

Fonts and Text in 2D

Version 1.0, 2002/03/23 - Tony Saveski (dreamtime), t_saveski@yahoo.com

Introduction

Finally, *Tutorial #2d*. I thought I'd never get here. After this tutorial, we will have a fairly complete 2D graphics library for the PlayStation 2! This tutorial is all about '*bitmapped*' fonts and printing text to the screen. I will not be covering anything about *True Type* or '*Vector*' fonts, except to suggest that someone else do it :-)

Most of the PS2 graphics code we will need has already been developed in the previous few tutorials, making fonts and text a fairly easy, but *tedious* feature to add. The main topics I want to cover here are:

- * Creating a Font Bitmap File.
- * Loading a Font.
- * Writing Text with a Font.
- * Making the Text Look Better.

I'll finish off with some ideas for projects that can be done using the library developed throughout these tutorials, and tell you what I'll be up to next.

Creating the Font

The first thing we need to do is create an image that contains all the characters we want to use when printing text to the screen. I've decided to make these bitmaps 256x128 in size, as we'll see later when I talk about the code.

There are a number of ways to create the bitmap. My first attempt was with *Photoshop* and can be found in the *resources/verdana.bmp* file. To create something like this, you need to do the following.

- * Create a new 256x128 pixels RGB image, and give it a black background (0, 0, 0).
- * Select the Text tool; Select the foreground color you want, making sure background is Black; Pick a font; then just go ahead and type out every character you want, making sure there's a bit of space between the individual letters.
- * Make sure antialiasing and blending are turned off because although it looks much better in *Photoshop* when blended with a static background, it can look absolutely awful in your own code where you use a different background.
- * When you're happy with the image and the characters you need, flatten the *Photoshop* image and save it as a 24-bit BMP. This can then be converted to C code using the *BMP2C* tool, just like in the other tutorials.

Loading the Font

We need to load this font into the GS memory before we can use it. This can be done using the `gs_load_texture()` function we already have, but we need to pick a memory address to load it to first. Looking at the `g2_init()` function, you can see that I've now put a single 256x128 'font buffer' just after the two frame buffers, and before the full-screen-sized texture buffer.

```
// Create a single 256x128 font buffer
g2_fontbuf_addr = gs_mem_current;
gs_mem_current += g2_fontbuf_w * g2_fontbuf_h * (v->bpp/8);
```

Now that we know where we're loading it, we can write the following function:

```
void g2_set_font(uint32 *data, uint16 w, uint16 h, uint16 *tc)
{
    g2_font_tc = tc;
    gs_load_texture(0, 0, w, h, (uint32)data,
                  g2_fontbuf_addr, g2_fontbuf_w);
}
```

What's this `g2_font_tc` thing, I hear you ask? It's an array of 128x4 texture co-ordinates: one rectangle is defined for each of the first 128 characters of the *ASCII* character set. For each character, we need to define the `x0`, `y0`, `x1`, and `y1` co-ordinates (in that order) of the rectangle enclosing that character; or the co-ordinates of a single black pixel for any characters not in our bitmap; or a nicely sized black rectangle for the 'space' character (32). I had to create the `verdana_tc.c` file manually:

```
uint16 verdana_tc[] = {
    0,    0,    1,    1,    // 000 - not used
    0,    0,    1,    1,    // 001 - not used
    ...
    0,    0,    16,   18,   // 065 - A
    20,   0,    32,   18,   // 066 - B
    36,   0,    48,   18,   // 067 - C
    ...
    0,    0,    1,    1,    // 127 - not used
};
```

