域は自動的に設定されないため、明示的に定義しなければなりません。この定義は、描画バッファやディスプレイバッファが占める VRAM 領域を定義する場合と同じ方法で行います。

具体的には sceGuDepthBuffer(void *zbp, int zbw) を使用しますが、サンプルの場合、zbp は SCEGU_VRAM_BP32_2 になります。libgu.h では、32 ビットに設定されている場合、描画バッファとディスプレイバッファ両方の後にある次の空きメモリ領域の先頭として定義されています。

深度バッファの幅は 16 ビットなので、128 の倍数として定義しなければなりません。そのため zbw は、マクロ SCEGU_VRAM_WIDTH で 512 として定義されます。

Z バッファは主に隠面消去に使用され、ピクセルをテストし、そのピクセルがすでに表示されているピクセルの背後にあれば除外します。Z バッファは、さまざまなテストを実施して、ピクセルを描画するか除外するかを選択します。サンプルでは、sceGuDepthFunc(SCEGU_EQUAL) で設定された深度の値が深度バッファ内の値以上であれば、ピクセルを描画するように設定しています。

深度の範囲を設定するには、次に初期化が必要です。ここで設定された値は、ビューポート変換でマップされた深度範囲になります。この範囲外のピクセルは、どれもラスタライズされません。Z バッファは、各ピクセルの z 値の比較結果を基にして、フラグメントを渡すか拒否します。深度範囲を非常に小さく設定すると、ピクセル深度がかなりばらつくため、ピクセルがラスタライズされるときに Z テストでの悪影響が発生してしまいます。そのため、サンプルでは、sceGuDepthRange(int nearz, int farz) を使用して、深度範囲を near z の値はデフォルト設定の 65535 に設定し、far z の値は 0 に設定しています。

sceGuClearDepth(int depth) は、Z バッファが消去されるときに GE が設定する値を設定します。

最後に、sceGuEnable(SCEGU_DEPTH_TEST) で深度テストを有効にします。

3 次元座標系と libGum

「参考文献」セクションの [3] で説明されているように、GE では、パイプラインで 6 種類の座標系を使用してプリミティブを画面に描画します。前の章では、プリミティブを THROUGH モードでレンダリングする方法を示しました。THROUGH モードでは、描画空間で指定した頂点が必要です。

この 3dscene サンプルでは、完全な 3 次元変換セットを使用します。そのため、6 種類すべての座標系を使用して、画面にレンダリングされる 3D プリミティブを変換します。PSP のプログラミングでは、変換パイプラインを通じて、手動でプリミティブを変換する必要はありません。これらの機能は GE がすべてハードウェアで処理します。これらの機能にアクセスするには、libGu および libGum を使用します。

利用可能な行列は、パースペクティブ、ビュー、ワールド、およびテクスチャの 4 種類です。

グラフィックエンジンのユーザーマニュアルで説明されているように、パイプラインでは、これら 4 つの行列によって、オブジェクトを 3 次元の空間から 2 次元の画面にレンダリングすることができます。これらの行列の機能は、次のとおりです。

- **ワールド行列**は、オブジェクトをローカル空間からワールド空間へ変換します。
- **ビュー行列**は、オブジェクトをワールド空間からビュー空間へ変換します。
- **パースペクティブ行列**は、オブジェクトを視点空間からクリップ空間へ変換します。
- **テクスチャ行列**は、射影されたテクスチャなどの効果に使用します。射影されたテクスチャについては、本ドキュメントでは扱いません。

libGum

基本的に、libGum [4] は行列マネージャ兼数式ライブラリで、行列変換やスタックなどの便利な関数が利用できます。OpenGL を理解していれば、OpenGL と libGum の構文には共通点があることがわかりだと思えます。行列変換関数をまとめると、以下のようになります。

- `sceGumTranslate(...)` 変換操作を適用します。
- `sceGumScale(...)` 拡大縮小操作を適用します。
- `sceGumRotate*(...)` x、y、または z、あるいはこれらすべてに回転を適用します。
- `sceGumLoadIdentity(...)` 行列を単位行列にリセットします。

これらの関数は、現在の行列に対する連続変換として機能します。そのため、関数が呼び出されると、次の変換は現在の行列に乗算されます。これらの行列は列順です。つまり、演算は後ろから乗算されるため、変換の順序は、後ろから前へと考える必要があります。

スタック機能は、変換を維持して後で復元しなければならないときに役立つ場合があります。一部のオブジェクトは基本変換を共有できますが、さらに個別の変換も必要です。たとえば、基本変換はスタックをプッシュしてから、各オブジェクトが専用の変換を追加して、次のオブジェクトのために基本変換を復元できます。この方法では、各オブジェクトの基本変換の処理を保存することができます。この簡単な例については、後で 3dscene のサンプルでデモしましょう。

PSP 上の行列スタックは、 4×4 の行列からなる配列を最初に宣言することで初期化します。理論上、ここで宣言できる数に制限はありません。スタックの行列を設定する関数は、次のとおりです。

```
sceGumSetMatrixStack( ScePspFMatrix *m,
                      int proj,

                      int view,
                      int world,
                      int tex
                      );
```

通常、パースペクティブ行列はアプリケーションごとに 1 回設定し、ビュー行列はフレームごとに 1 回設定します。このサンプルでは、アプリケーション中にパースペクティブ行列を変更しないため、初期化の段階で 1 回設定されています。

`sceGumMatrixMode(int mode)` は、現在の行列スタックを設定します。続く libGum の行列関数は、すべてスタックの最上位にある行列に対して変換を適用します。その結果が現在の作業行列となります。

サンプルでは、現在の行列を射影行列になるように設定し、`sceGumPerspective(float fovy, float aspect, float near, float far)` を呼び出します。この関数は、パラメータを使用してパースペクティブ行列を作成し、現在の作業行列に掛け合わせます。

GE では、フラットシェーディングとグローシェーディング(三角形で補間された頂点ごと)が可能です。フラットシェーディングとスムーズシェーディングのどちらを使用するか指定するには、関数 `sceGuShadeModel(int model)` を使用し、`model` パラメータとして `SCEGU_FLAT` または `SCEGU_SMOOTH` を指定します。

PSP では、ライティングはハードウェアで処理されます。libGu を介してアクセスできるハードウェアライティングは、最大で 4 つあります。次のセクションでは、ハードウェアライティングを設定する方法、およびそのプロパティとマテリアルについて説明します。

ライティング

すべてのシェーディングの計算では、フラットかグローかにかかわらず、ライティングの式の一部として法線が必要です。フラットサーフェスの場合、面法線が必要です。またグローシェーディングの場合には、頂点ごとの法線が必要になります。このサンプルでは、パッチを使用して、ジオメトリをレンダリングしています。この関数を使用すると、GE は、すべての法線を自動的に生成します。ライティングは頂点単位で行われるため、十分な結果を得るには、適切に高度にテッセレートされたメッシュが必要です。

GE のライティングモデルでは、環境(ambient)、拡散(diffuse)、鏡面(specular)、放射(emissive)という 4 つのパラメータを基に、ライティングを計算します。

