How to .... Display sprites

This document is intended to be a quick and simple guide to displaying sprites on your Sony Net Yaroze. It is aimed at anyone who is having trouble with any part of sprite displaying, or 2D graphics. Please contact me with any feedback you may have whether good or bad, I would appreciate it.

This is not an official Sony document, and I do not claim to have perfect knowledge of the Yaroze, but what I do know is the frustration of trying to display the most basic sprite on the screen. That is why I have decided to write this document and share my knowledge and experience with those that need it, after all, isn't that what the whole Yaroze project is supposed to be about?

I apologise in advance for any errors, as I do not claim to be the worlds top Yaroze coder. If you find any bugs/errors then please let me know and I will amend this document.

Ira Rainey - 02 June 1997

Note: Since writing this a lot of time has passed, and I've received a lot of e-mails of thanks about this tutorial, I'm glad it's helped so many people. But as I have received no critism (which is nice, 'cause even I know it isn't perfect - but OK), I have refrained from altering this document in any way whilst putting it into PDF format other than a couple of notes like this and the source code tagged on the end. I hope you find it as useful as many others have before you.

Ira Rainey - 02 February 1998

ira.rainey@btinternet.com

Sprites

In this document I will explain how to create, display, and move your sprites around the screen. I will not go into detail on things that are not directly relevant to displaying sprites, or into how efficient the routines used are - or not. All I intend to do is show you, in basic terms, how to get that character or background from your mind onto the screen.

Creating sprites is the most fundamental aspect of writing video games, whether you use them as moveable characters, or static backgrounds, it's something you just can't do without knowing. Writing code for games needn't be to difficult a job, so long as you understand the fundamentals of the machine that you are working on, that is where good documentation comes in, and I hope that for those with the need, that's what this is.

Game graphics basics

The PSX was designed purely as a games machine (and a damn good one at that), that is the simple down to earth fact. It is this dedicated design that makes game development on the Yaroze so easy.

To make a game look good you must have smooth, flicker free animation, that much is obvious. The flickering of graphics is generally caused by the program drawing to the screen whilst the vertical scan is still trying to draw it. One way to eliminate this, is to use the technique of double buffering. Double buffering is basically the use of two exact areas of memory - one onscreen, one offscreen - that are used for drawing and displaying graphics. The idea is simple, whilst you are displaying one area of memory onto the screen, you are re-drawing with your new sprite positions etc, to the offscreen area of memory that is not visible. When the drawing is finished you just switch the areas of memory around so that the visible screen becomes the offscreen drawing area, and the offscreen area becomes the visible screen. By synchronising this switching with the vertical retrace, you eliminate flicker and thereby produce smooth animation. The Yaroze makes all of these tasks very easy, but first you need to understand the structures and variables used by the Yaroze, and it's graphical memory layout.

Yaroze memory structure

The Yaroze has 2MB of RAM (the consumer PSX also has 2MB, and I believe the professional developer version has 8MB), this is used to store the OS, your applications and your graphics and sound data. Within reason you can store your graphical and sound data wherever you like, as long as it doesn't interfere with your application, or the OS. Also it has an area of memory called the frame buffer. Basically the frame buffer is an area of memory dedicated for video use. Within the frame buffer you must allocate space for your screen images, two of them since we will be using double buffering (incidentally I am assuming that you are planning on producing your sprites using a non-interlaced video mode - I do not cover interlaced mode here), and store data for any images that are currently being displayed. The best way to visualise the frame buffer is like a big piece of graph paper, or a big screen, with 1024 points along the horizontal axis and 512 points down the vertical axis, with the top left corner being 0,0 and the bottom right corner being 1024,512. Each of these areas of memory is a sixteen bit value, giving the frame buffer a total size of 1MB.

Yaroze graphics structures

There are a lot of structures and variables used in the Yaroze and it can get a bit confusing at first, but for basic 2D sprites the only ones you really need worry about are; **GsSPRITE**; **GsIMAGE**; **RECT**. You will also need to roughly understand how the ordering table (OT) system works, which I will cover next in this document.

GsSPRITE is the structure used for holding all the information relevant for any sprite you wish to display. Most of it's members are pretty self explanatory, but I will explain them briefly in the source code.

GsIMAGE is a structure that holds all the information about a relevant TIM file with which you wish to use it's contents as a sprite. Again, I will cover it's members in the source code.

