PS2 Linux Programming Using SPS2 – Tutorial 1

Introduction

This tutorial will provide a commentary on the file sps2_triangle.c that is used to draw a single static Gouraud shaded triangle using the PS2 Linux Development Kit and the SPS2 direct access module. Some prerequisite knowledge fundamental to the understanding of the program will also be provided. The tutorial is written for version 0.3.0 of SPS2. Thanks go to Jonathan Hobson (kazan) and Steven Osman (sauce) for their assistance in the production of this tutorial.

Getting Started

Initialising SPS2 is achieved by calling sps2Init() at the start of the program. This function gains access to the SPS2 kernel module and performs a number of additional functions, which do not concern us at this stage. The sps2Init() function returns a device descriptor which is a signed 32-bit integer. If this value is less than 0 the initialisation failed, most likely because the SPS2 module has not been loaded. The descriptor value must be retained since it will be needed to pass in to other SPS2 functions.

Since graphics are to be drawn, the screen must be initialised with sps2UScreenInit(0). This function will auto detect the video hardware and select an appropriate screen resolution, and create a virtual console. The screen is initialised for PSMCT32 pixel format, a 24 bit Z buffer and double buffering. The zero passed into the function indicates that all signals with handlers that close the virtual console are trapped before aborting the program thus allowing a graceful exit. The PCMCT32 pixel format describes each pixel by a 32-bit number, 8-bits for each of the red, green and blue colours and 8-bits for an alpha value, which is used for blending and transparency effects. The pixel storage formats can be seen on pages 27 and 28 of the GS Users Manual. The purpose of the Z buffer and the use of double buffering will be described later in this tutorial.

Towards the end of the program, after the main render loop, is sps2UscreenShutdown() which as the name suggests shuts down the screen, and sps2Release() which releases and closes down the sps2 module. These two functions must be called at the end of the program to provide proper termination and cleanup.

The Render Loop

The render loop consists of three functions: sps2UScreenClear(0,0,0); drawTriangle(); and sps2UScreenSwap(). A double buffered system is being used which means that there are two frame buffers in use. One frame buffer is being displayed on screen whilst the other is being drawn to. sps2UScreenClear(0,0,0) clears the buffer that is about to be drawn to. The three parameters are the rgb components of the colour that the buffer is cleared to – which is this case is black. drawTriangle() draws the required

triangle into the off-screen buffer – this function will be studied in more detail later. Finally, sps2UScreenSwap() swaps the display and draw buffers so that what was being drawn is now being displayed and vice-versa. Note that this function waits for a vertical synchronisation signal (VSync) from the graphics hardware before swapping the buffers, so that the buffers are swapped during the vertical fly back period of the display, thus preventing undesirable visual effects such as shearing and flicker. In essence, the graphics are scanned out onto the monitor in a series of horizontal scan lines starting from the top left hand corner of the monitor and finishing at the bottom right hand corner. When the scanning process reaches the bottom right hand corner of the monitor it starts over again from the top right hand corner in a continuous loop. The period when the scanning process returns from the bottom right hand corner to the top left hand corner is called the "fly-back" period, this being a very important time for graphics synchronisation purposes. It is during this "fly-back" period that the buffers are swapped, thus creating a smooth transition from one frame of graphics to the next. When all the drawing has been completed for a given frame, sps2UscreenSwap() waits for this fly-back period to commence before it swaps the buffers. The start of the fly-back period is signalled by a VSync interrupt from the graphics hardware. This render loop continues indefinitely until the program is terminated.

Defining The Triangle

Before the triangle can be drawn it is necessary to specify the position and colour of each of the three vertices. A 3 by 2 array (aVertices) is defined to hold the vertex positions. Each vertex is defined from an origin, which is located at the centre of the display. A 3 by 3 array (aColors) is defined to hold the colour of each vertex. An rgb triplet with r, g and b in the range 0-255 defines each vertex.