- "環境" ライティングは、基本的な環境のライティングと見なされます。このライトは、シーンから大きく広がって反射されます。つまり、全方位性となっていて、そのためフラットに見えるライトです。
- "拡散" ライティングは指向性があり、特定のソースから計算されます。拡散ライティングは、外見上はマットで、どのビュー角度からでもオブジェクト全体で同じように見えます。定義は、拡散ライティングはビュー独立であるということです。
- "鏡面" ライティングは、拡散ライトと同様に指向性があり、そのライトが照らすオブジェクトから特定の方向に反射されるように見えます。鏡面ライティングは、サーフェスにキラキラと光る外観を加える 1 つの方法です。鏡面ライティングは、ライトの位置、サーフェスの法線、およびカメラのビューポイントによって計算されます。オブジェクトやカメラが移動すると、ライティングはサーフェスを移動しているように見え、"ビュー依存" と呼ばれます。
- 式の "放射" 要素とは、オブジェクトの内部が輝いているかのように、オブジェクト自体から発せられるライトであると見なされます。これは、サーフェスのライトに対して全体的に追加されますが、別のサーフェスによってピックアップされたライトを実際に放出するわけではありません。

これらすべての要素は、サーフェスのマテリアルのプロパティと組み合わせられます。サーフェスのマテリアルのプロパティについては後述します。

GE では、最大で 4 個の頂点ライトをハードウェアで計算します。これらの各ライトは、指向性(directional)、点(point)、またはスポットライト(spotlight)にすることができます。すべてのライティング機能は、libGu からアクセスできます。そのため、ライティング効果を加えるのは非常に簡単です。ライティング効果により、レンダリングしたシーンのリアルさが増し、奥行きが加わります。

このサンプルでは、単体の拡散および鏡面の点のライティングを設定して、シーンを照らします。ライトのカラーは、`sceGuLightColor(int n, int type, unsigned int col)` を使用して設定されます。この場合、`n` は 0 ~ 3 のライト識別子番号、`type` はライトの式の構成要素で ambient、diffuse、specular のいずれかです。また、`col` は各コンポーネント用に定義されたカラーです。

`sceGuSpecular(float power)` は、ライトの鏡面指数を設定します。これは、レンダリングされるサーフェスのマテリアルのプロパティで輝度の値を設定するのと同じです。値を大きくすると、サーフェスの輝きが増

し、鏡面ライトは狭く、より収束した状態になります。このサンプルでは、すべてのマテリアルで同じ輝度値が設定されていると仮定しています。

拡散および鏡面ライティングに必要なのは、ライトの位置とビュー位置です。ビュー位置は、本ドキュメントで後述しますが、フレーム単位で設定されます。サンプルではライトを移動させる必要がないため、位置は、初期化段階で関数 `sceGuLight(int n, int type, int comp, const scePSpsFVector3 *vec)` を使用して設定されます。またサンプルでは、`SCEGU_LIGHT_POINT` でライトの `type` を定義しています。提供されるライティングの種類は、`comp` パラメータを使用して `SCEGU_DIFFUSE_AND_SPECULAR` に設定されます。ライティングが正しく機能するには、この関数に渡される位置ベクトルがワールド空間で定義されている必要があります。このベクトルは `*vec` として渡されます。

最後に、ライト "0" を `sceGuEnable (SCEGU_LIGHT0)` で有効にします。ライティングそのものは、`sceGuEnable(SCEGU_LIGHTING)` の呼び出しを使用することで、残りのレンダリングすべてで有効となります。以上で、サンプルプログラムの初期化は終わりです。

パッチ

このサンプルでは、GE の強力な機能であるハードウェアパッチテッセレーションが使用されています。以下のセクションでは、関係のあるベジェおよびスプラインサーフェスについて簡単に説明し、これらを使用する利点について述べましょう。ただし、スプラインとその使用についての数学的なチュートリアルが目的ではありません。数学に関連した情報については、巻末の「参考文献」セクションの [5] に説明されています。また、参考文献リストの [3] にも、GE がパッチサーフェスを計算するための方法について数多くの情報が記載されています。

GE ではベジェとそれ以外のスプラインサーフェスの両方をサポートしていますが、これらは両方ともパラメトリックサーフェスの一種です。一般に、パラメトリックサーフェスは、サーフェス計算の基底を成す一連の制御点で定義されます。ただし、サーフェスの計算で使用される式によっては、サーフェス自体がこれらの制御点を通過する必要はありません。

サーフェスの生成に使用される制御点は、一種のクレードル、または外端であるメッシュ (生成されるサーフェスの低解像度表示) と考えられます。GE では、一連の制御点で補間して、レンダリングされる三角形のサーフェス表示を生成します。この方法では各三角形の頂点データを GE に与える必要はなく、バス上で転送されるデータ量を大幅に削減できます。GE は、パッチにライティングを当てるのに必要な法線を計算するため、法線を頂点の形式に含める必要もありません。三角形のテクスチャ座標は、制御点ごとに提供されるテクスチャ座標を介して、サーフェス全体で補間されます。

サンプルでは、制御点の位置ベクトルの一部を初期化します。平面、球面、およびトーラスを記述する位置が作成されます。

ハードウェアツール用のコードでは、スプライン初期化を行うそれぞれの関数の最後に、関数 `sceKerneDcacheWritebackAll()` の呼び出しを行っています。この関数で、先に進む前にデータキャッシュ (D キャッシュ) 内のすべてがメイン RAM に書き戻されることを確認します。一般に、動的データを使用するときなど、この処理が必要な場合があります。理由は、`sceGuDrawArray(...)` などの `libGu` 関数は、メイン CPU に対して非同期な GE への DMA 転送を開始するためです。GE は指示されるとすぐに頂点データのフェッチを開始するので、初期化されたデータがまだ D キャッシュ内に存在していて準備できていない可能性がある頂点の描画を開始してしまう可能性があります。その結果、頂点が間違った場所にあるなど、奇妙な現象が発生してしまったり、画面外に生成されてしまうこともあります。

すべてのデータを書き戻す必要のない方法で D キャッシュを管理できます。しかし、D キャッシュの追跡は複雑な作業なので、ここでは説明しません。キャッシュ機能について詳しくは、「参考文献」セクションの [17] に記載されています。

コントローラの処理

このセクションでは、コントローラ入力データにアクセスする関数について詳細に説明します。この点では、ハードウェアツールとエミュレータの間には重要な違いがあります。つまり、同じ API を共有しません。このサンプルと次のサンプルでユーザからの入力を処理するコードは、すべて関数 `ReadPad()` に含まれています。次に、この `ReadPad()` に含まれるプラットフォーム依存コードについて説明します。

エミュレータ用のコントローラ入力

エミュレータがコントローラ用のコードを処理する方法は非常に簡単で、エミュレータライブラリに含まれています。そのため、特定のヘッダーをインクルードする必要はありません。サンプルで使用されている関数を次に示します。

