

# グラフィックプログラミングひとめぐり 前編

を行い、EE Coreでは次のフレームのためのDMA川データを作成します。

従って、1フレーム中にはDMA転送とVPU1、GSによる画面描画、EE Coreによる次のフレームのDMA転送のためのデータ作成が並行して行われることになります。

図2にこのイメージ図を示します。GS上のV-RAM、EE上のDMAバッファはダブルバッファになっています。

最初にGSの基本的な使い方、を説明するために、画面モードとGS上のV-RAM割り当てについて説明します。

GS 上にはフレームバッファ (RGBA)、Z バッファ (3D 表示の隠面処理で使用)、テクスチャが混在します。3D グラフィックスシステムでは 1 フレーム表示ごとに完全に V-RAM を描画し直します。従って、表示中と描画中の画像は別個に処理する必要があるため、フレームバッファを 2 枚用意して、表示中の画面と描画中の画面を交互に切り替えます。これをダブルバッファといいます。

フレームバッファ、Zバッファは固定サイズが必要ですから、それらを引いた残りの領域がテクスチャ格納域等にプログラムで自由に使用できる領域となります(図3)。

GSのV-RAMは全部で4Mbytesしかなく、お世辞にも余裕があるサイズとは、言えませんから、少しでもフレームバッファを節約する必要があります。そこで、GSは画面モード、特にインターレースモードのときには画面表示に必要なV-RAMが最小限になるようになっています。

本稿では、PS2Linuxで画面の見栄えをよくすることを目標に、一通りのプログラミングとその解説を行いたいと思います。前編ではPS2Linux上で3Dグラフィックス表示を行うための一連の技術について解説します。

PS2Linuxでは、他の3Dグラフィック環境では始めから用意されているような基本的な処理まで、すべて一から作成しなくてはなりません。その代わり、すべての資源の使用方法を自分で決めることができます。

一般的な多くのハードウェアと同じく、PS2は非常に多機能で複雑であり、1つの目的を実現するための手段は1つではありません。通常、これらはトレードオフによってどう処理するかを決定する必要がありますが、提供されるライブラリなどを使用する場合にはライブラリが想定して決めた使い方の範囲で従うしかありません。

しかし、PS2Linuxでは何も提供されない分、ハードウェアがすべて公開されており、すべてをプログラミングする側によって選択することができます。

PS2アーキテクチャ、3Dグラフィック表示、い  
ずれも大に奥が深く、とてもこのような数ページ  
の原稿で扱えるとは思えませんが、本稿でその  
片鱗に少しでも触れられればと考えています。

最初に、基本的なPS2で3Dグラフィックを表  
示するためのアーキテクチャについて説明し  
ます。

EE Core、VPU1、GS関係を簡単にまとめたブロック図を図1に示します。

PS2においてGSを利用してグラフィックを表示させるための方法としては、大きく分類すると2つの方法があります。1つは、EE CoreでVPU0などを用いて透視変換処理まで行い、2Dデータを求めメモリ上に配置し、DMAを用いてGSへ転送する方法です。

もう1つの方法は、EE Coreでは3Dの頂点データ、または頂点データを作成するためのヒントとなる情報のみを算出してメモリ上に配置し、後の作業はVPU1で処理をさせる方法です。メモリからはVPU1へ直接データを転送し、VPU1上ではGSの描画データを生成して直接GSを動作させます。

両者は同時に使用することができます。

EE Coreの実行とDMAによるVPU1、GSへの転送も並列に動作します。通常は、DMAへ転送するデータを格納するメモリをダブルバッファとし、1フレームの描画中にDMA転送

```

graph TD
    GS[GS] --- GIF[GIF]
    GIF --- VPU1[VPU1]
    VPU1 --- VIF1[VIF1]
    VIF1 --- GIF
    GIF --- EECore[EE Core]
    GIF --- MainMemory[Main Memory]
    EECore --- DMAController[DMA Controller]
    MainMemory --- DMAController
  
```

The diagram illustrates the VPU architecture. At the top is the GS block, which connects to the GIF block. The GIF block connects to the VPU1 block. The VPU1 block connects to the VIF1 block. The VIF1 block connects back to the GIF block. The GIF block also connects to the EE Core block and the Main Memory block. The EE Core block connects to the DMA Controller block. The Main Memory block connects to the DMA Controller block. The VIF1 block is labeled with 'VIF1 Channel (PATH1)' and the GIF block is labeled with 'GIF Channel (PATH3)'.

フレーム数 (1/60秒単位)	GS (GPU1)	EE
1	フレーム1で用意されたデータを描画	フレーム2の描画データを作成
2	フレーム2で用意されたデータを描画	フレーム3の描画データを作成
3	フレーム3で用意されたデータを描画	フレーム4の描画データを作成
4		フレーム5の描画データを作成
5		

Address 0

RGBA 1
RGBA 2
Z
作業領域 (テクスチャなど)

## ●VESA

VESAはその名の通り、TV接続ではなく、PCなどのモニターへ接続する画面モードです。PCと同じようにVGA、XGAなどの画面モードで表示できます。通常、VESAモードはノンインターレースでの使用が想定されますので、普通にV-RAMの領域を2画面分使用します。

## ●NTSC/PAL

NTSC/PALはノンインターレースで使用することもできますが、通常、インターレースモードの高解像度を選択するでしょう。GSにはインターレースモードの扱いによって、FRAMEモードとFIELDモードがあります。

### ・FIELDモード

FIELDモードは、V-RAMの読み出しが1ラインおきになります。従って、640×448ドットの場合ですと、NTSCのインターレースでも普通に640×448ドット分のフレームバッファを2枚用意してダブルバッファとして描画できます。実際には、フレームバッファを1画面分にしてもダブルバッファは実現できます。GSには描画中に偶数ライン／奇数ラインのどちらかをマスクすることができますので、1つの領域の奇数ラインと偶数ラインを分けることによってダブルバッファを実現できます。ただし、ラインのマスクをかけると処理が遅くなりますし、1フレームの描画が1/60秒以内に終わらなかったときには、描画中のものが表示されますので、画面が乱れます。

### ・FRAMEモード

FRAMEモードの場合は、V-RAMの読み出しは連続して行われます。フレームバッファはインターレースに必要なサイズ、すなわち解像度の半分を用意して描画します。例えば解像度が640×448ドットの場合は、640×224ドット分のV-RAMを用意し、常に640×224ドットに対して描画を行います。描画を対象とする表示中のグラフィックが偶数ラインか奇数ラインかによって、描画すべき画像を調整する必要があります。GS内の演算は整数ではなく4ビットの固定小数点を持っているので、インターレースモードでは、GSの描画時にオフセットとして0.5ドットを与えることによって、偶数画像と奇数画像の表示位置を調整しています。

FIELDモードと異なり、ラインマスクによるベナリティはありませんし、1フレームの描画が1/60秒以内に終わらなかったときにも、2フレーム以内だと解像度が半分になる程度で済みます（実際には突然解像度が落ちますし、3フレーム以上になると、オフセットの0.5ドットが反転して表示が乱れますので、遅悪く処理が遅

れても絵的に見られるといった程度の使い方にすべきでしょう）。

本稿では最後にサンプルプログラムを作成しますが、NTSC/PALのインターレースにはFRAMEモードを使用しています（本誌付録CD-ROMにサンプルプログラムを収録）。

PS2Linuxでは標準でlibps2devというライブラリが用意されており、これを利用してプログラミングを行うのが便利です。libps2devを用いたプログラムの基本形はリスト1のようになります。

Linux上でアプリケーションを動かす場合、必ず複数のアプリケーションの並列動作、つまり資源の共有を考慮する必要があります。PS2Linuxのアプリケーションでは、グラフィックを使用するときにはディスプレイを占いますの

で、他のアプリケーションを実行したい場合には実行中のプログラムを中断する構造になっています。

アプリケーションの中断は、仮想コンソールの切り替えによって発生します。プログラミング時にはlibps2devの関数、ps2\_gs\_vc\_enablevcswitch()を用いて中断処理、再開処理を記述します。切り替えを発生させたくない間は、ps2\_gs\_vc\_lock()、ps2\_gs\_vc\_unlock()で挟んでおきます。しかし、アプリケーションの切り替え時に使用しているすべての資源を開放するのは難しいでしょうから、どんなアプリケーションとでも完全に切り替え可能とすることは現実的には厳しいでしょう。

