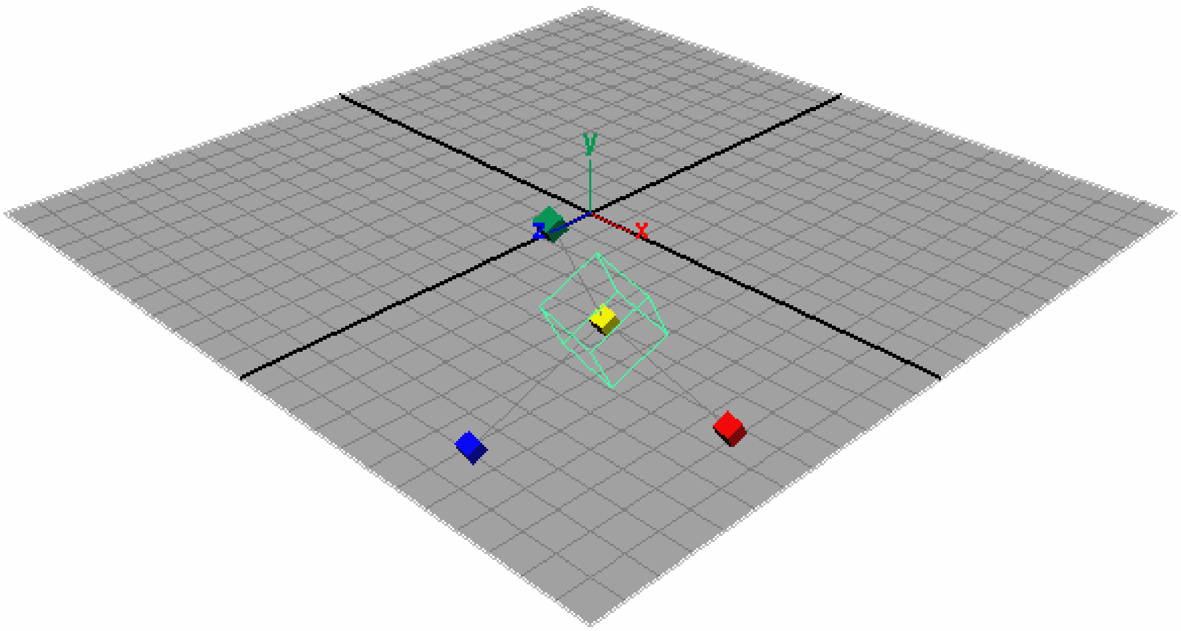
Programming PS2

Written by Alkampfer



Preface

The aim of this tutorial is to present a series of very simple programs to help the reader to familiarize with the architecture of Playstation2® and the Linux kit. All the examples are developed for the Linux Kit which can be easily purchased online. This document will not cover the installation of the kit and all the libraries, there are a plenty of documents in the Linux Kit home site http://playstation2-linux.com and I think that no one will have problem in installing and configuring the kit. The prerequisites to reading this document are essentially a basic knowledge of C/C++ and Linux Architecture as well as a knowledge of the basic notions needed to work with 3D graphics (Matrices, Vectors, homogeneous space, affine transforms and so on). Please excuse me for my English that is not very good, I will try to do my best to keep the document correct and smooth. Suggestions, corrections and comments on this document will be very appreciated as they contribute to improve the document and give me a feedback.

All the code is developed with basic distribution of linux kit except for SPS2 library downloadable at linux kit home site. Since header of SPS2 library are used through the code there is the need to specify to the compiler the directory where the SPS2 library resides, in my system for example is in directory /Develop/sps2dev-0.3.0a. Since SPS2 is surely located in different path in your system, it is sufficient to modify main makefile of the examples, changing variable SPS2DIR to the correct path. It is also possible to invoke make specifying the dir (ex. make SPS2DIR=/Develop/sps2dev-0.3.0a).

I hope you will like this work and have fun programming PS2.

1 Simple cube

1.1 First graphic application with playstation 2

This first example shows how to represent a simple cube with no texture and no lighting model applied on Playstation 2 machine, using standard PS2 library that is shipped along with Linux Kit. In this first example the aim is to achieve a basic knowledge of Graphic Synthesizer, consisting of initialization and communication with the GS engine. To keep things simple, for this first example, vector units are ignored and all calculations needed to represent the cube are done in C++ code by the main core of the EE.

First project is very simple, all interesting things about PS2 are contained in *main.cpp*, main file for the application. Others files contain code to work with matrix and vector and a more detailed explanation on them is found in appendix A. But now let's go exploring PS2.

1.2 Init Graphic Synthesizer

Initialization of the GS

First operation to be done in a PS2 system is to correctly initialize Graphic Synthesizer, the unit used to rasterize primitives on the screen. At the beginning of the program the instruction:

```
ps2_gs_gparam *gp = ps2_gs_get_gparam();
```

is used to obtain system information about current library. The function *ps2_gs_get_gparam()* cannot fail and fill the structure *ps2_gs_gparam* with all information needed to use the library. Detailed information about this structure could be found in documentation file of **libps2dev** library. Now the program must set video mode type (VESA, PAL, NTSC, …) this is done consulting first argument of the program: default video mode is VESA but it is possible to use switches *-ntsc* or *-pal* to select NTSC or PAL mode respectively. Then some standard configuration are stored in global variables:

```
      g_psm = PS2_GS_PSMCT32;
      g_zpsm = PS2_GS_PSMZ24;
      g_zbits = 24;
...
ps2_gs_vc_graphicsmode();
      g_fd_gs = ps2_gs_open(-1);
```

the program will use 32bit for front buffer and a Z-Buffer of 24 bits. Functions *ps2_gs_vc_graphicsmode*() and *ps2_gs_open()* are used respectively to set the console to graphics mode, and to initialize Graphic Synthesizer. Next step is specifying video mode and depth buffer mode:

```
ps2_gs_reset(0,
              PS2_GS_NOINTERLACE,
              g_out_mode,
              PS2_GS_FRAME,
              PS2_GS_640x480,
              PS2_GS_60Hz);

ps2_gs_set_dbuff(&g_db,
                  g_psm,
                  gp->width,
          gp->height,
                  PS2_GS_ZGREATER,
                 g_zpsm,
                 1);
```

How to set **Double Buffering**

Function *ps2_gs_reset()* is used to set the format of front buffer. Note that settings as 640x480 or 60 Hz are effective only if the video mode is set to VESA because PAL and NTSC have fixed dimensions and refresh rate. Function *ps2_gs_set_dbuff()* is used to set **double buffering**, this function essentially set the field of the *ps2_gs_dbuff* structure passed as first argument, remember also that only context one is stored. Second parameter specifies the format of front buffer, third and fourth parameters represent dimensions of the screen in pixel and their value is taken from structure initialized at the beginning of the program by the function *ps2_gs_get_gparam()*. Fifth parameter is the test of Z-Buffer, set to pass depth test if the z-coordinate of pixel is greater than the value already stored in the buffer. This is quite unusual because test is usually set to pass if the z-coordinate is smaller, because this means that the pixel is nearer to point of view. This difference arise because it is possibile to use left handed or rigth handed coordinate system, difference between these two representation are explanated in Appendix A. Finally there is the format of Z-Buffer and the last parameter that force GS to clean drawing area.

It is worth spending some time on *ps2_gs_dbuff* structure, that contains all information to deal with double buffering. It is necessary to set *clear0* and *clear1* members of that structure to specify color of the background of front buffer. They are of type *ps2_gs_clear* and for this first example the only member of interest is *rgbaq* used to specify back color. This is done by the instruction:

```
*(__u64 *)&g_db.clear0.rgbaq =
      PS2_GS_SETREG_RGBAQ(0x10, 0x10, 0x10, 0xFF,
                    0x3f800000);
```

Color is specified in a particular format that can be found in PS2 GS programming manual, is a 64 bit value in witch the lower 32 bits represent a standard RGBA color and upper 32 bits store the Q value used for perspective correction. This last value format is standard IEEE single floating point precision.

Default value used into the example is a gray, very near to black and 1.0f value for Q component. Note that *PS2_GS_SETREG_RGBAQ* macro expects data to be binary values, so it is necessary to specify binary form of single precision floating point number 1.0f that is well known to be 0x3F800000.

Now all settings are done, it is possible to initialize GS to made it ready for drawing:

```
//Initialize drawing environment and clear the buffers
ps2_gs_put_drawenv(&g_db.giftag1);

//Wait for all operations to be finished
ps2_gs_set_finish(&g_finish);
ps2_gs_wait_finish(&g_finish);

//Sync to V-Sync and start drawing in front buffer
odev = !ps2_gs_sync_v(0);
ps2_gs_start_display(1);
```

once these operations are done, the program enter in an infinite cycle, for every step of the cycle a frame is rendered in BackBuffer. First operation to do is lock the console to use it for drawing and swap the FrontBuffer with the BackBuffer:

```
ps2_gs_vc_lock();
ps2_gs_swap_dbuff(&g_db, frame);
```

After the scene is rendered into BackBuffer the program waits for V_Sync signal before switching to next frame, this is done to make frame rate equal to refresh rate of the monitor to minimize distortions. Now let’s take a deep look into the whole operation of setting geometry for this first cube.

1.3 Setting up geometry

Setting **Geometry** of the Cube, 3D coordinates and vertex colors

The object used for this first example is very simple because is a simple cube with colors stored in the vertex and no textures. No lighting model is used and all geometry stages are handled into EE without any use of Vector units. This is done because main target for this first tutorial is setting up GS and begin familiarizing with the PS2 engine, using Vector Units could made confusion for this first example. All vertex are stored in a global array of vertex.

```
PS2Math::Vector4 Cube[] = {
      PS2Math::Vector4(-5.0, -5.0, 5.0, 1.0),
      ...
```

Colors are stored in another array called CubeColor(), vertex colors are chosen to make the cube half green and half red, with green part up. Coordinate system used to represent vertex is a standard right-handed coordinate system (x, y, z).

Setting up, **World**, **View** and **Projection matrix**

Projection matrix is to be build taking into account aspect ratio of viewport, it is clearly necessary that near clipping plane has same aspect ratio of the viewport to avoid distortion. To calculate aspect ratio it is sufficient to use front buffer information contained into *ps2_gs_gparam* structure, remember also to take into account pixel dimensions of the monitor/TV used. Remember also that view matrix is left-handed and Projection Matrix has to be calculate accordingly.

```
float xyAspectRatio =
      static_cast<float>(gp->width * gp->pixel_ratio) /
                                 gp->height;
```

Last step is creating ViewportMapping transform, this is quite simple because all information about front buffer are stored in *ps2_gs_gparam*:

```
int xOffs = gp->width / 2;
int yOffs = gp->height / 2;
ProjMatrix.MapToViewPort(gp->center_x - xOffs,
                   gp->center_x + xOffs,
                   gp->center_y + yOffs,
                   gp->center_y - yOffs,
                   16777215.000000,
                    0);
```

Since Z-Buffer test was set to pass if the value is greater, there is the need to mirror coordinate system on Z direction, so object nearer to us have greater Z coordinate. This is done because a left-handed view matrix was used and PS2 machine Z-Buffer was build to be used with a right handed coordinate system. Note also that Y value of viewport grow from top to bottom so there is the need to mirror also Y coordinate of transformed vertex for the scene to be rendered correctly. These operation are done only once because view point is fixed for this example. After View and projection matrix are retrieved, for every frame, the position of the cube must be calculated.

To move the cube every frame, a World Matrix is generated storing a rotation value for each of the main coordinate axys, these increments are different to give to the cube a strange rotation movement. View point is fixed and it's chosen with eye position lying on z-axes at coordinate Z = 20 and looking towards the origin of coordinate system, up direction is the Y axis, this give us a good looking of the cube. Now it is sufficient to compose World, View, and Projection matrix to obtain final transformation matrix.

```
TransformationMatrix =
      ProjMatrix * ViewMatrix * WorldMatrix;
```

Since column vector notation is used, transformation of vectors is done premultiplying the vector with matrix. This means that to combine two or more transformations, matrix are to be multiplied in right to left order. Matrix representing first transformation is the rightmost one into the multiplication chain. In preceding example the order of application of three matrix are in fact: World, View and finally Projection matrix. Remember not to invert the order of these matrix because the result is very different from correct one.

1.4 Send data to the GS

Send data to GS with **DMA Controller**

To render the cube on the screen all is needed to do is sending information about vertices to Graphics Synthesizer; this operation is accomplished with a simple DMA transfer. Data transfer between various units of PS2 is in fact ruled by a DMA controller, this means that to do a DMA transfer, data is to be stored in a particular format called DMA packet. Manuals of PS2 contain all information about various DMA transfer modes, for this first example there is no need to do complex stuff and so a single packet is sufficient to transfer all data. First of all the application must allocate memory to build DMA packet into, this memory has to be 16 byte aligned to be used by DMA controller. A global area is declared using compiler directive to force alignment[1].

```
static __u8 __attribute__((aligned(16)))
      ScratchPad[16 * 1024];
```

This emulate scratchpad memory, but remember that it's normal memory. This area is passed to a routine called *MakeDMAPack()*, together with Transformation Matrix that is obtained by composition of World, View and Projection. Transformation Matrix is in fact needed to transform all the vertex of the cube from Model Space to the viewport coordinate system of the screen.

How to build **DMA Packet**

To build DMA packet first step is build the *DMA header*, a 128bit value specifying to DMA controller what is contained into the packet and the type of transfer to do.

```
int VertexNum = sizeof(Cube) / sizeof(Cube[0]);
int QWC = VertexNum * 2 + 1;

//This is the index into DMAPack
int DMAPackIndex = 0;

//Now create the HEADER
```

---

[1] Alignment can also be done manually allocating and aligning memory with malloc.

```
DmaPack[DMAPackIndex++].ul64[0] =
      MAKE_DMA_HEADER(QWC, 0, PS2_DMATAG_END, 0);
```

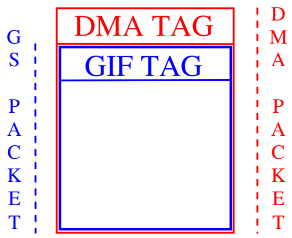
How to build a **Gif Packet**

To make simpler the creation of all bitfield representing: header, register setting, etc there are a lot of useful macro such as MAKE_DMA_HEADER:

```
#define MAKE_DMA_HEADER(QWC, PCE, ID, IRQ) \
      ((__u64)(QWC) | ((__u64)(PCE) << 26) | \
((__u64)(ID) << 28) | ((__u64)(IRQ) << 31))
```

this is a simple shift chain to build correct bit pattern with no pain for the programmer. This is useful because there is not the need to remember binary map of structure to build. QWC is the number of 128bit WORD that are to be transferred, PCE is a code that set priority of the packet, in this example there is no need of particular priority settings so 0 value is used. ID is the type of the packet and the value *PS2_DMATAG_END* means that after transferring this packed, DMA controller has to end transfer. Finally IRQ is used if an interrupt is needed after DMA transfer is ended.

To correctly build a DMA packet it is necessary to know exact structure of data to be transferred, this because there is the need to know the size of the whole packet to insert into DMA header. DMA controller in fact simply takes QWC WORD following DMA tag and transfer them to GS without any need to know what it's really contained into. Since there are a lot of information that could be sent to the rasterizer, Data sent to GS is organized in *GS Primitives* consisting in an header followed by the real data. This header is called *GIFtag* and specify format of data contained in the primitive so GS understand what to do with these data. More that one GS Primitive can be send at once, field EOP on GifTag is used to tell GS if current primitive is the last one or another follows. Last primitive have EOP = 1, this means that there is no more primitive following the current, a sequence of consecutive primitives are called *GS packet*. Structure of whole DMA packet is represented in the picture at the left, it is simply a GS packet with only a GS Primitive contained into one DMA packet.