Some Background

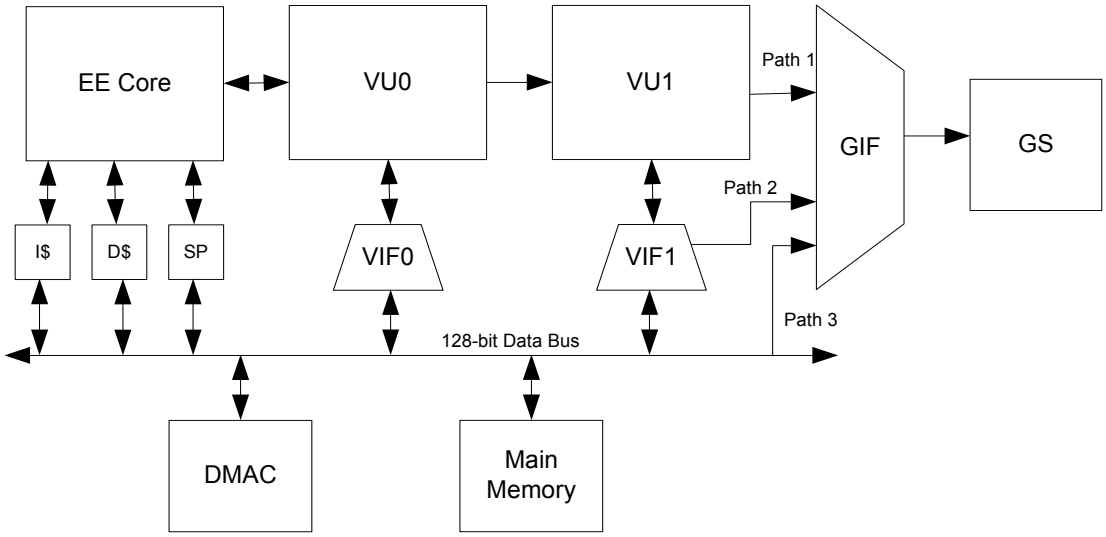

Figure 1

For the purposes of this tutorial, figure 1 shows the main internal data paths that exist within the PS2. It can be seen that the transfer of information to the Graphics Synthesiser (GS) is via the Graphics Synthesiser Interface (GIF). There are three ways to transfer information to the GS via the GIF: the only way which concerns us here is the transfer path which goes from the EE Core via the 128-bit wide data bus to the GIF, which is called Path 3. Path 3 is the slowest of the 3 transfer modes, but it is also the easiest to use.

In essence, the GIF is an interface between the EE core and the GS. The GIF functions as a pre-processor for the GS. The GIF performs a number of functions such as unpacking packed vertex data based on specifications that it has been sent through a GIFTag, and transfers the data directly into the registers of the GS. This process improves the throughput of the system.

Data that is sent to the GS is called a GS Packet. A GS Packet can be made up of a number of segments called GS Primitives. A GS Primitive is a GIFTag and the following primitive data. The overall layout of a GS Packet is illustrated in Table 1 which shows a GS packet consisting of three GS primitives.

GIF Tag 1 (End of packet = 0)
Primitive data
GIF Tag 2 (End of packet = 0)
Primitive data
GIF Tag 3 (End of packet = 1)
Primitive data

Table 1

Note that in the example code a GS Packet consisting of only one GS Primitive is being sent to the GS per frame.

A GIFTag is a single quad word (128 bit) piece of data with a number of fields that tell the GS what type of primitive is to be drawn. The required primitive data, such as the position and colour of the vertices, immediately follows the GIFTag.

There are 7 different primitive types that can be drawn by the GS: point, line, line strip, triangle, triangle strip, triangle fan, and sprite. See pages 38 and 39 of the GS Users Manual for diagrams of how each primitive is built from the vertex data that is specified. Note that a triangle primitive is being drawn in the example code

The primitive data that follows the GIFTag can be stored in 3 different formats, PACKED mode, REGLIST mode, and IMAGE mode. PACKED mode is used in the example code and only this mode will be described at this time.

GIFTag Format

In the example code the sps2GIFTag_t union is used for building the GIFTag data. The union is as follows:

```
typedef union {
    sps2uint128 i128;

    struct {
      unsigned int NLOOP    :15;
      unsigned int EOP      :1;
      unsigned int _PAD1    :16;
      unsigned int _PAD2    :14;
      unsigned int PRE      :1;
      unsigned int PRIM     :11;
      GIF_FLG_t FLG         :2;
      unsigned int NREG     :4;
      sps2uint64 REGS       :64;
    }s;

    sps2uint64 ai64[2];
}sps2GIFTag_t;
```

The union keyword specifies that its members share the same memory. This means that the whole structure “s” can be cleared to 0, simply by setting i128 to 0. Also, inside the struct “s” a colon and a number follows all the declared variables. This forces a bit field to be used for the variable, the length of the bit field being specified by the number after the colon. For example, NLOOP will use bits 0-14, EOP will use bit 15 etc. Values can be assigned just like with regular structures variables with the

advantage that the compiler packs the bits according to the defined specification. The bit-packing format used in this structure mirrors the packing format of the GIFTag, which can be seen on page 151 of the EE Users Manual.

