Bitmaps in 2D

Version 1.0, 2002/03/09 - Tony Saveski (dreamtime), t_saveski@yahoo.com

Introduction

Hi Again! I've decided to split this tutorial up into 4 separate parts. In total it will cover the following things:

* Tutorial #2a - Bitmaps in 2D.

* Tutorial #2b - Bitmaps in 2D...Really.

* Tutorial #2c - Animation in 2D.

* Tutorial #2d - Fonts and Text in 2D.

I'm doing this so I can release things to you quicker, and so I can provide snapshots of the code and demos in the state they're in at a certain point (before they change). I won't be able to spend a lot of time on this, but I hope I can still deliver these tutorials roughly one per week.

So let's get right to it. I recommend that you read and understand Tutorial #1 if you haven't done so already. At least get the code running on your own PS2.

Loading the Image Data

Before we can display an image, we need to get it into memory. I've chosen the BMP file format for these demos because it's so easy to process, and because I had some code already lying around that I wrote ages ago. Oh, and to make it even more simple, I'll only be using 24 bit BMP files (8 bits each for RGB).

The usual way to load an image file is to *open()* the file and then read the image data into a memory buffer as raw RGBA pixels. However, I can't do this on the PS2 yet because I have no standard C library (I haven't had a chance to download and play with *Gustavo's* implementation yet, but I hear it's great).

I'm also told that using *Naplink* and a certain PS2 BIOS syscall I should be able to open a file directly from my PC and load it via the USB cable! Again, I haven't had time to explore this yet because I'm spending too much time with real-life work and writing these tutorials in my spare time (If someone wants to pay me for doing this sort of thing, please send me an email at *t_saveski@yahoo.com* :-)

So, instead of loading the images dynamically at runtime, I've written a small utility to convert a BMP file into C array of the image data that will be linked with the executable file. This also makes it easier to distribute demos because everything is in a single file.

Here's a snippet of code generated by the tool:

```
// File Generated by BMP2C from resources/brick.bmp
#include "defines.h"

uint32 brick_w = 128;
uint32 brick_h = 128;

uint32 __attribute__((aligned(16))) brick[] = {
    // row 0
    0x00386a85,
    0x003f6c83,
    ...,
```

The image buffer is aligned to a 128 bit memory boundary, as needed by the DMAC, and each pixel is stored as a 32 bit RGBA value.

To produce a file with the above snippet of code, you would have called the tool like this:

```
C:\> bmp2c resources/brick.bmp brick > brick.c
```

The first parameter is the BMP filename; the second is the name you want for the array; and the output goes to STDOUT.

To have the C code automatically produced and re-compiled every time the BMP file is changed, make sure the *bmp2c* executable is in your path and add the following lines to the *Makefile*:

```
BMP2C=bmp2c
C_SRC = g2.c demo2a.c brick.c

%.c: $(BMP_DIR)/%.bmp
    @echo "--------------------------------------------------"
    $(BMP2C) $< $(*F) > $@

brick.c: $(BMP_DIR)/brick.bmp
```

Displaying the Image

Now that we have the image data available in memory (however you achieved it), it's time to write a routine to display it on the screen. At this stage you will need to already have initialised the graphics mode, probably using the *g2_init()* function from Tutorial #1.

Sending the image to the screen is a matter of setting a few GS registers (*BITBLTBUF*, *TRXPOS*, *TRXREG*, and *TRXDIR*) using a standard *GS Packet* transfer. This needs to be followed by a second DMA transfer of the image data.

Some New Registers

The registers mentioned above are new to us and need to be explained before we can write the image transfer code. If you look closely at the code with this tutorial, you will notice I've added the following convenience macros to the *gs.h* file:

```
//-----------------------------------------------------------------------
// BITBLTBUF Register - Setup Image Transfer Between EE and GS
//   SBP  - Source buffer address (Address/64)
//   SBW  - Source buffer width (Pixels/64)
//   SPSM - Source pixel format (0 = 32bit RGBA)
//   DBP  - Destination buffer address (Address/64)
//   DBW  - Destination buffer width (Pixels/64)
//   DPSM - Destination pixel format (0 = 32bit RGBA)
//
// - When transferring from EE to GS, only the Destination fields
//   need to be set. (Source fields for GS->EE, and All for GS->GS).
//-----------------------------------------------------------------------
```

```
#define GS_BITBLTBUF(SBP,SBW,SPSM,DBP,DBW,DPSM) \
    (((uint64)(SBP)    << 0)        | \
     ((uint64)(SBW)    << 16)       | \
     ((uint64)(SPSM)   << 24)       | \
     ((uint64)(DBP)    << 32)       | \
     ((uint64)(DBW)    << 48)       | \
     ((uint64)(DPSM)   << 56))

//-----------------------------------------------------------------------
// TRXPOS Register - Setup Image Transfer Coordinates
//   SSAX - Source Upper Left X
//   SSAY - Source Upper Left Y
//   DSAX - Destination Upper Left X
//   DSAY - Destination Upper Left Y
//   DIR  - Pixel Transmission Order (00 = top left -> bottom right)
//
// - When transferring from EE to GS, only the Detonation fields
//   need to be set. (Source fields for GS->EE, and all for GS->GS).
//-----------------------------------------------------------------------
#define GS_TRXPOS(SSAX,SSAY,DSAX,DSAY,DIR)    \
    (((uint64)(SSAX)    << 0)        | \
     ((uint64)(SSAY)    << 16)       | \
     ((uint64)(DSAX)    << 32)       | \
     ((uint64)(DSAY)    << 48)       | \
     ((uint64)(DIR)     << 59))

//-----------------------------------------------------------------------
// TRXREG Register - Setup Image Transfer Size
//   RRW - Image Width
//   RRH - Image Height
//-----------------------------------------------------------------------
#define GS_TRXREG(RRW,RRH)    \
    (((uint64)(RRW)    << 0)        | \
     ((uint64)(RRH)    << 32))

//-----------------------------------------------------------------------
// TRXDIR Register - Set Image Transfer Direction, and Start Transfer
//   XDIR - (0=EE->GS, 1=GS->EE, 2=GS-GS.
//-----------------------------------------------------------------------
#define XDIR_EE_GS            0
#define XDIR_GS_EE            1
#define XDIR_GS_GS            2
#define XDIR_DEACTIVATE       3

#define GS_TRXDIR(XDIR)    \
    ((uint64)(XDIR))
```