RECT is a simple structure that defines a rectangular area. It's members define the x and y position and it's width and it's height. It's used, in our case, for storing TIM files into the frame buffer.

The rough guide to ordering tables

The most confusing aspect of the graphics system to a new Yaroze programmer is the OT system. The Sony manual doesn't help much in implying it's use is for sorting 3D objects' depth and the like, but is still used for 2D sprites? Confused? I know I was. Basically, an ordering table is just a list of stuff the Yaroze has to draw to the offscreen buffer, and it puts them into a table so that it can allocate priorities to each object so that it knows which object goes in front/behind of which. This principle is exactly the same for 2D graphics. Say for example you have a character walking along a road or something, then you would want him/her to be drawn in front of the buildings etc, but maybe walk behind a tree or something. Using OT's you can assign priorities to each of these objects, so that they 'layer' the way you want them to. But what if, as in our sprite example here, we only have one object? Well, you simply only put one item into the OT.

Once you have registered your sprites/objects into the OT, the drawing of the OT to the offscreen buffer is carried out by the *GsDrawOt()* function, that's all you have to do.

I will also take a diversion here to say a quick word about PACKET's. Basically packets are the primitive work area for the OT, all you really need to know is that you should create a packet area big enough for the number of sprites you wish to use. A good way to do this is to define a global for the number of sprites you wish to use, *#define NO_SPRITES (x+1)*, where x = the total number of sprites, then defining our packet as, *PACKET GpuPacketArea[2][SPRITE_CNT*sizeof(GsSPRITE)]*. By adding one to the total number of sprites we give ourselves plenty of room.

Yaroze graphics programming methods

As we are only talking about 2D sprites here, this is obviously not a definitive guide to graphics programming on the Yaroze, I will only tell you what you need to know relevant to displaying sprites.

The *basic* algorithm is like this:

1) **Initialise graphics system**
2) **Load image data into the frame buffer**
3) **Initialise sprite to image data**
4) **Clear the ordering table**
5) **Register sprite into OT**
6) **Wait for drawing to finish**
7) **Wait for vertical retrace**
8) **Swap the display buffer**
9) **Sort the OT**
10) **Draw the OT to the offscreen buffer**
11) **Go back to step 4**

Following this sequence of steps, as written in the source code, you will get your sprite onto the screen.

One of the first lines of code you should have in *main()*, would be *SetVideoMode(mode)*, where mode is either *MODE_PAL* or *MODE_NTSC*, according to which you plan to use. *MODE_PAL* and *MODE_NTSC* are globals defined in *libps.h.*

The idea of double buffering is the fundamental basis behind graphics on the Yaroze. Because of this you must define, in the frame buffer your two screen memory buffers. This is done by using the functions *GsInitGraph()* and *GsDefDispBuff().*

GsInitGraph() initialises the Yaroze graphics system, no graphics functions can be carried out until this function has been called. The function needs five parameters passed to it when you call it; firstly the horizontal (x) resolution of the graphics screen you wish to run with (eg:320), second is the vertical (y) resolution (eg:240), third is the attribute variable, this is used to set interlace mode and double buffer offset mode, we're going to set it to 4 for GPU offset, non interlace. The fourth parameter we pass is to set dither on or off when drawing, we will set it to 0 (off), and the last parameter sets the video ram mode to either 16 bit or 24 bit, so we set it to 0 for 16 bit.

GsDefDispBuff() is the next function to call. This defines the position of the buffers used for drawing/displaying within the frame buffer. Four parameters are needed, first the x position of buffer 0, second the y position of buffer 0, third the x position of buffer 1, and fourth the y position of buffer 1. The co-ordinates refer to the top left hand corner of each of the buffers. You can put the buffers anywhere you like within the frame buffer, but 0,0 for buffer 0 is normal, and 0, screen_depth for buffer 1, where screen_depth is the y resolution of the display (eg: 240). This puts buffer 1 directly under buffer 0. If you have difficulty with this, try visualising the piece of graph paper again, and you are adding each of the buffers as pieces of coloured paper in the co-ordinates position.