Structure of GS packet is more complex than DMA packet, this because there are a lot of different type of data that the EE can send to GS. In this example only vertices information about coordinates and colors are sent and packet structure is very simple, but remember that general structure of GS Packet is the same even for more complex transfers. As for DMA tag a macro called *SCE_GIF_SET_TAG* is used to set Gif Tag.

```
SCE_GIF_SET_TAG(VertexNum, 1, 1, PrimReg,
      PS2_GIFTAG_FLG_PACKED, 2);
```

Since GS packet is used to transfer a series of identical data structures (usually array of vertices), first part of the GIF tag, called NLOOP, specify number of data structures contained in the primitive, in this first example this number is equal to the number of vertices. This transfer type is called PACKED mode, and means that a certain number of data structure follow GIF tag, NLOOP contains number of these structures. Then EOP is used to specify if there is another primitive following this packet, in this example is set to 1 because this is the only GS primitive contained in GS Packet. Field called PRE is particular and it is used to inform GS to set PRIM register with the value contained into the GIF Tag. PRIM register is the one used to set primitive rasterizing options.

Specifying **PRIM** content for **each GS Primitive**

Even if it's possible to set PRIM globally for all primitives, it is sometimes useful to set a different value for each primitive. To accomplish this operation, GS permits to specify new value of PRIM register directly into GIF Tag. In this way the content of PRIM reg could be changed for every data transfer.

Then PACKED transfer mode is chosen and finally the number of elements contained in a single data structure is specified in NREG field, this last value is 2 because for every vertex only color and coordinates are to be send.

Macro *SCE_GIF_SET_TAG* help to set lower 64bit part of the Gif Tag, upper 64bit part is used to specify type of the data contained into the structure, every field is specified with a 4 bit code, so a data structure can contain at most 16 different type of data. To easily set these value a particular structure called *sceGifTag* is used, here is the code that set data structure to contain color and vertex coordinates:

```
    sceGifTag *GifTag =
         (sceGifTag *) &DmaPack[DMAPackIndex];
         GifTag->REGS0 = PS2_GIFTAG_REGS_RGBAQ;
         GifTag->REGS1 = PS2_GIFTAG_REGS_XYZ2;
```

Specifying format of data contained in **GIF Packet**

This specify that every data pack (vertex) contains an RGBAQ data and a XYZ2 data. Note that whenever a XYZ2 data is written to the corresponding register, the primitive point will be rasterized on the screen, this means that for every vertex data structure, XYZ2 information has to be the last part of the structure. Consult PS2 manuals for further details.

Now let's take a step back, because the whole structure of GIF Packed is known, it is possible to calculate its size to set QWC part of DMA header. Size of the Gif Packet is equal to number of vertex times size of the structure (VertexNum * 2), plus the size of the Gif Tag that is 1. Remember that QWC size is give in 128bit words units, this means that total number of qwords to send is:

```
int QWC = VertexNum * 2 + 1;
```

Another step back is to be done to explain how to build format for PRIM Register:

```
__u32 PrimReg = SCE_GS_SET_PRIM(4, 1, 0, 0, 0, 0, 0, 0, 0);
```

structure of PRIM register is contained in GS manual, in this example primitive is given in triangle strip, Gouraud shading is used and texturing is disabled. Value of PRIM register is then passed to *SCE_GIF_SET_TAG* to be included into GIFtag, this is effective only if PRE field of GIFtag is 1.

Calculating **coordinates value** for vertex to be stored in **front buffer**

After setting DMA Tag and GIFtag, rest of the packet is filled with vertices information: color info are stored directly into CubeColor global variables and vertices coordinates are obtained transforming each vertex with TransformationMatrix. The whole process is simple and well commented but it is worth enough to spent time on the routine *ConvertToFixedPoint*() called before sending vertices data to GS.

```
float  PS2Math::Vector4::ConvertToFixedPoint(int Vertex[]) {

  float divw;

  divw = 1.0 / w;
  Vertex[0] = (int) (x * divw) << 4;
  Vertex[1] = (int) (y * divw) << 4;
  Vertex[2] = (int) (z * divw);
  Vertex[3] = 1;

  return divw;
}
```

Format of front buffer coordinate system is 16 bit fixed point number, with 12 bits for mantissa and 4 for fractional part, but vertices coordinates are stored in 32 bits float variables. Routine ConvertToFixedPoint() performs required transformation between these two different representations and also deomogenize the vector. To accomplish this, x and y values must be first divided by w, to obtain coordinate in 2d space of the screen and then transformed to fixed point format of front buffer. This transformation between floating point value and fixed point 16bit value is done in two parts, first the value is cast to an int, dropping fractional part, then a shift of 4 digit makes integer part fit into 12 bit mantissa of the Front Buffer. Dropping fractional part means that if the final (X, Y) value of the coordinate is (5.7, 10.3) pixel (5, 10) it is used to represent the vertex, so a little approximation is done, if necessary rounding can be used to minimize the error. Once DMA packet is build it is sufficient to send it to DMA controller.

Send packet to **DMA Controller**

```
ps2_dma_start(g_fd_gs, NULL, (ps2_dmatag *)ScratchPad);
```

This makes PS2 rasterize the cube on television or monitor. The only operation left is waiting for the next frame to come.

```
frame++;
odev = !ps2_gs_sync_v(0);
ps2_gs_vc_unlock();
```

Function *ps2_gs_sync_v(0)* is used to wait next V-Blank signal, and *ps2_gs_vc_unlock()* function is used to unlock virtual console so there is possibility for other application to use it.

1.5 SPS2 library

What is **SPS2** library

Insead of using standard library shipped with the kit, most of the people owning linux kit are using different library known as **SPS2**, written by **Steven Osman (AKA Sauce)**. The reasons of success of this library is essentially direct access to PS2 internal hardware DMAC and VU/GS registers. SPS2 make possible work with PS2 console in a similar way as professional ps2 programmers do. For this reason first example seen in previous example will be translated into SPS2 library and all following tutorials will be written mainly for sps2 library. To run *01-MovingCube_sps2* example, SPS2 module and header must be correctly installed and configured to make program compile and run.

Risk of using SPS2

Main difference between *SPS2* and *libps2dev* is that direct access to DMAC and internal register of the machine can crash the whole system if the program contain some bug. Crash mean that ps2 linux system must be rebooted, file system check will occur wasting precious time, so best thing to do is to double check content of DMA packet, GIF tag, etc… before actually send data to corresponding hardware. To accomplish this, a simple utility written in VB.NET (and so compatible only with Windows system with framework .NET installed) is included in this tutorial. This utility, still in a early beta developing stage, permits to build the content of a particular register or header of PS2 machine using an easy interface, it permits also to decode in a efficient way GIF tag and other register formats.

First thing to do is make a routine to dump memory content to console, since I use linux kit from my windows system with SSH connection, using a standard *printf()* function make text appears on my remote console, while the scene is rendered on my television. The routine *DumpDMAPack* is used to dump a number of 128 bit word to screen, starting from a given memory address:

```
void DumpDMAPack(void *Data, int Num) {

        //D is a pointer to sps2uint32[4] element.
        sps2uint32 (*D)[4] = (sps2uint32 (*)[4]) Data;
        for (int I = 0; I < Num; ++I) {

            printf("%d:\t%X %X %X %X\n", I,
            D[I][0], D[I][1], D[I][2], D[I][3]);
        }
}
```

memory is simply cast to a pointer to array of four int, every 128 bit word is dumped subdivided into four 32 bit part, from right to left because PS2 layout of data is little endian.

01-FirstCube_sps2

Converting program from libps2dev to SPS2 library is straightforward, since the hard part of the work is still the same: using matrix and build DMA packet for GS. The only things that are changed are: initialization part and the instructions used to use DMAC controller. All initialization code is reduced to a simple call to init function of the library, followed by the creation of a suitable memory area to build DMA packet into.

```
iSPS2Descriptor=sps2Init();

pMemory=sps2Allocate(4096,
        SPS2_MAP_BLOCK_4K | SPS2_MAP_UNCACHED,
        iSPS2Descriptor);

sps2UScreenInit(0);
```

Initing and **allocating memory** on SPS2

The function *sps2Init()* is enough to initialize the whole library, *sps2Allocate()* is then used to allocate aligned memory in multiple of 4Kb block. This particular routine is used because DMAC can only transfer block of memory that are 16 byte aligned and it wants Physical addresses not virtual ones. Function sps2Allocate() permit to allocate memory in a particular structure to retrieve its physical address. Finally *sps2ScreenInit()* is used to initialize the screen of the console. All these functions are well documented in the SPS2 library documentation, so there is no use to spent more word on them.

Projection matrix is build the same way as the preceding example with a slightly difference: aspect ratio is set to canonical 4/3 value, and the centre of the viewport is mapped at point (2048, 2048) of frame buffer. This value is specific to SPS2 library. Dimension of the viewport is simply retrieved with the functions *sps2UScreenGetWidth()* and *sps2UScreenGetHeight()*.

Talking to **DMAC**

After initialization it is time to start rendering cycle: function *sps2UScreenClear()* is used to clear the buffer, then we must build DMA packet containing vertex data to be sent to GS processor. Construction of DMA packet is identical to previous example, only one little thing is changed, there is no DMA header and packet begins directly with GIFtag. This because in the preceding example a simple address is passed to DMAC to start transfer, in this way the DMAC can control the header at that address to know type and size of data to be transferred. With SPS2, with direct access to PS2 hardware, it can be possible to instruct DMAC directly accessing its registers to specify type of transfer. DMAC registers are memory mapped and SPS2 library have suitable declaration that permit to access them through pointer syntax, every register is mapped as a simple pointer and data can be stored and read on it.

Setting and starting **DMA transfer**

To send data with DMAC three registers are used, **Dn_QWC**, **Dn_MADR** and **Dn_CHCR**, where n indicates channel to be used. Page 59 of EE manual states that GS channel have number 2, so registers used to transfer data to GS are called **D2_QWC**, **D2_MADR** and **D2_CHCR**. D2_QWC contains size of data to be transferred and is the first register to be set, D2_MADR contains address of data to be transferred and it is set immediately after QWC. Remember that DMAC takes no virtual address but only physical ones, so SPS2 comes with useful routine that permit to allocate memory and obtain its physical address. The only drawback of this is that memory should be allocated in chunk of 4KB and if more than one block is allocated, then physical block can be no contiguous, but for this first example this issue is not really a problem.

Last register to be set is D2_CHCR, that is called *channel control register*, it is used to specify type of the transfer and also to start transfer itself. In fact, when the byte 8 (STR) is set to 1, DMAC begins the transfer, using data contained in this 3 registers. Since in this application only a simple packet is send, D2_CHCR value is calculated before rendering cycle, using structure *Dn_CHCR_t* that help us to set correct values for the register.

Ending application.

After setting D2_CHCR register, the application needs to wait for DMA operation to complete, and then swap the frame buffer:

```
//Wait for the DMA transfer to finish
sps2WaitForDMA(2, iSPS2Descriptor);

//Swap displays now that we're done
sps2UScreenSwap();
```

Finally, after rendering cycle is ended, allocated memory must be free and library must be deinitialized.

```
//Shut down the screen
sps2UScreenShutdown();

//Free memory used to store DMA packet
sps2Free(pMemory);
```

```
    //Close library
    sps2Release(iSPS2Descriptor);
```

Particular attention must be paid before actually sending data to GS, this because, as stated before, sending wrong data can crash the system, especially if wrong values are written to DMAC registers. One way to avoid this problem, is dumping to screen packet to be send turning off transfer of data, in this way there is the possibility to check the data before doing some damage to the file system. Simple function *DumpDMAPack()* permits to dump content of memory area, as seen before. Then is possible to insert values of GIFtag in program RegMaker[2] included with the tutorial; pressing Sync button makes the program analyze content of the register actually showing values of the various parts that compose register itself. In this way, checking GIFtag value is really a breeze.

As final note remember that to use SPS2 library there is the necessity to specify directory of installation of the library, this can be easily accomplished by variable **SPS2DIR** present in makefile. In my system I installed SPS2 library in path **/Develop/sps2dev-0.3.0a** so to compile the code I invoke make utility with this command line:

```
make SPS2DIR=/Develop/sps2dev-0.3.0a
```

Since playstation2 linux community seems to prefer SPS2 library, rest of this tutorial will concern mainly on SPS2 library. Thanks again to Steven Osman (sauce) for making SPS2 available to linux kit community.

---

[2] This program is a very simple utility still in early beta form, only few registers format are supported, even if adding new register format is straightforward in its xml configuration file.

2 Controller

2.1 Knowing how to read JoyPad data

PS2 Peripherals: The **controller**

Using joypad in PS2 system is not difficult, but standard joypad library of linux system makes impossible to access all functionalities of the pad. To have maximum control over whole set of functions that DualShock offers to the programmer, there is the need to descend at low level, and read pad data in binary raw format. Linux kit can in fact map joypad as standard device with name **/dev/ps2pad00** and **/dev/psdpad10**, these two files represent devices connected in the two standard joypad ports of PS2 console. Presence of only two devices files means that multitap is not supported and is impossible using more than two joypad simultaneously.

Reading data and **interact** with the controller

To access a joypad corresponding device must be opened with standard C function *open()*, that returns an handle to the file. Once the file is opened, pad data can be read with standard *read()* function. To interact with the device *ioctl()* function must be used, comprehensive list of all the services supported by the pad are listed and explained with detail in document file *ps2pad_en.txt* located in standard Playstation documentation directory */usr/doc/PlayStation2/*, supposing that kernel documentation is correctly installed into the system. Documentation on pad still lacks the definition of structure of data read from the joypad devices. With read() function 32 bytes of data can be read from the joypad with every call, but there is no documentation file that tells us how to use these values. Solution to this problem is building a little program that dump these data on screen, in this way acting on the pad make possible finding how the button and the axis are mapped. Example *02-ControllerData* is build to make this check. Code of the example is straightforward: pad is first opened, than a check is made to see if the pad is functional, then analog pressure of button is activated and a loop, in witch data is read and dump from the pad, is started.

Dumping data content

Using output of this utility it is possible to find that digital states (pressed/unpressed) of all buttons are stored in byte number 3 and 4 of raw data, first byte is zero and byte 2 represents type and status of the pad. Byte 2 is in fact composed by three distinct values: bit 0-1 status of the pad, bit 2-3 status of asynchronous operation on the pad, bit 4-7 type of pad connected to the port. All the analog values (Buttons, sticks and directional cross) are mapped with a single unsigned byte value, ranging from 0 (no pressure) to 255 (maximum pressure). Every analog value needs a whole byte to be represented, so byte 4 through 19 are used to map analog values. In library header file *Pad.h*, some values are defined to help working with raw data, in particular there are the mask to check digital buttons. Remember also that a value of 0 means that the button is pressed while a value of 1 means that button is unpressed.

2.2 Interact with the joypad

Accessing joypad **features** with **ioctl**

Knowing structure of data read from joypad is not enough; to fully access all its features there is the need to know the various ioctl() services supported by the pad itself. Each of them is characterized by a constant value that identify the operation and an optional value if needed by requested operation. Whole set of services supported by the pad is contained in help file.

Different **Access Modes**

Before looking at most useful services supported by the pad, we must understand joypad features, first of all device can be open in two distinct mode: *block* mode (default) in witch program execution is suspended whenever some service is used or *non-block*, useful to manage asynchronous operations. This is useful to execute all long operation of the pad, in this way program can continue the execution while the pad has not completed requested task. Blocking mode is enough for use with simple applications, so do not mind using non blocking access type for now.