### [リスト1]

```
release()
{
    /* アプリケーション中断のための資源解放処理 */
}

acquire()
{
    /* アプリケーション再開のための資源確保、初期化、再設定など */
}

main()
{
    int id;
    ps_gs_dbuff gdb;
    DMABUFF *dmabuff[2];

    /* DMAバッファの初期化 */
    id = 0;
    DMAバッファdmabuff[]の確保

    /* GSの初期化 */
    ps2_gs_vc_graphicsmode();
    ps2_gs_open();
    ps2_vpu0_open();
    ps2_vpu1_open();
    ps2_gs_reset();
    ps2_gs_set_dbuff/_dc( &gdb, ... );
    ps2_gs_start_display(1);
    ps2_gs_vc_enablevcswitch( acquire, release );
    ps2_gs_vc_lock();

    /* メインループ */
    for (;;) {
        フレーム初期化
        :
        :
        プログラム実行
        dmabuff[id] にデータをセット
        :
        :
        前フレームで開始したVPU1、GSへのDMAが終了するのを待つ
        oddeven = ps2_gs_sync_v(0) (V-Syncを待つ)
        ps2_gs_vc_unlock();
        ps2_gs_vc_lock();
        ps2_gs_set_half_offset() (Interlaceのhalf offsetを設定)
        ps2_gs_swap_dbuff/_dc() (GSフレームバッファの切り替え)
        DMAバッファdmabuff[id]の内容をVPU1、GSへDMA転送開始
        id = !id; (DMAバッファの切り替え)
    }
}
```

## PS2のDMAアーキテクチャ

PS2ではグラフィックデータはすべてDMA転送バッファに入れてDMAで転送を行うことになります。ここではGSへ描画データを転送することを目的としたDMA転送について説明します。

PS2はDMAをベースとしたアーキテクチャです。何を行うにもDMAを使用するため、その目的を効率良く達成するべくDMAは非常に多機能な構造となっています。特徴的な機能としては、一般的な連続領域のデータをI/Oやメモリに対して転送する以外に、タグをつけて転送データを制御しながら転送するモード(Chain Mode)が挙げられます。グラフィックデータの転送にはこれを使用すると便利です。

DMA転送バッファへのデータの設定は、プログラミングの側から見ると、ひたすらバッファへデータを転送していくだけです。バタにデータを設定していくと分かりにくく、デバッグしにくいものとなりますから、ある程度抽象化してプログラミングしやすい構造を作る作業が必要でしょう。PS2Linux上にもDMA転送バッファ/パケットを設定するためのマクロがたくさん用意されていますので、それらを使ってもプログラミングができますが、ここではサンプルプログラムでも使用している、筆者のやり方を紹介します。

DMAについては、本誌の2001年12月号

【表1】DMAタグの種類

ID	動作
cnt	タグに続くデータを転送し、さらに次のデータへ進む
next	タグに続くデータを転送し、指定位置へジャンプ
ref	指定位置のデータを転送
refs	指定位置のデータをストロー制御しながら転送
refe	指定位置のデータを転送し、転送を終了
call	タグに続くデータを転送し、次の番地を保存して指定位置へジャンプ
ret	タグに続くデータを転送し、保存されていた位置へジャンプ
end	タグに続くデータを転送し、転送を終了

「PS2Linuxカーネル用パッチ」(今号の本誌付録CD-ROMに収録)でも扱いましたので、PS2Linux標準では扱うことのできない、パッチを当てたTTE=1時のDMA転送もまとめて扱ってみます。

PS2のDMA転送は16bytes(128bit)単位での転送となります。従って、DMA転送バッファは常に16bytesを1単位とした配列構造になっている必要があります。GSへのデータ転送で有用なSource Chain ModeのDMAバッファは、データ中のタグによって、連続しないメモリ空間上のバッファやデータを自由に参照できます。すなわち、転送データは連続しないメモリ空間上をポインタで連続しながら、いくつか存在することになります(図4)。DMAタグを表1に示します。

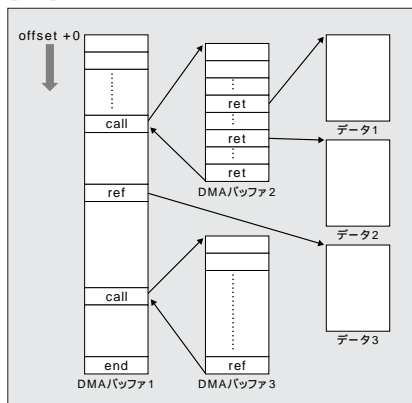
GSへデータを送る手段として、PS2ではVPU1を介する方法と直接GSへ転送する方法

があるのは始めに述べた通り(図1)ですが、各々の転送ルートをPATH1、PATH3と呼びます。実際のデータ転送時には、VPU1へのデータ転送にはVIFcodeと呼ぶタグ(以後VIFタグ)を、GSへのデータ転送にはGIFtagと呼ばれるタグを付けて、データの扱いを指示します。

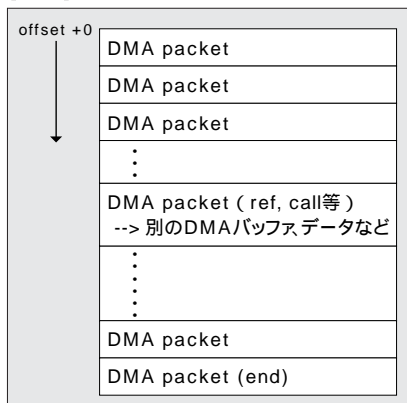
GIFについては後で扱いますので、ここではVIFタグの種類だけを表2に示します。PATH1を使用し、VIFタグでdirectを指定したとき、転送データはVPU1を介さず直接GSへと転送されます。この転送ルートをPATH2と呼びます。

注意すべきことは、図1の通り、DMAタグはDMAコントローラが、VIFタグはVIFが、GIFタグはGIFが解釈し、それぞれのステートは連続していないということです。この特徴は非常に有用で、1つのVIFデータをDMA転送時に分割して転送することが可能です。これは、異なる

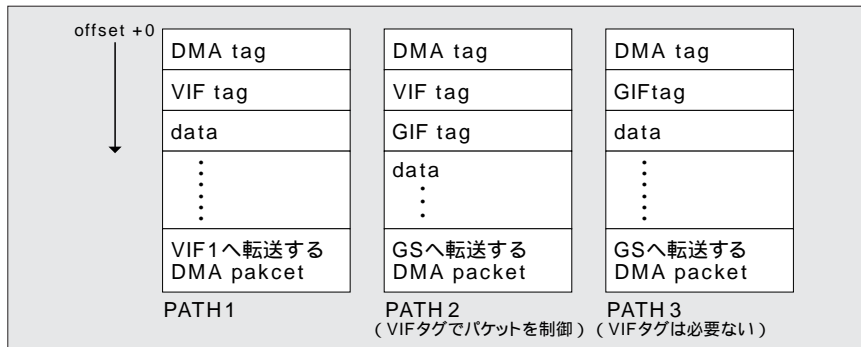
【図4】



【図5A】DMA転送バッファの構造



【図5B】各々のDMAパケットの内容



【表2】VIFタグの種類

NOP	動作しない
STCYCL	CYCLEレジスタの設定
OFFSE T	OFFSE Tレジスタの設定
BASE	BASEレジスタの設定
ITOP	ITOPレジスタの設定
STMOD	MODEレジスタの設定
MSKPATH3	GIFのPATH3転送マスク
MA RK	MA RKレジスタの設定
FLUSHE	マイクロプログラムの実行終了待ち
FLUSH	マイクロプログラムの実行終了待ちとGIF (PATH1、PATH2)の転送終了待ち
FLUSHA	マイクロプログラムの実行終了待ちとGIF (PATH1~PATH3)の転送終了待ち
MS CAL	マイクロプログラムの実行
MS CNT	マイクロプログラムの継続実行
MS CALF	マイクロプログラムの実行
STMASK	MA SKレジスタの設定
STROW	ROWレジスタの設定
STCOL	COLレジスタの設定
MP G	マイクロプログラムをロード
DIRE CT	GIFへのデータ転送 (PATH2)
DIRE CTHL	GIFへのデータ転送 (PATH2)
UNPACK	データの展開とVPUメモリへの書き込み

