CS 2261 Media Device Architecture

Mode 4, Mode 4 Drawing Functions, Mode 4 Drawing Functions with DMA

Introduction

* Currently, we have too little time to draw. The framerate decreases when there’s too much going on.
  + Implies we only show the frame when we are finished creating it.

Double Buffering

* Have two “buffers.” Draw on one while displaying the other, then switch.
  + You’re never looking at the one that is being drawn on.
  + Thus, flicker is impossible.
* Instead of two video buffers, we pretty much split the current video buffer in half
  + 38,400 bytes = 37KB (75KB for both)

Mode 4

* A bitmap display mode (just like Mode 3)
* Video Memory starts at 0x6000000 (just like Mode 3)
* 15-bit color (just like Mode 3)
* Video Memory consists of bytes (not like Mode 3)
* The problem is, the smaller memory size (bytes instead of shorts) proves a problem for storing color. The solution is to move color elsewhere (a Palette) and use indices to access it.

Palette

* Rather than each pixel looking at a 15-bit color, it looks at an 8-bit index. It finds that index in the palette, and displays *that* color.
* Palette Memory: 0x5000000
  + Array of 256 shorts
  + Each short is a 15-bit color (same RGB format as Mode 3)
* Video Memory: 0x6000000
  + 2 arrays of 38,400 bytes
  + Each byte is a 8-bit palette index
  + 28 = 256
* So we can still have the same range of colors (0 - 32767, Black - White)
  + We just have to pick a max of 256 of them to show at any given time
  + Not bad in comparison to full color

Mode 4 Double Buffering

* Two buffers: “front” and “back”
  + Front buffer: 0x6000000
    - Start of the video memory
  + Back buffer: 0x600A000
    - A little past halfway through the video memory
    - There are 2,560 bytes between the end of the front and the start of the back
* New process
  + 1. Display the back while drawing the front
  + 2. waitForVBlank
  + 3. Display the front while drawing the back

Controlling which buffer displays

* There is a register for this, the 4th bit of REG\_DISPCTL determines which buffer displays
  + 0: Front buffer displays
  + 1: Back buffer displays
* #define DISP\_BACKBUFFER (1<<4)
* Use Exclusive Or to toggle
* REG\_DISPCTL ^= DISP\_BACKBUFFER;

Controlling which buffer is drawn on

* All of our drawing functions draw to videoBuffer, so we just move wherever this points to
* #define FRONTBUFFER ((unsigned short\*)0x6000000)
* #define BACKBUFFER ((unsigned short \*)0x600A000)
* videoBuffer = FRONTBUFFER; //Drawing functions will now draw to the front
* videoBuffer = BACKBUFFER; //Drawing functions will now draw to the back

Flipping the Page

* When we are finished drawing, we flip the page.
* REG\_DISPCTL & DISP\_BACKBUFFER //tells us which page we are on
* True if we’re showing the back. False otherwise.

void flipPage() {

if (REG\_DISPCTL & DISP\_BACKBUFFER)

videoBuffer = BACKBUFFER;

else

videoBuffer = FRONTBUFFER;

REG\_DISPCTL ^= DISP\_BACKBUFFER;

}

Order of Program Flow: Mode 4

* update();
* draw();
* waitForVBlank();
* flipPage();

Why wait for VBlank before flipping?

* If we are not in the vertical blank period, then we are halfway through drawing the screen.
* What if half of the screen is one frame, and the other half is the next?
  + Tearing.

Reminders

* These are just display modes.
  + They only affect how the screen interprets the video memory
  + So (almost) all of our drawing functions will have to change
  + Nothing else changes
* You can switch between them mid-game
  + This is not only not difficult, it is encouraged.
  + For the final project, not homeworks unless otherwise specified

Mode 4 Drawing Functions

* Remember, the videoBuffer in Mode 4 is an array of chars
  + Each char represents where to find that pixel’s color in the palette
* Mode 3 drawing functions take in a short for the color
* Mode 4 drawing functions take in a char for the color
  + Remember, they don’t know or care which buffer (front or back) to draw to
  + They always draw where videoBuffer is pointing
* The problem is that Mode 4 is an array of chars, and we used shorts in Mode 3, so instead we grab the short, change the half we want to change, and put it back.

Pixel memory in Mode 4

* Each short controls two pixels
  + The right 8 bits (the 8 least significant bits) control the pixel on the right
    - The pixel with the even column
  + The left 8 bits (the 8 most significant bits) control the pixel on the left
    - The pixel with the odd column

Setting an even column pixel

* Grab the short.
* Clear the least significant 8 bits.
* Set the least significant 8 bits to the new color index.
* Put the short back.

Setting an odd column pixel

* Grab the short.
* Clear the most significant 8 bits.
* Set the most significant 8 bits to the new color index.
* Put the short back.

Putting it together

void setPixel4(int row, int col, unsigned char colorIndex) {

unsigned short pixelData =

videoBuffer[OFFSET(row, col, SCREENWIDTH)/2];

if (col & 1) { // column is odd

pixelData &= 0x00FF;

pixelData |= (colorIndex<<8);

} else { // column is even

pixelData &= 0xFF00;

pixelData |= colorIndex;

}

videoBuffer[OFFSET(row, col, SCREENWIDTH)/2] = pixelData;

}

Other Mode 4 Drawing Functions Without DMA

* Without DMA, the other drawing functions are mostly the same
* drawRect4, fillScreen4, and drawChar4 take in a char for color, not a short.
* drawRect4 and drawChar4 use setPixel4 instead of setPixel3
* fillScreen4:
  + Needs a short to put in each index to iterate correctly: colorIndex | (colorIndex << 8)
  + Only iterates half as many times as fillScreen3 (because it sets two-at-a-time)

Mode 4 Drawing Functions with DMA

* Remember, DMA has two chunk size options
  + 16 bits, or 32 bits
* The videoBuffer in Mode 4 wants chars (8 bits)
* So we have to copy two-at-a-time
* This makes things that want to edit only half of a short very difficult

fillScreen4

* You are copying two pixels’ worth of data at a time
  + SCREENWIDTH \* SCREENHEIGHT / 2
* For fillScreen4, you must make the color index cover both halves of the short so you can put that index in every pixel
  + volatile unsigned short pixelData = colorIndex | (colorIndex<<8);