Analog and **Digital** mode of the pad

DualShock and Analog pad have also two distinct operating mode: *analog* and *digital*, moreover, in analog mode, all buttons support reading analog pressure of the button, not only digital (1/0) state. User can choose to switch between analog and digital mode with a button located on the joypad, while *analog buttons mode* must be set with code, this because not every application needs to know analog pressure of button, and so it is enabled only when really needed.

Switching between analog and digital mode can be done even with code, it is sufficient to use PS2PAD_IOCSETMODE service with ioctl(). This service requires a structure made of two int as parameter: *offs* is used to specify the mode to operate with, 0 is for digital and 1 is for analog, while *lock* is used to determine what to do with that setting. Value of 0 left setting unchanged, value of 1 enable selected mode, value of 3 lock the pad in selected mode and finally value of 2 unlock previous locked setting. When an application lock the joypad in a specific mode (analog/digital) user cannot use button on the joypad to switch between these two. This is done to prevent user disable analog mode with application that does not work with digital mode. Remember that value of 2 does not change the state of the setting, but it only unlock a previous locked setting.

Choosing and **locking** mode of the pad

```
ps2pad_mode PS2PMode;
PS2PMode.offs = 1;
PS2PMode.lock = 3;
ioctl(Joypad0, PS2PAD_IOCSETMODE, &PS2PMode);
```

Remember to unlock all settings before exiting from the program, if not the device remains locked in selected state. Lock is particularly useful for application that needs to activate analog buttons, this because if the user switch to digital mode and then to analog mode again, analog pressure of the buttons is disabled and must be set again by the application. In this situation is vital locking the pad in analog mode to avoid this problem. Once analog mode is activated, analog buttons pressure can be enabled/disabled with a simple ioctl()call:

```
ioctl(Joypad0, PS2PAD_IOCENTERPRESSMODE); //Enable
ioctl(Joypad0, PS2PAD_IOCEXITPRESSMODE);  //Disable
```

Query pad for informations

Another fundamental service is PS2PAD_IOCGETSTAT, used to query status of the pad. Pad can in fact be in one of these states: *ready*, *busy*, *error* or *not present*. To read status of the pad it is necessary to make a cycle, reading again until the pad is busy:

```
int PadStatus =
      ioctl(Joypad0, PS2PAD_IOCGETSTAT, &PadStatus);
while(PadStatus == PS2PAD_STAT_BUSY) {

      ioctl(Joypad0, PS2PAD_IOCGETSTAT, &PadStatus);
}
```

The only parameter associated to PS2PAD_IOCGETSTAT is the address of an int variable that will contain the state of the pad at the end of ioctl() function.

Actuators and **DualShock**®

Last set of ioctl() services regard the actuators of the pad. Dual Shock has two different actuators, the first (ID 0) has fixed intensity while the second (ID 1) has an intensity that can be changed with continuity from 0 to 255. Both of these actuators must be set with a single ioctl() service call: PS2PAD_IOCSETACT that use a simple structure called *ps2pad* as parameter:

```
struct ps2pad_act {

  int len;
  unsigned char data[32];
};
```

Member *len* of the structure has to be set to value 6, accordingly to the documentation, *data* is a simple array of char values that are passed to corresponding actuator. Remember that indexes of data array does not correspond to actuator ID, in fact before actuators can be used with PS2PAD_IOCSETACT, ID must be assigned to the index of data array with service PS2PAD_IOCSETACTALIGN. This can be accomplished with a simple call:

```
ps2pad_act PSact;
PSact.len = 6;
memset(PSact.data, -1, sizeof(PSact.data));
PSact.data[0] = 0; //Actuator 0
PSact.data[1] = 1; //Actuator 1
ioctl(Joypad0, PS2PAD_IOCSETACTALIGN, &PSact);
```

This means that actuator ID number 0 is related to first element of data array while actuator with ID 1 correspond to second element, in this way actuator’s ID and index of data array in ps2pad structure are the same, this is the most natural setting. Now it is possible to send value 1 to actuator zero to have constant vibration or set an intensity value to actuator 2 to have a range of vibration.

2.3 Building a class to handle controller

JoypadManager class to handle controllers

Now that the structure of data read from the pad is well known it is possible to write a simple class to handle joypad data, this is necessary to access data more easily. The name of this class is JoypadManager, and is build to support basic functions, many things could be improved as for example access mode that is made only synchronous. Project *02-ControlledManager* contains the definition of the class together with a simple program to test basic functions of the JoypadManager Class.

How to use JoypadManager class

Construction of the class is very simple and so there is no need to analyze its structure, help file can show whatever information is needed on the class. Here follows a simple example that shows how to use JoypadManager objects to retrieve information about status of digital buttons. To access a joypad it is sufficient to declare a new JoypadManager object, passing the ID of the pad to class constructor, ID 0 means pad in port 1 and ID 1 means pad in port 2. Then is necessary to check if the joypad is ready to use.

To read data from the pad member function *ReadPadData()* must be called, this function read data from the pad and update its internal structure of button status. Possible states of digital buttons are:

- *Pressed*: button is pressed and was not pressed before
- *MPressed*: button is maintained pressed, it means that the button

was pressed last call

- *UnPressed*: Button is not pressed and was not pressed last call
- *Released*: Button is not pressed but it was pressed last call.

JoypadManager maintains information on last pressure state of the button, this is very useful for example to check when a button is released.

When in analog mode, digital status are still retrieved, but for every button it is possible to know its pressure value, normalized in range [0.0f, 1.0f]. Analog sticks are also enabled and their value is normalized to the interval [-1.0f, 1.0f] too. To access buttons data is sufficient accessing array of structure *AnalogButton* that contain information of all buttons present on the joypad.

Playing with **actuators**

Finally there is the possibility to use actuators, but first a call to member function *EnableActuators()* must be done to initialize both actuators. To make use of the class easier as possible, each of the two actuator can be used separately from the other, so there are two member functions to play with actuators, the first *SetActuatorConst()* enable/disable constant actuator, while *SetActuatorVariable()* is used to set vibration value of variable actuator, valid range is [0, 255].

Remember that is possible to lock joypad in analog mode, but is necessary to unlock it before exiting from the application, if not, joypad will remain lock even after the application is shuttled down. Moreover when playing with actuators it is also necessary to shut down both of them when exiting from the program, to avoid pad continue shaking without control after the application is closed. To avoid these problems, locking of pad in analog mode is automatically removed in destructor of the class, but actuators status must be reset from the application, this because there is the possibility that the user really wants the joypad to continue rumbling even when the JoypadManager object is destroyed.

Last example puts together first example of the cube and JoypadManager class, the result is a cube that can be moved with left analog stick and shaken with cross and circle buttons. Applying movement to the cube is a very simple process because it's sufficient to analyze pad data every frame updating world matrix that represents cube position in the scene.

3 Texturing

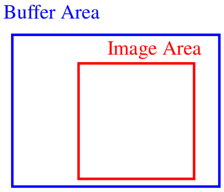
3.1 Transfer a texture to GS

Transfer Texture data to GS

A texture is nothing more than a sequence of pixel arranged in a square matrix, a simple raster image, that is applied to the primitive rasterized by GS processor. This means that texture data are to be transferred into GS memory before the texture can be used, this because GS can access only its internal memory. Transfer of large portion of data into GS memory is ruled by five internal register called BITBLTBUF, TRXPOS, TRXREG, TRXDIT and HWREG. Among these register, HWREG plays a special role, and will be discussed after the others. First four registers specify destination address and pixel format of the image to be transferred and must be set before HWREG is accessed.

Specifying **buffer** properties

BITBLTBUF register is used to specify propertied of the buffer to upload the texture into, it contains address of GS memory where to upload the texture, type of pixel and width of the buffer used. This register has six field, three related to transfer from host memory to local GS memory and the other used with transfer from GS local memory to host memory[3]. To transfer texture, direction is *from host to* local, this means that only upper word of the register is used. Remember that BITBLTBUF register contains *buffer* area properties where the image is located, but the image itself can be a subset of this area.



DBP field **address units**

It is worth spending some time on measure unit of the three quantity used to specify image data. Address in GS memory (DBP) is given in unit of 64 word, this mean that if we have size in bytes, since every word is 4 byte, original address has to be divided by 256 (64 * 4) to obtain correct value for DBP field. Usually first texture is uploaded into first free page of GS RAM, that is the first page after frame and Z buffer. This address can be retrieved with a call to sps2 library function *sps2UScreenGetFirstFreeGSPage()* that gives the number of the first free page in GS memory. Since a page in GS memory is 8192 bytes, to convert this value into correct measure unit for DBP, we must first multiply page address by 8192 and then divide by 256, resulting in a overall multiplication of original page address by 32. Even the DBW field, representing destination buffer width, must be expressed in unit of 64 pixel, if an image is 128x128 pixels, this value is 2. Finally DPSM contain an enumerated value representing format of the pixel.

---

[3] In a later section of this chapter will be analyzed the code to make a screenshot of a running program.

Coordinate offset of uploading address

TRXPOS is used to specify upper left point coordinate in buffer transmission area. For this first example only one texture at a time is transferred and it is positioned at the top of the page, so both X and Y coordinate must be zero. If a large number of small texture are to be uploaded it is more convenient arrange all texture in a bigger one, and then upload all of them as a single texture.

Size of the image

TRXREG specify width and height of the image to be transferred. With this last register, all information needed to store texture data into GS memory are set. Now to start the transfer, value 0 (Host to local) must be set in register *TRXDIR*, this makes the transfer start. Once the TRXDIR is accessed and the transfer started, to send data we must write each pixel, from upper left to down right, to the register **HWREG**, GS will do the rest, placing value in correct memory location. This particular way to transfer data makes possible to transfer other data while transferring texels, documentation says that transferring data in this way does not overtake a drawing primitive command issued before TRXDIR register was set. This means that it is possible to transfer texture and primitive at the same time.

How to send **whole texture** to HWREG

Since setting HWREG in packed mode can be done only with A+D addressing, it is clear that transferring texel data with this method is virtually impossible. For this reason GIFtag can specify also *IMAGE mode*, where NLOOP is the number of qword to be transferred, and all qwords following GIFtag are considered to be a continuous stream of dword (64 bits), that will be written sequentially to HWREG register. To transfer whole texture data a single GIFtag followed by all texel of the image would be enough, but remember that is impossible to allocate a large portion of physically contiguous RAM. This problem and the real structure of DMA packet used to transfer texture data into GS will be discussed in next paragraphs.

Ending and flushing data transfer.

Last operation is ending transfer of texture data setting TRXDIR to binary value 11, this makes the transfer ends. A better approach is access TEXFLUSH register (write any value on it), this makes GS wait for transfer operation to finish and then stop transfer.

3.2 DMA transfer of large portion of data

To transfer Texture from local memory to GS memory a DMA transfer is needed, but it is not possible to use **Normal** transfer mode because data often excess 4KB of size. Normal transfer mode is used to transfer data that

Lack of **physical contiguity**

are contiguous in physical memory[4] but remember that sps2Allocate() cannot guarantee physical contiguity of allocated pages. For this reason it is impossible to transfer whole texture with a single DMA packet, to handle this problem, DMAC controller supports *Chain Transfer Mode*, used to transfer large portion of data with a single DMA transfer splitted in many small packages, each of them smaller or equal to 4KB. Chain transfer mode supports both transferring from main memory to peripherals and back, to send texture data we are interested only in *Source Chain Mode*, used to transfer data from main memory to peripherals. This kind of transfer is started in a similar way as normal mode, setting Dn_MADR, Dn_QWC and Dn_CHCR registers, but another register is used, Dn_TADR. To start transfer is also needed to specify Source Chain Mode in Dn_CHCR register. When the transfer start, DMAC begins transferring QWC words from the address stored in Dn_MADR, but when this transfer is ended, transfer is continued reading the DMA tag stored in memory pointed by Dn_TADR. This tag contains three information

Source chain transfer mode

DMA tag in source chain mode

- Address that contain the data to be transferred next (goes to Dn_MADR)
- Size of data to be transferred next (goes to Dn_QWC)
- Instruction on how to locate next tag to continue transfer.

The name "Chain Transfer mode" means that a chain of DMA transfer is done, in with every transfer (ruled by a DMA tag) contains both the information of data to transfer as well as information to locate tag of next transfer.

DMA chain tag type

Different type of tag are supported by the DMAC to build different transfer chain, all these types are described in PS2 manuals and each of them will be explained in detail when encountered in the tutorial. In this chapter it is enough to focus on *ref* and *cnt* type, used to transfer texture data. All of these tag are based on the same structure represented at page 32 of GS manuals. From the picture it can be easily seen that they are composed by three distinct parts, usd to specify Size, Location of data and location of the next tag.

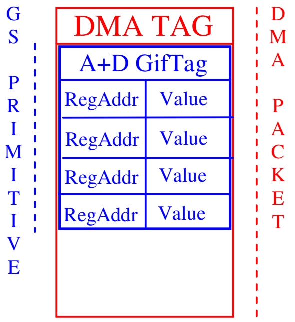
3.3 How to build DMA packet to transfer a texture to GS memory

Using **A+D transfer** with GIFtag

As for vertex data, first qword of DMA packet is a GIFtag, as seen before first step is setting the four registers used to identify texture data, this makes data transfer begin. Looking at GIFtag structure we find that these

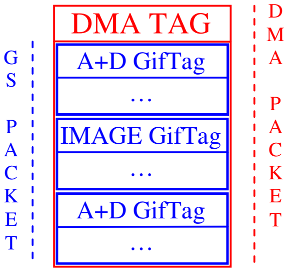
---

[4] Remember that when dealing with DMA transfer Physical addresses are used, memory allocated by the sps2Allocate function allocate a single continuous portion of memory in local address space not in physical one.



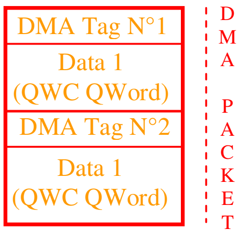
registers cannot be addressed directly in regs field, so *A+D data* must be specified. A+D data means that the NLOOP qwords following GIFtag represent values to be stored into GS registers. Each of these 128 bit value contains both the address and the data to store into GS register, address of the register is its hexadecimal value found in GS manual. Since four register are to be accessed, four qword are needed. Structure of the beginning of packet is represented in the picture.

After these registers are set, another GS Primitive, containing texture data, must be sent to GS with *IMAGE* format as discussed before. Structure of IMAGE packet is very simple, NLOOP QuadWord after GIFtag are transferred into HWREG register and stored into memory according tothe four image transfer registers. Finally, after all the texel are transferred to GS memory, it is necessary to set TEXFLUSH register to end image transfer, this is accomplished with another GS Primitive (standard PACKED mode) with a single A+D register.



To transfer whole texture to GS we need three GS Primitive, that could be contained in a single DMA pack, as represented in the picture. Unfortunately pixel data often take more than 4KB of space making the whole DMA packet larger than a single page so it is impossible to transfer all data in one shot, because of lack of physical continuity in DMA pack. This happens because sps2Allocate() function could not allocate physically contiguous RAM, as states into the documentation.

The obvious solution to this problem is using multiple DMA packet, now it's the time to use Chain transfer mode. To understand overall structure of the packet remember that it is possible to split a GS Primitive in more than one DMA packet if this two packet are to be sent contiguously.