位置に配置されたデータを連結して転送する場合などに役立ちますし、DMAタグで指定できる転送サイズの限界も分割転送によって解決できます。

さて、このDMA転送バッファですが、16bytes単位で扱うといっても、実際の内部のデータは、32ビット単位または64ビット単位で設定することになります。しかし、DMAデータは常に16bytes単位での扱いとなりますから、必要に応じてパディングを行いながら値を設定していくことになります。

残念ながら、標準のPS2Linuxでは使用できませんが、チャンネル制御レジスタDn\_CHCRのビット6にはTTEビットというのがあり、このビットを立てるとDMAタグ自身も転送先ペリフェラルへ転送されます。このTTEビットは、VPU1へのデータ転送において威力を発揮しますが、結局のところ、TTE=1、TTE=0での違いは、DMAタグの空き領域64ビットにVIFタグを2つ埋め込むことが可能かどうかという点だけです。VIFタグは4bytesであるため、TTE=1のDMA転送ではDMAタグにVIFタグを入れられるため、本当に転送したいデータを16バイトアライメントに無駄なく揃えることができるようになります。

GSへデータを転送するDMAのパケット構造は図5A、図5Bのようになっています。

DMAデータ転送方法のPATH2とPATH3における実際のデータの違いは、VIFタグを挟むかどうかということだけです。TTE=1の場合、VIFタグはDMAタグに埋め込めますが、PATH3ではVIFタグを挿入しても無視されま

## 【リスト2】DMAバッファ切替え用マクロの抜粋

```
int __grl_ps2gdma_path      /* PATH1/PATH2かPATH3か */

int __grl_ps2gdma_sysbuf    /* システムで用意されたダブルバッファか
                             ユーザーがDMAでつなぐことを目的に自分
                             で用意したバッファか */

#define grd_set_closedrawpath() \
    if (__grl_ps2gdma_sysbuf) { \
        __grl_ps2gdma_sysbuf = FALSE; \
        __grl_ps2gdma_buf[__grl_ps2gdma_path] = __grl_ps2gdma_buf_p; \
    }

#define grd_set_currentdrawpath(_PATH) { \
    grd_set_closedrawpath(); \
    __grl_ps2gdma_path = (_PATH); \
    __grl_ps2gdma_buf_p = __grl_ps2gdma_buf[__grl_ps2gdma_path]; \
    __grl_ps2gdma_sysbuf = TRUE; \
}

#define grd_set_currentdrawpath_user(_PATH, _ADR) { \
    grd_set_closedrawpath(); \
    if ((_PATH) >= 0) \
        __grl_ps2gdma_path = (_PATH); \
    __grl_ps2gdma_buf_p = (_ADR); \
    __grl_ps2gdma_sysbuf = FALSE; \
}
```

すので、同じデータを利用できます。

さて、各データですが、DMAタグは16バイト単位、VIFタグは4bytes単位、GIFタグは8bytesと4bytesの混在で扱うことになりま

すし、データ型もintだけではなくfloatまで混在します。

PS2のプログラミングでは一般的に、CやC++を使用すると思いますが、通常、このようなバッファならば、バッファ操作関数を用意してすべてのデータを操作し、複雑なサイズや型の違いは操作関数やプログラム言語の機構を用いて解決するのが正しいプログラミングのやり方でしょう。そのようにしないと、「この人はオブジェクト指向プログラミングはおろか、抽象データ構造さえ知らないのか」ということになってしまいそうです。しかし、プログラムの大部分がDMA転送バッファへデータを設定することにあり、EEに少しでも負

担をかけたくないとなると話は変わってきます。このような方針で行くと、可変要素がない限り、データの設定は定数代入1回とポインタの移動のみとなるように、また、なるべくマシン語に展開されたとき効率良くメモリアクセスを行うように記述することが、言語レベルでは最良の方法となります。

最適化後のコードを予想しにくい要因を減らすためにも、単純なプログラミング言語がよいと思いますので、今回はC++ではなく、C言語でコードを記述します。

このような複数のデータ形式を扱う場合は、C言語の場合、unionで管理するのが正しく綺麗な方法ですが、今回はそうではなく、DMAバッファはunsigned intの配列として定義し、すべてマクロを使ってアクセスしています。複数のバッファを切り替えながらバッファに値を設定するために、実際のマクロはリスト2、リスト3のように定義しています。

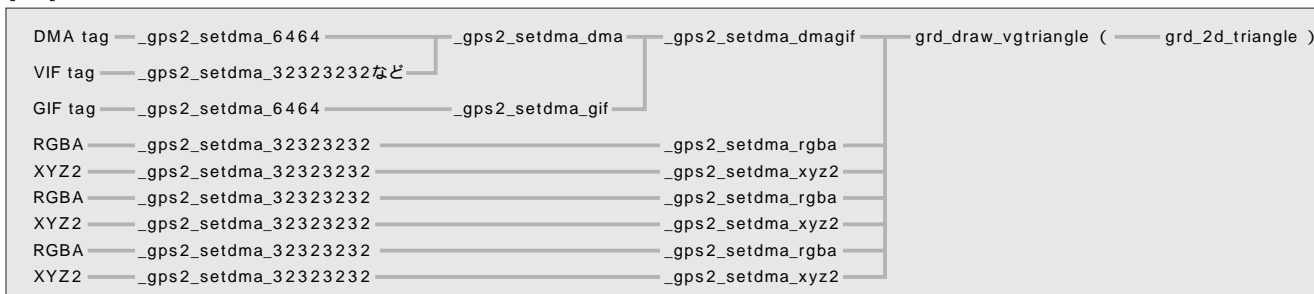
## GSへ転送するデータの作成

ここで、具体的にPATH1を使用してパケット

【図6】

offset +0	DMAタグ CNT
↓	VIFタグ DIRECT
	GIFタグ レジスタ(RGBA,XYZ2)x3, PACKED
	RGBA 頂点1のRGBA
	XYZ2 頂点1のXYZ
	RGBA 頂点2のRGBA
	XYZ2 頂点2のXYZ
	RGBA 頂点3のRGBA
	XYZ2 頂点3のXYZ

【図7】

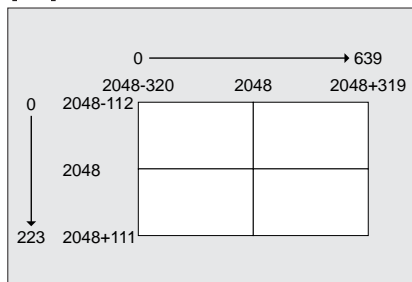


を積み、GSでプリミティブを描画させてみます。GSへのパケットはREGLISTモードとPACKEDモード、主にテクスチャ転送で使われるIMAGEモードがあり、GIFタグで指定できますが、ここではPACKEDモードを使用しています(REGLISTモードを使った場合、転送できるパケットの種類に制限がありますが、実際に転送するパケットを置くメモリの領域は半分になります)。三角形を描画するパケットの構造は図6のようになります。

【表3】プリミティブの種類

POINT	点
LINE	線
LINE STRIP	線(ストリップ)
TRIANGLE	三角形
TRIANGLE STRIP	三角形(ストリップ)
TRIANGLE FAN	三角形(ファン)
SPRITE	四角形

【図8】



三角形は、GSへ3つ目のXYZZが入力された瞬間実際に描画が開始されますが、問題はデータの設定方法です。三角形に限らず、多種の図形を描画するたびにこのようなパケットをすべて直接記述していると、実に手間がかかりますし、プログラムが分かりにくくなってしまいます。従って、ある程度抽象化することになります。EEに無駄な処理をさせずに実現したいので、なるべく生成されるコードが単純になる範囲で抽象化する構造を設定します。これは、今回はマクロで定義します。