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

この関数は、ボタンの状態とアナログスティックの動きについて情報を収集します。押されたボタンの情報を含むビットフィールドとして、符号なし整数を返します。`unsigned char* buffer` パラメータは、4 つの符号なし 8 ビット整数からなる配列へのポインタです。`sceEmuAnalogRead` は、アナログスティックについての情報をこれらの整数に代入します。サポートされているアナログスティックは最大で 2 つです。サンプルでは左手用のアナログスティックを入力に使用し、そのデータは、*x* と *y* について、`sceEmuAnalogRead` が戻った後に配列要素の最後の 2 つに格納されます。

ハードウェア用のコントローラ入力

ハードウェア用のコントローラ処理コードは別の API であるため、ヘッダーを別途インクルードする必要があります。

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

コントローラライブラリは、動作中のデジタルボタンを区別するように設定することも、アナログスティックをサービスに含めるように設定することもできます。これは、アプリケーションで必要なければアナログスティックを処理しないようにするためです。サンプルでは、次の呼び出しで、コントローラサービスライブラリを初期化します。

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

サンプルでは、アナログとデジタル両方のコントロールの使用をデモしています。そのため、関数に渡されるパラメータは `SCE_CTRL_DIGITALANALOG` になります。ハードウェアソースコードの場合、`ReadPad()` 内部では、次の関数で、コントローラの状態について必要な情報を収集します。

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

`SceCtrlData` データ構造体は、コントローラ入力パラメータを判断するのに必要な情報を格納するように設計されています。

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

`sceCtrlReadBufferPositive(...)` の呼び出しは、このデータ構造体に値を代入します。次に、ボタン情報を抽出するために、この結果に対して適切なビットマスクで AND を使用します。

アナログスティックのデータはそれぞれ 8 ビット以内で符号化され、x 軸と y 軸の情報は、構造体の Lx および Ly 要素に代入されます。これには正負の情報も含まれ、軸ごとに 0 ~ 127 が負値、128 ~ 255 が正值というように、unsigned char で符号化されます。サンプルでは、このデータを -1.0 ~ +1.0 の範囲に変換しています。これは、コントローラ入力を読み取る標準的な方法です。

一般に、アナログコントローラには一定の不正確さが現れるため、この例では、中心から 30% の移動が不感帯として設定されています。この不正確さが原因で、コントローラのリリース後に特定の方向に固定されたように見える "浮動動作" が起こることがあります。

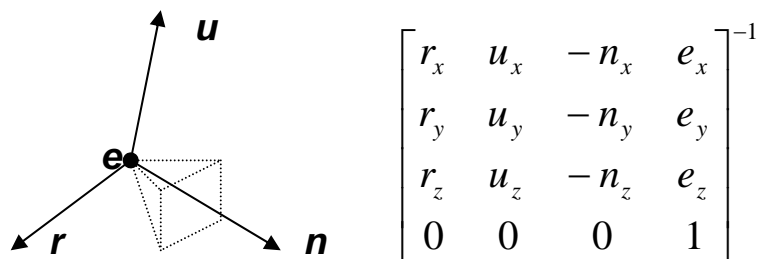
この場合、ユーザのインタラクションによって、カメラがシーンを回転させるために使用する回転角が増えるだけです。これについては、レンダリンググループ内のビュー変換を扱う次のセクションで説明します。

ビュー変換: 簡単なカメラ

各フレームで最初に行うことは、ビュー変換を設定することです。先に説明したように、鏡面ライティングでは、ビュー位置を設定する必要があります。これは、ビュー行列内で符号化されます。このサンプルでは、ビューは移動されているため、各フレームで設定する必要があります。ビュー行列は、カメラの情報を表す 4 つのベクトルで構成されています。最初の 3 つのベクトルは、3 つの単位長の正規直交ベクトルの組として定義されたカメラ "基底" です。これは、"正規直交基底" と呼ばれます。この基底は、行列の回転部分を構成します。4 番目のベクトルは、ワールド座標でのカメラの位置です。

ビュー行列を簡単に定義するには、回転要素と変換要素だけで構成されたワールド行列の逆行列を求めます。この行列に拡大縮小操作を導入すると、行列が正規直交でなくなるため、操作に失敗します。

図 4-1: カメラとビュー行列の関係



まず、`sceGumMatrixMode(...)` が呼び出されて、現在の作業行列として `SCEGU_MATRIX_VIEW` が設定されます。これは、次に単位行列に設定されます。その後、シーンから離れたところとシーンの付近では、`sceGumTranslate(...)` および `sceGumRotate(...)` を使用して、それぞれ変換関数および回転関数が必要な分だけ適用されます。渡される回転の値は、コントローラからのユーザー入力によって変更される変数です。

この行列を `sceGumStoreMatrix(ScePspMatrix4 *m)` で取得して、逆行列にします。libGu にも libGum にも、ネイティブの逆行列関数はないため、ここで指定されています。GE で使用されるビュー行列の内部形式は 4×3 なので、逆行列関数では、 4×4 行列の 16 個の要素のうち 12 個を逆数にするだけで済みます。

この行列は、逆行列になると、カメラが回転する原点を一定の距離から眺めるカメラを表します。次に、`sceGumLoadMatrix(ScePspMatrix4 *m)` は、ビュースタックの最上位の行列として、この行列をロードし直します。これは、ビュー行列のカメラをすばやく簡単に定義する方法です。

このフレームの残りの部分では、モデル行列による操作が使用されます。これは、現在の行列として設定され、単位行列となります。ライティングは有効ですが、ライト表現を描画するために、レンダリンググループの終わりの方ではライティングが無効になります。

このシーンで最初にレンダリングされるオブジェクトは、フロア平面です。

マテリアル

マテリアルのカラーは、全体的なライティングの式の別の部分を形成します。ライティングの式は、ライティングが有効なときに頂点の最終的なカラーを決定します。ライティングの完全な式については、グラフィックエンジンのユーザーマニュアル [3](p.56)で説明されています。同じ情報を再掲するのを避けるため、ここでは、マテリアルのカラーが各レベルの式で乗算されていることだけを説明します。つまり、拡散ライティングコンポーネントは、マテリアルの拡散コンポーネントが乗算され、鏡面ライティングコンポーネントは、マテリアルの鏡面コンポーネントが乗算されるといった具合です。一般にマテリアルは、単に、レンダリングされるオブジェクトのカラーであると考えることができます。しかし、これらのマテリアルパラメータを使用すると、繊細な効果を得ることができます。詳細については、この章で後述します。

式 4-1: ライトとマテリアルによる頂点のカラーに対する寄与

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

式 4-1 は、ライトとマテリアルが頂点のカラーに対して寄与する完全な式を示しています。ここでは、単一のライトが定義されています。複数のライトの場合、式は、各ライトにおけるライトとマテリアルの寄与を合計したものになります。

関数 `sceGuMaterial(int type, unsigned int col)` は、マテリアルのカラーを連続する頂点に設定するために使用されます。`type` は `SCEGU_AMBIENT`、`SCEGU_DIFFUSE` または `SCEGU_SPECULAR` で、`col` は 32 ビットのカラーフィールドです。

パッチ

フロア平面は、GE のハードウェアスプラインサーフェス機能を使用してレンダリングされます。レンダリング関数で、`libGum` で変換する適切な行列を使用するために、`libGum` には、`libGu` から各 `sceGuDraw*()` 関数を複製しています。

```

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