Now look for the *gs_put_image()* function in the *gs.c* file to see how to set these registers up. There is nothing new here from how we set up GS Packets in Tutorial #1. Note however that I have set the frame buffer address to 0 because we are continuing on from Tutorial #1 and still only have the one frame buffer located right at the start of the GS memory.

```
    BEGIN_GS_PACKET(gs_dma_buf);
    GIF_TAG_AD(gs_dma_buf, 4, 1, 0, 0, 0);
    GIF_DATA_AD(gs_dma_buf, bitbltbuf,
        GS_BITBLTBUF(0, 0, 0,
            0,                      // frame buffer address
            (g2_max_x+1)/64,        // frame buffer width
            0));                    // pixel format = 32bit
    GIF_DATA_AD(gs_dma_buf, trxpos,
        GS_TRXPOS(
            0,
            0,
            x,
            y,
            0));    // top left -> bottom right
    GIF_DATA_AD(gs_dma_buf, trxreg, GS_TRXREG(w, h));
    GIF_DATA_AD(gs_dma_buf, trxdir, GS_TRXDIR(XDIR_EE_GS));
    SEND_GS_PACKET(gs_dma_buf);
```

Next, we need to tell the GIF how much image data there is to transfer. The *DIF_TAG_IMG()* macro is slightly different to the A+D GIF macro (see *gif.h*). It tells the DMA and GIF that what follows is IMAGE data and how much there is.

```
BEGIN_GS_PACKET(gs_dma_buf);
GIF_TAG_IMG(gs_dma_buf, w*h/4);
SEND_GS_PACKET(gs_dma_buf);
```

We can now send the actual image data with a standard DMA transfer on the GIF channel.

```
SET_QWC(GIF_QWC, w*h/4);
SET_MADR(GIF_MADR, data, 0);
SET_CHCR(GIF_CHCR, 1, 0, 0, 0, 0, 1, 0);
DMA_WAIT(GIF_CHCR);
```

The Demo

At this stage I created a small demo that put the image at random places on the screen and found that I couldn't see a bloody thing because it was so fast! I really wasn't expecting this machine to push data around this quickly.

I though a more interesting 'demo' would be to load a full-screen image and then do something. After spending more time looking for a 'cool-enough' image than writing the code, I settled on what is now loaded with *demo2a.elf*. Hope you like it.

Loading the large image (640x256x32) identified a bit of a bug in my code. It turns out that the PS2 can't transfer the entire buffer in one go, so only something like the top third of the image was being displayed. Scanning through the *funslower* demo source, I discovered that they load 128x128 qwords at a time.

I added some code to break up the transfer and everything was fine. It seems like the DMAC can transfer 'only' 16,384 qwords of IMAGE data per transfer.

Conclusion

We've successfully loaded an image and displayed it onto the screen! I just have to say again that I'm still impressed with how fast this box is (although, the last time I did any serious graphics programming was with Watcom C++ / MS-DOS on a 486, so maybe I'm easily impressed). There are a few things wrong with the method I used in this tutorial:

* There's no obvious way to handle transparent regions in the bitmap.

* It can't handle negative coordinates. The image just 'wraps'.

* The bitmap doesn't get clipped at all, even when you have setup a clipping region with the SCISSOR registers.

Don't despair. This code will still be of use to us to move data around between the EE and GS when we tuck it away with the generic GS routines in the *gs.** set of files. Next week, I'll have to come up with a better way of drawing bitmaps, this time with proper clipping and support for transparent bits.

Finally, even though I've used their source code over the last week as a reference while writing these tutorials, I finally loaded and ran the *funslower* demo by *SoopaDoopa* for the first time...and was completely blown away! If you haven't downloaded this demo yet make sure to get it from *http://ps2dev.LiveMedia.com.au* right now. It shouldn't be missed.