今回のサンプルプログラムでは、図7のような構造になっています。このマクロ定義は階層になっていますが、実際に展開された後は、メモリへマクロの引数を直接代入することになります。

さて、このgrd\_draw\_vgtriangle()はGS上であらかじめ設定された表示座標上に描きます。GSの表示座標はGSのレジスタ、XYOFFSETやSCISSORで設定したものが、初期値はPS2Linuxではps2\_gs\_set\_dbuff/\_dc()で設定した構造体ps\_gs\_dbuffの中にGSへ転送するためのタグとして格納されています。例えば、NTSC 640×448(インターレース)の場合、図8のような座標となっています。

私は、2Dで扱うときには、画面左上を(0, 0)

にしたいですし、インターレースかどうかに関係なく解像度そのままでも扱いたいので、grd\_draw\_vgtriangleは、さらにgrd\_2d\_gtriangleというマクロでラッピングして、この座標変換作業を行います。

普通、ゲームなどでは解像度は固定になる場合がほとんどだと思いますから、コンパイラに値を整数で固定してしまうならば、マクロの引数で変数を使用しない場合、コンパイラによって定数展開され座標変換を追加したことによる処理のペナルティはありません。

GSのプリミティブに対して以上のような抽象化を行うことによって、PS2のDMA転送データ作成は効率を犠牲にすることなく格段に扱いやすくなると私は考えています。

ところで、GSのプリミティブの種類はPRIMレジスタで指定します。これはGIFタグに直接埋め込むことができます。プリミティブの種類には表3のようなものがありますので、これらを実用的なレベルまで抽象化するためには、使用するプリミティブすべてに対してマクロの定義を行う必要があります。

さて、同じくPRIMレジスタで指定し、GIFタグに直接埋め込める要素として、いくつかの属性があります。これは表4のようになっています。

これらは、使用する場合によって、指定する属性の組み合わせが異なります。これも効率を落とさずに指定すべきですが、grd\_draw\_\*やgrd\_2d\_\*で実際に定義することを考えると、最初から指定しておいた方が都合がいい属性と使用時に判断したい属性があります。言い、最終的にはビットをセットすることになるため、マクロのオプションとして値を渡し、マクロ内で論理和を用いて処理しています。このようにすれば、変数を指定しない限り、すべてコンパイラの定数展開で処理されますので、EE使用時の処理に対するペナルティはありませんし、指定したい時に自由にPRIM属性を指定できます。

さて、GSの描画時に切り替え可能なものとして、DMAの転送によるPATHの他に、コンテキストが存在します。GSの描画には先だって設定を行うことで描画方法を左右するレジスタがいくつかあります。その中のいくつかはコンテキストによって環境を切り替えることができます。PATH1PATH3で並列させて描画を行う場合、描画条件をそれぞれのPATHで独立して管理する必要があり、コンテキストの使用用途としては代表的なものです。また、平行描画を行わない場合でも、テクスチャ2枚を切り替えながら描画させたい場合など、有効利用できる箇所はいろいろと存在します。

従って、コンテキストの切り替えも、PATHと同様に切り替える機構を用意したいところです

【リスト3】DMAバッファ設定用マクロの抜粋

```
#define _gps2_setdma_32( _V ) \
    *__grl_ps2gdma_buf_p++ = (_V);
#define _gps2_setdma_f32( _F ) \
    *((float *)__grl_ps2gdma_buf_p)++ = (_F);
#define _gps2_setdma_64( _V ) \
    *((u_long64 *)__grl_ps2gdma_buf_p)++ = (u_long64)(_V);

#define _gps2_setdma_32323232( _V1, _V2, _V3, _V4 ) { \
    __grl_ps2gdma_buf_p[0] = (_V1); \
    __grl_ps2gdma_buf_p[1] = (_V2); \
    __grl_ps2gdma_buf_p[2] = (_V3); \
    __grl_ps2gdma_buf_p[3] = (_V4); \
    _gps2_setdma_skip_4(4); \
}

#define _gps2_setdma_f32323232( _F1, _F2, _F3, _F4 ) { \
    *((float *)__grl_ps2gdma_buf_p)[0] = (_F1); \
    *((float *)__grl_ps2gdma_buf_p)[1] = (_F2); \
    *((float *)__grl_ps2gdma_buf_p)[2] = (_F3); \
    *((float *)__grl_ps2gdma_buf_p)[3] = (_F4); \
    _gps2_setdma_skip_4(4); \
}

#define _gps2_setdma_6464( _V1, _V2 ) { \
    *((u_long64 *)__grl_ps2gdma_buf_p[0]) = (u_long64)(_V1); \
    *((u_long64 *)__grl_ps2gdma_buf_p[2]) = (u_long64)(_V2); \
    _gps2_setdma_skip_8(2); \
}

#define _gps2_setdma_6464ul( _V1, _V2 ) { \
    *((u_long64 *)__grl_ps2gdma_buf_p[0]) = (u_long64)(_V2); \
    *((u_long64 *)__grl_ps2gdma_buf_p[2]) = (u_long64)(_V1); \
    _gps2_setdma_skip_8(2); \
}
```



が、ここで変数を使用してしまうと、せっかくの定数関数が意味をなさなくなりますので、基本的にコンテキスト1以外を使用するときは、オプションとして自分で設定します。一見、不便に見えますが、実際に自動で切り替えて使用するのが便利なのは、実験目的以外にはないように私は感じています。

これらのマクロ定義はリスト4のようになります。

さて、このコンテキストごとに設定されるデータを始めとして、GSではプリミティブ描画に対して、描画に先だって設定しておくことで描画方法を左右するレジスタがいくつかあるのは前述の通りですが、中でもALPHA、TEST、ZBUFは頻繁に使用するのはないかと思います。

ALPHAは $\alpha$ 値の扱いを制御できますので、TEST( $\alpha$ テストなど)と組み合わせて、いろいろな描画や特殊効果が実現できます。TEST(デプステスト)とZBUFの組み合わせも、ZBUFを更新するか、参照するかなどで頻繁に使うことになると思います。

これらについても綺麗に扱いたいところです。X Window Systemのグラフィックコンテキストや、Microsoft WindowsのGDCを参考に、デフォルト色の管理も扱って、即戦力的にしたような機構を作れば効率を落とさず使いやすくて良いでしょう。ただし、残念ながら、今回のサンプルプログラムでは特に何も行っておりません。

GSは既存の機能を組み合わせることによって、非常に多くの視覚的効果を作ることができますが、これについては実験した回数だけやり方が存在するといってもいいくらいですから、本稿で扱うつもりはありませんが、ごく簡単に基本的なテクニックに触れておきます。

GSはフレームバッファはいつでも白黒の設定が可能で、V-RAM上の位置を変更したり、サイズを自由に設定したりできます。従って、作業領域に描画させて、描画した内容をテクスチャとして本来の領域へ描画することができます。この応用として、現在描画中のV-RAMの内容や前フレームのV-RAMの内容をテクスチャとして扱うこともできますから、自由に編集が可能です。このことと、ALPHA、TESTなどを始めとするさまざまなGSのレジスタを組み合わせることによって、実に多くの美しい表現を得られます。PS2のゲームでよく用いられるばかりしも、GSの機能としては見当たりませんが、これらの応用としてテクスチャ描画の拡大縮小時の補完を利用して実現できます。

## ●PS2プログラミングの傾向

さて、このようにして定義したDMAバッファとパケット設定マクロですが、「最良の方法とは何か」となると実に難しい話となってきます。その議論をする前に、PS2のプログラミングの傾向について少し触れておかなければなりません。

実際にPS2のプログラミングを行っていて意外に思うことは、わずかなプログラム変更が大きくパフォーマンスを左右することが頻繁にあるということです。この原因はPS2のアーキテクチャに起因します。すなわち、EE側におけるキャッシュの小ささ(2次キャッシュがない)、キャッシュミスヒット、RD-RAMの特性、コンパイラの最適化などに加えて、並列動作するGSとVPU側のプログラムも関係し、原因が分かづらくなっています。これに加え、ややこしいプログラミングを試行錯誤しながら検証せざるを得ないためチューニングが容易でないこと、そしてコツをつかむまでにやたらと時間がかかることなどが、PS2のプログラミングが難しいといわれる原因になっているのでしょう(ゲーム開発現場では、苦労のわりに目に見える結果が出ないことから、本当に技術的に詳しい人間以外は何が問題なのかさえ分からず、つまらないことで苦労させられて余計……ということもあるかもしれませんね)。