```

この関数には、先に説明した `sceGuDrawArray(...)` と同じパラメータの `type`、`*index`、`*p` があります。一般に、パラメトリックサーフェスは 2 次元表記で定義されます。2 次元テクスチャ座標の変換と同様に、これらは `u` と `v` の範囲として定義されます。`ucount` と `vcount` は、各方向におけるそれぞれの制御点の個数です。この例では、 10×10 ベクトルの配列が制御点のグリッドとして設定されているため、`ucount` と `vcount` はそれぞれ 10 になります。

`uflag` および `vflag` パラメータは、パッチが構成される方法に関連があります。これらの値は、B スプラインサーフェスの計算で重要なノットに関係しています。ノットは、`u` および `v` の開始と終了の両側が開いているか閉じているかに設定できます。この効果は、両端が開いている状態に設定されている場合、パッチはベジェパッチのように機能します。B スプラインサーフェスでは、小さい領域を覆う場合でも、制御点あたりの三角形が多く生成されます。大きい領域の B スプラインサーフェスの作成する場合には注意してください。状況に応じてテッセレーションを管理しなければなりません。そうしないと、パフォーマンスに影響を与えるおそれがあります。

パッチレンダリングハードウェアでは、`u` と `v` の各方向で、1 ~ 64 のサブディビジョンレベルが可能です。値を大きくするとより滑らかなジオメトリが生成されますが、多くの処理を必要とするため、結果としてレンダリングにかかる時間が長くなります。パフォーマンスと品質のバランスをどのように取るかを決定する必要があります。サンプルでは、`sceGuPatchDivide(int udiv, int vdiv)` を使用して、フロア平面とトーラスで別々にサブディビジョンレベルを設定しています。

ベジェと B スプラインのパッチを比較した場合の長所、短所、特異性、および数学的表記についての詳細は、グラフィックエンジンのユーザーマニュアル [3] および [5] のテキストに載っています。

行列変換

ワールド変換の次のセットでは、この章で前述した行列スタックを簡単にデモしています。

トーラスに設定された制御点は、モデル空間内にあり、原点の周囲にあります。この例では、両方の制御点で基底変換行列を共有します。これは、`sceGumTranslate(&offset)` を呼び出して設定されます。描画される最初のトーラス面は、後ろから乗算された回転を提供し、モデル空間内のまま回転させることができます。この回転により、トーラス面は水平方向に横たわっているように見えます。この回転はトーラス #2 で使用される基底変換に干渉するため、行列は、最初にスタックに格納されて維持されます。次に、以下の関数を呼び出して、トーラスがレンダリングされます。

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

この関数は、スプラインの関数と構文が同一です。ただし、パッチが開いているか閉じているかを指定する必要はありません。これで、`sceGuPopMatrix()` の呼び出しにより、保存されている行列がスタックの一番上に復元されます。

2 番目の行列は、1 番目の行列による環を回る軌道で回転させます。この変換では、原点からの一定の距離をもう一方にぴったり合うように変換します。これにより原点からオフセットされるため、これに回転を適用すると変換が完了します。これらは後ろから乗算されるため、変換は、逆順で宣言しなければならないことに注意してください。こうして、トーラスがレンダリングされます。

先に説明したように、オブジェクトのマテリアルを使用してオブジェクトの表示を制御でき、適切に使用すれば、魅力的な効果を生み出すことができます。トーラスの例では、鏡面の値が、非常に暗いグレーとして宣言されていました。これはライトの鏡面の値に乗算されるため、最終的な鏡面コンポーネントの強度が抑えられ、適切なつやつやした仕上がりが生成されます。

シーンで最後のオブジェクトは、ライトの表現です。複雑な物体ではなく、単純な白い球体を使用されます。これを実現するため、ライティングを無効にし、描画するカラーを白に設定しています。これにより、GE は頂点のカラーをライティング経由で計算せずに、一定の描画カラーを使用するようになります。モデル行列は単位行列にクリアされ、レンダリング前に、ライトの位置が変換として適用されます。これで、3dscene サンプルは終わりです。

まとめ

この章では、3 次元の概要をサンプルコードで説明しました。深度バッファとビュー行列を設定し、もう 1 つのライブラリ libGum を導入しました。ライティングとパッチテッセレーションという PSP のハードウェア機能を紹介し、ほんの数行による簡単な 3 次元シーンをデモしました。次の章では、VFPU を紹介し、現在のサンプルを VFPU アセンブリコード例で拡張する方法を説明します。

第 5 章:

VFPU (Vector Floating Point Unit)

このページは空白です。

はじめに

前の章で説明したソースコードには、ハードウェアツールとエミュレータの両方で互換性がありました(一部の例外を除く)。これは、本ドキュメントの構成上、意図的に決定したものです。しかし、VFPU はエミュレートされません。そのため、この章で説明するソースコードは、エミュレータには対応しません。

この章では、VFPU の概要と、VFPU とシステムのほかの部分との関係について説明します。また、VFPU を使用するアプリケーションと、アプリケーション内で VFPU を初期化する方法についても説明します。そして、前の例を拡張する簡単な数学関数で、VFPU がどのように使用されるかについて説明します。この例は、VFPU のコードを記述し始めるのに役立ちます。

アセンブリ言語

PSP VFPU は、C や C++ のような高水準言語内でプログラミングすることはできません。つまり、プログラマは、VFPU アセンブリ言語命令セットを使用しなければなりません。本ドキュメントでは、従来のアセンブリ言語について、少なくとも入門レベルの知識があることを前提としています。「参考文献」セクションの [6] と [7] は、関連アセンブラのチュートリアルや説明が記載されている参考書です。この章では、各関数に存在する特定のアセンブラ演算コードだけを説明し、gcc インラインアセンブラ構文などのトピックについては、明示的に説明しません。この情報は付録 A で説明しています。

VFPU

VFPU は強力な 128 ビット Vector Floating Point Unit (ベクトル型浮動小数点演算ユニット) です。ALLEGREX CPU のコプロセッサとして動作し、行列やベクトルの演算などの数学関数に適しています。VFPU には 128 個の 32 ビットレジスタが含まれており、すべてをプログラムで使用できます。これらは、「行列レジスタファイル」と呼ばれ、「**行列レジスタ**」セクションで後述します。

VFPU が動作できるのはコプロセッサモードだけなので、ALLEGREX と並列実行させることはできません。しかし、ベクトル演算の計算はメイン CPU よりもはるかに高速であるため、パフォーマンスを大きく向上させることができます。この導入部の最後の例は、前章の例に対して、VFPU を使用してシーンで光源を周回させます。

libvfpv

SCE では、「libvfpv」という名前の C ライブラリをツールチェーンでソースコードと共に提供しています。このライブラリには、2、3、および 4 要素ベクトルと行列の演算を含む、フルセットの数学関数が含まれます。これらの数学関数は、すべてインライン VFPU アセンブリコードを使用します。ソースコードは、`/usr/local/psp/devkit/src/vfpv` にあります。次のセクションでは、次のサンプルで使用する関数を説明します。

libvfpv 関数をプログラム内で使用するには、`#include libvfpv.h` を使用し、さらに、リンクされたライブラリのリストに「libvfpv」を追加して、makefile 内でアプリケーションを libvfpv とリンクする必要があります。