Next you must initialise your image and link the sprite structure to the image data. This is first done by loading your image from it's main memory location, in our case referenced by the global spriteAdd. To do this you use the *GsGetTimInfo()* followed by *LoadImage()* function. *GsGetTimInfo()* gets all of the relevant data from the TIM file and stores it in a structure of type **GsIMAGE**. From this you can find out all of the details of the TIM file, such as size, position in the frame buffer etc. The most important member of the **GsIMAGE** structure, once you have the TIM data, is the **GsIMAGE**.*pmode member*. This member will tell you what type of image it is 4, 8, 16 or 24 bit and whether or not the image has a colour look up table (CLUT).

The valid pmode values are:

4 bit wo/CLUT	**0x00**
4 bit w/CLUT	**0x08**
8 bit wo/CLUT	**0x01**
8 bit w/CLUT	**0x09**
16 bit	**0x02**
24 bit	**0x03**

Once you have this information you can use *LoadImage()* to load the image into the frame buffer. Like all things in the frame buffer, the image data must sit at a certain location, and this location is defined in the header of the TIM file. You can also 'hard code' these positions in your code by setting the **RECT**.*x* and **RECT**.*y* variables to your frame buffer position, but obviously this is not as flexible when it comes to writing that general sprite handling routine.

Now, as I found out the hard way, as a Macintosh user with no real TIM utilities this is something of a pain in the ass. As anyone who uses the Sony tool, TimUtil will know, you can set the image's position in the frame buffer from within the program, however as the only graphics tool us Mac owners have are the crap conversion things that MetroWerks give out on the CodeWarrior CD, the frame buffer position can only be set by hacking the TIM file with a hex editor, or by 'hard coding' the position - neither of which are very flexible. All I can suggest to fellow Mac users at the moment is to try and befriend someone with a PC who will let you run the Sony tools, or if you have a PC at work (like me luckily (?)), then you could use that.

Note: As of the time of creating this PDF document, Photoshop plugins are available on my web site for Macintosh users that will import and export TIM files directly.

Obviously you should set your TIM file to load the data into an area of the frame buffer that is not used by either of the buffers. So as in our source code example our screen resolution is 320x240, buffer 0 sits at 0,0 in the frame buffer, buffer 1 sits at 0,240 in the frame buffer, we will put our image data into 320,0. This will place it nicely alongside buffer 0. Imagine that graph paper again, now with the rectangle of the image added to it.

If you are creating 4 or 8 bit sprites then you also need to load the CLUT into the frame buffer. This is done the same way as loading the image data, by using the details got from *GsGetTimInfo()* that have been stored in the **GsIMAGE** structure. First set the **RECT** values from the CLUT members in the **GsIMAGE** structure and use then *LoadImage()* to put the CLUT into the frame buffer.

Once the image data is loaded you can then assign your sprite structure the relevant variables and pointers to your image data. This is done using something called the texture page.

Texture page addresses are another thing that seems chucked in to confuse you, but again it's really quite simple. Texture page addresses are a simple co-ordinate system within the frame buffer. All you need to do to find out the texture page address for your image, is to use the function *GetTPage().* Calling this function requires four parameters, first the type of image you are looking for where 0 = 4 bit, 1 = 8 bit and 2 = 16 bit, second is an attribute variable used for transparency which we are not bothered with so we set it to 0, third is the x position of the image in the frame buffer, and fourth is the y position of the image in the frame buffer. This routine will then return a value which you assign to your sprite structure tpage member. NB: The texture pages are defined in multiples of 64 in the x direction and multiples of 256 in the y direction.

Once you have you texture page address you are 90% of the way there, you have set up the graphics system, defined where the drawing buffers go in the frame buffer, loaded your image into the frame buffer, found your texture page address and assigned it to your sprite structure.

Now you set up the rest of your sprite structure, such as x (**GsSPRITE**.*x*) and y (**GsSPRITE**.*y*) position on the screen, sprite width (**GsSPRITE**.*w*) and height (**GsSPRITE**.*h*), the position of the CLUT in the frame buffer (**GsSPRITE**.*cx* & **GsSPRITE**.*cy*), scaling, brightness, and rotation. You can also set an offset to the texture page address using the members **GsSPRITE**.*u* for the x direction, and **GsSPRITE**.*v* for the y direction. Using these you can have multiple sprite images in one large TIM image and change the visible sprite by changing the offset values. However, the offset values can only be set from 0 ~ 255. By using this adjustment you can achieve frame animation of your sprite.