First type of Chain tag used is *cnt* type, it makes DMAC transfer QWC qword of data following the tag, and read next qword as the next tag of Chain transfer. This is useful but is not enough to transfer a whole texture, because it is impossible to specify the address of a different page. Structure of DMA packet composed only by *cnt* tag is showed in figure at the left, it is used to contain first GS Primitive (four A+D registers) and the GIFtag of the Gif packet containing texture pixels. This means that IMAGE GIF Packet is splitted in more than one DMA pack, but as seen before this is not a big problem.

Reading texture data in different memory area

Since texture is a long file, it is convenient to read pixel data in a different memory location respect DMA packet, as we will see later this memory is allocated with a separate call of sps2Allocate inside the constructor of TextureManager class. But this will be explained later in this chapter.

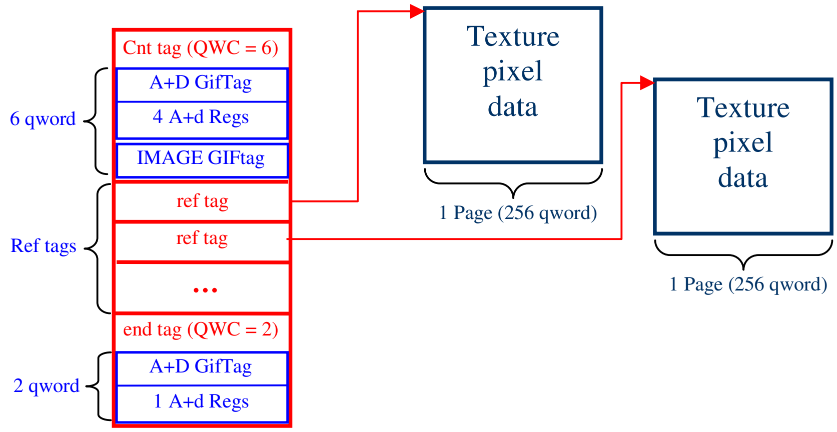
If texture is stored in completely different pages from DMA packet, most obvious solution is to use *ref* tag to transfer pixel data. This tag instruct DMAC to transfer QWC qwords of data at the address contained in its ADDR field and take next qword following current tag as next tag. With this tag, we can use a single ref tag for every 4KB page used to store texture data, all these ref tag are positioned after the first cnt tag.

Complete **structure** of DMA packet

Finally *end* tag tells DMAC to transfer QWC qwords of data following the tag and stop the transfer, this last tag is useful to write into TEXFLUSH register and terminate texture upload. Structure of whole Chain used to transfer texture data is represented in following picture.

Physical contiguity solved

From this picture is clear that texture data is allocated into as many pages as needed, and then each of these page is transferred with a ref tag during chain mode. Since 10 qwords of DMA packet are used by cnt and end tag, there is rooms for 246 ref tag, and since each of them can address a whole page (4KB) it is possible to transfer a texture up to 1MB fitting all DMA tags into a single page of ram, this means that there is no more risk of having DMA packet going across page boundary[5].



CLUT data can be send to GS with the same schema used for texture with the only difference that whole CLUT fits into a single page of ram and so a single ref tag is needed, moreover texture transfer can be flushed after both texture data and CLUT data is transferred to GS.

3.4 TextureManager Class

How to handle a texture

To help handling texture a simple C++ class was build to manage loading a texture from disk, uploading texels to GS and building register for

---

[5] 1 MB texture is impossible to fit into GS memory together with frame buffers and depth buffer, it is also not useful having memory wasted with such a big texture.

texturing to be set before a primitive is rendered. This class is contained in example project *03-TextureManager*[6], and will be hopefully expanded in the future to handle more feature as CLUT, Alpha, swizzling and so on. This class is made for the purpose of showing how to deal with texture, code is not optimized, and memory is not handled efficiently because each instance of TextureManager class load and free texture memory. A better approach is allocate enough space to contain bigger texture for the whole program and use that area to handle one texture at time.

Build own texture **file format**

Instead of making TextureManager class to access some known file format as targa or bitmap, a better solution is to build own image format to make texture manager class simpler and faster. A simple utility in Visual Basic is build to convert an image (jpg, tiff, bmp) into this proprietary file format, this program is called TextureConverter and can be used also to convert pixel format (in this first release only RGBA and RGB are supported) and change size of the image.

Format of the file is very simple, and consist of a simple header containing image information (format, size..) followed by RGB data of texels, starting from upper left to bottom right. This make TextureManager class to load the texture quickly because no decompression or other operation has to be done. The only drawback of using this format is that texture files tends to be big file, but if really space is a problem a simple zip algorithm can be introduced to shrink the size of the image. Hawing own file format makes possible to handle the texture off-time, reducing run time cost.

TextureManager class in details

TextureManager constructor simply access file containing the texture and load all texel data into a region of memory allocated with sps2MemoryAlloc(). Once TextureManager object is successfully created a call to *SendTexture()* public method upload texture data to GS. This methods accept two arguments, the first is the address of GS where to upload the texture, the second is a sps2Memory_t object that represent the memory in witch to build DMA packet to send the texture. GS address must be a beginning of a GS page, its value can be obtained by the function *sps2UScreenGetFirstFreeGSPage()* if only one texture at a time is uploaded, remember that this address must be expressed in unit of 64 word.

TetxtureManger object assume that texture is to be uploaded at the beginning of the address specified by GSAddress parameters, this makes the upload function straightforward. Structure of the packet to send data is already discussed in preceding paragraph.

---

[6] Thanks to guardian for giving me permission to use his mammoth symbol as texture file.

3.5 How to use uploaded texture in main program

Set **currently used** texture

After all textures are uploaded, to choose witch of them the GS must use, register TEX0 must be set. This register specify the address of current texture in GS memory, in this way the user can choose any of the uploaded textures to use with next primitives that will be rendered. Since it is possible to upload more than one texture at a time, this register must be set by main program.

TextureManager **build** TEX0 register

Since *TEX0* register is used to specify basic information about the texture such as height, width, pixel format etc, it needs to know texture information stored by TextureManager object used to upload the texture. This can suggest that function to build TEX0 register is to be included in texture manager. But it is worth to know that in TEX0 TextureFunction (DECAL, MODULATE, etc) is set and this value is logically managed outside TextureManager class. For this reason TextureManager class has a member function called *BuildTEX0()* that return a valid value for TEX0 to send to GS. The only parameter accepted by this function is the TextureFunction to use. In this way main program can correctly set whichever texture is desidered.

Different **Texture functions**

GS support four type of texture functions, details on them are given in GS manual; the only peculiarity it is worth to spent time on, is the way GS handle MODULATE function. When texture colour is modulated with fragment colour (colour stored into vertices), texture colour is represented same as original image when fragment colour is 0x80. This means that if texturing is used, vertex colour should range from 0x00 to 0x80 in all the three component. Using greater valued makes pixel of the texture to appear brighter. Pressing TRIANGLE button on the pad switch between MODULATE and DECAL texture function so you can appreciate the difference.

TEX0 register

To completely set texture environment another two register must be set, first of them is called *TEX1* and it's used to store information on mipmapping, not used in this tutorial, as well as texture filtering functions. If mipmapping is not used then only NEAREST and LINEAR value are accepted for filtering functions. Usually only LINEAR value is used because it give best result, remember also that filtering can be separately selected for minification and for magnification, and magnification filter supports only LINEAR and NEAREST functions, even if mipmapping is used, this is obvious because there is no other texture level to interpolate with magnification. In the example project *03-TexturedCube* press SQUARE button to switch between these two mode, but remember to move the camera (L1, L2) closer to the cube to well appreciate

Setting **CLAMP** register

the difference.

Last register related to texture environment setting is CLAMP, used to set clamping mode. If value outside interval [0.0, 1.0] is used as ST coordinates on a vertex, GS has four method to determine colour of the texels that lies outside this range. Default mode is REPEAT, that is used to repeat texture as if it is tiled onto the triangle, another useful mode is CLAMP, that repeat the colour at the border, this means that texture address values are clamped to range [0.0, 1.0]. To see different clamping in action, modify texture coordinate pressing R1 and R2 and then press CROSS button to alternate between CLAMP and REPEAT wrapping mode.

Build DMA packet to **send cube** information to GS

DMA packet used by main program to draw the cube is now changed into source chain mode, first part of the packet contains register texture settings (TEX0, TEX1 and CLAMP), while last part contains vertices information. It is interesting looking on how texture coordinate value are calculated:

```
if (GS_PRIM_FST_STQ == TexCoordinateMode) {

      //Calculate S/W, T/W, 1/W, and store value into DMAPACK
      STQ[0] = TextureCoordinates[VertexIndex][0] * RecW;
      STQ[1] = TextureCoordinates[VertexIndex][1] * RecW;
      STQ[2] = RecW;
      DMAPacket++->ul128 = *(sps2uint128 *) STQ;
}
else {

      //Calculate texture in UV format,
      UV[0] =(int)(TextureCoordinates[VertexIndex][0]
                          * TM.Width()) << 4;
      UV[1] = (int)(TextureCoordinates[VertexIndex][1]
                          * TM.Height()) << 4;
      DMAPacket++->ul128 = *(sps2uint128 *) UV;
}
```

Building **texture address** vectors

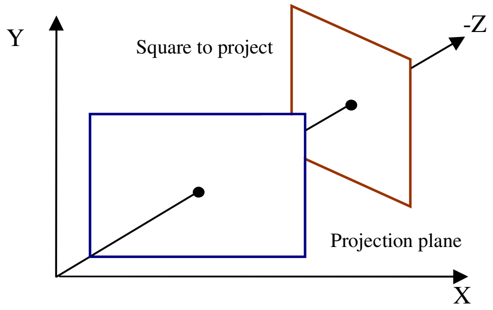
When STQ coordinate system is used, S and T value are divided by W (RecW contains in fact value 1/W), when UV coordinates are used we must remember that UV coordinates does not range from 0.0 to 1.0 as ST values, but are expressed in units of texels. Since Texture address for the cube is expressed in ST coordinate system, value must be converted multiplying original S and T value for texture dimension. Finally, format of UV coordinates is expressed in fixed point format, with 12 bit of integer part and 4 bit of fractional part. To convert between plain integer and this format a simple shift can be done, dropping fractional part.

3.6 STQ texture coordinates and perspective correction

Why **perspective correction** is needed

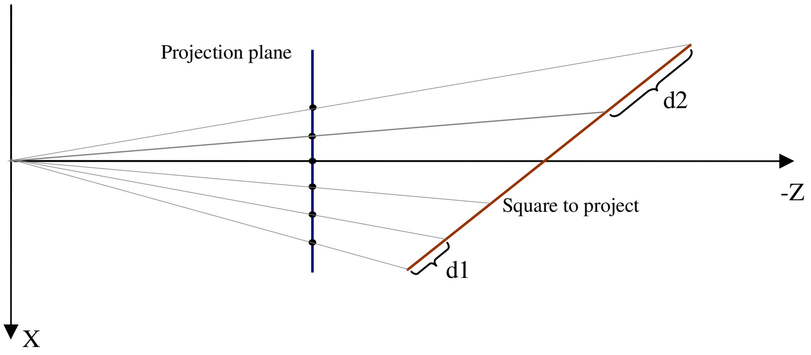
Texturing process takes place after a triangle is projected on the screen, to find the texel that correspond to each pixel of rasterized triangle, a linear interpolating process could be used on texture coordinates associated with the vertices. This usually lead to image artefact because of perspective distortion.

The problem arise because we are doing a linear interpolation of coordinates of projected triangle, since projection is not a linear transformation in $\Re^3$, this linear interpolation on projected coordinates lead to non linear interpolation in original coordinates of the triangle. Suppose for example that a square is positioned in world space in the following location.



Now consider what happens if we do linear interpolation on projected vertices. Picture now represent the same scene, but viewed from the positive side of Y axes.

How to **correct** perspective distortion on PS2

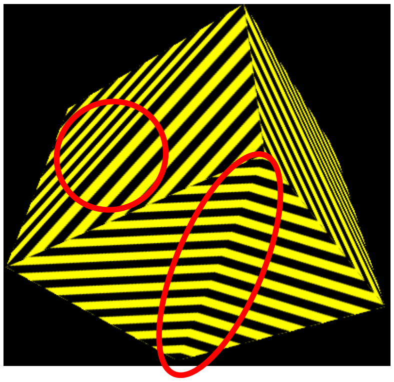


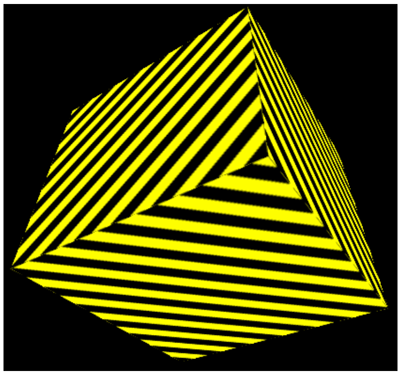
It can be easily seen that linear interpolating projected square, produces non uniform interpolation on original object leading to artefact in applied texture. Since this problem was early addressed by Jim Blinn[xx], modern hardware support a *perspective correction algorithm* in their internal rasterizer to help minimize this problem. A simple approach, as found by Blinn and called *hyperbolic interpolation*, does not interpolate (u,v) texture coordinate but coordinate $\left({}^{U}\!/_{W}, {}^{V}\!/_{W}, {}^{1}\!/_{W}\right)$, where W represent homogeneous coordinate of transformed vertex. Playstation2 hardware support this correction with the using of STQ addressing mode instead of classic UV, where texture perspective correction is not used. For each vertex the value 1/W must be calculated and used to multiply original (S, T) texture coordinates, then this value must be stored in Q part of texture coordinate, GS will do the rest of the process. Looking at the code in *main.cpp* it can be easily seen this process during the building of cube data to be sent to the GS.

When data is uploaded to GS, it is necessary that texture coordinate STQ are sent before the RGBAQ, this because Q component is stored in RGBAQ register and not together with S, T coordinate value.

Perspective correction at work

To appreciate the usefulness of texture perspective correction press L3 when the *03-TexturedCube* demo is running, it changes texture coordinate system from STQ to UV and vice versa effectively enabling/disabling texture perspective correction. If a pattern of uniform line is used as texture, we can easily see that when perspective correction is not performed, lines appears not uniform on triangle border.





Picture on left shows that if normal UV coordinate are used, some errors happens into rendered image. First artefact consist in discontinuities on lines on bottom face of the cube, second artefact happens on upper faces, where lines appear to have different width and different spacing. Picture on the right shows same geometry with texture correction enabled (STQ coordinates on PS2), this time the image is rendered without any defects.

For those who are interested in texture perspective correction I suggest reading the exceptional article of Chris Hecker freely downloadable at http://www.d6.com/users/checker/misctech.htm.

3.7 Grab the screen on PS2

How to grab screen content, thanks to Sparky

Preceding paragraph shows some screenshot of the running application, the question now is: "How it is possible to take screenshot of a running 3D program?". The answer is in GS manual and in exceptional code made by Sparky (http://playstation2-linux.com/projects/sps2demo/). I adapted the code to work in my application, the hard work is all made by Sparky and I only done little modification to make the routine simpler to use. Code itself it is well commented so lets have a little briefing on how this routine works.

Local to host transfer mode

First of all GS manual section 4.2 deal with transfer data to and from GS, this is all is needed to know to do a screenshot. First of all it is necessary to know how to handle signals such as FINISH, this is done reading *ps2dev_ioctls_en.txt* documentation file. Transfer from host to GS memory was already be explained in preceding paragraphs regarding how to send a texture to GS; grabbing screen content is very similar because it is simply a transfer from GS memory to Local memory, but code that does that is more complicated respect texture uploading code, because some problems must be solved. First problem consist in the impossibility to use DMA channel 2 (transfer to GS), because it works only *to GS*, to make data flow from GS to memory, channel 1 is to be used, so data pass through VIF1 that is directly connected to.