Writing Text to the Screen

To write text to the screen, we need to loop through each letter in the text and draw a sprite textured with the font bitmap, using the texture co-ordinates of the character being drawn. The image code is the same as the `g2_put_image()` function, but I've duplicated it because I might want to do something a bit differently for text in the future.

```
void g2_out_text(int16 x, int16 y, char *str)
{
    char    c;                // current character
    uint16  *tc;              // current texture coordinates [4]
    uint16  x0, y0, x1, y1;  // rectangle for current character
    uint16  w, h;            // width and height of above rectangle

    x += gs_origin_x;
    y += gs_origin_y;

    c = *str;

    while(c)
    {
        // Read the texture coordinates for current character
        tc = &g2_font_tc[c*4];
        x0 = *tc++;
        y0 = *tc++;
        x1 = *tc++;
        y1 = *tc++;
        w = x1-x0+1;
        h = y1-y0+1;
    }
}
```

```

// Draw a sprite with current character mapped onto it
BEGIN_GS_PACKET(gs_dma_buf);
GIF_TAG_AD(gs_dma_buf, 6, 1, 0, 0, 0);

GIF_DATA_AD(gs_dma_buf, tex0_1,
GS_TEX0(
    g2_fontbuf_addr/256,          // base pointer
    (g2_fontbuf_w)/64,          // width
    0,                            // 32bit RGBA
    gs_texture_wh(g2_fontbuf_w), // width
    gs_texture_wh(g2_fontbuf_w), // 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));

GIF_DATA_AD(gs_dma_buf, uv, GS_UV(x0<<4, y0<<4));
GIF_DATA_AD(gs_dma_buf, xyz2, GS_XYZ2(x<<4, y<<4, 0));
GIF_DATA_AD(gs_dma_buf, uv, GS_UV((x1+1)<<4, (y1+1)<<4));
GIF_DATA_AD(gs_dma_buf, xyz2, GS_XYZ2(
    (x+w*g2_font_mag)<<4, (y+h*g2_font_mag)<<4, 0));

SEND_GS_PACKET(gs_dma_buf);

// advance drawing position
x += (w + g2_font_spacing) * g2_font_mag;

// get next character
str++;
c = *str;
}
}

```

Better Fonts Faster

If you run the demo now, you should see that the first part of the first message looks OK. It uses only capital letters that are nicely aligned.

But in the rest of the message, which uses lower and upper case letters, things are slightly out of alignment. Originally I thought it was caused by bad texture co-ordinates, but I've actually traced the problem back to the way *Photoshop* outputs text. It seems that for each row of text output, *Photoshop* may draw the text with a slightly greater height than other lines, depending on what letters you've used on that row. Even though the height is only one or two pixels different, it is enough to make my font look pretty crap when used. If anyone knows how to stop *Photoshop* from doing this, can you please tell me!

One way to fix it is to make sure that each row of text has the letter 'j' in it, which almost always seems to have full height and forces all rows to have consistent height. But this is a complete hack and a waste of GS memory that I'd rather not have. I also want a faster way of generating the texture co-ordinate file because doing it by hand for the *verdana* font took way too long.

A bit of a search on the Web, and I discovered a nifty little tool called "*Bitmap Font Builder*", by *Thom Wetzel Jr*, which can be downloaded from <http://www.lmnopc.com>. All you need to do is choose your font, set some properties, select 256x256 as the bitmap size, and you magically have a bitmap that can be used on the PS2! The program actually draws the characters for two separate fonts on the same bitmap, so I crop it to 256x128 using *Photoshop*.

Next, we need to create the texture co-ordinates source file. This is easy, because Thom's program will evenly space every single character it prints. So, knowing that the bitmap is 256x128 and there are 16x8 characters, calculating the texture co-ordinates is a no-brainer. I've created a file called *fixed_tc.c* that can be used with any texture created with *Bitmap Font Builder*. You will only need to experiment with character spacing to make it look nice (*HINT: sometimes, negative spacing is best!*).

The above is OK if you're happy with a *'fixed-width'* font (which is good for fonts like *courier-new*), but what do we need to do if we want each letter to only be as wide as it needs to be, giving us a *'variable-width'* font? The simplest thing I can think of is to tell Thom's tool to left-justify all the characters and then copy and modify the existing co-ordinates file, changing the *x1* co-ordinate for each character to make it as wide as needed (in fact, the tool can export a file with all the widths for you). If someone writes a tool to automatically do this (hint!), please tell me :-)

I've created fixed-width *courier-new* and *tahoma* fonts for this tutorial. Throw out the *verdana* that I did manually with *Photoshop*, and check out the demo to see how these new ones look.

Changing Font Color

The simplest way to change the color of a font is to create the bitmap with that color in the first place! But I know you all want some way of changing the same font at runtime.

Something I don't want is to go to a CLUT-based pixel format because I want to allow full 32-bit textured fonts, and I don't want to support two different font types for now.