しかし、実際、どんなハードウェアにも似たような問題はありますし、これだけのアーキテクチャを低価格に実現した以上、これらの犠牲は仕方ないことです。むしろ、この程度の犠牲でこんな素晴らしいハードウェアが提供されたのですから、文句を言うのは賢沢というものでしょう。PS2の場合、EEやGSに関して隠されている部分はありますが、原因をきっちり調べて理由を押さえれば、ほとんどの問題は解決できます。すなわち、チャレンジし甲斐のあるハードといえますから、これはもうしゃぶ倒すしかないでしょう。

PS2の場合、EEの速度が不自然に遅い原因は、キャッシュミスヒットが大きく関係しています。これを回避するために、キャッシュを直接コントロールしたり、スクラッチパッドとDMAを組み合わせてガリガリと利用する方法があります。しかし、いかんせんLinuxのユーザー空間のプログラムでの実現となると、少々きついものを感じます。

## ●最良の方法とは?

話を戻し、今回の方法がベストであるかどうかということに言及したいと思います。ここでいうベストなものとは、最良の方法(最も簡単に使いやすい)で、最良の解(最も効率の良いもの)を得られるものだと私は考えます。すなわち、記述時点では非常に分かりやすく抽象化されており、簡単に使用でき、コンパイル後はマシン語のレベルでCPUが最も効率良く実行されるということになります。しかし残念ながら、両者の要求は相反していますから、絶対的な解というものはなく、両者の間でうまくバランスをとることになります。

今回の方法では、抽象化する上で犠牲にしたものについてはすでに言及したので、CPU資源の効率利用について述べます。

まず1つは、どのようなコードを記述しても最適化されるかどうかという点が挙げられます。これはさまざまなケースでアセンブリリスト出力を見て調べる必要がありますが、いくらコンパイラ技術が進歩して良いコードを出力するようになったとはいえ、「コンパイラの出力するコードが常に最適」というのは無理な要求です。本当に速度を要求するところは、マシン語レベルで厳密に最適化するしかありません。今回のサンプルは、C言語のレベルではほとんど無駄がないということと、現実としてCPUの実行効率を意識したコードが出力されるという点で、私は良いものだと考えています。

もう1つは、#defineを多用してしまうため、すべてがinlineに展開され、実際のコードが大きくなり、キャッシュに収まらなくなりやすいという点です。gccで-O3を指定すると、finline-functionsが有効になり、可能な限り関数をinline展開し始めますが、この場合でも同じことがあてはまります。同じ関数呼び出しが複数ある場合、関数の規模が関数

【表4】PRIMレジスタで指定し、GIFタグに直接埋め込める要素(属性)

IIP	シェーディング方式(ゲロー/フラット)
TME	テクスチャマッピングのON/OFF
FGE	フォグのON/OFF
ABE	アルファブレンディングのON/OFF
AA1	1パスアンチエイリアシングのON/OFF
FST	テクスチャ座標の指定方法(STQ,UV)
CTXT	コンテキスト1,2の選択

【リスト4】コンテキスト設定に関するマクロ定義と使用方法

```
#define grd_set_currentcontext(_CTXT) \
    __gr1_ps2draw_context = (_CTXT);

#define grd_get_currentcontext() __gr1_ps2draw_context

grd_2d_gtriangle( x, y, ...,
    GRDRAW_OPT_GOURAUD | GRDRAW_OPT_ALPHA |
    GS_PRIM_CTXT_V(grd_get_currentcontext()) );
```

【図9】



呼出のオーバーヘッドよりも十分に大きく、関数を展開しないことによってキャッシュを有効に使える場合、inlineは逆効果です。

この問題の選択は実に難しく、特にキャッシュの小さいPS2では判断しにくいのですが、コンパイラが生成するコードが予想しやすいことや、十分に最適化したコードは小さいループにすることができ、予定外のキャッシュミスヒットさせない記述方法が大抵可能であること、実際に実験してみると結局この方が高速になったことなどから、私はこのように実現しています。

結局のところ、今回の方法は現実的に実装できるレベルでCPUを効率良く使用でき、抽象化についても使いやすいくレベルで処理できているということで、要求される問題を使いやすく効率を落とさずにうまく解決できますから、トレードオフとしては望ましいところをとっていると私は考えています。

私にはPS2のゲーム開発の経験がありますが、以前開発した際には、このような綺麗な手法は取れず、結局、最初に中途半端に決めた構造のままマスターアップを迎えてしまいま

した。このような構造は1度決定してしまうと、チューニングの問題と絡み、後で変更するのが面倒(特に複数人で開発していると分担の内容によっては事実上不可能)ですから、最初に時間をかけて検討したいところです。

## テクスチャ転送

さて、ここまでGSを扱って、プリミティブを表示させる方法について述べましたので、次にプリミティブにテクスチャを張り付けます。

前述の三角形にテクスチャを張り付けるとなると、図9のように指定できるでしょう。追加すべきところは、GSのPRIMヘテクスチャ描画の属性を指定することと、三角形へのテクスチャ座標を指定するために、STQを指定することです。

テクスチャ座標をテクセル座標で直接指定したい時には、UVを使います。PS2のテクスチャサイズは2の巾乗でなければなりません、UVを使うとこのような制限を回避できますのでただ画像を表示させる場合などに便利です。しかし、UVを実際に使用する際、うまく指定できなくて悩んだ経験が誰でも1度はあるのでは

ないでしょうか。忘れがちですが、UVで指定するテクセル座標は、0.5ドットずらす必要があります。

さて、実際に描画バケットを積む前に、特にTEX0、TEX1よりも先にテクスチャをV-RAMへ転送しておく必要があります。

テクスチャをV-RAMへ転送する方法は、DMA転送バッファに転送命令を入れるだけです、これはGIFタグのImageモードを使います。転送サイズの制限(DMAの転送サイズは16ビット指定でき、128ビット単位で処理、GIFタグのNLOOPは15ビット指定でき、IMAGEモード転送時には128ビット単位で処理)がありますので、リスト5のようなプログラム構造で記述できます。

さて、このV-RAMへの転送ですが、PS2のV-RAMはご存知の通りあまり大きいとは言えませんので、最もV-RAMサイズの使用量が小さいNTSCなどであっても、V-RAMは簡単に一杯になってしまいます。

その代わり、PS2のGS-EEメモリ間は128bit 150MHzという非常に高速なバスで結ばれていますので、必要に応じてテクスチャデータを転送するといった作業が実目的に実行できます。従って、実際にはテクスチャの使用量にV-RAMを使用する前にV-RAM上の空き領域を確保して、転送先をダイナミックに決定して転送することになります。

GSのV-RAMの構造は図10のようになっています。

テクスチャ転送アドレスはブロック単位で指定できます。ただし、テクスチャフォーマットによってGSがインターリーブするメモリの構成が変化しますので、ページ単位より下のサイズで複数のテクセル格納フォーマットを混在させて管理したい場合には注意が必要です。少しでもV-RAMを有効に使いたいところですが、その努力によるGSの描画時間の短縮と、EEのCPU時間の消費を天秤にかける、バランス

【リスト5】

```
DMAタグ、VIFタグ、GIFタグをセット
BITBLTBUFをセット
TRXPOSをセット
TRXREGをセット
TRXDIRをセット
rest = 転送サイズ
while (rest > 0) {
    if (rest > 16*32767)
        len = 32767;
    else
        len = (rest+15)/16;
    DMAタグ、VIFタグ、GIFタグ(GIF_FLG_IMAGE)をセット
    VIFタグをセットして、テクスチャデータを
    DMAのrefタグで転送(p, len)
    rest -= len*16;
    p += len*16;
}
TEXFLUSHをセット
```