PSP のカーネルソフトウェアには、マルチスレッド化された広範な機能が用意されていて、優先度ベースのシステムでスレッドを管理します。このソフトウェアはスレッドコンテキスト内で VFPU レジスタを管理するため、スレッドが切り替わると、VFPU レジスタが保存および復元されます。VFPU コンテキストには、128 個のレジスタやその他の多くの情報が含まれるため、VFPU コンテキストの保存および復元時には、パフォーマンスに影響が出ることがあります。この問題を解決するために、PSP カーネルには、スレッドがその有効期間中に VFPU を使用するかどうかを指定する機能が含まれています。この機能のデフォルトは「オフ」です。有効にせずにスレッドが VFPU にアクセスしようとすると、コプロセッサ例外が発生してプログラムは終了します。ただし、スレッドで VFPU を有効にすることは、非常に簡単です。ここでは、この点について説明します。

実行は `main()` で開始します。このスレッドは、このサンプルで管理しなければならない唯一のスレッドです。そのため、VFPU は使用する前のある時期に有効であることが、ただ 1 つの要件です。使用する関数は、`main()` の内部で呼び出される最初の関数です。

`sceKernelChangeCurrentThreadAttr(SceUInt clearAttr, SceUInt setAttr)` によって、呼び出し先スレッドの属性が変更されます。この場合は、特定の属性を消去する必要がないため、1 番目のパラメータとして 0 が渡されます。

2 番目の引数として渡される `SCE_KERNEL_TH_USE_VFPU` は、このスレッドの VFPU 属性を設定するように、カーネルに通知します。これで、このサンプルにおけるスレッド管理は完了します。PSP のカーネルおよびスレッドライブラリについての詳細は、「参考文献」セクションの [9] および [10] に示されたドキュメントで学習してください。より詳細な情報が載っています。

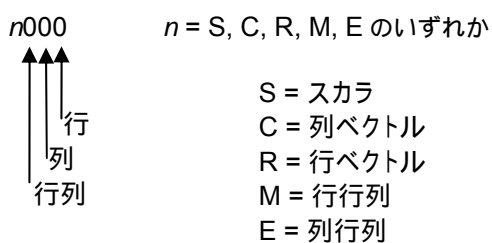
行列レジスタ

VFPU コードを記述するために、VFPU チップ自体の行列レジスタについて少し学んでおくのが効果的です。VFPU 内の行列レジスタファイルは、128 個の 32 ビットレジスタで構成され、これらのレジスタにはさまざまな方法でアクセスできます。単一スカラと、2、3、および 4 要素ベクトルがサポートされています。また、 2×2 、 3×3 、または 4×4 行列としてレジスタにアクセスすることもできます。すべてのベクトルと行列のアクセス性オプションは行と列の形式で利用できます。特に『VFPU User Manual』([8])で初めてオプションを見たときは、大量のオプションがあるように見えます。

行列レジスタファイルは、\$0 ~ \$127 の名前が付けられたレジスタのシーケンシャルなリストです。各レジスタの大きさは 32 ビットワードになります。

レジスタファイルを想像するには、8 つの 4×4 行列が一列に並んでいる状態を考えてみると良いでしょう。このとき、8 つの行列すべてが一番上の行がレジスタ \$0 ~ \$31 の連続セット、2 番目の行が \$32 ~ \$63 というようになります。これは、VFPU アセンブリ言語が規定する命名規則にアクセスすることに対応しています。この命名規則とは、次の性質を持つ 4 桁の英数字の文字列です。

図 5-1: VFPU 行列レジスタの命名規則



そのため、たとえば行列レジスタのアクセスコード S000 は、1 番目の 4×4 行列で、1 列 1 行目のスカラを意味することになります。

ベクトルや行列を定義することは、もう少し複雑です。レジスタファイル内でベクトルや行列が開始される位置に数字を変換し、可能な場合は、他のコンポーネントを定義するために要素をラップします。

図 5-2: M000 アクセスコード

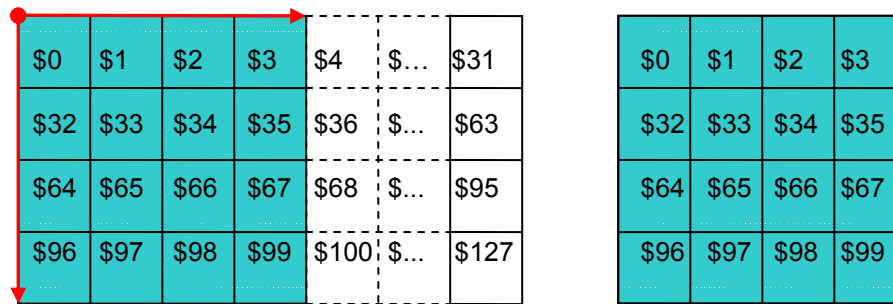


図 5-2 は、M000 アクセスコードを図示したものです。これは、行行列で、行列レジスタファイルの 1 番目の行列の、1 列 1 行目から始まっています。右側の行列は、M000 から得られる行列レジスタの配置を示しています。

図 5-3: M012 アクセスコード

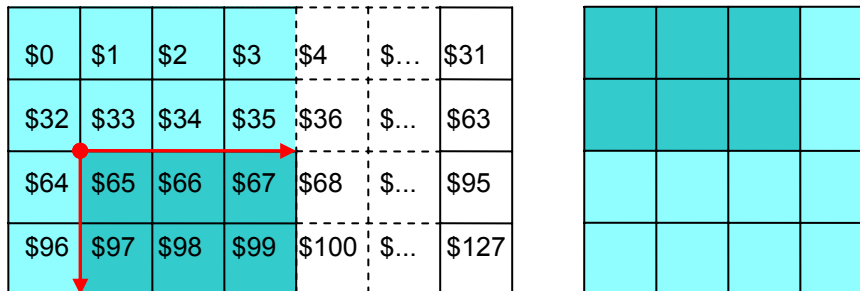


図 5-3 は、行列アクセスコード M012 を表しています。これは、1 番目の行列の 2 列 3 行目から始まる行行列として定義されています。この図が示すように、この行列の開始位置は、行列レジスタファイルへのオフセットになります。また、色の明るい部分は、ラップされる値を示しています。右側の行列は、M012 から得られる行列要素の配置を示しています。しかし、すべての構成が可能とは限りません。アクセスコードの完全なリストと、アクセスコードに対応する要素の説明は、「参考文献」セクションの [8] に記載されています。

コード

このセクションでは、このサンプルで libvfpv から使用する VFPU 関数を説明します。アセンブリコード命令を詳細に見ていきます。

関数

ライトの位置は、ワールド座標のベクトルです。ジオメトリのように GE 変換を受けません。そのため、ライトの位置を再定義するには、ライトの位置ベクトルを手作業で移動してから、`sceGuLight(int n, int type, int comp, const scePSPsFVector3 *vec)` を呼び出す必要があります。ここでの目的は、浮かんでいるトーラスに対して、ライトを時計回りに周回させることです。それには、Y 軸に対する 3×3 の回転行列を作成して、この行列をライトの位置ベクトルに掛けます。そのために libvfpv の 3 つの関数を使用します。

```
sceVfpvMatrix3Unit( scePSPFMatrix3 *pm0 );
```