The _PAD parameters can be ignored, as they are only there to give the structure the correct alignment. The rest of the parameters will be describe from the bottom up starting with REGS, this will require a bit more information on how data is stored in a GS Packet.

Primitive data is stored in user defined “batches”. An example of a batch might be the colour and position of the vertex of a triangle. The REGS field is 64 bits long, which can be thought of as an array of 16 4-bit elements, with each element represents a different register. Page 152 of the EE Users Manual details all the registers that can be used in the REGS field. Note that in the example code only the XYZ2 and RGBAQ are used.

One thing to note when setting up a batch is that writing to XYZ2 performs a “Vertex Kick” and can also perform a “Drawing Kick”. A Vertex Kick causes the current contents of the GS registers set up so far to be placed in the vertex queue, which is the queue of vertex data waiting to be processed by the GS. A Drawing Kick is performed when the necessary vertex information is arranged in the vertex queue in such a way that drawing of a primative can begin. In our example code, a Drawing Kick will be performed when the three pieces of vertex information necessary to draw the triangle are available in the vertex queue. The order that vertex information is given to the GIF is therefore very important. If a vertex is to be a specific colour, the colour register must be set before the XYZ2 register so that when the Vertex Kick is performed the correct vertex colour is in the RGBAQ register. If the register order sent to the GIF is reversed, the drawing kick will be performed before the RGBAQ register is set and the vertex will take the colour of whatever value was already in the RGBAQ register. See page 40 of the GS Users manual for more information on Vertex and Drawing Kicks.

In the example code a Gouraud shaded triangle is being drawn which has three vertices each with a position and colour. Thus the registers are specified in the following order: RGBAQ first, then XYZ2.

The following macro is applied to pack the register information into the REG field:

```
SPS2_SET_GIF_REG(gifTag, gifTag.s.NREG++, GIF_REG_RGBAQ);
SPS2_SET_GIF_REG(gifTag, gifTag.s.NREG++, GIF_REG_XYZ2);
```

RGBAQ is packed first followed by XYZ2. GIF_REG_RGBAQ is the register address of RGBAQ, which is defined as 0x01, and GIF_REG_XYZ2 is the register address of XYZ2, which is 0x05. gifTag is the sps2GIFTag_t union that is being used and gifTag.s.NREG is the NREG field of the GIFTag structure. In essence, each time this macro is executed, NREG is incremented by one (remember that NREG was initialised to zero at the start) and the specified register address is packed into the REGS field. It can therefore be seen that NREG specifies the number of registers that are being used, which in this case is 2.

The FLG field specifies the data format that is being used to specify the primitive data that follows the GIFTag. Packed mode is being used so FLG is set to GIF_FLG_PACKED, which is defined as 0x00.

The PRIM register of the GS is a packed bit-field specifying the kind of primitive to be drawn. There are several fields in this register, which will not all be explained here. For the purposes of this tutorial only the first 3 bits (0-2) that specify the primitive type (PRIM) to be drawn, and bit 3 that specifies the primitive shading (IIP) will be used. See page 116 of GS Users manual for information on all of the fields of the PRIM register. In the case of the example code, a triangle primitive is to be drawn (PRIM = 011) and Gouraud shading is to be applied (IIP = 1) so the complete PRIM register is set to 1011 binary, 0x0B hexadecimal or 11 decimal.

The EOP field simply indicates if this is the last GIF Tag in the GS Packet. If EOP=1 the GIF will not expect any more GIFTags and primitive data in the graphics packet, effectively meaning that the current graphics packet has ended transmission. In the case of the example code EOP is set to 1 to indicate that this is the last (and first) packet that the GIF should expect. If there were more packets to follow, EOP would be set to 0 with the final packet having EOP set to 1.

The PRE field indicates whether or not to transfer the PRIM field of the GIFTag to the PRIM register of the GS. There are other methods available to set the PRIM register of the GS but for now PRE is set to 1 indicating that the information in the PRIM field of the GIFTag should be transferred to the PRIM register of the GS.