A class to take screenshot of running program

All the hard code is contained in a class called ScreenShot, in function *DownloadVram()* originally made by Sparky. Original version use a lot of helpful class included in sps2Demo, the version inside the class is adapted to run alone, so it is very easy to include in own project. Just add Screenshot.h and Screenshot.cpp and the work is done. To take a shot, create a ScreenShot object passing sps2descriptor to the constructor and then take a shot after screen swapping with member function *TakeShot()*. Remember to pass valid filename (file generated is a targa uncompressed so it's best if the file ends with "tga") and optionally the type of frame buffer, PAL or NTSC, remember that the function is defaulted to NTSC. Here it's a typical use, extracted from example 03-TexturedCube:

```
sps2UScreenSwap();
if (Joy.Button(R3).State == Pressed) {

       static int   ShotIndex = 1;
       char                buffer[255];
       sprintf(buffer, "./Shot%3d.tga", ShotIndex++);
       SH.TakeShot(buffer, NTSC);
}
```

How to **download a line** from frame buffer

It is clear that using this class is very simple. Now take a look deeper into the operations needed to grab data from frame buffer. The code download a line at a time, so all data fits into a single page of ram, routine TakeShot() then reads all the lines of the frame buffer and build a simple uncompressed targa image that is very easy to build. Consulting paragraph 4.2 of GS manual it can be seen that transferring data from Local Buffer to Host consist of 7 step, step 1 through 4 talk about accessing the FINISH register of GS[7] and wait for

---

[7] "access a register" actually means write any value into it, usually value zero is used.

FINISH signal (FINISH bit of CSR register become 1), then set transmission parameters. To access FINISH a simple A+D addressing can be used but remember that to access FINISH signal, ioctl() must be used. First of all a VIF header must be build to instruct the VIF to send data to GS:

```
pDmaMem[0] = 0; //NOP;
pDmaMem[1] = 0x06008000; //MSKPATH3(0x8000); Disable path 3
pDmaMem[2] = 0x13000000; //FLUSHA;
pDmaMem[3] = 0x50000006; //DIRECT(6);
```

Direct(6) means that 6 qword of data are to be transferred to GS, these qwords are: 1 GIFtag specifying 5 A+D registers, and the four transmission registers as seen before plus FINISH register. Before actually send the packet to the VIF, we must first enable reading of FINISH event:

```
ioctl(eventfd, PS2IOC_ENABLEEVENT, PS2EV_FINISH);
ioctl(eventfd, PS2IOC_GETEVENT, PS2EV_FINISH);
… //Packet is send
ioctl(eventfd, PS2IOC_WAITEVENT, PS2EV_FINISH);
```

First line actually enable intercepting the FINISH event while the second clear the FINISH bit of CSR. After the packet is sent to VIF wait for the FINISH event.

Point 5 tells to set 1 into the privileged register BUSDIR, this reverse the direction of data from local memory to host, then read data and finally set again 0 in BUSDIR to restore direction setting. This is enough, but remember that, at the end of DownloadVram routine, it is necessary to enable again path 3 that was disabled at the beginning of the routine and finally enabling again PS2EV_VBSTART event that gets disabled when ioctl() is used to enable FINISH event.

Now screen content can be grabbed and used as screenshot.

4 Vector Units

4.1 VU0 in Macro mode

What are those **Vector units**?

Real power of EE processor lies on vector units VU0 and VU1, two powerful math processors especially designed to operate on typical data of computer graphics at increased speed in respect on using a general purpose processor. These processors are able to execute SIMD (Single Instruction on Multiple Data) instruction working on *packed vector data*, consisting 4 float values packed together in a single 128 bit data. Internal structure of EE is discussed with great detail in Playstation 2 manuals, so in this chapter it's assumed that the reader read at least EE and VU manuals and knows internal structure of PS2 console.

Using VU0 as a coprocessor **COP2**

To familiarize with the instruction set of Vector Units it is possible to use VU0 in macro mode. VU0 is tightly coupled with EE core and can be *used as COP2 math coprocessor*. When operating as coprocessor, the user can access its internal registers and instructions directly with inline assembly embedded into C++ code. Since most important instructions can be used in macro mode it is worth an overview to understand basic concept of VU before moving on *Micro Mode* that shows the real power of Vector Units.

Gcc **assembly syntax**

Before actively examine VU0 macro mode code, a little overview of GCC inline asm syntax is needed, especially for those used to work in Windows environment. Since inline asm must often work with variable declared in C/C++ code, GCC has a special syntax that makes possible to access a variable in inline asm code independently of machine or environment used. It is well known that layout of the stack is not the same in Windows and Linux system, and it can differ even with compiler options, such as optimization. This makes accessing C/C++ declared variable not portable and difficult, gcc solve this problem with the use of particular syntax structure that specify to the compiler the variable that are used inside the asm block. As an example this is a simple inline asm block consisting of a single instruction:

```
int x = 1;
int y = 2;
int result = 0;
asm ("
  mul %0,%1,%2
":"=r"(result):"r"(x),"r"(y));
```

This fragment of code simply declare 3 C++ variables and use them into a block of assembly language composed by a single instruction, a multiplication. The asm block is enclosed into "…", this means that asm block finish at the beginning of the last line, then a colon is present to state that variables list begins. After the first colon the expression **"=r" (result)** specify to the compiler that the variable result is used as output variable, its name into asm block is **%0** since it's the first variable declared in the list. Gcc manuals specify in fact that after the first colon list of output variable has to be placed. Character **r** specify to the compiler that variable is an integer value so integer store has to be used to store data on it. After output variables list another colon delimits the beginning of input variables list. Each of these two blocks can contain more than one entry separated by comma character. Input variable list is similar to the output variable list, just remove '=' character from the declaration.

Variable name in asm block is composed by a '%' character followed by the index with the variable appears in the final output/input variable list. Both output and input variables are considered to form a single list so variable **x** hae name **%1** in asm block. The whole block takes the name of *assembler instruction template*, for more details consult online GCC documentation freely consultable at http://gcc.gnu.org.

4.2 Some vectors and matrices operations in VU0 macro mode

Some **math** with **COP2** (VU0)

Most obvious operations that can be executed with V0 are packed math operations, as example this is the code to calculate sum of two vectors:

```
float V1[4] QWALIGN = {1.0f, 0.0f, 1.0f, 1.0f};
float V2[4] QWALIGN = {2.0f, 2.0f, 0.0f, 1.0f};
float V3[4] QWALIGN = {0};

//Sum two vectors
asm __volatile__("

        lqc2 vf1, 0(%0)
        lqc2 vf2, 0(%1)
        vadd.xyzw vf3, vf1, vf2
        sqc2 vf3, 0(%2)

": : "r" (V1), "r" (V2), "r" (V3));
```

First of all three vectors are declared as simple array of 4 float element with simple C++ syntax, remember that to load these value into V0 registers with *lqc2* instruction data are to be 16 byte aligned in memory. A simple macro called *QWALIGN* is used to apply *__attribute__((aligned(16)))* attribute to the three arrays. To use C++ variable in asm block the assembler instruction template list states that V1, V2, V3 are three input variable labelled respectively %0, %1 and %3 in asm block. Even if V3 is used as output variable it is declared as input ones, moreover all three variables are declared as integer number ("r" specification). This is done because into the asm block only the addresses of the three vectors are used not their contents and the address is a simple integer number. In fact %2 is never used as destination operand so it is effectively an input value. Asm code is straightforward, first of all content of V1 and V2 are loaded into VU0 registers *vf1* and *vf2* respectively, then instruction *vass.xyzw* is used to compute the sum and the result is stored back with sqc2 into main memory. With this simple block the sum of two homogeneous vectors[8] can be performed with a single instruction (not counting loading and storing ones).

Vector **Cross Product** with COP2

Instruction set supported by Vector Units is expecially designed to handle homogeneous (4 component) vectors and affine transform expressed by 4x4 matrix, in this paragraph we will see how to implement with VU0 basic math functions operating with vectors and matrices.

Before looking on how to implement dot product between two vectors, lets look at the code used to perform cross product:

```
asm __volatile__("

        lqc2 vf1, 0(%0)
        lqc2 vf2, 0(%1)
        vopmula.xyz ACC, vf1, vf2
        vopmsub.xyz vf3, vf2, vf1
        sqc2 vf3, 0(%2)

": : "r" (V1), "r" (V2), "r" (V3));
```

only two instructions are needed since *vompula* and *vopmsub* instructions are especially designed to handle vector cross product. Remember to write down ACC in uppercase because gcc does not compile the code if this requirement is not met.

Matrix product with COP2

Now lets move to matrices multiplication, used to compose two or more transformations into one. Since this operation is very common in 3D computer graphics programs, VU have a special set of instructions that make possible to perform this operation with only 12 instructions. First of all lets examine how the two matrix are declared:

```
float mat1[4][4] = QWALIGN {{1, 2, 3, 4},
                             {5, 6, 7, 8},
                             {9, 10, 11, 12},
                             {13, 14, 15, 16}};
float mat2[4][4] = QWALIGN{{0.1, 0.2, 0.3, 0.4},
                            {0.5, 0.6, 0.7, 0.8},
                            {0.9, 1.0, 1.1, 1.2},
                            {1.3, 1.4, 1.5, 1.6}};
```

A matrix in memory is nothing more than 16 float elements stored

---

[8] This is true if the w component of both vectors have the value one. Ex: V=(2, 0, 0, 2), W = (0, 2, 0, 2) and V+W = (2,2,0,2) that is different from (2,2,0,4).

together as a linear array, one element after the other. Now lets review how matrix multiplication is done.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix} =$$

$$\boxed{a_{11}b_{11}} + \boxed{a_{12}b_{21}} + \boxed{a_{13}b_{31}} + \boxed{a_{14}b_{41}}, \boxed{a_{11}b_{12}} + \boxed{a_{12}b_{22}} + \boxed{a_{13}b_{32}} + \boxed{a_{14}b_{42}},$$
$$\boxed{a_{11}b_{13}} + \boxed{a_{12}b_{23}} + \boxed{a_{13}b_{33}} + \boxed{a_{14}b_{43}}, \boxed{a_{11}b_{14}} + \boxed{a_{12}b_{24}} + \boxed{a_{13}b_{34}} + \boxed{a_{14}b_{44}}$$

In above expression only first row of the resulting matrix is represented, because all the other are calculated in the same way. Since multiplication is made row by column it is difficult multiplying together first row of left matrix with first column of the second matrix because it is impossible to store first colum of second matrix into a V0 register with a **single** load instruction. Moreover, even if data in memory will be reorganized to make this possible, at the end of this multiplication we will end with the four elements marked with blue stored into the components of one register and then we have to sum them together to find first element of destination matrix. This is call horizontal addiction (adding together four values contained in a register) and it is not performed with a single instruction another way must be found.

How to use broadcast instruction

Looking at the element marked with red we can notice that they are a vector obtained multiplying each element of the first row of mat2 with the element a_{11} of mat1. Looking at the element in green we see that this vector is obtained multiplying each element of the second row of mat2 for the element a_{12} of mat1, analog things happens for vectors marked with yellow and orange. First row of result matrix it is obtained by a simple sum of these four vectors. To accomplish this task VU's are equipped with *mulbc (broadcast product)* and *maddbc (broadcast product sum)* instructions. These instruction permits to multiply all component of a register for a single component of another register, instead of multiplying each component with the corresponding one as for a normal mul. Since a chain of multiplication and addition is to be done, the accumulator register is used to avoid stalling the pipeline.

vmulax ACC, rs, rt.x

	127 96	95 64	63 32	31 0
rs	RS3	RS2	RS1	RS0
	127 96	95 4	63 32	31 0
rt				**RT0**
	127 96	95 64	63 32	31 0
ACC	RS3*RT0	RS2*RT0	RS1*RT0	RS0*RT0

With vmulax we can store in acc register all the elements marked in red into accumulator register with a single instruction, supposing mat1 is stored into vf1-vf4 registers and mat2 is stored into vf5-vf8 registers.:

```
vmulax      ACC,   vf5,   vf1
```

now it is sufficient to use multiply and add instruction to calculate green elements and adding at the same time with the content of the accumulator register, then the same for yellow and orange elements. Last instruction store the result in vf9 register.

```
vmadday     ACC,   vf6,   vf1
vmaddaz     ACC,   vf7,   vf1
vmaddw      vf9,   vf8,   vf1
```

It is very important to notice that the result of the first instruction is used by the second instruction, this would normally cause a stall of the pipeline if the result would be stored in normal register. This would happens because a mul operation has a throughput/latency of 1/4. The use of accumulator register avoid this problem because accumulator registers are often implemented into the core of the ALU itself as temporary register, when an instruction has Acc as destination register, the result of the operation is stored into the Acc register immediately after the result is ready, so next instruction can use it. The others rows of matrix result are calculated in the same way.

4.3 Horizontal add and transforming a vector.

Needs for horizontal adding register components

To calculate dot products between two vectors we have to use a normal vmul operation but then we have to perform an horizontal addiction that is calculate the summation of the first three component of a vector[9]. This operation is not present in VU's instruction set, but can be easily done with broadcast add, even if this is not so efficient:

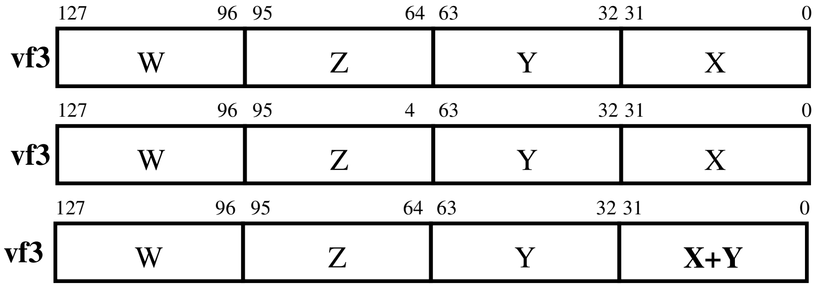
```
vmul.xyzw   vf3, vf1, vf2
vaddy.x     vf3, vf3, vf3
vaddz.x     vf3, vf3, vf3
```

first instruction do normal multiplication between vector components then the second instruction means: *multiply component of vf3 by the y component of vf3 with a broadcast, but execute this operation only for x component of the vectors.*

---

[9] This is true only if w component of both vectors have the value 1.

vaddy.x vf3, vf3, vf3



Horizontal add with broadcas instructions

The y in the instruction indicate the register component to broadcast, then dot separate instruction from the mask component list, this means that only x component will be update. With this instruction VU0 execute the operation only for the x part of the registers, ignoring remaining 3. The second vaddz.x instruction store the result of dot product in the x part of vf3. If a complete horizontal addition is to be done, *vaddw.x* is last instruction to be used. From this example we see that to execute an horizontal add three add instructions must be used, this means that there is no speed up using VU's respect to normal floating point unit. Consider also that the above sequence of instructions cause a stall into the pipeline of VU0 because we must wait for the first vmul instruction to be completed (4 clock ticks) before using *vaddy* instruction.

