

Animation in 2D

Version 1.0, 2002/03/16 - Tony Saveski (dreamtime), t_saveski@yahoo.com

Introduction

For this tutorial I promised to show you how to do *smooth* animation in 2D. But there's a small problem: It only takes about three lines of code to implement it. This sucks because it's not enough code to base a whole tutorial on, so I'm going to have to add some more topics to make things a bit more interesting. So here are the things I'll cover in this tutorial:

- * Smooth 2D animation using a double buffering technique.
- * Total beginners guide to Video standards.
- * How to synchronise page 'flips' with the Vertical Retrace.
- * Add some NTSC video mode support to the library.
- * How to automatically detect a PAL or NTSC system.

Ok, so it might not be such a short tutorial after all.

Smooth Animation

If you ran the demo from the last tutorial, you would have noticed the ugly 'flashing/flickering' and 'tearing' effects on the bitmaps while it was moving. You probably also noticed that the scrolling background also looked a bit 'jerky' as it was moving down the screen. There are a couple of things that contribute to this ugliness.

The main problem with the demo was the way everything was being drawn and moved on the same visible frame buffer. For example, to move the bitmap I'd: draw the background; draw the bitmap on top; draw the entire background again, over the bitmap; draw the bitmap at the new position; and so on. Because it is all happening right on the screen, our human visual system is able to perceive everything.

The other thing contributing to the tearing and jerkiness of the bitmaps is that the changes to the screen buffer due to animation are happening at different intervals and part way through the drawing of a frame. If we could somehow synchronize our drawing with the Vertical Retrace of the TV (when nothing is actually being drawn), the animation would look much cleaner.

Double Buffering

The first thing we need to do is 'allocate' a second frame buffer. To do this, I will just define the GS memory starting addresses of the two buffers, and store them in the `g2_frame_addr[2]` array. The size of the buffers is determined by the width, height and color depth of the selected video mode. This is all set up in `g2_init()`.

```
// Save address and advance GS memory pointer by buffer size (in bytes)
// Do this for both frame buffers.
g2_frame_addr[0] = gs_mem_current;
gs_mem_current += v->width * v->height * (v->bpp/8);

g2_frame_addr[1] = gs_mem_current;
gs_mem_current += v->width * v->height * (v->bpp/8);
```

While the first frame is being displayed, we can be drawing in the other. When we've finished drawing, we can tell the CRTC to display what's in the second buffer. While the second buffer is being displayed, we can draw again in the first frame. When finished with the drawing, we can tell the CRTC to flip back to the original frame (I think you get the idea).

This technique is called "*Double Buffering*". By only actually drawing to the currently 'off-screen' buffer, the eye can't see the bitmaps being drawn over each other. This has the effect of removing flickering and flashing during animation (but not tearing).

To implement this, remember from *Tutorial #1* that we control the GS output location using the `FRAME_1` register, and we tell the CRTC where do get its display data using the `DISPFB_2`. (Recall that I'm using the CRTC's Rectangular Area Read Circuit #2 only, hence the `DISPFB_2` and not `DISPFB_1`).

Now, to allow you to control the GS output address, and the CRTC's data-read address, we only need to create two simple functions that I won't even bother explaining:

```
void g2_set_visible_frame(uint8 frame)
{
    GS_SET_DISPFB2(
        g2_frame_addr[frame]/8192,    // Frame base pointer = Addr/8192
        cur_mode->width/64,           // Buffer Width (Pixels/64)
        cur_mode->psm,                // Pixel Storage Format
        0,                             // Upper Left X in Buffer = 0
        0                             // Upper Left Y in Buffer = 0
    );
    g2_visible_frame = frame;
}

void g2_set_active_frame(uint8 frame)
{
    BEGIN_GS_PACKET(gs_dma_buf);
    GIF_TAG_AD(gs_dma_buf, 1, 1, 0, 0, 0);
    GIF_DATA_AD(gs_dma_buf, frame_1,
        GS_FRAME(
            g2_frame_addr[frame]/8192,    // Frame base pointer = Addr/8192
            cur_mode->width/64,           // Frame buffer width (Pixels/64)
            cur_mode->psm,                // Pixel Storage Format
            0));
    SEND_GS_PACKET(gs_dma_buf);

    g2_active_frame = frame;
}
```

An alternative it is to set up the `FRAME_2` register from the start with the settings for the second frame buffer, and then just tell the `PRIM` register which context to use during drawing. This would also involve some conditional code to correctly set all the other registers for the appropriate context (or some smart register selection code using +0 and +1 register offsets). I can't say which would be more efficient without doing some tests, but I'll leave it like this because I might want to make use of the Context 2 register set when I start doing 3D anyway.