Finally NLOOP tells the GIF how many times to repeat reading the register data specified in the REGS field. In the example code there are two registers being used to specify each vertex and there are three vertices. NLOOP is set to 3 indicating that the register data should be read three times, once for each vertex. Note that this means that the GIF will be expecting a total of NLOOP x NREG (3 x 2 = 6) register values in the primitive data section after the GIFTag. It is important to get these calculations correct otherwise things will not work as expected.

Primitive Data

The primitive data comes immediately after the GIFTag and in the case of the example code is specified in Packed mode. Packed mode uses a whole 128 bit quad word to store data for one 64 bit register within the GS. One advantage of this is that it is quick and easy to set the primitive data. The disadvantage of using packed mode is that twice as much data (as is necessary) is needed to specify each GS register.

The format used for packing each register is specified in section 7.3 of the EE users Manual. The packing formats are mirrored in the structures contained within the sps2GIFPackedRegister_t union. The structures used in the union can be found in the sps2tags.h file. Each of these structures packs the bits according to the formats described in section 7.3 of the EE Users Manual.

```
typedef union {
    sps2uint128                 i128;
    sps2GIFPackedPRIM_t         PRIM;
    sps2GIFPackedRGBAQ_t        RGBAQ;
    sps2GIFPackedST_t           ST;
    sps2GIFPackedUV_t           UV;
    sps2GIFPackedXYZF2_t        XYZF2;
    sps2GIFPackedXYZ2_t         XYZ2;
    sps2GIFPackedFOG_t          FOG;
    sps2GIFPackedA_D_t          A_D;
    sps2GIFPackedNOP_t          NOP;
}sps2GIFPackedRegister_t;
```

This union can be used to pack any register into the required format using the same qword of memory. For now there are only two register formats that concern us, XYZ2 and RGBAQ:

```
typedef union {
    sps2uint128 i128;

    struct {
      unsigned int R        : 8;
      unsigned int _PAD1    : 24;
      unsigned int G        : 8;
      unsigned int _PAD2    : 24;
      unsigned int B        : 8;
      unsigned int _PAD3    : 24;
      unsigned int A        : 8;
      unsigned int _PAD4    : 24;
    }s;

} sps2GIFPackedRGBAQ_t;

typedef union {
    sps2uint128 i128;

    struct {
      unsigned int X        : 16;
      unsigned int _PAD1    : 16;
      unsigned int Y        : 16;
      unsigned int _PAD2    : 16;
      unsigned int Z        : 32;
      unsigned int _PAD3    : 15;
      unsigned int ADC      : 1;
      unsigned int _PAD4    : 16;
    } s;
} sps2GIFPackedXYZ2_t;
```

The active fields of RGBAQ are R, G, B and A. The red, green and blue and alpha fields are in the range 0-255. The Q field is of no concern at this time.

The active fields of XYZ2 are X, Y, Z, and ADC. The ADC is of no concern at this time but if this ADC bit is set to 1, the drawing kick is not performed for this vertex - this can be a useful feature if some triangles are not to be drawn but it is also important in the case of the example code that ADC is cleared to zero (this being done when the union memory is set to 0). The PS2 uses a Z-Buffer to automatically sort out which pixels are closest to the viewer and should or should not be drawn. The Z value of a vertex is a 32 bit unsigned integer. The higher the Z value the closer the primitive is to the screen. If two pixels are to occupy the same position on screen, the pixel with the higher Z value will be drawn and the pixel with the lower Z value will not be drawn. This feature is used to sort out which objects (or parts of objects) are closest to the viewer in 3 dimensional graphics. Note that the Z value cannot be 0 or the primitive will not be drawn. The X and Y values are the horizontal and vertical coordinates of the vertex. The centre of the PS2 screen is at the coordinate (2048, 2048). The positive X axis goes from left to right across the screen, the positive Y axis goes from top to bottom down the screen. Therefore for a vertex 150 pixels to the left, and 150 pixels up from the centre of the screen the coordinates (2048 – 150, 2048 – 150) would be used – this can be seen in the code below. Finally the X and Y coordinates are in 12:4 fixed point format which means that the coordinate data must be left shifted by 4 – again, this can be seen in the code below. The alpha value is set to a default value of 128

