

Bitmaps in 2D...Really!

Version 1.0, 2002/03/13 - Tony Saveski (dreamtime), t_saveski@yahoo.com

Introduction

Our mission in this tutorial is to develop a 'proper' 2D *put_image()* function, supporting transparent bitmap regions and proper clipping when negative co-ordinates are passed. Some things we will cover here are:

- * How to appropriately load a bitmap into the GS memory (using the routine developed in *Tutorial #2a*).
- * How to display this bitmap using the *SPRITE* primitive and UV texture co-ordinates.
- * Define a mapping between the *Primitive* co-ordinate system and *Window* co-ordinate system to allow the image to be placed at a 'negative' location and properly clipped against the top and left of the screen.
- * Using the fast alpha test to allow transparent bitmap pixels.

There's quite a bit of functionality we are gaining here without a great deal of effort, as you will soon see. This tutorial will also help you better understand how the Playstation2 really works, so pay attention! :-)

The 'New Method'

In *Tutorial #2a* I developed a *put_image()* function that loaded the image directly into the display buffer. I mentioned a number of reasons why this is not ideal, and that in this tutorial I would be presenting a better method of displaying a 2D bitmap.

This new method consists of two steps:

- * Load the bitmap into a 'hidden' part of GS memory.
- * Display the bitmap on the screen by turning on texture mapping, drawing a *SPRITE* primitive, and specifying appropriate texture co-ordinates for each vertex.

Note that we still do not need a Z-buffer for anything we are doing, so why waste the memory by creating one?!

Loading the Bitmap

We can use a modified version of the old *put_image()* function to transfer the bitmap into GS memory. It can be found in *gs.c* as:

```
void gs_load_texture(uint16 x, uint16 y, uint16 w, uint16 h,
                    uint32 data_adr, uint32 dest_adr, uint16 dest_w);
```

Some important things to note are:

* The destination buffer has to be aligned on a 256 byte boundary (I prefer to think in terms of bytes instead of words, dwords, and qwords...especially when dealing with memory addresses!). You don't have to 'allocate' this buffer, just make sure you're not using the space for anything else, such as a frame or z-buffer.

* The last parameter is the 'width' of this buffer you're loading the image into, and it has to be a multiple of 64 pixels (remember the *BITBLTBUF* register fields!). You shouldn't think of it as a linear address space of words or bytes, but instead think of it as a 2D array, or 'screen', were you specify things in *X* and *Y* pixel locations and *W* and *H* pixel distances. And that's exactly what effect the *X* and *Y* parameters above have: they will load the image at that 'screen position' in the texture buffer. *W* and *H* are the width and height of the bitmap.

Displaying the Bitmap

This step is now a bit different to what we did previously, and I'll have to introduce some new registers. The other registers used have been described in a previous tutorial.

```
#define texflush    0x3f    // Tickle before using newly loaded texture
#define tex0_1     0x06    // Texture Buffer Setup (Context 1)
#define tex0_2     0x07    // Texture Buffer Setup (Context 2)
#define uv         0x03    // Specify Vertex Texture Coordinates

//-----
// TEX0_x Register - Set Texture Buffer Information
//   TBP0 - Texture Buffer Base Pointer (Address/256)
//   TBW  - Texture Buffer Width (Texels/64)
//   PSM  - Pixel Storage Format (0 = 32bit RGBA)
//   TW   - Texture Width (Width = 2^TW)
//   TH   - Texture Height (Height = 2^TH)
//   TCC  - Tecture Color Component
//         0=RGB,
//         1=RGBA, use Alpha from TEXA reg when not in PSM
//   TFX  - Texture Function
//         (0=modulate, 1=decal, 2=hilight, 3=hilight2)
//-----
#define TEX_MODULATE  0
#define TEX_DECAL    1
#define TEX_HILIGHT  2
#define TEX_HILIGHT2 3

#define GS_TEX0(TBP0,TBW,PSM,TW,TH,TCC,TFX,CBP,CPSM,CSM,CSA,CLD) \
((uint64)(TBP0) << 0) | \
((uint64)(TBW) << 14) | \
((uint64)(PSM) << 20) | \
((uint64)(TW) << 26) | \
((uint64)(TH) << 30) | \
((uint64)(TCC) << 34) | \
((uint64)(TFX) << 35) | \
((uint64)(CBP) << 37) | \
((uint64)(CPSM) << 51) | \
((uint64)(CSM) << 55) | \
((uint64)(CSA) << 56) | \
((uint64)(CLD) << 61)

//-----
// UV Register - Specify Texture Coordinates
//-----
#define GS_UV(U,V) \
(((uint64)(U) << 0) | \
((uint64)(V) << 16))
```