The most important member of the **GsSPRITE** structure is the **GsSPRITE**.*attribute* member. Using this member you set which type of sprite you wish to display (4, 8 or 16 bit), whether or not you want to scale it, rotate it, adjust the brightness etc. Check the reference manual for details on the relevant bits to set for what you wish to do, here we just want to display an 8 bit sprite so we will set it to **GsSPRITE**.*attribute = 0x01000000,* which tells the sprite that it is 8 bit, display on, brightness adjust on, rotation on, semi transparency off.

Once all of the above have been completed the sprite has been set up, and all that remains to get it on the screen is sorting out the ordering table registration and stuff.

Getting it on the screen

Now you have all of the relevant structures initialised, all you need to do is run through the main loop of the code that runs the OT routines. Remember, because of the double buffering technique used, you must run the program in a loop - at least twice - or you won't see anything on the screen.

At the start of your code you need to set the OT as a global like this:

```
// Ordering table variables
#define             OT_LENGTH       1

GsOT                WorldOT[2];
GsOT_TAG            OTTags[2][1<<OT_LENGTH];
```

This is ok if you only plan to use one OT. Don't worry about what it does right now, just that it works is enough.

After the initialisation stuff in the main program you should set up your OT, like this:

```
// Initialise the ordering table
  for (i=0;i<2;i++)
        {
        WorldOT[i].length = OT_LENGTH;
        WorldOT[i].org = OTTags[i];
        }
```

Now you should set up some kind of loop for the main program, such as checking the controller pad for a button being pressed, and execute the following commands.

First use an integer variable to find out which of the buffers is currently he one being drawn on, and which is the display buffer. This is achieved by using the function *activeBuff = GsGetActiveBuff()*. The current active buffer must be identified so that you know which one you need to draw to.

Then set the packet work area to the current one using *GsSetWorkBase()*. Again, necessary for the correct data to be worked upon. Clear all the entries from the OT using *GsClearOT()* ready for new entries to be registered. Then register all your entries into the OT. With sprites you can use two different functions, *GsSortFastSprite()*, or *GsSortSprite()*. The difference between the two is that obviously *GsSortFastSprite()* is faster, but you pay for the extra speed by not being able to rotate, or scale sprites. If you wish to carry out any of these features then you need to use *GsSortSprite()*.

Once the items are registered, and in our sample code we only have one sprite so we only need one entry, we call *DrawSync(0)*, which holds the program until all the background work

is carried out. When *DrawSync(0)* releases the program, it waits until a vertical retrace by making a call to *VSync(0)*. This returns when the CRT scan is returning back to the top of the screen, and allows you to switch your buffers by using *GsSwapDispBuff()*, thereby giving you a flicker free change over. With that done, all that needs to be carried out now is to register a sort OT command using *GsSortClear()*, which doesn't execute until a *GsDrawOt()* function has been called. Once the OT has been sorted, it's redrawn to the offscreen buffer ready for the loop back around and the buffer switch.

That's all there is to it. One extra point I should note is that, the frame rate you will achieve will depend on how many objects are registered in the OT. This is due to the fact that we have told the program to wait (*DrawSync(0)*) until all background activity is completed (namely drawing), before switching buffers, thereby several V-Blanks may pass before the switching of your buffers.