The section of example code that packs the primitive data is repeated here for clarity:

```
// First, prepare and sent the RGBAQ register
gifRegister.i128=0;
gifRegister.RGBAQ.s.R=aColors[iPointLooper][0];
gifRegister.RGBAQ.s.G=aColors[iPointLooper][1];
gifRegister.RGBAQ.s.B=aColors[iPointLooper][2];
gifRegister.RGBAQ.s.A=0x80;
DPUT_EE_GIF_FIFO(gifRegister.i128);

// Next, prepare and send the XYZ2 register
gifRegister.i128=0;
gifRegister.XYZ2.s.X=(aVertices[iPointLooper][0]+2048) << 4;
gifRegister.XYZ2.s.Y=(aVertices[iPointLooper][1]+2048) << 4;
gifRegister.XYZ2.s.Z=1;
DPUT_EE_GIF_FIFO(gifRegister.i128);
```

gifRegister is a declared instance of the sps2GIFPackedRegister_t union. Firstly the union is cleared to zero then the colour of the vertex is set from the colour array. This data is then sent to the GIF using DPUT_EE_GIF_FIFO(gifRegister.i128); - this macro will be discussed in a bit more detail below. The union is again cleared then the X and Y vertex coordinated are set from the vertex array. Note that the Z value is simply set to 1, which is fine for this example since only one triangle is being drawn. Once again this data is sent to the GIF using DPUT_EE_GIF_FIFO(gifRegister.i128).

It is seen from the example code that this process is repeated three times in a loop, once for each vertex, providing primitive data that consists of 6 packed registers.

Sending the GS Packet to the GIF

The DPUT_EE_GIF_FIFO() macro writes a qword of data directly onto the FIFO of the GIF. The GIF FIFO is effectively a memory buffer housed within the GIF that can contain 16 qwords of data, this data being processed on a First In First Out basis – hence FIFO. Notice in the example code, that for each frame being drawn a GifTag is sent to the GIF- DPUT_EE_GIF_FIFO(gifTag.i128), followed by the 6 registers which make up the primitive data – 6 x DPUT_EE_GIF_FIFO(gifRegister.i128).

It is important to mention at this point that that DPUT_EE_GIF_FIFO has a hidden problem in that if all 16 entries into the FIFO are full and more data is "DPUT" into the FIFO, the data just sent to the FIFO will be discarded. This is obviously not a problem in the example program provided since only a small amount of data is being sent to the GIF each frame. However, this problem may manifest itself in programs where a lot of information is being transferred to the GIF in a frame. The code segment given below can be used to circumvent this problem:

```
while (*EE_GIF_STAT & (0x1F << 24));
```

This code reads the status register of the GIF and loops until the effective data count (FQC) field of the register has reached zero. The effective data count is the number of qwords remaining in the FIFO. If FQC = 0 then it is possible to send more data to the FIFO. In order to be safe, this code could be put before the DPUT_EE_GIF_FIFO macro, but it will introduce an additional performance hit. It is possible to arrange this fix and associated code to maximise performance but the exact details will depend upon the application in question.

For best performance from the PS2 the Direct Memory Access Controller (DMAC) will be used in future tutorials to transfer data to the GIF. The use of, DPUT_EE_GIF_FIFO however, is excellent for introducing the many important techniques and concepts required for programming the PS2, without the added complications introduced by the use of the DMAC.

Conclusions

Considerable content has been introduced in this tutorial, some of which will be covered again in future tutorials. Some general graphics theory and techniques, which are not specific to the PlayStation 2, have also been mentioned such as double buffering, Z-buffers and monitor scanning. If the reader is not familiar with these, it is recommended that they be investigated in the many Graphics Texts and Internet sites that describe them in more detail.

Finally, in order to enhance understanding of the techniques introduced in this tutorial, it is also recommended that the reader alter the supplied program to, for

example: draw the triangle in a different place; change the colours of the vertices; draw more than one triangle; draw a different kind of primitive; etc; etc. Experimentation leads to understanding.