この関数は、 3×3 の行列ポインタを取り出して、単位行列に設定します。ここで注意する VFPU 命令は 2 つしかありません。これらの命令については、次で説明します。

```
__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)
);
```

この例では、この関数から VFPU コードを再生成します。淡色で表記されている部分は、すべての例で使用されている標準 gcc インラインアセンブリ構文です。この章では構文の一部について説明しますが、詳細は付録 A で説明しています。

ほとんどの VFPU 命令には、ALLEGREX または FPU の命令と区別するために、ベクトルを表す "v" が先頭に付くか含まれています。命令が有効なベクトル要素の大きさと区別するために、必要に応じて命令にはサフィックスも付けます。サフィックスは次のとおりです。

s	スカラー要素関連
p	2 要素関連
t	3 要素関連
q	4 要素関連

一般に命令の名前は、その命令が実行する操作の頭字語となります。

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

vmidt.t 命令は、 3×3 の単位行列を作成して、行列レジスタファイルに、指定された位置から格納します。この場合は、命令の "midt" 部分が "Matrix Identity" (単位行列) に対応し、"t" というサフィックスは、単位行列で 3 重ワード、つまり 3×3 行列を使用することを表します。一部の頭字語は、最初に見た他のものよりもっと簡単に認識できますが、すぐに慣れるでしょう。

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

この命令は、指定したオフセットを加えたメモリアドレスに単一ワードを格納します。この場合では、メモリアドレスはオフセット(0)+ アドレス(%0)で指定されています。コードのオフセット部分は、書き込むアドレスからオフセットするバイト数を決定します。アドレス部分は、gcc インラインアセンブリ構文を使用して、アセンブラ部分へ渡されるアドレス変数です。上述のコード部分では、閉じ括弧の前の最後の行に、以下が含まれています。

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

これは、gcc アセンブラディレクティブで、書き込み用メモリ変数を渡します。この変数は、アセンブラ内では "%0" として参照されます。"%" 記号は、コードの下にあるリストで宣言された変数であることを表し、"0" は、最初に宣言された変数であることを意味します。多くの変数を宣言している場合は、0 から始まって順番に、宣言した順序でアクセスします。詳細については、付録 A を参照してください。

2 要素または 3 要素を格納する特別な命令は用意されていません。そのため、すべての行列要素を正しい位置に格納するには、適切にオフセットを増加させたいと、"sv.s" を 9 回呼び出す必要があります。しかし、4 要素ベクトルおよび行列を使用する場合は、sv.q を使用して、1 回でメモリにクアドワードを書き込むことができます。

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

この関数では、Y 軸に対して ry ラジアン of 回転行列を作成し、行列 pm0 に格納します。行列 pm1 が NULL の場合は、格納前に作成された行列が乗算されます。この関数の VFPU コードは、次のようになります。

```
__asm__ (
".set          push\n"
".set          noreorder\n"
"mfc1         $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"(pm1), "f"(ry * (2.0f/3.14159265358979323846f))
: "$8"
);
```

VFPU アセンブリ関数に渡された浮動小数点変数は、まず、FPU 内の汎用レジスタに格納されます。VFPU でこれらの値を使用するには、VFPU 行列レジスタファイルに移動させる必要があります。この関数の最初の 2 つの命令が、移動を実行します。

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

ALLEGREX 命令 "mfc1" は、"Move from co-processor 1" (コプロセッサ 1 から移動) を意味します。この命令で、変数が FPU の汎用レジスタから CPU の汎用レジスタへ移動します。命令 "mtv" は、"Move word to VFPU" (ワードを VFPU へ移動) という意味で、変数を CPU のレジスタから VFPU レジスタファイル内の領域へ移動し、今後の操作に備えます。

その次の 3 行では、回転行列の作成を行います。Y 軸に対する列回転行列は、次のように定義できます。

図 5-4:

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

これら 3 つの VFPU 命令では、行列要素に各変数を代入します。

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

"vrot.t" 命令は、この作業に最適です。スカラから、変数の余弦と正弦を計算し、その結果を行列の 3 重ワードの領域内に定義どおりに配置します。正または負の変数を操作するように命令させることもできます。

ここで注意しなければならないのは、VFPU の内部角度形式です。この関数のアセンブリパラメータリストでは、ry 変数は、次のように変更されています。

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

これは、単位の変換手順です。VFPU の内部角度形式は、ラジアンでも度でもなく、象限です。象限とは、円の 1/4 のことで、円に沿って完全に回転させると 4 象限になります。ここでの変換は、ラジアンによる角度を象限に変換する方法です。"vrot.t" 命令の形式は次のようになります。

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

vDst および vSrc は、それぞれ操作の宛先とソースです。Writemask は、5 ビットの値に変換されるニーモニックで、VFPU はこの値を使用して、vDst レジスタに変数を書き込む際のパターンを判断します。このニーモニックではバリエーションのリストを使用できますが、いくつか制限があります。"c" および "s" 要素は、余弦または正弦のどちらを使用するかを表し、"-" 記号は負の数字が有効であることを表します。括弧内でこれらを使用する順序によって、宛先の 3 重内での書き込み位置が決まります。"0" は、各要素を 0 に設定します。使用できる有効な文字はこれらだけです。

"vidt.q" 命令は、単位行列の対応する部分として指定されたベクトルを設定します。この場合は 2 列目が、回転行列の要件に対応しています。

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

この ALLEGREX 命令は、"Branch on equal likely"(等価の時に分岐)を意味します。この場合は、2 番目の入力パラメータをレジスタ "\$0" に対してチェックします。このレジスタは常に 0 です。この入力パラメータは、行列ポインタ pm1 です。行列ポインタが NULL の場合、プログラムカウンタは、指定されたアドレスに設定されます。この場合は、"0:\n" で指定されたサブルーチンとなり、コード内では "0f" で参照されます。この命令は、分岐テストに成功した場合だけ、遅延スロット内の命令(分岐の後の命令)を実行します。

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

この命令は、3 重ワードベクトルを "e000" から "e200" にコピーします。少し見てみれば、"e200" が行列レジスタセクションで、すべての値がスカラ形式でこのセクションからメインメモリに書き戻されることがわかります。そのため、行列ポインタが NULL の場合、回転行列が新しい位置にコピーされてから、データをメインメモリに書き戻すセクションへの分岐が実行されます。つまり、NULL をこの関数に渡せば、sceVfpMatrix3Unit(...) を前提要件として呼び出す必要はありません。しかし、ここでは、例として残してあります。分岐が実行されない場合は、行列データはコピーされません。その代わりに、次に説明する一連の命令が実行されます。

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


一般に、変数をメインメモリから VFPU にロードするときは、VFPU と同等のサイズに揃える必要があります。そのため、ワードアドレスはワード境界に揃え、クアドワードはクアドワード境界に揃える必要があります。メモリアドレスがこの前提条件を満たさない場合、CPU が例外を生成し、実行は終了します。"lvr.q" および "lvl.q" 命令を使用すると、この問題を回避できます。これらの命令は、このコードでデモされているとおり順番に使用するのが最適です。これらの命令で、指定されたアドレスから VFPU 行列レジスタファイルに、クアドワードが格納されますが、順序と格納揃えが重要です。この点が、ワードのロードまたはベクトルのロードを行う標準の命令とは異なります。