The most obvious way is to modify the bitmap in EE memory, setting each 'non-black' pixel to the new color, and then load this new font into the GS memory. I think this method would be acceptable and not too slow (as long as you don't do it for every frame!), but I want to try something a bit more interesting, maybe using the GS hardware.

Another (probably silly) way to implement font colors is to: Set the *FBMSK* field (Frame Buffer Mask) of the *FRAME_x* register to only allow the color you want through; Draw your text in white; Reset the *FBMASK* to allow everything again. I tried this out and it can probably be made to work.

I also thought about using the *Alpha Blending* stage of the GS to make it draw an arbitrary fixed color onto the frame buffer wherever the alpha or color is non-zero. I couldn't see a way to do this in the short amount of time I thought about it, but if anyone more creative than me comes up with a way to do this, please tell me!

What I would actually do is to draw a great big dirty colored rectangle over the font bitmap directly in the GS memory. I'd treat the *font-buffer* as a *frame-buffer*, and draw the rectangle telling the GS to not draw over any pixels where the target alpha is zero. This way, only the used font pixels get changed to the new color. I've left this as an exercise for the reader. (*Note: This has nothing to do with the fact that there's some silly bug in my code that I can't find, that's stopping me from finishing the code and releasing this tutorial earlier ;-)*

This all seems like a lot of effort, considering the only thing that looks decent on my monitor is a White Font on a Blue Background. There must be a good reason why Square used this combination in the Final Fantasy games. Use good HCI theory when picking text and background colors, and be considerate to color-blind people like me (blue/green)!

Blending and Antialiasing

Although I've only shown you boring white fonts so far, there's not reason why we can't do chunky, pretty, textured fonts like you see in most demos and games. In order to do this, I have to go back to *Photoshop*. I won't explain how I created these fonts (*yes I DID do them myself, I'm not a TOTAL spastic :-)*), but I have included the PSD's in the `./resources` directory. You should be able to work it out and modify them easily.

The first one I did was a simple font, filled with a gradient, on a black background, with no antialiasing or blending. This suffered from a shocking case of the jaggies and looked pretty bad on the screen.

So I thought I'd let *Photoshop* do some antialiasing. This looked great in Photoshop, but all the pixels around the edges were blended with the black background. When I dumped some text onto a different background, it looked even worse than before.

You can find the next font I created as `./resources/textfont.psd`. I took the first font from before, and added a black border around it (*color [1,1,1] actually, so it doesn't get ignored during the first Alpha Test on the PS2*). I let *Photoshop* antialias inside the black outline, but not on the outside. You can see in the demo that this looks OK when it's small (*although still a little jaggy*), but pretty bad when the font is drawn using a bigger size.

The only thing left was to use the Alpha Blending stage of the PS2. Before I could do this, I needed some way of getting correct alpha values into the alpha channel of the font bitmap. To do this we need an enhanced version of the *BMP2C* utility I built in a previous tutorial (remember it currently only outputs 0x00 or 0x80 in the Alpha Channel).

I quickly modified it to accept a third command-line parameter, which is another bitmap file to be used as the Alpha channel (*Make sure to overwrite your old one with this new-and-improved one*). The easiest way to create the Alpha bitmap is to:

- * Take the font PSD file;
- * Hide the gradient or texture layer so you are left with the white font antialiased onto the black background;
- * Change the white text to color `0x808080` (because the valid Alpha range for the PS2 is 0x00-0x80);
- * Save it as a 24-bit BMP file.

It's that simple! You can find my sample font in `./resources/textfont2.psd`.

Now, check out the *Makefile* to see how to automate all the code generation and compiling, and run the demo to see the results.