Dr Henry S Fortuna
University of Abertay Dundee
h.s.fortuna@abertay.ac.uk

```
/*
    Copyright (C) 2002 Terratron Technologies Inc.  All Rights Reserved.
    Author: Steven Osman, Morten Mikkelsen, Lionel Lemarie

    This file is part of sps2.

    Before using this file you MUST agree to the license agreement in
    the file LICENSE provided with this package.

    For more information visit:
    http://www.playstation2-linux.com/projects/sps2/
    http://window.terratron.com/~sosman/ps2linux/
    http://www.terratron.com/

    7/12/2003 - Sauce
    8/26/2003 - hsfortuna (some bug fixes)
*/
#include <stdio.h>

#include <sps2lib.h>
#include <sps2tags.h>
#include <sps2util.h>

void drawTriangle();

// Define the coordinates of our triangle
typedef int vertex_t[2];
vertex_t aVertices[]={
  {-150, -150},  // Up from the center, to the left
  { 150, -150},  // Up from the center, to the right
  {   0,  150}   // Down from the center, in the middle
};  // triangle coordinates

// Define the colors of the points
// Make sure there's one of these for each coordinate!
typedef int color_t[3];
color_t aColors[]={
  {255,  0,  0}, // All red
  {  0,255,  0}, // All green
  {  0,  0,255}  // All blue
}; // triangle colors

/**
 * This is the main function of our tutorial.  It will perform some simple
 * initialization and then set up a small loop to repeatedly draw our
 * triangle.
 */
int main() {
  int iSPS2Descriptor;      // The handle we use for sps2

  iSPS2Descriptor=sps2Init();       // Initialize sps2

  // Initialize the screen.  Zero tells sps2UScreenInit to exit gracefully
  // when it receives a signal.
  sps2UScreenInit(0);

  while (1) {
    // Clear the draw area.   The three zeros mean we want to clear it to a
    // color that has zero red, zero green, and zero blue respectively.
    // In simpler terms, we're clearing the screen to black.
    sps2UScreenClear(0,0,0);

    // Draw the triangle
    drawTriangle();

    // Swap displays now that we're done
```

```
    sps2UScreenSwap();
  }

  // Shut down the screen
  sps2UScreenShutdown();

  // and close sps2
  sps2Release(iSPS2Descriptor);
  return 0;
}

/**
 * This function will draw our triangle.  It's a little bit inefficient
because
 * it builds the whole GIF tag every frame (the GIF tag could have been
 * prepared earlier, for instance).  On the other hand, since we're only
 * drawing one triangle on the screen, this is of very little concern.
 */
void drawTriangle() {
  sps2GIFTag_t gifTag;
  sps2GIFPackedRegister_t gifRegister;
  int iPointLooper;

  // Prepare the GIF tag
  gifTag.i128=0;       // Blank it out
  gifTag.s.NLOOP=3;    // 3 entries.  One entry for each of the points.
  gifTag.s.PRE=1;      // We are providing a valid PRIM value
  gifTag.s.EOP=1;      // End of packet -- draw me please!
  gifTag.s.PRIM=11;    // 3 = triangle | 8 = gourard shading
  gifTag.s.FLG=GIF_FLG_PACKED; // We're using the packed format for data
  gifTag.s.NREG=0;     // Start off with zero registers, increment as we
                         // populate the descriptions

  // Add a register.  It will be an RGBAQ register
  SPS2_SET_GIF_REG(gifTag, gifTag.s.NREG++, GIF_REG_RGBAQ);
  // Add another register.  It will by an XYZ2 register
  SPS2_SET_GIF_REG(gifTag, gifTag.s.NREG++, GIF_REG_XYZ2);

  // Send the GIF tag
  DPUT_EE_GIF_FIFO(gifTag.i128);

  // Now that we promised three sets of RGBAQ and XYZ2 registers, we should
  // deliver them.
  for (iPointLooper=0; iPointLooper<3;iPointLooper++) {
    // First, prepare and sent the RGBAQ register
    gifRegister.i128=0;
    gifRegister.RGBAQ.s.R=aColors[iPointLooper][0];
    gifRegister.RGBAQ.s.G=aColors[iPointLooper][1];
    gifRegister.RGBAQ.s.B=aColors[iPointLooper][2];
    gifRegister.RGBAQ.s.A=0x80;
    DPUT_EE_GIF_FIFO(gifRegister.i128);

    // Next, prepare and send the XYZ2 register
    gifRegister.i128=0;
    gifRegister.XYZ2.s.X=(aVertices[iPointLooper][0]+2048) << 4;
    gifRegister.XYZ2.s.Y=(aVertices[iPointLooper][1]+2048) << 4;
    gifRegister.XYZ2.s.Z=1;
    DPUT_EE_GIF_FIFO(gifRegister.i128);

  }
}
```