How to **transform** a vertex

Now it is time to examine one of the most important operation executed on vectors, *vertices transformation*. To transform a vertices a multiplication between a matrix and a vector is to be done, lets review how it is done:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \\ v_w \end{bmatrix} = \begin{bmatrix} a_{11}v_x + a_{12}v_y + a_{13}v_z + a_{14}v_w \\ a_{21}v_x + a_{22}v_y + a_{23}v_z + a_{24}v_w \\ a_{31}v_x + a_{32}v_y + a_{33}v_z + a_{34}v_w \\ a_{41}v_x + a_{42}v_y + a_{43}v_z + a_{44}v_w \end{bmatrix}$$

First of all we must consider that even if vector is a *column* vector, its layout in memory is linear making it possible to multiply together first row of the matrix with vertex, then an horizontal add with storing the result in x component will complete the first component of transformed vertex. This sequence of operation has to be repeated for the other three components of the vertex. This is not so efficient because we had seen that horizontal add is not efficiently handled by VU0, remember also that this operation is critical because it has to be done for all the vertices that make a scene. It is necessary to find an alternative way to do this operation.

If we use row vector notation as DirectX do, transforming a vertex has now this form:

$$\begin{bmatrix} v_x & v_y & v_z & v_w \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

Switching between **column** and **row** vectors

It is clear that this operation is equal to the first part of matrix multiplication, leading to only four VU's operations and great execution speed. This means that it is better to use row vector notation but for those who prefer to use column vectors, that are more mathematically correct, it is worth to notice that to obtain transformation matrix for row vector notation from the transformation matrix used in column notation a simple transpose operation is needed. This makes possible to use column notation throughout whole program and transpose transformation matrix before sending it to VUs.

To understand the difficulty of transforming a vector using column notation directly into vector units it is possible to examine code contained in *main.cpp* of *04-MacroMode* example. Two possible implementation are presented there, both of them not very efficient. Both begins with the multiplication of the vector with the four rows of the matrix in register \$8-\$11, then an horizontal add is to be done for these four vectors. Horizontal add of \$8 represent x component of transformed vertex, horizontal add of \$9 is the y component and so on.

MMI solution, swizzle and add

One solution consist in moving these vectors to main memory, use swizzling algorithm with MMI[10] to transpose the component and finally a simple vadd is enough to find result. This method is not efficient because data are to be passed from VU0 to EE core and then to VU0 again.

Vector units solution, broadcast to add

A different approach is used in the second solution that is completely done in VU0 since we have seen before that broadcast add can do horizontal add. It is important now to store each horizontal add in the corresponding part of the register.

```
vaddy.x        vf8,   vf8,   vf8   #Begin HorAdd of vf8 in x
vaddx.y        vf9,   vf9,   vf9   #Begin HorAdd of vf9 in y
vaddx.z        vf10,  vf10,  vf10  #Begin HorAdd of vf10 in z
vaddx.w        vf11,  vf11,  vf11  #Begin HorAdd of vf11 in w
vmr32          vf1,   vf0         #store (0, 0, 1.0, 0) in vf1

vaddz.x        vf8,   vf8,   vf8
vaddz.y        vf9,   vf9,   vf9
vaddy.z        vf10,  vf10,  vf10
vaddy.w        vf11,  vf11,  vf11
vmr32          vf2,   vf1         #store (0, 1.0, 0, 0) in vf2

vaddw.x        vf8,   vf8,   vf8
vaddw.y        vf9,   vf9,   vf9
vaddw.z        vf10,  vf10,  vf10
```

---

[10] This algorithm will be discussed in detail in the next paragraph

```
    vaddz.w         vf11,   vf11,   vf11
    vmr32           vf3,    vf2          #store (1.0, 0, 0, 0) in vf3
```

Since we have to do 4 horizontal adds, we can interleave them to avoid stall, we see in fact that register vf8 is accessed another time after 5 instructions avoiding stall. Various *vmr32* instructions are used to calculate constant (0, 0, 1.0, 0), (0, 1.0, 0, 0) and (0, 0, 1.0, 0) from the value in constant register vf0 (0, 0, 0, 1.0). These value are needed because at the end of horizontal adds, in register vf8 the value of horizontal add is stored in x component but the other components contains old values that are not necessary anymore. A possible solution to this problem is multiplying vf8 by vector (1.0, 0, 0, 0) to maintain only the x component. Same thing happens to the other three registers. Finally to compute transformed vertex it is sufficient add together vf8-vf11 while multiplying for the various constant vector to eliminate unwanted component.

Masking and adding.

```
    vmula.xyzw  ACC,    vf11,   vf0
    vmadda.xyzw ACC,    vf10,   vf1
    vmadda.xyzw ACC,    vf9,    vf2
    vmadd.xyzw  vf12,   vf8,    vf3
```

Even if this algorithm can be interesting and can help to understand Vector Units it is clear that using row vector to use only four instruction is highly preferable.

4.4 Multimedia registers to transpose a matrix

Fast **transposition** of a matrix

As we see in previous paragraph it is more convenient to use row vector notation perform vertex transformation with only *four instruction in Vector Units*. Since this tutorial does not concern speeding up code to the limit, to maintain a correct mathematics notation column vector is used through the text. To avoid making excessive complicated VU code all transformation matrices are transposed before they are uploaded to Vector Units. To perform a fast transposing, *Multimedia Instructions* comes at hand because they permit to transpose a matrix with only 8 instructions. To understand how the code works lets before have a look at the various packing instructions.

The “**swizzling**” operation

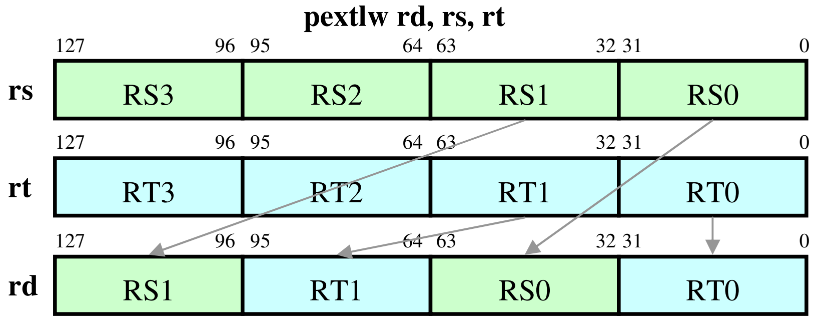
First of all we must load matrix elements into 4 128 bit registers with the instruction LQ (Load Quadword), in this way matrix rows are loaded in registers $8, $9, $10 and $11.

```
    lq     $8,    0(%0)
    lq     $9,    16(%0)
    lq     $10,   32(%0)
    lq     $11,   48(%0)
```

Now using packing instruction *pextlw*, *pextuw*, *pcpyld* and *pcpyud* we will perform an operation known as *swizzling*, consisting essentially in transposing a 4x4 matrix. This operation is used in more general situations to transform data layout from *AoS* (array of structure) to *SoA* (Structure of array). At this time we are not interested into real meaning of the swizzling operation and it is sufficient to know that is the same thing of transposing a matrix.

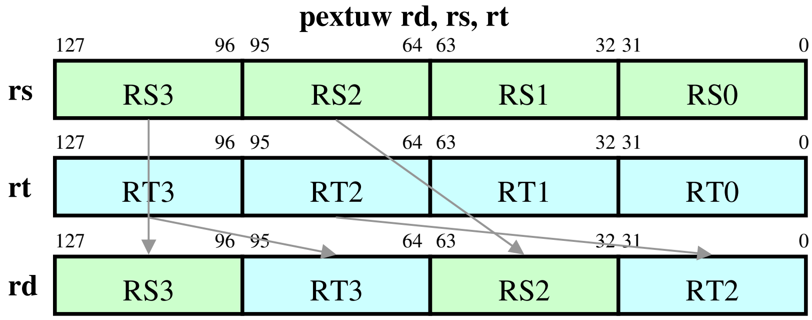
Packing instructions: **pextlw**

First of all examine the instruction ***pextlw*** (Parallel EXTend Lower from Word) that operate with two source 128 bit register mixing word values into another 128 bit destination register as represented in following picture.



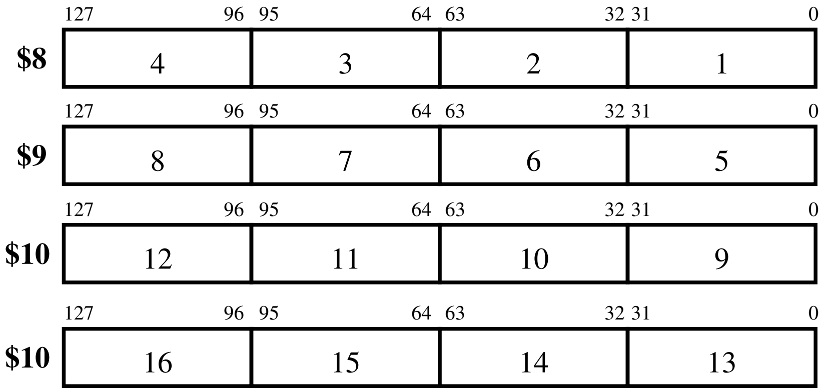
Packing instructions: **pextuw**

Instruction ***pextuw*** (Parallel EXTend Upper from Word) operate in similar fashion:



Even if these instruction seems strange and not so useful, they are especially implemented to do multimedia operation such as swizzling. To understand how these instruction are used we will now examine the swizzling operation with great detail.

Register after loading operations

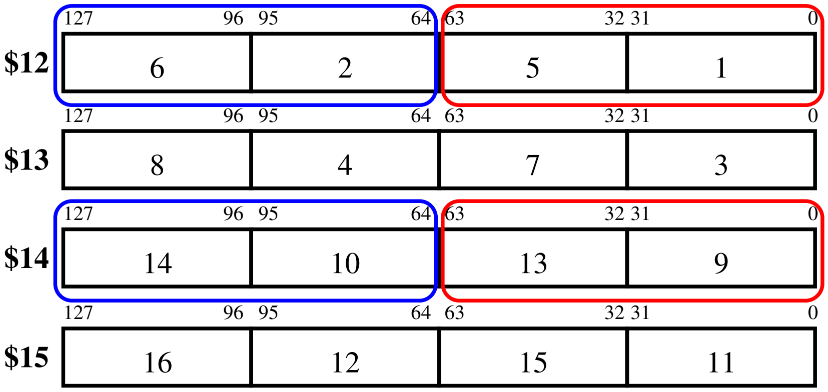


After the four load (lq) instructions the whole matrix is loaded in four registers. Exact content of the register after loading operation is represented in the picture in previous page. Using parallel extend instructions it is possible to begin the swizzling operation with these four instructions:

```
pextlw    $12,     $9,    $8
pextuw    $13,     $9,    $8
pextlw    $14,     $11,   $10
pextuw    $15,     $11,   $10
```

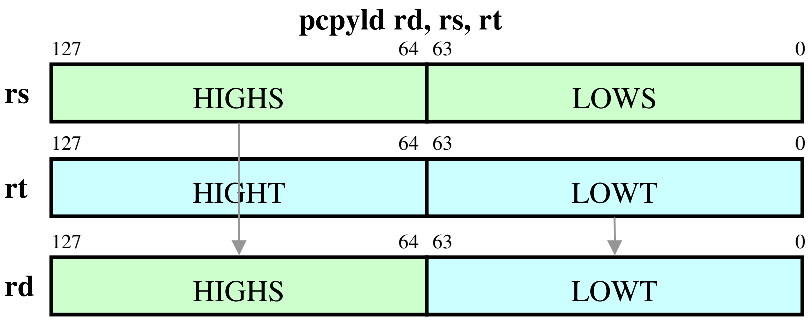
After this operation matrix data is transferred to registers $12-$15 and swizzle operation is begun. Content of registers is now the following.

Register after the first part of the swizzling



Packing instructions: **pcpyld**

Now, recalling that first row of transposed matrix is the vector (1, 5, 9, 13) we can notice that it is now decomposed into two separate parts highlighted in red. It is time to use *pcpyld* instruction to reconstruct first row of transposed matrix.



Instruction *pcpyld* operate in similar fashion and it used to build second row of transposed matrix, composed by the two parts highlighted in blue. It is now possible to obtain final matrix with the instructions:

```
pcpyld    $8,      $14,   $12
pcpyud    $9,      $12,   $14
pcpyld    $10,     $15,   $13
pcpyud    $11,     $13,   $15
```

now final content of $8-$11 registers is:

End of swizzling operations

	127 96	95 64	63 32	31 0
$8	13	9	5	1
$9	14	10	6	2
$10	15	11	7	3
$10	16	12	8	4

This is exactly what we wanted, our original matrix transposed.

4.5 Micro Mode

using Vector Units as **separate processor**

Real power of vector units shows up in Micro Mode, moreover, VU1 can only work in micro mode and this is the only way to use it. Micro Mode consist in uploading code into VU micro memory, then upload data into VU memory and start the execution of the microprogram. In this way vector units are used as a separate processors, even if they are contained into EE. Micro mode makes EE core free to do other stuff while VU works on vector data, making the application runs faster. In the example *04-MicroMode* a simple VU1 program, that transforms the vertices of the cube, is uploaded into VU1 avoiding to transform each vertex with EE core.

Direct connection between VU1 and GS, **PATH1**

VU1's main difference from VU0 is the direct connection with the GS, called **PATH1.** The presence of PATH1 means that VU1 can directly communicate with GS, sending data using **XGKICK** instruction. Lets see why this direct connection is necessary

Input data of VU1 for this example are: *transformation matrix* and *vertices data* (coordinates, colors, texture STQ coordinates), these data are transformed by VU1 executing uploaded micro code and finally output data must be sent to the GS. It would be source of great inefficiency if output data must be moved again into main memory to do transfer with EE core as seen before. To avoid this the XGKICK instruction it is used to send a GS primitive to GS directly from VU1 memory. This means that main application has only to send vertices data to the VU1 and then start execution of microprogram, this is enough to make cube be rasterized on screen. Now it is time to look at how VU1 microcode is build.

ASM code for micro mode is a little strange at the beginning, this because for every instruction, two opcode are needed, one for the upper unit and the other for the lower unit. This happens because Vector Units are composed by *two distinct units* (upper and lower), each of them having own instruction set. With this architecture VU is able to perform two different instructions at the same time, achieving great performances. To give to the reader an idea on how micro code looks, it is showed, as a short example, the beginning of the microprogram included into the example *04-MicroMode*:

Upper and **lower** execution units

```
NOP[D]              IADDIU VI01, VI00, 0
;Load Transformation matrix
NOP                 LQI           VF01, (VI01++)
NOP                 LQI           VF02, (VI01++)
NOP                 LQI           VF03, (VI01++)
NOP                 LQI           VF04, (VI01++)
```

Only lower execution unit is used in this snippet, witch purpose is loading transformation matrix into the first four floating point registers of VU1. This code will be explained later in great detail, now it is time to focus on various aspect of micro mode, ranging from compiling code, to uploading binary code into VU1 micro memory.

Compiling **.vsm** files

To compile a program written in Micro Mode Assembly we can use the assembler *ee-dvp-as* that comes with the kit. A little problem is that this tool create an object files filled with extra stuff that must be removed to upload code to the VU1. This happens because an object file contain usually more information than plain binary assembled code, such as symbols, header information and so on. To strip out all unwanted information we must use *objcopy* tool, used to manage content of object files. This utility strips out everything that is not useful to us, leaving only *real binary code* for VU1. This operation can be automated into makefile do be done automatically:

```
# Rule for compiling .vsm -> bin
.vsm.bin:
        ee-dvp-as -o $*.vo_ $*.vsm
        objcopy -Obinary $*.vo_ $*.bin
        rm $*.vo_
```

First of all obj file is assembled by the ee-dvp-as assembler generating object file with extension *.vo_*, then *objcopy* analyze structure of the file and copy only part of the file regarding binary assembled code into a new file with extension *.bin*. The option to do this operation is *–Obinary* that means: “copy only binary part of the object file”. Now resulting .bin file contains only assembled binary code for VU1 and can be directly read and uploaded into VU1 by the main application. Another way to proceed is using *bin2as* tool to convert bin to an assembly file that can be assembled by standard linux assembler as, making possible to include object file directly into the application. I prefer working directly with bin files, avoiding to store binary Vector Units microcode directly into executable file.

