CS 2261: Media Device Architecture - Week 6 part 2

Quiz 2

- Stuff to the sides of the room
- If you need scratch paper, I have some up front!
- You have 28 min.
- Come back for lecture afterwards! Please... :'(

Overview

- DMA
- Intro to Mode 4

Direct Memory Access (DMA)

- What makes a program slow?

```
for(int i=0; i<34800; i++)
{
  VIDEO_BUFFER[i] = 0;
  // super slow clear screen
}
```

Is there a solution?

- Processor is slow
- Moving blocks of memory or filling blocks of memory is common and slow

Custom Circuits

- Can do a few things really well
- Used for basic operations
- If things become complicated must revert to general purpose processor

Enter DMA

- DMA = Direct Memory Access

- Hardware supported data copy
  - Up to 10x as fast as array copies
    - Especially for larger arrays
  - You set it up, the CPU is halted, data is transferred, and CPU gains back control
  - It's completely blind to the contents/type of the data being copied (so we'll use void pointers with it).

DMA Channels

The GBA has 4 DMA channels: (we'll only use 3 for now)

- 0
  - Highest Priority
  - Time Critical Operations
  - Only works with IWRAM
- 1 & 2
  - Transfer sound chunks to sound buffer
- 3
  - Lowest Priority
  - General purpose copies, like loading tiles or bitmaps into memory

Using DMA

- Source
  - REG_DMAxSAD (0x040000B0 for channel 0)
    (where x is the channel number: 0, 1, 2, 3)
  - The address of the data that will be copied
- Destination
  - REG_DMAxDAD (0x040000B4 for channel 0)
  - The address to copy the data to
- Amount
  - REG_DMAxCNT (DMA control) (0x040000B8 for channel 0)
  - How much to copy plus options

REG_DMAxCNT

- Lower 16 bits contain amount to transfer
- Upper 16 bits contain other options
  - Turn on a DMA channel
  - When to perform the DMA
  - How the copy source and destination behave
  - How much to copy at a time
  - Whether or not to throw an interrupt on completion
  - Repeat or don't repeat on finish
- Can be treated as one 32 bit register, or two 16 bit registers
  - They're all bits, after all!

DMA Control bits

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DMA_ON	IRQ	When			Chunk Size	Repeat	SRC		DST						

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Number of Transfers (max: 65,536)															

This is the number of *chunks* to copy. Not the number of bytes.
Zero here means do it 2^16 times (since zero made no sense to be an option here)

DMA Control Bits explained

bits	name	bits description
0-15	N	Number of transfers (where 0 means 2^16).
16-20	NOT USED	
21-22	Destination Adjustment bits	
	DMA_DST_INC	00: increment after each transfer (default)
	DMA_DST_DEC	01: decrement after each transfer
	DMA_DST_FIXED	10: none; address is fixed
	DMA_DST_RESET	11: haven't used it yet, but apparently this will increment the destination during the transfer, and reset it to the original value when it's done.
23-24	Source Adjustment bits	
	DMA_SRC_INC	00: increment after each transfer (default)
	DMA_SRC_DEC	01: decrement after each transfer
	DMA_SRC_FIXED	10: none; address is fixed
	DMA_SRC_RESET	11: "forbidden" for source
25	DMA_REPEAT	If set, repeats the copy at each VBlank or HBlank if the DMA timing has been set to those modes.

DMA Control Bits explained

bits	name	bits description
26	CS (Chunk size)	0: Copy by half-word (16 bits) 1: Copy by word (32 bits)
27	NOT USED	
28-29	TM (Timing Mode)	00: Start immediately (still a small delay before it takes over) 01: Start at VBlank 10: Start at HBlank 11: Start at Refresh - advanced and probably not what you expect. Ignore for now (or forever?)
30	I (Interrupt Request)	If set, Raise an interrupt when finished.
31	En (Enable)	Enable the DMA transfer for this channel (turn it on!)

Source/Dest Adjustment

1. Increment Source and Dest:
   - Both set to defaults (00)
   - This copies source to dest
   - Ex: drawImage, drawFullScreenImage
2. Source Fixed, increment Dest:
   - Source set to (10), Dest set to (00)
   - Fills dest with one value from source
   - Ex: drawRect, fillScreen

All the Dest. fixed options probably make no sense at all (lots of overwriting). Decrementing just lets you do things backwards (or reverse things, if src and dest are going in opposite directions).