And so the new `gs_put_image()` function looks like this:

```
//-----  
void gs_put_image(uint16 x, uint16 y, uint16 w, uint16 h, uint32 *data)  
{  
    // - Call this to copy the texture data from EE memory to GS memory.  
    // - The g2_texbuf_addr variable holds the byte address of the  
    // 'texture buffer' and is setup in g2_init() to be just after  
    // the frame buffer(s). When only the standard resolutions are  
    // used this buffer is guaranteed to be correctly aligned on 256  
    // bytes.  
    gs_load_texture(0, 0, w, h, (uint32)data, g2_texbuf_addr, g2_max_x+1);  
  
    BEGIN_GS_PACKET(gs_dma_buf);  
    GIF_TAG_AD(gs_dma_buf, 7, 1, 0, 0, 0);  
  
    // Access the TEXFLUSH register with anything  
    GIF_DATA_AD(gs_dma_buf, texflush, 0x42);  
  
    // Setup the Texture Buffer register. Note the width and height! They  
    // must both be a power of 2.  
    GIF_DATA_AD(gs_dma_buf, tex0_1,  
        GS_TEX0(  
            g2_texbuf_addr/256,    // base pointer  
            (g2_max_x+1)/64,      // width  
            0,                    // 32bit RGBA  
            gs_texture_wh(w),     // width  
            gs_texture_wh(h),     // height  
            1,                    // RGBA  
            TEX_DECAL,            // just overwrite existing pixels  
            0,0,0,0,0));  
  
    GIF_DATA_AD(gs_dma_buf, prim,  
        GS_PRIM(PRIM_SPRITE,  
            0,                    // flat shading  
            1,                    // texture mapping ON  
            0, 0, 0,              // no fog, alpha, or antialiasing  
            1,                    // use UV register for coordinates.  
            0,  
            0));  
  
    // Texture and vertex coordinates are specified consistently, with  
    // the last four bits being the decimal part (always .0 here).  
  
    // Top/Left, texture coordinates (0, 0).  
    GIF_DATA_AD(gs_dma_buf, uv,    GS_UV(0, 0));  
    GIF_DATA_AD(gs_dma_buf, xyz2,  GS_XYZ2(x<<4, y<<4, 0));  
  
    // Bottom/Right, texture coordinates (w, h).  
    GIF_DATA_AD(gs_dma_buf, uv,    GS_UV(w<<4, h<<4));  
    GIF_DATA_AD(gs_dma_buf, xyz2,  GS_XYZ2((x+w)<<4, (y+h)<<4, 0));  
  
    // Finally send the command buffer to the GIF.  
    SEND_GS_PACKET(gs_dma_buf);  
}
```

Note that I'm always loading the bitmap to position (0, 0) in the texture buffer. This isn't the best thing to be doing, but for now I accept simplicity over performance. I will probably change it later to allow you to load multiple bitmaps into different parts of the texture buffer so they only need to be loaded once and used many times. You will then only need to get the texture co-ordinates correct.

Also, the more observant of you would have noticed some mistakes in the comments to my address field assignments in the previous tutorials (ie. divide by 64 instead of 256 for texture buffers, and 2048 instead of 8192 for *FRAME* and *DISPLAY*). I'm just lucky that everything was being loaded at address 0! Oh well, that's what you get for thinking in bytes instead of words. Nice of you all to totally ignore it and not mention it to me :-)

I nearly forgot to mention something important: In order for the fields of the *PRIM* register to be used, you have to write a 1 to the *PRMODECONT* register (I've added it to *g2_init()*). This tells the GS to use the primitive drawing attributes specified in the *PRIM* register, instead of the ones in the *PRMODE* register (which is the default). My guess is that it is more efficient to change drawing attributes using the *PRMODE* register, *if you don't want to change the primitive type currently in use* (like when drawing millions of triangles with different colors and textures). Something to remember for a later tutorial.

Clipping

There are some problems with the *G2* library and clipping with the way things are at the moment:

- * All primitive co-ordinates are unsigned integers. So how the heck can you put an image past the left and top of the screen (ie. at a negative location) and have it clipped appropriately against the top and left of the screen?

- * Even if we fixed the above problem, the negative numbers of the top/left become very large unsigned positive numbers and the bitmap gets drawn with what should be the bottom/right as the top/left, a very large bottom/right, and a texture that is very stretched.

However, I've come up with a "cunning plan" for allowing you to specify negative primitive co-ordinates.

I'm going to 'move' the Origin of the Window co-ordinate system to (1024.0, 1024.0) of the Primitive co-ordinate system (by setting the *XYOFFSET* register in *g2_init()*), and add this amount to all co-ordinates passed to the primitive drawing functions in the *G2* library (ie. *g2_line()*, *g2_put_image()*, ...). In this way we can avoid 'really' passing negative Primitive co-ordinates to the drawing routines.

I chose 1024 as the offset amount because this is the maximum size allowed by the GS for a single texture. I should probably be 'centering' the Window in the Primitive space (4095 x 4095) instead, or moving it to the bottom/right, depending on the mode and how the GS behaves (ie. I haven't tried this yet). I'd draw a picture of it all for you if I could be bothered.

A good way to think of it is that the Window co-ordinate system is a 'viewport' into the Primitive co-ordinate system, with the two origins offset by some amount. By adding this offset to any primitive so-ordinates, we are forcing them into the Window, but still keeping them within the valid Primitive area.

Alpha Transparency

One last feature I want to add to the *put_image()* function is support for transparent regions. One way to achieve this is by taking advantage of the Alpha Test in the pixel pipeline (not to be confused with the *Alpha Blending* stage later in the pipeline).