Source Code

```
//
// A basic 8 bit sprite viewer
//
// Ira Rainey 2/6/97
//
// ira.rainey@btinternet.com
//

// Include the necesary header files
#include <libps.h>
#include "pad.h"

// Total number of sprites plus one
#define     SPRITE_CNT  (1+1)

// Screen size detail
#define     SCREEN_WIDTH   320
#define     SCREEN_HEIGHT  240

// Tim address
// Load your TIM file here
#define     spriteAdd   0x80100000

void InitSprite ();
static u_long PadRead(long);
static long CheckPad ();

// Set ordering table length
#define OT_LENGTH 1

// Ordering table variables
GsOT        WorldOT[2];
GsOT_TAG    OTTags[2][1<<OT_LENGTH];
int            activeBuff;

// Primitive related vaiables
PACKET         GpuPacketArea[2][SPRITE_CNT*sizeof(GsSPRITE)];

// Sprite structure pointer
GsSPRITE    TestSprite;

// Controller related variables
volatile u_char   *bb0, *bb1;

// Main program section

void main ()
   {
   // Working variable
   int   i;

   // Initialise the drawing system
   GsInitGraph(SCREEN_WIDTH,SCREEN_HEIGHT,4,0,0);
   GsDefDispBuff(0,0,0,SCREEN_HEIGHT);
```

```
   // Initialise the ordering table
   for (i=0;i<2;i++)
      {
      WorldOT[i].length = OT_LENGTH;
      WorldOT[i].org = OTTags[i];
      }

   // Initialise the sprite
   InitSprite();

   // Get controller
      GetPadBuf(&bb0,&bb1);

   // Keep playing until select pressed
   while ((CheckPad()) !=99)
      {
      // Find out which buffer is being used
      activeBuff = GsGetActiveBuff();

      // Packet area
      GsSetWorkBase((PACKET *)GpuPacketArea[activeBuff]);

      // Clear the ordering table
      GsClearOt(0,0,&WorldOT[activeBuff]);

      // Register sprite into ordering table
      // Uses GsSortFastSprite for speed - no rotation/scaling needed
      GsSortFastSprite(&TestSprite,&WorldOT[activeBuff],0);

      // Wait for all drawing to finish
      DrawSync(0);

      // Wait for vertical blank interrupt
      VSync(0);

      // Switch display buffer & offscreen buffer
      GsSwapDispBuff();

      // Sort the ordering table
      GsSortClear(0,0,0,&WorldOT[activeBuff]);

      // Draw the objects registered in the Ordering table
      GsDrawOt(&WorldOT[activeBuff]);
      }

   // Finished
   return;
   }

//
// Sprite initialisation routine
//

void InitSprite()
   {
   RECT     rect;                     // Define RECT structure
   GsIMAGE  tim_data;                    // Define GsIMAGE structure
```

```
   // Get the TIM file info - skip 4 bytes from start of file
   // to get to info we need
   GsGetTimInfo ((u_long *)(spriteAdd+4),&tim_data);

   // Load the image into the frame buffer
   rect.x = tim_data.px;                    // x pos in frame buffer
   rect.y = tim_data.py;                    // y pos in frame buffer
   rect.w = tim_data.pw;                    // width of image
   rect.h = tim_data.ph;                    // height of image
   LoadImage(&rect, tim_data.pixel);           // load data into frame buffer

   // Load the CLUT into the frame buffer
   rect.x = tim_data.cx;                    // x pos in frame buffer
   rect.y = tim_data.cy;                    // y pos in frame buffer
   rect.w = tim_data.cw;                    // width of CLUT
   rect.h = tim_data.ch;                    // height of CLUT
   LoadImage(&rect, tim_data.clut);         // load data into frame buffer

   // Initialise sprite structure
   TestSprite.attribute = 0x01000000;                // sprite attribute - 8 bit
   TestSprite.x = ((SCREEN_WIDTH/2)-(tim_data.pw));      // sprite x axis
position
   TestSprite.y = ((SCREEN_HEIGHT/2)-(tim_data.ph/2));// sprite y axis position

   // The width of an 8 bit sprite is half that of a 16 bit hence the *2
   TestSprite.w = (tim_data.pw*2);          // sprite width
   TestSprite.h = tim_data.ph;              // sprite height
   TestSprite.tpage = GetTPage(1,0, tim_data.px, tim_data.py);
   TestSprite.u = 0;                        // Texture page offset x axis
   TestSprite.v = 0;                        // Texture page offset y axis
   TestSprite.cx = tim_data.cx;             // CLUT x axis position
   TestSprite.cy = tim_data.cy;             // CLUT y axis position
   TestSprite.r = 128;                      // Regular brightness
   TestSprite.g = 128;                      // Regular brightness
   TestSprite.b = 128;                      // Regular brightness
   TestSprite.mx = 0;                       //
   TestSprite.my = 0;                       //
   TestSprite.scalex = ONE;                 // Defined as 4096
   TestSprite.scaley = ONE;                 // Defined as 4096
   TestSprite.rotate = 0;                   // No rotation

   // Wait until LoadImage() has finished before returning
   DrawSync(0);
   }

//
// Pad read routine
//

static u_long PadRead(long id)
   {
   return (~(*(bb0+3) | *(bb0+2) << 8 | *(bb1+3) << 16 | *(bb1+2) << 24));
   }

//
// Routine to check the pad status
//
```

```
static long CheckPad ()
   {
   u_long padd = PadRead(1);

   if(padd & PADselect) return(99);

   }
```