To actually use this new Alpha channel, we need to make two small changes to the code. First we need to set up the `ALPHA_I` register correctly. The `ALPHA_I` register let's you set the variables of the Alpha Blending Function $Cout = (A-B) * C >> 7 + D$. Here are the register parameters:

```
//-----  
// ALPHA_x Registers - Setup Alpha Blending Parameters  
// Alpha Formula is: Cout = (A-B)*C>>7 + D  
// For A,B,D - (0=texture, 1=frame buffer, 2=0)  
// For C - (0=texture, 1=frame buffer, 2=use FIX field for Alpha)  
//-----  
#define GS_ALPHA(A,B,C,D, FIX) \  
(((uint64)(A) << 0) | \  
 ((uint64)(B) << 2) | \  
 ((uint64)(C) << 4) | \  
 ((uint64)(D) << 6) | \  
 ((uint64)(FIX) << 32))
```

For the effect I want with the colors I've used, we need to set:

```
A = color of the font pixel
B = color of the frame buffer pixel
C = alpha of the font pixel
D = color of the frame buffer pixel
```

I've put this code in the `g2_init()` function, so we only need to do it once:

```
// Setup the ALPHA_1 register to correctly blend edges of
// pre-antialiased fonts using Alpha Blending stage.
// The blending formula is
// PIXEL=(SRC-FRAME)*SRC_ALPHA>>7+FRAME
GIF_DATA_AD(gs_dma_buf, alpha_1,
GS_ALPHA(
    0,          // A - source
    1,          // B - frame buffer
    0,          // C - alpha from source
    1,          // D - frame buffer
    0));       // FIX - not needed
```

The last thing we need to do is tell the GS to draw the primitive with Alpha Blending, in `g2_out_text()`. Also, remember that the Alpha Test is still correctly rejecting 'pure' black pixels much earlier on in the pipeline.

```
GIF_DATA_AD(gs_dma_buf, prim,
GS_PRIM(PRIM_SPRITE,
    0,          // flat shading
    1,          // texture mapping ON
    0, 1, 0,    // no fog or antialiasing, but use alpha
    1,          // use UV register for coordinates
    0,
    0));
```

Conclusion

There are MANY different ways to achieve what I've done in this tutorial. For example, you could use a Color LookUp Table and different pixel format with the one-color font bitmaps, thus saving memory and giving you the ability to change colors just by modifying the CLUT (probably :-). Just keep in mind that I've usually gone for the simplest and most direct method in these tutorials and it may not be the ideal way in all circumstances. I also wanted to stick with a full 32-bit pixel format for now.

So that's the end of the (first set of?) 2D tutorials! Hopefully some of you will get inspired to develop a nice demo or game using the code. Just make sure to tell me when you've got something, because I want to see some creative things done in 2D :-). If someone can think of something more useful to do with the Alpha byte of each texture pixel than pre-antialiased fonts, please tell me!

I'll be making fixes and additions to the code I've developed so far, especially in consolidating the few different image and font functions that can be generalised a bit. Keep an eye out for updates on <http://www.livemedia.com.au> or watch the ps2dev@topica mailing list.

Ok, so where do we go from here? These are some potentially useful things I've thought of in the last five minutes:

- * Change the font primitive from Sprite to Triangles and allow text rotation about a point. This would be useful for doing cool rotating-text and spinning bitmap demos.

- * Make some nice bitmapped fonts to share with everyone. I'm hopeless with *Photoshop*, as you saw, so maybe some of you artist types can help out here. I'll create the texture co-ordinate file and alpha bitmap if you're too busy or lazy!

* Write a nice generic GUI Library with "Skins". I think this would be extremely nice to have, and doesn't require too much effort. Probably a good idea to port an existing one.

* 'Port' some old Amiga, PC, or even C64 demos to the PS2. Write tutorials about doing demo effects like Plasma, Fire, Particle systems, etc... HEAPS OF FUN! :-)

* Write a game or 'port' an existing 2D (or fake 3D) one!

* Port an emulator! Using the GUI library to select ROMS. Check out [Sjeeps](#) SMS Emulator for an example of what's possible! And I hear someone from [#ps2dev](#) is well on the way to having MAME ported across (the bastard beat me to it).

* Port a True Type font engine to PS2 :-) I know there are free ones out there.

Come on! Pull your fingers out and do something!! :-) Now there's no reason why we can't see a new demo or two EVERY WEEK from people. Also, I personally think that everyone should share ALL their code. Otherwise what's the point?!

I think the very next thing I'm going to do is a Simple 2D Game. It should only take a few days and be good for a laugh. After that, it's probably going to be more tutorials, either 2D/3D using the VU's, or Sound. Cast your votes now if you have a preference.

Again, ***keep those e-mails coming***, especially if you have some suggestions and ideas.

Cheers,

Tony (dreamtime)

t_saveski@yahoo.com.