After performing *Texture Mapping*, *Fogging*, and *Antialiasing*, the GS does a number of *Pixel Tests* (*Scissoring*, *Alpha Test*, *Destination Alpha Test*, and *Depth Test*). The *Scissoring Test* determines whether a pixel falls within the Window viewable area defined by the *SCISSOR* register. If this test is passed, then the GS performs an *Alpha Test* on the pixel as defined by the *TEST* register of the current context, to determine what to do next.

Here are the fields of the *TEST* registers:

```
//-----  
// TEST_x Register - Pixel Test Settings  
//   ATE   - Alpha Test (0=off, 1=on)  
//   ATST  - Alpha Test Method  
//           0=NEVER: All pixels fail.  
//           1=ALWAYS: All pixels pass.  
//           2=LESS:  Pixels with A less than AREF pass.  
//           3=LEQUAL, 4=EQUAL, 5=GEQUAL, 6=GREATER, 7=NOTEQUAL  
//   AREF  - Alpha value compared to.  
//   AFAIL - What to do when a pixel fails a test.  
//           0=KEEP:   Don't update anything.  
//           1=FBONLY: Update frame buffer only.  
//           2=ZBONLY: Update z-buffer only.  
//           3=RGBONLY: Update only the frame buffer RGB.  
//   DATE  - Destination Alpha Test (0=off, 1=on)  
//   DATM  - DAT Mode (0=pass pixels whose destination alpha is 0)  
//   ZTE   - Depth Test (0=off, 1=on)  
//   ZTST  - Depth Test Method.  
//           0=NEVER, 1=ALWAYS, 2=GEQUAL, 3=GREATER  
//-----  
#define ATST_NEVER      0  
#define ATST_ALWAYS    1  
#define ATST_LESS      2  
#define ATST_LEQUAL    3  
#define ATST_EQUAL     4  
#define ATST_GEQUAL    5  
#define ATST_GREATER   6  
#define ATST_NOTEQUAL  7  
  
#define AFAIL_KEEP     0  
#define AFAIL_FBONLY  1  
#define AFAIL_ZBONLY  2  
#define AFAIL_RGBONLY 3  
  
#define ZTST_NEVER     0  
#define ZTST_ALWAYS   1  
#define ZTST_GEQUAL   2  
#define ZTST_GREATER  3  
  
#define GS_TEST(ATE,ATST,AREF,AFAIL,DATE,DATM,ZTE,ZTST) \  
(((uint64)(ATE)   << 0) | \  
 ((uint64)(ATST)  << 1) | \  
 ((uint64)(AREF)  << 4) | \  
 ((uint64)(AFAIL) << 12) | \  
 ((uint64)(DATE)  << 14) | \  
 ((uint64)(DATM)  << 15) | \  
 ((uint64)(ZTE)   << 16) | \  
 ((uint64)(ZTST)  << 17))
```

I want to tell the GS not to draw any pixels with an Alpha value equal to zero. To achieve this, I have added a GS command in the *g2_init()* functions as follows, and updated the BMP2C tool to output an Alpha value of zero for any pitch-black pixels (0, 0, 0).

```
// Setup TEST_1 register to allow transparent texture regions where A=0  
GIF_DATA_AD(gs_dma_buf, test_1,  
  GS_TEST(  
    1, // Alpha Test ON  
    ATST_NOTEQUAL, 0x00 // Reject pixels with A=0  
    AFAIL_KEEP, // Don't update frame or Z buffers  
    0, 0, 0, 0)); // No Destination Alpha or Z-Buffer Tests
```

You might be wondering why I'm 'wasting' a whole byte of GS memory per pixel per frame buffer when there are probably other ways to do transparent bitmaps only using 24 bits per pixel. The reason is that in a future tutorial I want to experiment a little with the Alpha Blending stage of the pipeline. If I can't think of anything interesting to do with the Alpha byte of each pixel I will change everything to 24 bit then.

I've actually written all of this text without testing any of the code. I hope it works! Let's find out by doing a simple demo.

The Demo

The background image is from <http://www.gameart.org> and is called "*Hover Tank Rush*" by *Zman*, and holds a hidden message that I will reveal in *Tutorial #2d* when fonts are working. (No it's not in the blurry text at the bottom of the screen...nor is it something you will magically see by applying a FFT to the image).

Those of you who are not totally blind from decades of programming, or umm... something else, will notice how crap the animation in this demo is. I did this on purpose so I could smoothly lead into the next tutorial, which is all about *Smooth Animation in 2D using Double Buffering*, plus some other miscellaneous goodies. I will use the same demo in the next tutorial, but with double buffering. We should see a huge difference!

Finally, I'd like to thank the *very few* of you who have emailed me to say HI, and to let me know that you're reading the tutorials and enjoying them. I really would like to hear from more of you with comments, queries, ideas, and especially criticisms. Or even just to say HI! It will make my day and keep me motivated to do more tutorials. So drop me an email at t_saveski@yahoo.com.

See you next week (or the week after that depending on free time).