をとる必要があるでしょう。

4bit、8bit画像データの場合は、インデックスデータとクラットの2つに分かれてしまいうから、クラットの管理も必要になります。8bit画像のクラットはオフセット「0x08-0x0f」、「0x10-0x18」が入れ替わりますので注意が必要です。テクセル格納フォーマットについても、クラット付画像を意識してRGB24ビットテクスチャデータとインデックスデータを同時に配置するなど、活用できそうなものはいくつか用意されていますが、現実的に上手に管理して、全体として効率アップにつながるように使用すると、なかなか難しいものがあります。

ここまでくると、利用できるかどうかはPS2上で実際に実現する内容に左右されるのではないかと思います。

このような動的な領域の配置方法については、仮想記憶のメモリ割り当てアルゴリズムが応用できるでしょう。例えば、特別なハードウェアが必要ない単純なアルゴリズムを応用し、テクスチャが参照される回数を数えて、回数が少ないものからテクスチャ使用領域を開放する方法などがあるでしょう。この程度ならCPUをあまり余分に使わずにバランス良くテクスチャ割り当ての問題を解決できます。

しかし、動的なメモリ確保と配置についての最適な解を現実的な時間で得られる良いアルゴリズムは、現在のところありませんし、あらかじめフラグメンテーションの発生が起きない順番でデータを用意しておくようなことをしない限り、フラグメンテーション問題を完全に解決することはできません。どういう方法でどこまで実装するかは、CPUの使用量として許される時間と、実際に得られる意味のある効果とのトレードオフとなります。これは描画内容次第ですから、一般的な解決方法はないでしょう。

私は、こんなところに凝るくらいなら、V-RAMを使い切ったら潔くクリアしてしまった方がいいのではないかと考えます。理由は、描画される内容によりますが、PS2のV-RAMに対して転送するテクスチャは十分に小さいものから十分に大き過ぎるものまでバラエティに富んでいる上に、再利用されるテクスチャは実は非常に少ないので、勝手にテクスチャがV-RAMに残っていると、再利用可能という利点よりも、テクスチャ配置の邪魔になる確率の方がずっと高いのではないかと考えるためです。

実際にテクスチャ転送位について動的な配置管理を実装するにあたって、テクスチャ特有の特徴と思われることをいくつか挙げておきます。

- ・最適化されたプログラムでは、1フレーム

内で同じテクスチャが使用されることは、実はそう多くありません。そのようなテクスチャは非常に少数です。

テクスチャキャッシュを生かすためにも、一般的にテクスチャはソートしてまとめて使用します。例外は、描画の順番が重要となる半透明描画や、その他の特殊な理由があるものとなります。GSの半透明描画は半透明を使用しないときと比較してペナルティはありませんが、描画の順番を制御するとVPU1の利用が難しくなりますので、そもそもポリゴンをたくさん使いたいなら、半透明の少ないデザインにする必要があります。抜け道として、半透明に似た効果を生む加算などが場合によっては代用できるでしょう。

- ・テクスチャのうち頻繁に使用するものをV-RAMの領域固定にしたいと考えるかもしれませんが、実際の使用状況を考えたとき、効率がよくなる用途が考えられませんでした。結局ほとんどのテクスチャは1度しか使用されないことや、PS2のグラフィックパワーを十分に活用するアプリケーションの場合、V-RAMの内容が1フレームの描画で何度も内容を完全に置き換えられるため、これらを考慮すると固定したテクスチャはV-RAMアドレス割り当ての邪魔となるだけです。

さて、テクスチャ処理の実装にあたっては、他に、MIPMAP対応、複数クラット対応なども必要な作業でしょう。簡単に触れると次のようになります。

MIPMAPは描画サイズによって、いくつかのレベルの分けて、テクスチャを選択する技術です。小さくなったテクスチャが縮小の都合でちらついて汚くなるのを防いだり、小さなポリゴンに対して適したテクスチャを対応させて、描画速度の低下を防ぐことができます。

MIPMAPは、MIPMAP分のテクスチャをV-RAMに同時転送しておき、TEXレジスタやMIPTBPレジスタでパラメータを指定するだけです。動的に転送するテクスチャのサイズはMIPMAPを使わないときと比較して2倍近くになってしまいます。MIPMAPは、全部つなげてV-RAMに配置することもできます。この場合、V-RAMのフラグメンテーション問題が多少軽減されるでしょう。

複数クラットは、クラット付テクスチャ(4bit、8bit)のクラット部分のみを何種類か用意し、クラットのみを変更することでほとんどCPUを使わずに色を変更させる技術です。複数クラットをPS2で実現するためには、描画時にTEXレジ

スタに指定するクラットを実際に変更する方法がありますが、4bitクラット時に限り、クラットを8bitとして256色以内で用意してTEX0レジスタのCSAのoffsetで選択する方法があります。

さて、ここで1つ問題が生じます。複数クラットとMIPMAPを扱えるような、PS2のテクスチャにぴったりの汎用2Dフォーマットに、メジャーなツールが扱えるものが存在しないということです。2Dのフォーマットは別になくても作ればいいのですが、ツールまで用意するとすると、実に困ったことになります。

ライセンス向けにはサードパーティからTIM2形式というものが公開されています。実はこのフォーマットはPS2Linuxの登場とともにライセンス向けに限定する理由がなくなったので、一般公開予定の予定があるそうです。公開されるならばこのようなものが利用できるようでしょう。

TIM2形式は、もともと私が設計した\*1)もので、今回このような記事を書くにあたって使えれば楽でよかったのですが、残念ながらそううまくはいかないようです。

実際に、自分でフォーマットを作成する場合、汎用的にGSで必要な機能を網羅するための条件を簡単に挙げると次のようになります。

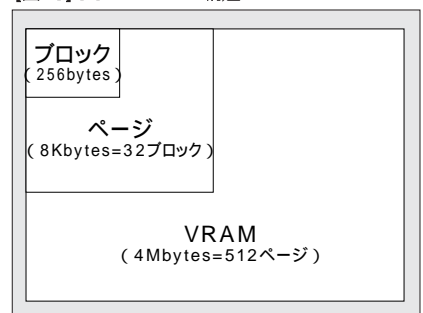
最低条件として、MIPMAPが持て、複数クラットが持て、DMA転送を意識した16バイトアライメントに対応したものであることが挙げられます。

あったら便利なものとしては、128バイトアライメントへの対応や、クラットのオフセット「0x08-0x0f」、「0x10-0x18」反転(4ビット複数クラット時の反転をサポート)、テクスチャ単位で変更したい可能性のあるパラメータのON/OFFが挙げられるでしょう。

パラメータは可能なものすべてを用意するとやたらと管理情報が大きくなってしまいます。使用するもの以外は領域を削れる構造を用意できれば、管理情報のサイズを小さく抑えられてベストだと思います。

このようなフォーマットは、たいしたノウハウ性がないにも関わらず、乱立するとツールがいろいろ

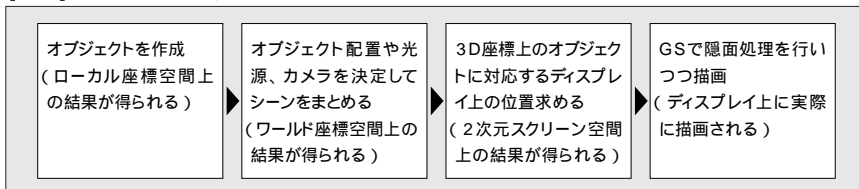
【図10】GSのV-RAMの構造



\*1) 以前、ゲームの開発と共に2Dのグラフィックツールの開発を担当していたのですが、オフィシャルな2Dフォーマットがなかったので用意したものです。このようなフォーマットは特にならぬノウハウがあるかもしれませんが、乱立するとツールがたくさんのフォーマットに対応しなければならぬので、たくさんありません。スタンダードなフォーマットが存在することが重要だと思います。



【図11】GSのV-RAMの構造



【リスト6】

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} m \\ n \end{pmatrix}$$

【リスト7】

$$\begin{pmatrix} x' \\ y' \\ w' \end{pmatrix} = \begin{pmatrix} a & b & m \\ c & d & n \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