Chunk Size and Number

- Ex. Copy an array of 43 shorts
  - chunk size 16, copy 43 chunks
- Ex. Copy an array of 103 ints
  - chunk size 32, copy 103 chunks
  - chunk size 16, copy 206 chunks (half an int each time)
- Ex. Copy an array of 82 chars
  - chunk size 16, copy 41 chunks (two chars each time)
  - can't chunk by 32, because then you'd want to copy 20.5 chunks

Memory

X	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000

```
typedef struct
{
   const volatile void *src;
   volatile void *dst;
   volatile u32 cnt;
} DMAREC;

DMAREC *dma = X;
```

Memory

X	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000

```
dma++;
```

Why are we doing this?

DMA control registers are all consecutive in memory!

Memory

0x040000B0	00000000	00000000	00000000	00000000	DMA0SAD
	00000000	00000000	00000000	00000000	DMA0DAD
	00000000	00000000	00000000	00000000	DMA0CNT
	00000000	00000000	00000000	00000000	DMA1SAD
	00000000	00000000	00000000	00000000	DMA1DAD
	00000000	00000000	00000000	00000000	DMA1CNT
	00000000	00000000	00000000	00000000	DMA2SAD
	00000000	00000000	00000000	00000000	DMA2DAD
	00000000	00000000	00000000	00000000	DMA2CNT
	00000000	00000000	00000000	00000000	DMA3SAD
	00000000	00000000	00000000	00000000	DMA3DAD
	00000000	00000000	00000000	00000000	DMA3CNT

Memory

0x040000B0	00000000	00000000	00000000	00000000	DMA0SAD
	00000000	00000000	00000000	00000000	DMA0DAD
	00000000	00000000	00000000	00000000	DMA0CNT
	00000000	00000000	00000000	00000000	DMA1SAD
	00000000	00000000	00000000	00000000	DMA1DAD
	00000000	00000000	00000000	00000000	DMA1CNT
	00000000	00000000	00000000	00000000	DMA2SAD
	00000000	00000000	00000000	00000000	DMA2DAD
	00000000	00000000	00000000	00000000	DMA2CNT
	00000000	00000000	00000000	00000000	DMA3SAD
	00000000	00000000	00000000	00000000	DMA3DAD
	00000000	00000000	00000000	00000000	DMA3CNT

Memory

0x040000B0	00000000	00000000	00000000	00000000	DMA0SAD
	00000000	00000000	00000000	00000000	DMA0DAD
	00000000	00000000	00000000	00000000	DMA0CNT
	00000000	00000000	00000000	00000000	DMA1SAD
	00000000	00000000	00000000	00000000	DMA1DAD
	00000000	00000000	00000000	00000000	DMA1CNT
	00000000	00000000	00000000	00000000	DMA2SAD
	00000000	00000000	00000000	00000000	DMA2DAD
	00000000	00000000	00000000	00000000	DMA2CNT
	00000000	00000000	00000000	00000000	DMA3SAD
	00000000	00000000	00000000	00000000	DMA3DAD
	00000000	00000000	00000000	00000000	DMA3CNT

```
typedef struct
{
  const volatile void *src;
  volatile void *dst;
  volatile u32 cnt;
} DMAREC;

DMAREC *dma = (DMAREC *)0x040000B0;
```

CS2261

Memory

0x040000B0	00000000	00000000	00000000	00000000	DMA0SAD	dma[0];
	00000000	00000000	00000000	00000000	DMA0DAD	
	00000000	00000000	00000000	00000000	DMA0CNT	
	00000000	00000000	00000000	00000000	DMA1SAD	dma[1];
	00000000	00000000	00000000	00000000	DMA1DAD	
	00000000	00000000	00000000	00000000	DMA1CNT	
	00000000	00000000	00000000	00000000	DMA2SAD	dma[2];
	00000000	00000000	00000000	00000000	DMA2DAD	
	00000000	00000000	00000000	00000000	DMA2CNT	
	00000000	00000000	00000000	00000000	DMA3SAD	dma[3];
	00000000	00000000	00000000	00000000	DMA3DAD	
	00000000	00000000	00000000	00000000	DMA3CNT	

Memory

0x040000B0	00000000	00000000	00000000	00000000	DMA0SAD
	00000000	00000000	00000000	00000000	DMA0DAD
	00000000	00000000	00000000	00000000	DMA0CNT
	00000000	00000000	00000000	00000000	DMA1SAD
	00000000	00000000	00000000	00000000	DMA1DAD
	00000000	00000000	00000000	00000000	DMA1CNT
	00000000	00000000	00000000	00000000	DMA2SAD
	00000000	00000000	00000000	00000000	DMA2DAD
	00000000	00000000	00000000	00000000	DMA2CNT
	00000000	00000000	00000000	00000000	DMA3SAD
	00000000	00000000	00000000	00000000	DMA3DAD
	00000000	00000000	00000000	00000000	DMA3CNT

```
typedef struct
{
  const volatile void *src;
  volatile void *dst;
  volatile u32 cnt;
} DMAREC;

DMAREC *dma = (DMAREC *)0x040000B0;
```

```
dma[3].src;
```

