Cycling the Color Palette

Methodology

There’s a couple of ways of accomplishing colors that rotate. The basic idea is to read the palette, and write its contents one index over. We can use the Load_DAC_Palette(..) and Write_DAC_Palette(..) functions to do this very very easily. All undefined functions can be found in the *Setting the VGA Color Palette* tutorial. The only problem is that our background color gets written over with this method. If you have a full screen picture, this may not be TOO bad, but personally it bugs the crap out of me! I decided to do the function in inline assembly so that the background always remains the same. Another point to mention is that with most video cards, if you don’t wait for a vertical retrace and just decide to start rotating the colors, it will result in MASSIVE flickering. If we do wait we get a flawless looking picture with little or no performance loss. You decide! If this seems a little confusing, look below. Each accomplishes color palette cycling. You should be able to find the example that fits your needs, and your programming style.

BIOS Functions

Good old BIOS provides several functions that allow the reading and writing of DAC registers. These should be used as a starting point only. They are VERY slow. Then again, compared to the time it takes the inline assembly code to do a rotate (1/70 of a second for 1 color shift) anything else seems slow :) BIOS doesn’t provide function that actually DO the rotating, we’ll just be using the functions we created earlier that use the BIOS, to do it.

```
void Video::Rotate()
{ uchar Palette[256];
  Load_DAC_Palette(0,256,Palette);
  Write_DAC_Palette(1,256,Palette);
}
```

Right now you should be asking yourself, if we are starting at color one when we are writing, how can we write 256 colors? Shouldn’t that be 255 colors? The BIOS is smart enough that if we specify more colors than what is actually available, it will wrap around and start setting the colors from 0! Neeto huh :) With this example color 0 does get shifted so the background color shifts as well, for better or worse. The next function should take care of that problem as i view it.

```
void Video::RotateNoBlack()
{ uchar R,G,B,*Palette;
  Get_DAC(255,R,G,B);
  Load_DAC_Palette(1,255,Palette);
  Write_DAC_Palette(2,254,Palette);
  Set_DAC(1,R,G,B);
}
```

Here we are just reading in the last color of the palette, the one that was destined to go into color 0. We set the rest of the palette, starting with 1 and only 254 colors so black isn't touched. Then set the 1st color (besides black) to our saved color, and voila! I'm writing this from memory, but i think it will work ok!

C++ Functions

This is the same structure as the above functions that use the BIOS functions. The only difference is that these will be a little faster because we are communicating right with the metal baby!

```
void Video::Rotate()
{ uchar Palette[256];
  Load_DAC_C_Palette(0,256,Palette);
  Write_DAC_C_Palette(1,256,Palette);
}

void Video::RotateNoBlack()
{ uchar R,G,B,*Palette;
  Get_DAC_C(255,R,G,B);
  Load_DAC_C_Palette(1,255,Palette);
  Write_DAC_C_Palette(2,254,Palette);
  Set_DAC_C(1,R,G,B);
}
```

These functions do the exact same thing as the BIOS versions above, except like I mentioned before they will be faster since they are communicating directly with the hardware!

Inline Assembly Functions

```
void Video::FastRotate()
{Load_DAC_Palette(0,256,Palette); // use inline asm version
 address=(long)&Palette;
  while(inp(0x3DA) & 0x08);
  while(!(inp(0x3Da)&0x08));
  __asm__("
    movw $762,%%cx          //counter
    movw $0x03c6,%%dx        // dx is our port
    movw $0xff,%%ax         // ax is the input number to go to port
    outb %%ax,%%dx
    movl $2,%%eax           //Set Starting color
    movw $0x03c8,%%dx
    outb %%ax,%%dx
    movl _address,%%ebx
    add  $3,%%ebx           //skip over color 0!
    LoopStart:
    movl (%%ebx),%%eax
    movw $0x03c9,%%dx
    outb %%ax,%%dx
    add  $1,%%bx
    Loop LoopStart
    mov  $0,%%eax          //set black
    outb %%ax,%%dx
    outb %%ax,%%dx
    outb %%ax,%%dx
    movl (%%ebx),%%eax      //set wrap around color
    outb %%ax,%%dx
    add  $1,%%bx
    movl (%%ebx),%%eax
    outb %%ax,%%dx
    add  $1,%%bx
    movl (%%ebx),%%eax
    outb %%ax,%%dx
    "
    :
    :
  :"%cx","%eax","%dx","%ebx");
}
```

Lets go through this in Chunks if you will :) First off, for this to work, address must be defined globally since its inline assembly. We load up the color palette and assign address to the address of our buffer in memory. The next two while's wait for a vertical retrace to start before proceeding. That gives us a 0 flicker screen! Try it without and then write me saying how grateful you are :) yeah right! In the first chunk of the assembly we set cx to be the number of loop iterations we have to do. Loop uses cx for this. Next, we set the DAC mask register to 0xff so that we have access to all colors. The next chunk tells the DAC that we will start with color number 2. Next, we move the address of address into bx so we can use it as a pointer, then we add 3 onto it because we know the first 3 values are going to be 0's, because who in their right mind is going to set color 0 to a REAL color!? I think i may have created some enemies from that statement, but moving right along..! Inside the Loop (LoopStart) we create a pointer (ax) and set the port to which we will be writing this information (dx). We exit out of the loop with all colors set except 0 (which we want to leave anyway) and 1. At this very point, the DAC will reset to start setting data for color 0, so lets just set it to 0's. That is what the chunk after the loop does. The final chunk sets color 1 manually. Hope this helps you out some. If you have any question, comments, complaints, whatever! send me some Feedback.

Contact Information

You can get a hold of me through my website or direct email.

Email : deltener@mindtremors.com
Webpage : http://www.inversereality.org

Inverse Reality