いち対応しなくてはならないため、多くの方で共有するほうが良いと思います。

## ●サンプルプログラムの実装

今回のサンプルプログラムでの実装について触れますと、特に $\alpha$ 値を持つテクスチャを使う必要がありませんでしたので、Windowsの画像の縦横がそれぞれ2の中乗であるBMP形式ファイルを使用しています(4bit、8bit、24bitに対応)。ただし、文字表示の抜きなどでは、 $\alpha$ は0となります。プログラムとしては複数クラットも管理されていますが、BMP形式にはありませんので、使用することはできません。MIPMAPもやはり意味がありませんので、テクスチャの管理のコードは実装されていますが、MIPTBPなどのGSレジスタの設定が中途半端です。テクスチャの領域の確保はページ単位で動的に確保していきます。V-RAMがいっぱいになった場合はそのままV-RAMをクリアしてしまいます。このようになっている理由は、前述したような理由もありますが、複数のテクスチャをまとめて管理する機構が今回のサンプルプログラムには用意されていないことにあります。

以上、とても奥の深いPS2を扱うには簡単にしすぎましたが、本稿でのPS2の2次元のグラフィックとDMAの扱いについては一通り終了とします。

## グラフィックパイプラインとジオメトリ変換

続いて、3次元グラフィックの表示を行います。3次元グラフィック処理の手順は、まずローカル空間上で1つ1つのオブジェクトを作成し、続いてワールド座標空間上に作成したオブジェクトを配置し、光源、カメラなどを決定します。

続いて、作成された3次元座標上のデータを2Dスクリーン座標上に変換して、実際にGSで描画を行います(図11)。

GSでの描画は、すでに述べた通り、求めら

れた2D座標をそのままキックだけです。GSにはZバッファがありますので、半透明でないプリミティブについては、そのままZ座標を指定して描画するだけで描画の上下関係は解決されます。半透明やアンチエイリアスを扱うためには描画の順番を考慮する必要がありますから、Zソートが必要に応じて用意するなどといった処理が必要です。

残る問題としては、3次元グラフィックを実現する一連の流れで他に必要な技術として、3D座標 $(x, y, z)$ を視点 $C$ に従って2D座標 $(x', y')$ へ変換する作業が必要ですので、以下にこれを説明します。この処理を透視投影といいます。

## ●同次座標

座標変換というと、リスト6のような式を高校の数学あたりで勉強したことを思い出されるのではないのでしょうか。

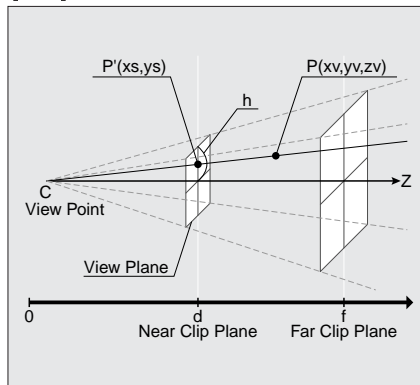
この式では、「平面上の点 $(x, y)$ を点 $(x', y')$ へ」という場合には任意の位置へ変換することができますが、平行移動のためには $(m, n)$ が必要です。これは同次座標(homogeneous coordinate)を導入することによって解消できます。この場合では、行列の次元を1つ上げることによって平行移動も乗算の行列に埋め込むことができ、ある点に対する任意の変換は1回の行列乗算のみで可能となります。

結果として、先程の座標変換式はリスト7のように記述できます。

今、対象としているものは2次元座標でしたが、これを3次元座標で実行する必要がありますので、3次元 $(x, y, z)$ の座標を4次元 $(x, y, z, w)$ で扱うことになります。

PS2はVPUを使用すると、4つの32ビット浮動小数の積和演算を1度に処理できますから、4x4行列と4次元ベクトルの演算は4命令で演算可能です。

【図12】3次元グラフィックレンダリングパイプライン



## ●透視投影

$P(xv, yv, zv)$ を視点 $C$ から見たView plane上の点 $P'(xs, ys, zs)$ に変換する透視投影は図12のようになっています。 $zs$ は描画時の上下関係の判断に使用します。リスト8のようになります。

視点からの距離を $d$ とし、 $d \sim f$ までの間の3次元空間上の座標を透視投影します( $d$ =視点からの距離=Near clip plane、 $f$ =far clip planeです。clip planeの内部をディスプレイに対して描画します)。これを $x$ に対して注目した図は図13のようになります。

ここで、View plane上で $xs$ が $[-1, 1]$ となるようにするならば、リスト9のようになります。同様にして $ys$ もリスト10のようになります。

$zs$ は、 $[0, 1]$ となるように、 $zv \in [d, f]$ を対応させると、リスト11のように表せます。

従って、リスト12になり、ただし、リスト13となるような $xyzw$ はリスト14のようになります。

従って、 $T_{pers}$ はリスト15のようになります。

$1/w$ は応用範囲が広く利用価値があります。1つ例を挙げると、テクスチャのテクセル座標(STQ)のQで、パースペクティブ補正などに使用されます。

実際には画面の比率はNTSCならば4:3ですので、今回のサンプルプログラムでは $T_{pers}$ の $d/h$ の $h$ を縦横で分離して、 $x, y$ の画面比率を扱っています。

さて、これで得られた領域はスクリーン解像度やZバッファの深度が考慮されていません。これらを $T_{pers}$ に埋め込むことは簡単にはできませんが、それを行うと、次に説明するポリゴン単位のクリッピングが実行できなくなってしまいます。

## ●クリップ

View plane内の $(xs, ys, zs)$ は、必ずリスト8となります。従って、描画しようとする図形がこの条件あるいは、リスト16を満たさない頂

点を含むかどうかによって、ポリゴンが画面外に出る可能性があるかどうかを調べられます。PS2のVPUにはこの目的のために利用できるCLIP命令があります。

クリッピングを行うときには、この判定の後、値を実際のPS2の解像度とZバッファの深度に対応させて図形を実際に描画します。NTSCインターレース 640×448ドット、Zバッファが24ビットの場合はリスト17のように処理できます。

GS上では2D座標上のシザリングによって画面外のポリゴンは描画させないことができますが、2D座標上になった時点で画面外かどうかは正しく判定できませんので、オブジェクトが画面外にかかる可能性があるときには、何らかのクリップ処理が必須です。

ところで、これで判別できるのはポリゴンの各頂点が画面内に収まっているかどうかということだけです。すべて収まっている場合は良いのですが、それ以外の場合は、完全に画面外かどうかすら判断できませんので、頂点が1つでも画面外にある場合は、もっと詳細に判断して、画面に収まるようにポリゴンを整形するなどの処置が必要になります。

今回のサンプルでは少し手を抜きまして、ポリゴンの表示判定の画面を大きめに行うことによって、正しく見えるように処理を行っています。

このようなクリップの他に、裏表判定によるクリップ処理もあります。裏側となるポリゴンは一般的に影になるポリゴンですから、描画を行わないようにしてGSに無駄な処理をさせないようにできます。

これらの話題は奥が深く、深く追求するときがありません。上記では、最低限のことに触れたのみですが、本稿での3DグラフィックスをPS2で実現するための方法については終わりにします。

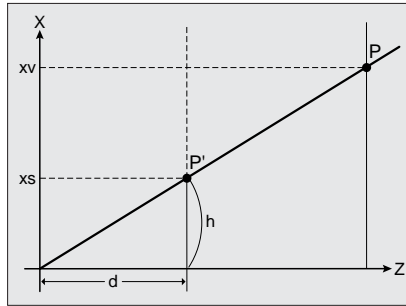
## パフォーマンス測定

PS2Linuxでのプログラミングにおいては、パフォーマンスの測定が重要になってきます。

PS2Linuxでは /usr/doc/PlayStation2 の下のドキュメントを見るだけで、precコマンドやgcc-profilingなど、いくつかこの目的のための方法が用意されていることが分かります。これらを用いて詳細にデータを調べるのが正しい方法でしょう。