Video Standards

In order to understand something about this topic and a bit more about the mode setting code I developed in *Tutorial #1*, I decided to learn a little about Video standards and their differences. A short time of searching on the Web has yielded some interesting information, but remember that there's a ton of info about this that I don't know which I'm sure is important. So go out and research it (and tell me what you learn).

It turns out that the early TV-set designers decided to use the frequency of the Mains power supply to provide timing for the box. Because there are two main power standards in use in the World, there were immediately two different video standards: one operating at 60Hz (30 frames per second) and the other at 50Hz (25 frames per second). After the introduction of color and other small changes, we have ended up with the *NTSC* and *PAL* standards (among other less popular ones). (*Mental Note*: Don't use the vertical synch for any timing that will need to be consistent on all PS2s around the world).

Here are some technical details about the most popular video standards:

Name	Frame Rate	Field Rate	Scan Lines
NTSC	29.97	59.94	525
PAL	25	50	625
SECAM	25	50	625

The USA and Japan use *NTSC*, while most of Europe and Australia use *PAL*. The French have developed their own system, called *SECAM*, whose specs look very much like *PAL* to me. Can someone tell me what sort of PS2 they use in France and other *SECAM* countries (Russia?). Most of the differences between the standards seem to be related to the field frequency and in how they have 'implemented' support for color.

Why is all this important to us? There are two main reasons:

- * It's important to be able to write code that will run the same on the different systems around the world.
- * If we synchronise our drawing (or frame buffer 'flip') with the above frequencies, we can achieve 'tearing-free' and flicker-free animation.

Horizontal and Vertical Retrace

Drawing occurs Left-to-Right and Top-to-Bottom. As the electron gun is moving Left-to-Right, it is drawing the pixels for the current scan-line. When it finishes drawing the line, the gun is turned off while it is being moved back to the left to draw the next scan-line. This is known as the *Horizontal Retrace* or *Horizontal Blanking* interval.

The *Vertical Retrace* or *Vertical Blanking* interval is the part of the drawing process when the CRT has finished drawing an entire frame and is moving the gun, again turned off, from the bottom of the screen back to the top. This event will occur 25 times per second on a *PAL* system and just under 30 times a second on an *NTSC* system.

Even with Double Buffering in place, if we change the image being drawn part way through a frame, it will result in the tearing effect. However, if we limit these changes only to the times when a Vertical blanking interval is happening, we greatly reduce or eliminate this tearing effect.

Most computer graphics systems generate an interrupt at the start of a blanking interval that we can write an interrupt handler for. While it's possible to do this on the PS2, it's much easier to just poll the appropriate bit of CSR register, which will be set at the start of the interval. Just remember that this method hogs the CPU until the interrupt is generated.

```
void g2_wait_vsync(void)
{
    *CSR = *CSR & 8;
    while(!(*CSR & 8));
}

void g2_wait_hsync(void)
{
    *CSR = *CSR & 4;
    while(!(*CSR & 4));
}
```

The Demo

I've kept the same demo from the previous tutorial so you can compare the difference in quality of the animation.

Just to repeat, 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. The message is actually for now3d, who you can frequently find loitering at #ps2dev on EFnet :-)

NTSC Support

A few of you have correctly pointed out to me that the code is currently lacking support for NTSC. The reason for this is that I didn't think I had any NTSC device I could test things on, but I was wrong. The software that came with my crappy TV tuner card allows you to switch it into NTSC mode. I've now added some standard NTSC video modes, and a PAL/NTSC auto-detect routine that I ripped straight from *funslower* :-)

It seems to work fine for all the NTSC modes I've set up, and the animation even looks a bit smoother. Can someone please tell me if it all detects and works OK on a proper NTSC TV. Thanks.

Also, if anyone has detailed info about getting a higher vertical resolution and using interlaced modes I'd really appreciate an explanation.

Conclusion

So that's the end of this tutorial. I really can't wait until I've finished the next one, so I can move onto something a bit more interesting...although I might do a simple 2D Game first. We'll see. I have cunningly dropped a few hints in these Tutorials about a simple game idea that I'm thinking about.

Just a reminder, the next Tutorial is all about developing a 2D Font Engine based on the image code developed so far.

Oh, and keep those e-mails coming: t_saveski@yahoo.com.

Cheers,

Tony (dreamtime)