4.6 VIF Packet

DMA transfer to and from Vector units

To communicate data from EE to Vector Units a DMA transfer is needed. Rules to make DMA packets for GS are still valid, but with a different organization, there is no more GS primitives but *VIF packets*, formed by a 32 bit *VIFcode* followed by some data if needed. VIFcode is basically different from GIFtag because it does not only contain information on data that has to be sent to the VU, but can contain also commands that are executed by the VU. VIFcode is 32 bits long and since it can contain meaningful information by itself (not followed by any other data) there is the possibility to insert two VIFcode into upper 64 bits of DMAtag in chain transfer mode and transfer them together with DMAtag itself, providing to instruct DMAC to not discard DMAtag during data transfer. Details on how to do this kind of transfer can be found in EE manual.

Uploading Micro Code **MPG VIFcode**

To use Vector Units in micro mode first thing to do is upload microcode to micro mem of the VU that is to be used, this is easily accomplished by a FIVPacket with *MPG VIFcode* at the beginning of the packet, followed by binary data representing assembled code. It is also possible to read the code directly into VU1 memory, this because micro mem of VU1 is memory mapped into the address space of the EE.

```
FILE *VU1Code;
VU1Code = fopen("BasicTransform.bin", "rb");
if (NULL == VU1Code) {

       cout << "Unable to open BasicTransform.bin...Exiting\n";
       exit(5);
}
int CodeSize = fread(VU1_MICRO_MEM, 1, 4096, VU1Code);
fclose(VU1Code);
```

This is simple but not the preferred way to upload code into VU1, this mainly because all transfer should be regulated by DMAC, moreover the VU1 must be in idle mode to permit direct access from the EE. It is surely most interesting sending the microprogram with a DMA packet, moreover the code for this example is very short and fits all into a page of ram avoiding the need to do stitching of the packet.

Transfer code with a **VIFpacket**

First quadword of DMA packet will contain DMATag as well as *MPG VIFcode* so at the beginning of the *SendMicrocodeToVU1()* routine, two pointer are declared:

```
PS2_QWORD *DMAHeader = (PS2_QWORD *) DMAPackAddress->pvStart;
PS2_QWORD *DMAPacket = DMAHeader + 1;
```

All assembled code is read from the file directly into the memory pointed by DMAPacket and size of code in bytes can be simply obtained by the fread function return value.

```
int CodeSize = fread(DMAPacket, 1, 4092, VU1Code);
```

Now data is filled into the packet, and size of the data is known, this makes possible to build DMAtag and MPG VIFcode.

```
DMAHeader->ul128 = PS2_DMA_SET_HEADER((CodeSize + 15) >> 4,
       0, DMA_ID_SOURCE_END, 0, 0, 0);
DMAHeader->ul32[2] = 0;
DMAHeader->ul32[3] = PS2_DMA_SET_VIFCODE(0, CodeSize >> 3,
       VIF_CMD_MPG);
```

Size of DMAPaket is the smaller number of QWord that contains all code, so a little calculation has to be done, remembering that CodeSize is expressed in byte and a shift of 4 bit makes a division by 16 to find size in QWords. Remember also that ee-dvp-as compiled code is multiple of 16 bytes, this because even if the instruction are a odd number, a pad of all zero is inserted, this makes us sure that code will fill an integer number of QWord.

Insert VIFcode directly into the DMAtag QWord

After DMAtag is build, word number 3 is zeroed because it must contain a NOP VIFcode to guarantee alignment[11], then fourth word will contain *MPG* VIFcode that tell VIF to upload following data into Micro Memory, starting at address zero. Size of the code must be given in unit of instructions, since every instruction is a DWord long, a simple division by 8 convert from byte units to DWord units. Now it is possible to start transfer in chain mode in the usual way as seen for preceding examples.

4.7 Sending data and executing the code.

Send Data to Vector Units

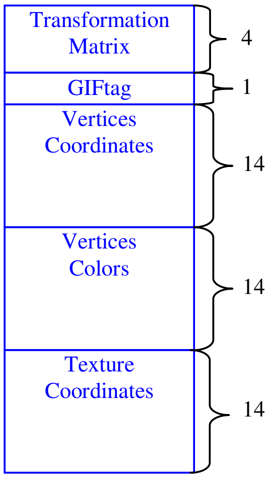
Before examining Micro Code of the example it is necessary to know how to send data to the VU1 and start the execution of loaded microprogram. In this example VU1 performs only vertex transformation and STQ texture coordinate calculation, remember in fact that original S and T value must be divided by value 1/W resulting from vertex transformation. To transform the vertices, VU1 needs to know transformation matrix and since we know that it is preferable to use row vector notation and main program use column vector notation, transformation matrix has to be send in transposed form. Then, since

---

[11] Check the EE manuals MPG VIFcode alignment requirements for further information.

VU1 has to kick vertices data to GS, it needs to prepare a GIFtag to build a GSPrimitive in VU1 memory. This is required because kicking a primitive actually means sending a GS primitive stored in VU1 memory. Since we know in advance the structure of vertices, we can calculate GIFtag in main application and send it to VU1 immediately after transformation matrix data. Then cube vertices data must follow, beginning with coordinates and prosecuting with colors and texture coordinates. Since we do not use double buffering for this first example, all these data are stored starting from the address 0 of VU1 internal memory. Layout of memory after uploading is showed in picture.

VU1 mem (16Kb) after data uploading



All these data must be stored into a single VIFPacket, and since all of them are QWords it is possible to do a single transfer with *UNPACK* VIFtag of *V4_32* structure. To make transfer simpler, DMAtag is transmitted during DMA transfer, this permits to put two VIFtag together with DMAtag in the first QWord of whole DMA packet, in this way all the other v4_32 data will follow.

```
int VertexNum       = sizeof(Cube) / sizeof(*Cube);
int VIFPacketSize = 4 + 1 + VertexNum * 3;

DMAPacket->ul128 = PS2_DMA_SET_HEADER(VIFPacketSize + 1, 0,
      DMA_ID_SOURCE_END, 0, 0, 0);
DMAPacket->ul128|= (sps2uint128) PS2_DMA_SET_VIFCODE_UNPACK(0,
      VIF_UNPACK_SIGNED, VIF_UNPACK_TO_ADDR,
      VIFPacketSize, V4_32) << 96;
```

Beginning a **V4_32** data transfer to Vector Units 1

First of all the application must know the size of VIFPacket, this value is equal to the size of the matrix (4 QWord) plus size of GIFtag (1 QWord) plus size of all data regarding vertex. This last value is equal to number of vertices times 3 (coordinates, colors and texture coordinates component). The size of DMA packet is 1 QWord longer, this is done because we have to transfer other two VIFpakets. Layout of the first QWord of DMAPacket after PS2_DMA_SET_HEADER is the following.

127 96	95 64	63 0
0x00000000	0x00000000	DMA CHAIN TAG (END)

Adding VIFcode to DMA header

After the DMA header is set, upper 64 bits of the DMA tag are cleared to value zero, this is important because value zero represents *NOT VIFtag*. Now another macro is used to build the UNPACK VIFcode, since double buffering is not used VIF_UNPACK_TO_ADDR is specified, meaning that the address passed as first parameter is an absolute address. Data type are V4_32 and starting address is location 0 of VU1 memory. This VIFcode has to be shifted by 96 and combined with *or* previous content of DMA header. This makes VIFcode fitting the fourth Word of DMA header as represented in figure.

127 … 96	95 … 64	63 … 0
UNPACK	NOP	DMA CHAIN TAG (END)

When DMA transfer start, DMAC must be instructed to transfer DMA tag together with the data, this means that first VIFcode (NOP) is the first thing passed to VIF1, then UNPACK code comes and VIF1 now waits for data to come.

```
DMAPacket++;
//Put matrix into the packet
DMAPacket++->ul128 = *(sps2uint128 *) TransformationMatrix[0];
DMAPacket++->ul128 = *(sps2uint128 *) TransformationMatrix[1];
DMAPacket++->ul128 = *(sps2uint128 *) TransformationMatrix[2];
DMAPacket++->ul128 = *(sps2uint128 *) TransformationMatrix[3];
```

Starting microprogram after data are sent

This is the data of transformation matrix that follows DMAtag into DMA packet, then GIF tag is build and inserted into the packet and finally vertices data. After all the data are stored in VU1 memory it is time to start the execution of the microprogram, this is done with *MSCAL VIFcode*, but remember that before starting to execute the code a *FLUSH VIFcode* must be send to be sure that loading of data in Vector unit memory is completed. These two VIFcode are sent immediately after texture coordinate data:

```
DMAPacket->ul128          = 0;
DMAPacket->ul32[0]        = PS2_DMA_SET_VIFCODE_FLUSH;
DMAPacket->ul32[1]        = PS2_DMA_SET_VIFCODE_MSCAL(0);
```

First of all the whole QWord is reset to value zero, then it is sufficient to put the two VIFcode in the right order, first the FLUSH to wait for loading operation to finish, then MSCAL VIFcode that actually starts the microprogram at the address 0 into VU1 Micro Memory.

127 … 96	95 … 64	63 … 32	31 … 0
NOP	NOP	MSCAL 0	FLUSH

4.8 Microprogram

Vector Units first microprogram

Finally it is time to examine file *BasicTransform.vsm* that contain assembly code for the microprogram that transform the vertices. First of all transformation matrix is load into the first four registers of VU1 then number of vertices is retrieved by the NLOOP field of the GIFtag that come with the data:

```
NOP                     ILWR.x        VI02, (VI01)x
NOP                     IADDIU        VI03, VI00, 0x7fff
NOP                 IAND          VI02, VI03, VI02
```

Since VI01 contains pointer to the GIFtag, *ilwr* instruction load first Word of GIFtag into VI02 register, then a simple mask is build to mask out NLOOP part, the result is number of vertices stored in VU1 memory. Then is time to calculate address of vertices, colors and texture coordinates:

```
NOP                 IADDIU        VI10, VI01, 1
NOP                 IADD          VI11, VI10, VI02
NOP                 IADD          VI12, VI11, VI02
NOP                 IADD          VI13, VI12, VI02
NOP                 IADDIU        VI04, VI13, 1
NOP                 IADDIU        VI05, VI13, 2
NOP                 IADDIU        VI06, VI13, 3
NOP                 LQ VF5,       0(VI01)
NOP                 SQ VF5,       0(VI13)
```

Storing pointer to **input data**

First of all adding 1 to VI01 we store in VI10 the pointer to the first vertex, then adding number of vertices to this register make possible to obtain pointer to first color vector. Same thing happens for texture coordinate. After all input data pointer are calculated, output data pointer must be found. Output data is stored immediately after input data so adding the number of vertices to texture coordinate pointer (VI12) produces address of first free memory address, here GIFtag will be placed and all vertices data will follow. VI04 store pointer for output texture component, VI05 will store pointer for colors and finally VI06 store pointer for XYZF2 coordinates. Remember from chapter 3 that is very important that texture data comes before color data. Finally GIFtag is moved to output location. Now main loop begins and it is executed for each input vertex data..

Loop that transform the vertices to produce output

```
NOP                 LQI    VF10, (VI10++) ;load vertex
NOP                 LQI    VF11, (VI11++) ;Load Color
NOP                 LQI    VF12, (VI12++) ;Load texure

;Apply transformation matrix.
MULAx.xyzw ACC, VF01, VF10x       NOP
MADDAy.xyzw ACC, VF02, VF10y      NOP
MADDAz.xyzw ACC, VF03, VF10z      NOP
MADDw.xyzw VF15, VF04, VF10w      NOP

;Wait for operation to be executed
NOP                 NOP
NOP                 NOP
NOP                 NOP
;Calculate 1/W
NOP                 DIV Q, VF00w, VF15w
```

Load with increment instruction is used to load data for current vertex in register VF10, VF11 and VF12, then transformation matrix is applied to vertices coordinates and finally value 1/W is calculated, dividing the VF00 w component (VF00 is the constant register with value (0, 0, 0, 1)) by the w component of transformed vertex. Now it is time to deomogenize vertex transformed coordinates, same value 1/w is also used to find real S,T,Q texture coordinate values:

```
MULq VF16, VF15, Q          NOP

;Now normalize texture coordinate values.
MULq VF17, VF12, Q          NOP
NOP                         NOP
NOP                         NOP

;Convert to fixed point
FTOI4 VF19, VF16            NOP
```

Apply texture perspective correction

Even texture coordinate vector must be multiplied for 1/W value to apply perspective correction (See paragraph 3.6 for further details).

Dealing with Z-Buffer value

Since format of frame buffer is fixed point format (12:4) instruction *ftoi4* perform the conversion between standard floating point format to fixed point format. Here a strange thing happen, z component of the vector will go to the Z-Buffer for depth test, but format of Z-Buffer is 24 bit integer, but ftoi4 transform float value into 16 bit fixed point value, that is very different from Z-buffer value. To maintain code simple a little trick is used to avoid artefact on depth test in rendered cube. If we consider a fixed point value 12:4 as a simple integer value, we are actually multiplying real value by 16. Lets make an example: if we have value 1,0f converting with ftoi4 lead to binary value 0000000000010000, representing value 1 in format 12:4, if we consider this value a simple integer value we obtain value 16. This means that after transformation with ftoi4, Z real value is multiplied by 16, to avoid this problem the Viewport Mapping transform is done with a range for Z that is different from the one really used. Since output value is 12:4 we have only 16 bits for Z value and valid range is (0, 65535) but since original value will be multiplied by 16 by the ftoi4 function, real range become (0, 4095).

```
ProjMatrix.MapToViewPort(2048-  wpWidth / 2,
      2048 + wpWidth / 2,
      2048 + wpHeight / 2,
      2048 - wpHeight / 2,
      4095.0, 0);
```

If original value of 16777215 is used, artefacts will occur because every value greater than 4095 will overflow during the conversion and makes strange things happens on the cube. Finally vertex output data is stored after GIFtag

```
NOP                         SQ VF17, 0(VI04)
NOP                         IADDIU VI04, VI04, 3
;Now color and vertex
NOP                         SQ VF11, 0(VI05)
NOP                         IADDIU VI05, VI05, 3
;Now coordinates.
NOP                         SQ VF19, 0(VI06)
NOP                         IADDIU VI06, VI06, 3
```

Every time that some data is stored the corresponding pointer is incremented by 3, this because for every vertex we have 3 distinct output data. Then the end of the loop is reached.

```
NOP                       IADDI VI02, VI02, -1
NOP                       NOP
NOP                       IBNE VI02, VI00, LOOP
NOP                       NOP
```

Don't forget branch delay slot

To close loop it is sufficient to decrement VI02 register, that contain number of vertices, and jump to the beginning of the loop if this value has not reached zero. When VI02 reach zero, all input vertices are transformed and the loop can end. Remember that the instruction after a branch such as IBNE will be executed even if the instruction will branch, this is known as *Branch Delay Slot*. In the example above a NOP NOP instruction is placed after the IBNE to avoid problems. If the XGKICK instruction would be inserted immediately after IBNE instruction, kicking vertex will start at the end of every cycle iteration, making a mess on the screen. Last instruction end the microprogram and kick the vertex to the GS

```
NOP[E]                    XGKICK  VI13
NOP                       NOP
```

This conclude our first VU1 microprogram. Remember that the code above is not optimized and to make faster and cleaner microprogram, VCL tool can be used because it perform automatically optimization and makes easier to write code. Finally do not forget to use *svvudb*, sauce's VU's debugger that will help you a lot in coding VUs.

