CS2261 Media Device Architecture

* Quiz 3 on Wednesday next week, focuses on Mode 3 and will have a high-level question or 2 on Mode 4

Review/examples of questions:

* Constant and volatile: USENTI generates a constant array, volatile is important when it comes to fillScreens parameter. If it’s not volatile, then the color parameter will default to black
  + The call stack is usually ignored when it comes to color and we get a junk address
* Why have a generic void pointer? Because in DMA we want to move blocks of data of different sizes (like 32 bits or 16 bits). If we declare a type, like int pointer, we can’t move shorts with it
  + We don’t know yet syntactically what we are moving, and making it void allows for freer movement
* Why are we using DMA?
  + Faster
  + Limitations? Data copy must be contiguous, order matters.

Notes

* DMA: speeds up painting during VBlank
* Still have a lingering problem that VBlank isn’t long enough
* Solutions:
  + Currently we have: Video memory 🡪 video controller 🡪 screen
  + What if we address the issue of being forced to do our transfers in the VBlank period by having a separate memory unit
    - **Shadow Video Memory**
  + This is extra memory not being read by the video controller, which avoids any conflict and any flicker (flicker result of trying to read and write at the same time)
    - Draw whatever complex graphics you want in the shadow memory and transfer the contents to real video memory during VBlank
      * Called **Double Buffering**
    - Still inefficient because you’re writing twice: first to shadow memory and then copy things over
    - New solution: affect where Video Controller gets its data
* Video Controller: looks to video memory for the data, but we can change where video controller looks for data
  + Use pointers to change video controller to look at the shadow video memory
  + Same idea: 2 buffers = like 2 screens
  + Video controller had a bit that determines which buffer to use (0 for standard, 1 for shadow)
  + During VBlank, flip the bit so it shows data in the buffer that was just drawn on
  + Update other buffer for the next frame
  + Another problem: too expensive!
    - Solution: get rid of the current other shadow video memory and instead split the main video memory in half and flip between using them
      * Front buffer and back buffer
      * You have two now
      * Theoretically, just looking at it as 2 buffers of 8 bits/pixel
  + Problem: you now have 8 bits per pixel instead of 16 bits
    - This will limit things like color
      * 3 bits to color (RGB) instead of 5 bits to color
    - Primitive, this will lower the intensity of colors we can have
* A Hardware Solution
  + Index color
  + Store the colors in a table, give each an index, the memory locations hold an index that you can use to go fetch a color
    - Have each location represent a 16 bit color: A palette
  + Just adding a little bit more memory
  + A fast lookup in an array for the color using the index
  + Yet another problem:
    - only 256 colors at a time
    - Which colors do we use for the palette?
    - Solution: figure out what you are going to show and try to figure out which colors you need
    - Tools for this: USENTI
      * Can generate an array of most important colors (palette)
      * Two possibilities: Your image might go under the 256 color limit or can have more than 256 (will cut you off at 256)
        + Honestly you cannot tell when colors are cut short
* Memory Access:
  + Example of changing a pixel from red to blue
* Mode 4
  + Mode 4 pixels are 8 bits each, packed with one other into a 16 bit video buffer entry
    - Left and right bits
    - [8 bits on right 8 bits on left]
* Implementing Mode4:
  + Use the REG\_DISPCTL
  + You can have a game switch modes
  + Bit 4 page select: allows you to pick which buffer is being displayed
    - 0 for the front buffer
    - 1 for the back buffer
    - Buffer 0 – x6000000
    - Buffer 1 – 0x600A000
    - Have macros for both
    - Palette address: 0x5000000
  + For bitmapped graphics, use background2
  + Code now:

while (1){

calculatePositions()

drawStuff()

waitForVBlank()

flipPages()

}

* + flipPages()
    - if the display H/W (hardware) is drawing on one screen, switch to other
    - make sure drawscreen() draws on the appropriate buffer
    - just switch the address reference
* Now that we have two buffers:
  + Mapping pixels to Memory
  + Pair of consecutive pixels map to a short
    - PIXEL 0 1 | 2 3 | 4…
    - SHORT 0 | 1 | 2
  + For 384000 pixels we need 38400 shorts since we split video memory into two buffers
  + How would you find the short number of a pixel:
    - Already have the offset function
    - Examples and explanation on slides
  + Odd columned pixel is controlled by the higher ordered byte (bits 8-15 of videobuffer)
* Reviewed how to set pixels (both even and odd columned pixels)
  + To determine if its an even or odd column, use the % operator on the pixel column
    - Even side of short or odd side based on what you’re working with
  + Alternatively, look at binary numbers, if last value is 1, its odd, 0 if it’s even