しかし、PS2のグラフィックプログラミングでは、詳細なデータを採取してその内容を解析するというよりも、多くの場合、1フレーム当たりの計測時間を計測して、リアルタイムに表示し、処理内容に応じた時間の変化を調べ、時間のかかる処理を単純化していく作業を基本とした方が効率良く作業できます。

【図12】



【リスト8】

```
-1 < xs < 1, -1 < ys < 1, 0 < zs < 1
```

【リスト9】

$$xs = \frac{d}{hx} \cdot \frac{xv}{zv}$$

【リスト10】

$$ys = \frac{d}{hy} \cdot \frac{yv}{zv}$$

【リスト11】

$$zs = \frac{f}{f-d} \cdot \frac{df}{f-d} \cdot \frac{1}{zv}$$

【リスト12】

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = Tpers \begin{pmatrix} xv \\ yv \\ zv \\ 1 \end{pmatrix}$$

ここでは、このような作業方法を実現します。実際に計測する時間は、各フレームのEEの処理時間、GSの処理時間(PATH1、PATH3)です。GSの処理時間については、ハードウェアの仕様としてはDMAの終了割り込みやGSの割り込みから正確な値が測定できますが、今回は話を簡単にするために省略して、EEより処理時間が長いときにのみ正しい値を測定します。最終的にチューニングした際には、パフォーマンスの測定できなくなるのはGS側なので、これで当面の用は足りるでしょう。

計測した値は、棒グラフにして画面上に表示させます。

測るデータのバリエーションによって、測定方法のプログラム構造はいろいろと考えられますが、サンプルプログラムではリスト18のような構造で測定しています。

getcurrenttime() は、現在の時刻を得るところです。この目的のためには、

【リスト13】

```
xs = X/W
ys = Y/W
zs = Z/W
```

【リスト14】

$$X = \frac{d}{hx} \cdot xv$$

$$Y = \frac{d}{hy} \cdot yv$$

$$Z = \frac{f}{f-d} \cdot \frac{zv}{f-d} \cdot \frac{df}{f-d}$$

$$W = zv$$

【リスト15】

$$Tpers = \begin{pmatrix} \frac{d}{h} & 0 & 0 & 0 \\ 0 & \frac{d}{f} & 0 & 0 \\ 0 & 0 & \frac{f}{f-d} & -\frac{df}{f-d} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

【リスト16】

```
-W < X < W, -W < Y < W, 0 < Z < W
```

【リスト17】

$$Sx = \frac{640}{2} \cdot xs + 2048$$

$$Sy = \frac{224}{2} \cdot ys + 2048$$

$$Sz = -\frac{24}{(2-1)} \cdot zs + \frac{24}{2-1}$$

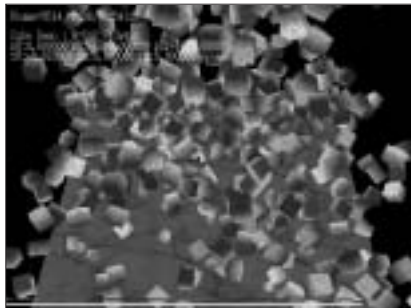
【リスト18】

```
[ フレーム前処理 ]
start = getcurrenttime()
[ EE処理 ]
[ 前フレームの処理時間を画面に描画 ]
ee_end = getcurrenttime()
wait DMA(path1) stop and
DMA(path3) stop
gs_end = getcurrenttime()
wait V-Sync
[ フレーム後処理、GSのダブルバッファ切替など ]

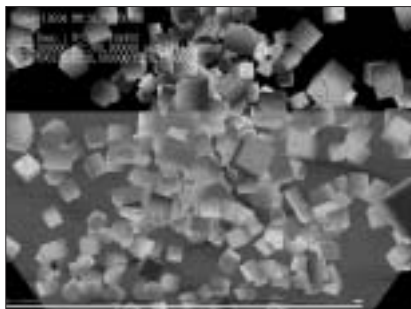
GSの処理時間 = gs_end - start
EEの処理時間 = ee_end - start
```

/dev/perf\_\* を使えそうですが、今回のサンプルプログラムでは別の方法を使用しています。今回は2種類用意しており、Linuxのgettimeofday()を用いた方法と、本誌12月号で解説したパッチを用いてI/Oポートを叩いて直接タイムT3を直接参照するものです。T3を扱うためには、正しいシステムクロックなどの情報が必要になります。ここはLinuxということで、カーネルソースが参考になります(/usr/src/linux/arch/mips/ps2

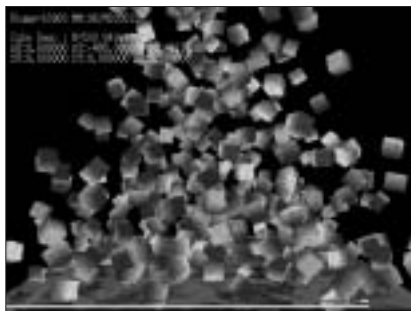
【画面1】



【画面2】



【画面3】



【画面4】



/kernel/time.c)。

## フレームバッファダンプ

GSの細かい処理を見たい場合、フレームバッファをファイルに落として見ることになるでしょう。

この場合、単にファイルへ落とすだけなのですが、NTSC/PALのインターレースの場合はフレームバッファが半分しかありませんので、解像度通りのV-RAMの内容を得るためには、1つ前のフレーム(ダブルバッファの裏側)から持ってくる必要があります。

1つ前のフレームと現在の表示内容が異なっている場合、当然フル解像度の画像を得る方法がないことが問題となりますが、これを回避する方法はありません。このため、画面の物体を一時的に止めておくなどの工夫が必要です。

いろいろな画面モードに対応させる場合は、このような違いを考慮したコードを記述する必要があります。

## サンプルプログラム

以上のものをまとめて、実装したサンプルプログラム「Cube Demo」を付録につけます。このプログラムは本稿の内容を実装して、以下を実現しています。

- ・ テクスチャを用いて複数の正方形の表示
- ・ パッドを用いてカメラをコントロール
- ・ フォント表示
- ・ フレームバッファのダンプ
- ・ パフォーマンス計測用棒グラフ表示

立方体をただ表示させるだけでは芸がありませんので、動きを付けてデモにしています。画面1～画面4のようにデフォルト480個の立方体が画面の中で飛び跳ねます。

プログラムの操作方法は次の通りです。

- ・ 左右のアナログと左の4ボタンでカメラ位置を移動します
- ・ ○ボタンと×ボタンで正方形の数を増減します
- ・ SELECTボタンを押している間一時停止。SELECTボタンを押している間にSTARTボタンを押すと、フレームバッファがそのままファイルにセーブされます。

プログラム構成は次のようになっています。

- ・ 初期化、設定ファイルなど  
Makefile  
MCONFIG  
config.h  
config\_ps2linux.h

common.h  
init.c

- ・ 立方体デモのメインルーチン、データ  
main.[hc]  
cube.[hc]

- ・ DMA処理、コンテキスト、GS設定など  
def\_dma.h  
def\_gs.h  
context.h  
draw.[hc]

- ・ PS2の2Dグラフィック処理  
prim.h  
prim2d.h  
bmp.h  
tex.[hc]  
font.[hc]

- ・ PS2の3Dグラフィック処理  
matrix.[hc]  
geometry.[hc]  
prim3d.[hc]

- ・ プロファイラ、フレームバッファセーブ  
debug.[hc]

- ・ その他のサブルーチン  
fileio.[hc]  
memory.[hc]  
input.[hc]

これらをコンパイルして実行するには、次のように入ります。

```
# make depend
# make
# make run
```

コンパイル時に表5A～表5Cのオプションを選択できます。その他、本文で触れていないことについて簡単に説明します。

立方体表示モデルは図14のようになっており、データはプログラム埋め込みで与えられ、Triangle Stripで描画させています。ジオメトリの計算はEEでべたに実行しています。結果として、ジオメトリの計算と表示データの設定だけでEEの95%も使っており、GSの方にかなりの余裕があることが想像できます。最後にテクスチャですが、フォントはX11のフォントから適当なものを、その他はRed Hat Linux 7.2J (FTP版)の/usr/share/nautilusのドにあるものから適当に持って来て、サイズやヒストグラムを修正し、BMP形式で保存したものです。