Moving the Offscreen Buffer to Video RAM

Intro

By now we know what an offscreen video buffer is, but how the heck do we move it to Video RAM!? This tutorial will go through each of the possible video configurations and show you how to make it happen. At the end we go through a master function that will tie everything together. These functions are encapsulated in my video class, which explains the Video:: in the function definitions. Remember that in our Video class, we have defined a function pointer called Move2Vid which was is supposed to be set when we switch to a video mode, the pointer assignment convention is Move2Vid = &Video::AppropriateFunction. Now we can finally see what we were setting our pointer to!

Standard Video Modes

If we know where Video Ram resides, we can move all of our Video Buffer there in one memmove function! Because I'm such a nice guy, I'll let you see the same function using DJGPP inline assembly. This version is pretty darn quick because it moves the buffer in 4 byte amounts. zoom zoom zoom :) Here are the two functions for moving our Offscreen Buffer to Video Ram.

```
void MoveMemToScreenStandardMode()
{asm (" LOOPB:
        movl (%%esi),%%eax
        movl 4(%%esi),%%ebx
        movl %%eax, (%%edi)
        movl %%ebx,4(%%edi)
        addl $8,%%esi
        addl $8,%%edi
        decl %%ecx
       jnz LOOPB"
       : /*No output*/
       :"c" (Screen_Size>>2), "D" (video_screen), "S"(video_buffer)
       :"%eax","%ebx");
}
```

or standard C++

```
void MoveMemToScreenStandardMode()
{memmove(video_screen,video_buffer,64000);
}
```

Although the inline assembly version is longer and an eyesore to look at, it's faster! We follow the convention defined in the DJGPP and Protected Mode programming tutorial. The asm function moves 4 bytes at a time and 8 bytes total per loop. ecx,esi and edi are automatically loaded with our input register scheme. Notice since we load our registers this way, we don't have to list them in the trashed register listing since the compiler knows before hand that those registers will not be valid! video_screen is simply the location of video ram, 0xA000 in real mode and 0xA000,0 passed in our protected mode MK_FP function. video_buffer is a char* to our offscreen video buffer, and Screen_Size should have been set when we switched video modes! The second version is a lot smaller, but isn't as fast since it will be moving memory one byte at a time :(We simply pass the appropriate char * 's and our screen size to make it work. Simple enough huh? Let's make it a little more difficult and start going into some weird video modes!

Tweaked Video Modes

This is the function listing that covers how to move our Offscreen Buffer to Video Ram when we are dealing with a Tweaked Video Mode. If you don't remember how Video Ram is configured in these modes, please take a look at the explanation here. In the previously mentioned Using Unchained Video Modes tutorial, we briefly described how to gain access to the different memory pages and how to access individual pixels. Now we are going to construct a function that will be able to move all of our Offscreen Buffer to Video Ram!

Functions

This function moves our Offscreen Buffer to Video Ram in a very unique manner. It will select one plane of Video Memory and write the appropriate byte to it. In essence we want our function to select one plane of Video Memory and then move every 4th byte to that memory location. Here we go!

```
void Video::MoveMemTweaked()
{ for(int l=0;l<4;l++)
   {outportb(0x3c4,0x02);
    outportb(0x3c5,(1<<l));
    asm(" PlanerLoop:
         movl (%%esi),%%edx
         movb %%dx,(%%edi)
         addl $4,%%esi
         addl $1,%%edi
         Loop PlanerLoop"
        :
        :"c" (Screen_Size>>2), "D" (video_screen), "S"(video_buffer+l)
        :"%edx");
  }
}
```

or Normal C++

```
void MoveMemTweaked()
{ unsigned char *vbuff=video_buffer;
  unsigned int SS = Screen_Size>>2;
  for(int plane=0; plane<5; plane++)
   { outp(0x3c4,0x02);
     outp(0x3c5,(0x01<<(plane&3)));
     for(int pixel=0,count=0; pixel<SS; pixel+=4,count++)
        {video_screen[count]=vbuff[pixel];
        }
     vbuff++;
   }
}
```

For each we are going to divide Screen_Size by four since each memory location actually counts for 4. In each function we select which plane we want to write to, then move every 4th byte from our Video Buffer to Video Ram. After each movement we add one to the location of our Video Buffer since we want to start with the next set of pixels. So we start to move 0,4,8,12..then when we add one on to it after its finished with the first plane it will begin to move 1,5,9,13 and soon. Both functions are pretty darn inefficient, but I thought I'd make them as simple as possible so that you get the feel of how they work, that way you can make your own optimizations later on! More detail on how to deal with the memory organization of a tweaked video modes lies under the Using Unchained Video Modes tutorial.