A Three dimensional view

After all the objects are settled into three dimensional space and the scene is ready to be rendered, it is necessary to apply a transformation to map the scene itself on a two dimensional space, the monitor. This transformation, that maps 3D vectors in 2D ones, is clearly a projection and it's a linear transformation in Homogeneous space, so it can be expressed by 4x4 square matrix as for World Matrix. Now let's look on how to derive it's representation.

To find a solution is convenient to divide the original problem into more smaller ones. The final transformation is in fact derived in three step, because each of these is conceptually separated from the other. So the Projection Matrix is composed by: *View transform, Projection transform and ViewportMap transform*. In next section these transformations will be analyzed with great detail to find 4x4 transformation matrices that will form final projection matrix.

A.1 View Transform

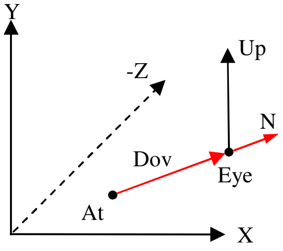
Once the scene is ready to be rendered, the first thing to do is knowing the position and the orientation of the viewer, this is done with *View transform* that is represented by a matrix called *View Matrix*. The aim of this transform is to change coordinate system to simplify the subsequent projection operations. In the new coordinate system the camera (the eye of the viewer) is located at the origin, and the direction of viewing is coincident with one of the coordinate axes.

To define a new coordinate system, four distinct data are necessary, the position of the new origin and the three axis that will form the new coordinate system. In a view transform the new origin is the position of the camera, one of the axis is the direction of view, another is the *up* direction and the third is determined by other two with a cross product to have an orthonormal basis. The transformation that map a point from the original coordinate system $(O,\bar{X},\bar{Y},\bar{Z})$ to the new coordinate system $(O,\bar{X}',\bar{Y}',\bar{Z}')$ is well known and is composed by a translation of $-O'$ and a rotation expressed by a matrix in witch the row are the coordinates of the new coordinate axes expressed in the original system. The whole operation is so reduced in finding the expression of the new reference coordinate system.

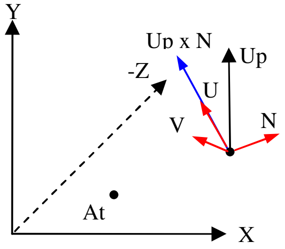
$$\begin{bmatrix} X_x & X_y & X_z & 0 \\ Y_x & Y_y & Y_z & 0 \\ Z_x & Z_y & Z_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -O'_x \\ 0 & 1 & 0 & -O'_y \\ 0 & 0 & 1 & -O'_z \\ 0 & 0 & 0 & 1 \end{bmatrix} =$$

$$\begin{bmatrix} X_x & X_y & X_z & -(X_x O'_x + X_y O'_y + X_z O'_z) \\ Y_x & Y_y & Y_z & -(Y_x O'_x + Y_y O'_y + Y_z O'_z) \\ Z_x & Z_y & Z_z & -(Z_x O'_x + Z_y O'_y + Z_z O'_z) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

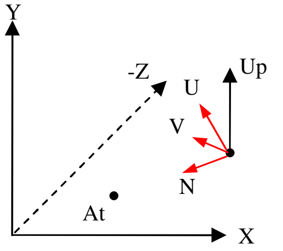
There are many techniques to find the expression of View Coordinate System, but one of the simplest and easiest to use is to define the camera with only three vector: *Eye, At, Up*. The first is the position of the camera, the second is the point that the camera it is looking at and the third is the general direction of Up. Since the Up vector must be perpendicular to the others, it is simpler to give only its general direction and let the routine that build the matrix adjust the value. Given these three values, it is possible to build two different view matrix, one right handed and the other left handed; over these two, the latter is preferred because maintains a more intuitive direction of view. In this section we derive both of them, so the user is free to use whichever form he prefers.



To determine *right handed view matrix* it is necessary to find the expression of the *view coordinate system* $(Eye, \bar{U}, \bar{V}, \bar{N})$ expressed in current coordinate system. Eye is given by the user, so the first vector to find is the direction of view (Dov) that is determined by $(Eye - At)$, even if this seems strange because the direction of view is from At to Eye this is needed to build a right handed system, vector N is found normalizing Dov.

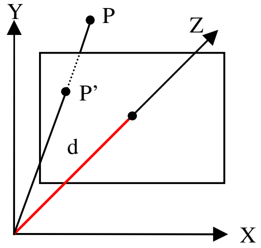


Next step consist to find vector U that is obtained normalizing cross product of the general vector Up with N. Remembering the definition of cross product, resulting vector is perpendicular both to Up and N and has the direction show in figure. Last step is to find the real Up vector called V, this is accomplished with another cross product between N and U as shown in figure. Right handed view coordinate system has the unpleasant characteristic that the camera looks towards the negative part of the z-axis, this lead to the fact that object nearer to the viewer have greater z value respect to farer object.

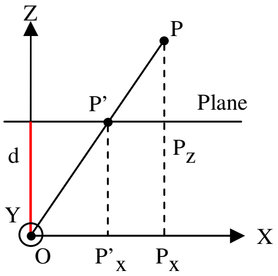


Deriving a left handed view coordinate system is straightforward, simply find N by normalizing $(At - Eye)$, U is found by normalizing cross product of N and Up and finally V is found with cross product of U and N. Now the reference system is left handed but the direction of view is coincident with z-axis and this is more intuitive and simpler to work with.

A.2 Projection transform after Left Handed view transform



After view transform there is the need to project the object from 3D space to 2D space to represent the scene on a monitor. Since the *Direction of View* is coincident with the z-axis the projection transform is straightforward. The only parameter is the distance of the screen of projection from the origin, called **d**, as represented in red in the figure. To determine the operation needed to project a point P into the plane, to obtain P', a simple consideration can be done.



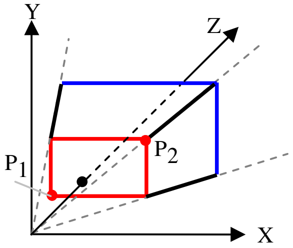
First of all it is better to observe the scene Y axis, as shown in the left picture. From the similitude of the two triangles $OP'P'_x$ and OPP_x it is easy to see that:

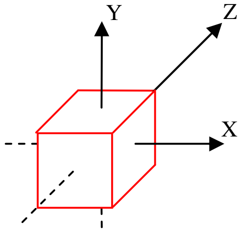
$$P'_x = \frac{P_x d}{P_z}$$

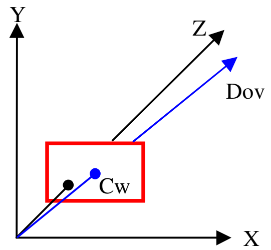
Same thing happens for y coordinates. The matrix that project a point P into the plane perpendicular to z-axes at distance d is easily computed as

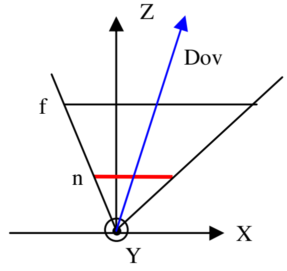
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

Division by z is obtained after the deomogeneization of the coordinates (division by W). Even if this matrix correctly implement a projection it is not used because of clipping. The problem is that our viewport has a finite extension but the plane used for projection is infinite, a better way to proceed is to define a finite projection area that represent our viewport called *near plane*, represented in red in the figure. To define this area two point must be defined: the bottom left corner $P_1 = (l, b, n)$ and the upper right corner $P_2 = (r, t, n)$, also the maximum z value is given, and it's called f, to define the area in blue called *far plane*. The volume determined by these two quadrilateral is called *Viewing Frustum* and contains all the objects that must be projected on our viewport. The goal is to find a transformation that map the Vieving Frustum in the unit cube, that is the cube having one corner at $(-1,-1,-1)$ and the other at $(1,1,1)$. This transformation actually implement a perspective projection of the point inside the frustum and makes clipping very easy because it is sufficient to check if the coordinate lies outside the interval $[-1,1]$.









First transformation to do is making direction of view (Dov), determined by the near plane, coincident with z-axis. This operation is needed if the z-axis intersect near plane in a point different from it's center as showed in the picture. To obtain this it's sufficient using a shearing transform in x and y direction to make the *Center of view* (C_W) lie on z-axis. The general form of the shearing matrix is:

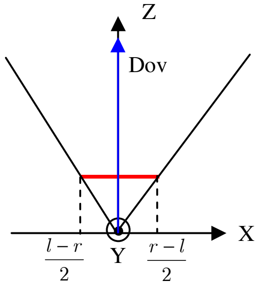
$$Sh = \begin{bmatrix} 1 & 0 & Shx & 0 \\ 0 & 1 & Shy & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We must find the value of Shx and Shy that map Cw into z-axis, the coordinate of the center of projection are:

$$Cw = \left[\frac{r+l}{2}, \frac{t+b}{2}, n\right]^T$$

After the shearing transform its coordinate will be:

$$\begin{bmatrix} 1 & 0 & Shx & 0 \\ 0 & 1 & Shy & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{r+l}{2} \\ \frac{t+b}{2} \\ n \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{r+l}{2} + nShx \\ \frac{t+b}{2} + nShy \\ n \\ 1 \end{bmatrix}$$



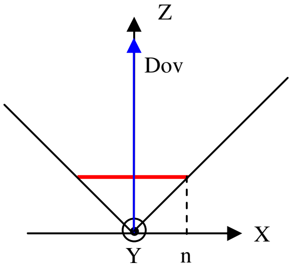
Now it is sufficient to set to zero x and y transformed coordinates to find the value of the shearing parameters that maps center of projection over the z-axes.

$$Shx = -\frac{r+l}{2n}$$
$$Shy = -\frac{t+b}{2n}$$

After this transformation near plane is centered on z-axis and the new value for the bottom left and upper right corner are:

$$P_1 = \left[\frac{l-r}{2}, \frac{b-t}{2}, n\right]$$
$$P_2 = \left[\frac{r-l}{2}, \frac{t-b}{2}, n\right]$$

Next step is to do an uniform scaling in x and y coordinates to make the slope of bounding plane equal to 1, this means that near plane has to become a square with side length of 2n. Is easy to see that the scale factor must transform the x coordinate of P_2 from the value (r - l) / 2 to new value n. the

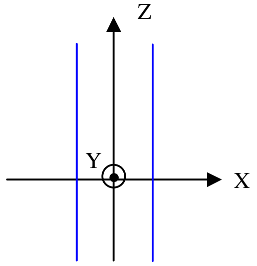


matrix that represent this scaling is well known:

Now that the sides of the frustum have unit slope it is possible to find

$$Sc = \begin{bmatrix} \frac{2n}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2n}{t-b} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

the last transformation, that makes viewing frustum become unit cube. This particular transform has to modify the sides of the frustum so they become parallel to plane X = 0 and Y = 0. To achieve this it is sufficient to divide x and y coordinate by coordinate z, with the same procedure shown at the beginning of the chapter. This determines the first, second and fourth row of



$$\mathrm{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ P_{31} & P_{32} & P_{33} & P_{34} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

the matrix that represent this transform:

First two rows are same as unit matrix because x and y coordinates should not be changed. Fourth column is calculated so w coordinate of transformed vertex will store original z value, in this way the division by z it is performed during deomogeneization.

Third row of the matrix determines the z coordinates of transformed vertices and it has to be calculated to satisfy the condition that near plane has to be transformed in plane $Z = -1$ and far plane to $Z = 1$. First thing to take into account is that planes parallel to $Z = 0$ maintain this property after transformation, this means that new z coordinate does not depend on original coordinate x or y and $P_{31} = P_{32} = 0$. There is also to remember that transformed z value has to be divided by new w value to omogenize the vector and find the real z value. The expression of transformed z value is:

$$z' = \frac{P_{33}z + P_{34}w}{z}$$

This expression takes into account that transformed w has the value of the original z. Now if we apply the condition seen before a simple system is obtained.

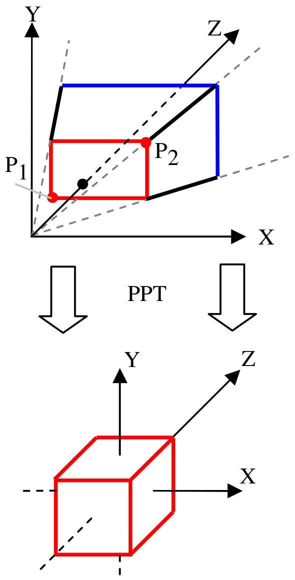
$$\begin{cases} \dfrac{P_{33}nw + P_{34}w}{nw} = 1 & with \quad \dfrac{z}{w} = n \\ \dfrac{P_{33}fw + P_{34}w}{fw} = -1 & with \quad \dfrac{z}{w} = f \end{cases}$$

This is a simple linear system that gives back the value of P_{33} and P_{34} , this will complete the third part of the projection transform.

$$\begin{cases} P_{33} = \dfrac{n+f}{f-n} \\ P_{34} = \dfrac{2fn}{n-f} \end{cases}$$

So the whole matrix is:

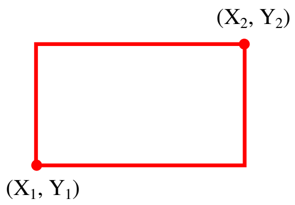
$$\mathrm{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \dfrac{n+f}{f-n} & \dfrac{2fn}{n-f} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$



Now the viewing frustum it is mapped into the unit cube and it's sufficient to combine these three transform together to find the matrix that represent the fully *perspective projection transform.*

$$PPT = \mathrm{P} \cdot Sc \cdot Sh = \begin{bmatrix} \dfrac{2n}{(r-l)} & 0 & \dfrac{(l+r)}{(l-r)} & 0 \\ 0 & \dfrac{2n}{(t-b)} & \dfrac{(b+t)}{(b-t)} & 0 \\ 0 & 0 & \dfrac{f+n}{f-n} & \dfrac{2fn}{n-f} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

A.3 Mapping to the screen



Once the viewing frustum is mapped into unit cube there is the need to represent the point into a 2 dimensional screen that has finite coordinate (X1, Y1) (X2, Y2) as represented into left figure. This means that it is sufficient to transform coordinate from the unit cube to screen. To transform a generic interval $[X_1, X_2]$ into an interval $[X_3, X_4]$ three transformation are needed: first is a translation by quantity $-X_1$ to make interval start at 0, then a scaling with a factor of $(X_4 - X_3)/(X_2 - X_1)$ makes the two interval extension equal, finally a translation by $-X_3$ complete the transform. This transformation has to be applied to X, Y and Z transformed coordinates, having as destination interval the extension of the viewport. Since a screen does not have a Z coordinate, the extension of the Z-Buffer is usually used, to fully use the extension of the Z-Buffer.

The whole transform matrix become:

$$VMT = \begin{bmatrix} \frac{X_2 - X_1}{2} & 0 & 0 & \frac{X_2 + X_1}{2} \\ 0 & \frac{Y_2 - Y_1}{2} & 0 & \frac{Y_2 + Y_1}{2} \\ 0 & 0 & \frac{Zf - Zn}{2} & \frac{Zf + Zn}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Combining together *View transform, Projection transform, ViewportMap transform* we can obtain the full transformation that map a single point on 3D coordinate system to 2D coordinate screen of the monitor.