図 5-5: lvr.q および lvl.q 命令

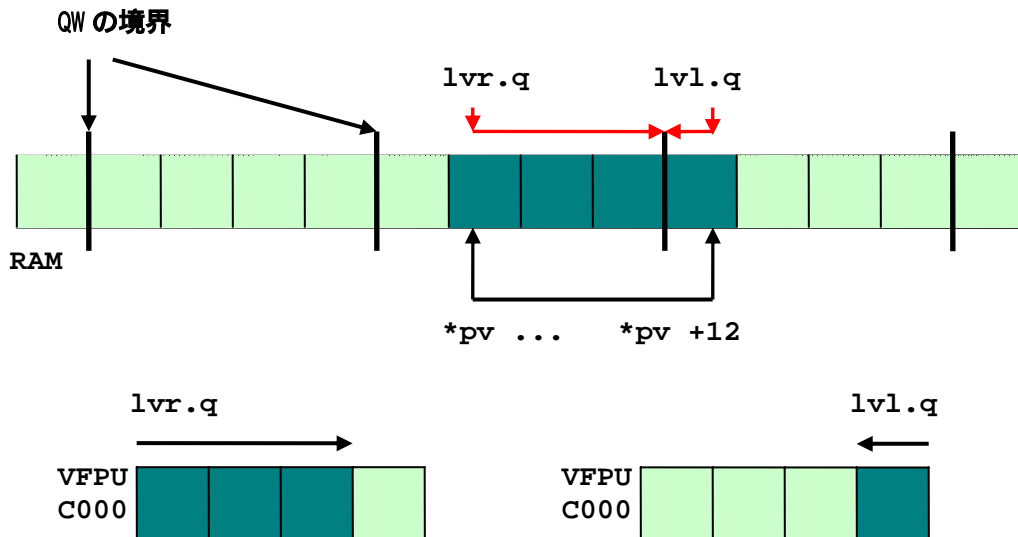


図 5-5 は、"lvr.q" 命令と "lvl.q" 命令がメモリで操作する方法とこの例の VFPU 行列レジスタを示しています。図の上側は、メイン RAM のセクションで、色の濃い部分に、"pv" でポイントされるクアドワードがあります。図に示すように、ベクトルは、クアドワード境界をまたがっています。"lvr.q" 命令は、事実上 "pv" からクアドワード境界までを読み込みます。ワードは、行列レジスタファイル内で左から右へと格納されます。"lvl.q" 命令は、"pv+12" から左にクアドワード境界まで読み込み、行列レジスタファイル内では右から左にワードを格納します。この方法では、両方の関数を使用することで、クアドワード要素がクアドワード境界に揃っていても、すべてのクアドワード要素を行列レジスタファイルにロードします。

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

"e100" にロードされた行列を使用して、"vmmul.t" 命令が実行されます。この命令は 3 x 3 行列の乗算命令で、この場合は、"e200" に結果を格納します。その後、宛先アドレスに書き戻されます。

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

説明する関数は上記のもので最後です。この関数は、3 x 3 行列を 3 要素ベクトルで変換します。ベクトルを Y 軸に沿って実際に回転させるのは、この関数です。

```

__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)
);

```

この例で使用している命令のほとんどを説明してきました。前の例では、行列とベクトルは"lvr.q" および "lvl.q" 命令を使用してロードされていました。新しく追加した命令は次の命令です。

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

この単一命令で、行列の乗算を実行します。得られるベクトルは、列形式で格納されて、"sv.s" 命令でメモリに書き戻されます。

まとめ

この章では、VFPU コプロセッサを紹介し、SCE が提供する VFPU 数学ライブラリである libvfpu 内の関数を説明しました。第 4 章のサンプルを発展させて、VFPU を使用してライトを周回させました。また、コードの説明では命令を 1 つずつ見て、関数を説明しました。

VFPU は PSP システムの非常に強力なパーツであり、ハードウェアから最大のパフォーマンスを引き出そうとすると、VFPU の使用は不可欠です。VFPU は、数学関数が提供されているだけでなく、粒子系や物理シミュレーションなど他の多くのアプリケーションにも適しています。開発者はすでに VFPU を利用しており、今後も斬新な使い方をを見つけ続けると予想されます。

最後に

これで、PSP のプログラミングの詳細を説明するドキュメントは終わりです。本ドキュメントでは、2 次元と 3 次元グラフィックの基本、およびコントローラ入力について説明しました。また、アプリケーション内で VFPU を使えるようにする方法についても簡単に説明しました。

印刷時点では、PSP は、最も強力に興味深いポータブルゲームコンソールであり、このガイドで説明した以外にも多くの機能があります。これらの素晴らしい機能の多くをアプリケーション内でいかに簡単に実装できるかをご理解いただき、ハンドヘルドエンターテインメントの新時代を画すシステムをもっと深く調べたいとお考えになることを期待しています。

付録 A:

gcc インラインアセンブリ構文

このページは空白です。

インラインアセンブリは、C または C++ のソースファイル内にアセンブリのコードを挿入できる、非常に便利な機能です。たとえば、関数の内部に、最適化された VFPU コードを簡単に作成できます。gcc では、このオプションをネイティブで提供していますが、使用するには、いくつかルールに従う必要があります。この付録では、一般的な用途における gcc インラインアセンブリ構文の概要について説明します。典型的な VFPU インラインアセンブリ関数を次に示します。

```
ScePspFVector4 *sceVfpuVector4Scale( ScePspFVector4 *pv0, const ScePspFVector4
*pv1, float t)
{
    __asm__ (
        ".set          push\n"
        ".set          noreorder\n"
        "mfc1          $8,    %2\n"
        "mtv           $8,    s0l0\n"
        "lv.q          c000, %1\n"
        "vscl.q        c000, c000, s0l0\n"
        "sv.q          c000, %0\n"
        ".set          pop\n"
        : "=m" (*pv0)
        : "m" (*pv1), "f"(t)
        : "$8"
    );
    return (pv0);
}
```

この関数は、4 要素ベクトルをスカラー倍します。このセクションではインラインアセンブリ構文だけを説明するため、実際の VFPU コードは淡色表示し、前章とは同様の方法では説明しません。

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

すべてのインラインアセンブリコードは、このステートメント内に記述する必要があります。このステートメントはコンパイラディレクティブです。これにより gcc に対し、これがアセンブリコードであり、このコードはアセンブラ用に予約されているため、通常のソースコードと同様にはコンパイルしないよう通知します。

この付録で説明する関連命名規則に合うようにこのコードを変更すると、以下ようになります。

```
__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"
);
```

アセンブラのオプションの設定

アセンブリコードは、3 つの ".set ... \n" ステートメントで囲まれています。これらの set アセンブラオプションは、このコードで非常に重要なだけでなく、最も一般的と言ってもいい使用法です。