VESA Modes

Moving Offscreen Buffers to Video Ram in a VESA Mode is easier than having to deal with a Tweaked Video Mode, but certainly not as easy as a Standard Video Mode. For each function we will be selecting which bank we are going to fill, then we just have to fill in the normal 64k of memory, set to the next bank and start all over again. If you don't remember how Video Ram is organized under a VESA Video Mode, then take a gander here!

Functions

Here's our wonderful listing of inline assembly mayhem and the easy-to-understand C++ version. The inline assembly version has been slightly optimized by unrolling its inner loop a bit.

```
void Video::MoveMemVESABanked()
{long index=0;
 unsigned long Bank;
 for(Bank=0;Bank<NumberOfBanks;Bank++)
    { SetBank(Bank);
     asm("LOOP:
         movl (%%ebx),%%edx
```

```
          movl %%edx,(%%eax)
          movl 4(%%ebx),%%edx
          movl %%edx,4(%%eax)
          movl 8(%%ebx),%%edx
          movl %%edx,8(%%eax)
          movl 12(%%ebx),%%edx
          movl %%edx,12(%%eax)
          addl $16,%%eax
          addl $16,%%ebx
         Loop LOOP”
          :
          :”a” (video_screen),”b” (video_buffer+index),”c” (4096)
          :”%edx”);
      index+=65536;
     }

  if(MemAfterBanks)
  {SetBank(Bank);
   asm(“LOOP3:
        movl (%%ebx),%%edx
        movl %%edx,(%%eax)
        addl $4,%%eax
        addl $4,%%ebx
       Loop LOOP3"
        :
        :”c” (MemAfterBanks>>2),”a” (video_screen), “b” (video_buffer+index)
        :”%edx”);
   }
 }
```

or Normal C++

```
 void MoveMemVESABanked()
 { unsigned char *tempptr= video_buffer;

  for(Bank=0;Bank<NumberOfBanks;Bank++)
    {SetBank(Bank);
     memmove(video_screen,tempptr,65536);
     tempptr+=65536;
    }

  SetBank(Bank);
  if(MemAfterBanks)
    { memmove(video_screen,tempptr,MemAfterBanks);
    }

 }
```

In the inline assembly version, the first loop fills the as many full screens as possible, then enters another section of inline assembly and fills in the remainder. Notice how the pointers are being directly loaded into the registers. Since we are preloading the registers, we don't have to list them in the trashed register listing. All we really need to know is that we have to set which memory bank we are going to start filling fill as many full screens as possible and then move the remaining portion. The SetBank function is detailed in the Video Modes with Banks tutorial.

Master Move2Video Function

Now that we can move our Offscreen Buffer to Video Ram under Standard, Tweaked and VESA Video Modes, let's create one function that can be used in any Video Mode no matter how Video Ram is arranged! Well thanks to us for creating separate self-sufficient functions for each mode, we now have that power! Without any further delay:

Function

```
void Video::Move2Video()
{while(inp(0x3DA) & 0x08);
 while(!(inp(0x3Da)&0x08));
(this->*Move2Vid)();
}
```

Well I think we made this function out to be a lot harder than it really was! This function basically boils down to waiting for a vertical retrace to occur, then run our function pointer. Notice the strict guidelines we are following for calling the function pointer. We have our function in parenths, and access it with the this->* combination. This is a must in order to be compliant with GCC 2.8.1. The previous version allowed us free reign and worked, but those old conventions are no longer going to be accepted, so learn this method!! Here's where the real power of function pointers comes into use. We no longer have to test what video mode we are in, we don't even care. We know that our function pointer was set when they set the video mode. Since screen refresh is certainly time dependent, eliminating the if's in this function by using function pointers was a real smart move!

Now we can move our Video Buffer to Video Ram no matter WHAT Video Mode we are in! Now THAT'S power! It's nice to have one function that does all the work. I just wish I could find some people to do all my work...Wish me luck! If you have any questions, comments, rude remarks please give me some feedback!

Contact Information

I just wanted to mention that everything here is copyrighted, feel free to distribute this document to anyone you want, just don't modify it! You can get a hold of me through my website or direct email. Please feel free to email me about anything. I can't guarantee that I'll be of ANY help, but I'll sure give it a try :-)

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