".set push \n" は、現在のアセンブラオプションをスタックに保存します。そのため、必要に応じて現在の状態を復元できます。

".set noreorder \n" は、命令を並べ替えないように通知する重要なステートメントです。命令の並べ替えとは、アセンブラでコードの最適化が必要だと判断された場合に、命令を並べ替えようとする機能です。通常、手

動で最適化したコードは、アセンブラで作成されたコードよりも効率的です。そのため通常、コードの作成者は、命令の並べ替えを管理するべく最善を尽くしています。

".set pop\n" は、".set noreorder\n" の前の状態に、アセンブラオプションを復元します。

アセンブラのコード

アセンブリコードそのものは、入出力変数やレジスタに対して読み書きされたデータに対して操作を実行する一連の命令です。入出力変数は、"%" 記号の後に数字を付けたものをアセンブラ内で宣言します。これらの変数は、メインのコードブロックの後に、コロンで区切られた最初の 2 つのセクションで宣言されています。

入出力変数

これらは、アセンブラコードが読み書きする変数です。インラインアセンブラの変数と、C または C++ の変数の一番の違いは、その順序です。C や C++ では、変数は使用する前に宣言します。インラインアセンブリで変数を使用できるようにするには、アセンブラコードの後、入出力変数のリスト内で宣言します。その構文は次のとおりです。

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

入出力変数は、上記の順序でアセンブラコード内で宣言され、%0 から順番に数字が増えていきます。コード例で出力変数は、ベクトルポインタ pv0 で、パラメータとして関数に渡されています。入力変数も、同様の構文規則に従い、同様の方法でコード内に記述します。

制約

制約とは、変数を使用すべき方法を定義するための、アセンブラに対する追加情報です。一般的な例を次に示します。

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

この制約は、"r" は整数レジスタ内に、"f" は浮動小数点レジスタ内に、変数を置く必要があるということです。

```
"m"
```

この制約は、変数がアドレスであるということです。

また、制約に追加できる修飾子もあります。一般的な修飾子は次のとおりです。

```
"="
```

この修飾子は、変数が書き込み専用であると規定します。

```
"+"
```

この修飾子は、読み書き可能な変数であるということです。

クロバー

ブロックにおける最後のコロン区切り部分は、クロバーリストです。クロバーリストとは、アセンブラ コードが使用するレジスタのリストのことです。コードで CPU レジスタを使用する場合は、クロバーリストに含めなければなりません。VFPU レジスタを含める必要はありません。

詳細については、以下を参照してください。

[12], [13], [14]

付録 B:

エミュレータとハードウェアツールライブラリの違い

このページは空白です。

この付録では、エミュレータとハードウェアツールでコードが異なるサンプルで使用されているさまざまな関数について説明します。

エミュレータ固有

エミュレータの `#include` および関数

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

このファイルは、エミュレータの関数を呼び出すためにインクルードする必要があります。

`Init()` 関数では、`libGu` が初期化される前に、次の関数を呼び出してエミュレータを初期化する必要があります。

```
sceEmuInit()
```

同様に、プログラムがシャットダウンするときは、次の関数を呼び出します。

```
sceEmuTerm()
```

この関数で、エミュレータライブラリをシャットダウンします。

エミュレータの垂直同期では、次の関数を呼び出します。

```
sceEmuVSync(SCEEMU_SYNC_WAIT)
```

第 4 章で説明したコントローラの違いを除けば、エミュレータ固有で呼び出す必要がある関数はこれだけです。

ハードウェアツール固有

まず、エミュレータ用に作成されたコードをハードウェアツールで動作するようにするため、前のセクションで説明したエミュレータ用関数の呼び出しをすべて削除します。その次に、このセクションで説明する重要な点をコードに追加します。

次のファイルをインクルードする必要があります。

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

PSP アプリケーション、つまり "モジュール" を生成するために、ハードウェアツールで次のマクロが必要です。

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

このマクロは、モジュール用の情報を生成します。多くのモジュールは同時にロードすることができます。モジュールは PSP カーネルが管理します。このマクロは、カーネルがモジュールを識別できるようにするために必要となります。

モジュールの名前は、実行可能ファイルの名前と同じにする必要があります。

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

これは、次のようなディスプレイ関数呼び出しで必要です。

```
sceDisplayWaitVblankStart()
```

この関数は、エミュレータ用のコードで使用される `sceEmuVSync(SCEEMU_SYNC_WAIT)` の代わりとなる、ハードウェア用の垂直同期関数です。

`InitGFX()` 関数内で呼び出さなければならないハードウェア固有のディスプレイ関数には、もう 1 つ、次の関数があります。

```
sceGuDisplay(SCEGU_DISPLAY_ON);
```

この関数は、PSP のハードウェア画面を明示的に動作状態にします。この関数を使用しないと、画面には何も表示されません。

このチュートリアルで説明するサンプルでは、メモリを動的に割り当てていませんが、この機能は、一定のサイズを超えるほとんどすべてのアプリケーションで必要になります。そのため、これに関連したメモリ管理の問題 (次に説明) を認識することが重要です。

メモリ管理

PSP ハードウェアには、メインメモリとして 32 MB の DDR RAM が搭載されています。しかし、このうちの 8 MB はカーネルによって予約されています。実行可能ファイル(デバッグ情報を除く)は、メインメモリの残りの部分に存在しなければなりません。したがって、たとえば実行可能ファイルが 3 MB の場合だと、アプリケーション用として 21 MB の RAM が残ります。この残りの 21 MB ブロック内から `malloc()` でメモリを割り当てるには、コードに次の行を含める必要があります。

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

この値は、アプリケーション内で `malloc()` が利用できるメモリの量です。残りのブロックを `malloc()` とは関係なく管理するために、独自のメモリ管理スキームを用意することもできますが、メモリを割り当てる方法としては、この方法が最も一般的です。

参考文献

1. 『ALLEGREX Manual』(SCE)
<https://psp.scedev.net/projects/hwmanuals>
2. 『LibGu Reference Manual』(SCE)
<https://psp.scedev.net/projects/library>
3. 『Graphics Engine Users Manual』(SCE)
<https://psp.scedev.net/projects/hwmanuals>
4. 『Computer graphics, C version – Second Edition』
(Donald Hearn および N. Pauline Baker, Prentice Hall)
5. 『Introduction to RISC Assembly language Programming』
(John Waldron, Addison Wesley)
6. 『See Mips run』
(Dominic Sweetman, Elsevier, Morgan Kaufmann)
7. 『Vfpu Users Manual』(SCE)
<https://psp.scedev.net/projects/hwmanuals>
8. 『Kernel Overview』(SCE)
<https://psp.scedev.net/projects/library>
9. 『PSP Thread Manager Reference』(SCE)
<https://psp.scedev.net/projects/library>
10. 『Vfpu Instruction Reference Manual』(SCE)
<https://psp.scedev.net/projects/hwmanuals>
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. 『Inline Assembler Constraints』
http://gcc.gnu.org/onlinedocs/gcc-2.95.3/gcc_16.html#SEC175
14. 『OpenGL Programming Guide』
(第 4 版、Shreiner, Woo、その他、Addison Wesley)