Memory

0x040000B0	00000000	00000000	00000000	00000000	DMA0SAD
	00000000	00000000	00000000	00000000	DMA0DAD
	00000000	00000000	00000000	00000000	DMA0CNT
	00000000	00000000	00000000	00000000	DMA1SAD
	00000000	00000000	00000000	00000000	DMA1DAD
	00000000	00000000	00000000	00000000	DMA1CNT
	00000000	00000000	00000000	00000000	DMA2SAD
	00000000	00000000	00000000	00000000	DMA2DAD
	00000000	00000000	00000000	00000000	DMA2CNT
	00000000	00000000	00000000	00000000	DMA3SAD
	00000000	00000000	00000000	00000000	DMA3DAD
	00000000	00000000	00000000	00000000	DMA3CNT

```
typedef struct
{
  const volatile void *src;
  volatile void *dst;
  volatile u32 cnt;
} DMAREC;

DMAREC *dma = (DMAREC *)0x040000B0;
```

```
dma[3].dst;
```

Memory

0x040000B0	00000000	00000000	00000000	00000000	DMA0SAD
	00000000	00000000	00000000	00000000	DMA0DAD
	00000000	00000000	00000000	00000000	DMA0CNT
	00000000	00000000	00000000	00000000	DMA1SAD
	00000000	00000000	00000000	00000000	DMA1DAD
	00000000	00000000	00000000	00000000	DMA1CNT
	00000000	00000000	00000000	00000000	DMA2SAD
	00000000	00000000	00000000	00000000	DMA2DAD
	00000000	00000000	00000000	00000000	DMA2CNT
	00000000	00000000	00000000	00000000	DMA3SAD
	00000000	00000000	00000000	00000000	DMA3DAD
	00000000	00000000	00000000	00000000	DMA3CNT

```
typedef struct
{
  const volatile void *src;
  volatile void *dst;
  volatile u32 cnt;
} DMAREC;

DMAREC *dma = (DMAREC *)0x040000B0;
```

```
dp[3].cnt;
```

DMA Setup

- Map a struct array over the DMA registers

```
typedef struct {
  const volatile void *src;
  volatile void *dst;
  volatile u32 cnt;
} DMAREC;

#define DMA ((volatile DMAREC*)0x040000b0)
```

Filling the Screen (Demo this!)

```
#define DMA ((volatile DMAREC*)0x040000b0)

typedef struct
{
  const volatile void *src;
  volatile void *dst;
  volatile u32 cnt;
} DMAREC;

void fillScreen(volatile u16 color){
  DMA[3].cnt = 0;  // clear it first!
  DMA[3].src = &color;
  DMA[3].dst = VIDEO_BUFFER;
  DMA[3].cnt = 1 << 31 | // turn it on!
               1 << 26 | // set chunk size to 32 bits
               1 << 24 | // set src as fixed
               16200;    // 32400 / 2
}

int main(){
  REG_DISPCNT = MODE3 | BG2_ENABLE;
  u16 i = 0;
  while (1) {
    fillScreen(i);
    i+= 127;
  }
}
```

Why doesn't this work?!
Let's fix it!

Filling a rectangle

Fill row-by-row using DMA for each row.

As long as each row is >10 pixels, it should still be faster than manually drawing every pixel.

When to DMA?

- Copying/filling a lot of data (more than ~20 pixels) with NO LOGIC to the copy.
  - drawRect, fillScreen
  - drawImage
  - arrayCopy
  - arrayReverse

- drawChar with DMA - nope, since you want logic (though you could copyChar with DMA)

Intro to Mode 4

- So far, everything we've been doing uses Mode 3, which is a bitmapped mode
  - 240x160 resolution
  - Starts at 0x06000000
  - Each pixel is a short with 15-bit color information: XBBBBBGGGGGRRRRR (32768 colors!)
  - (38400 shorts! -- ~76kB of the 96kB of VRAM available)
- Mode 4 is obviously somewhat different:
  - Still a bitmapped mode
  - Still 240x160 resolution
  - Still starts at 0x06000000 (sort of)
  - Each pixel is a single byte (256 colors -- which you have to pick!)
  - 38400 bytes -- ~38kB -- that leaves a lot of <u>extra room in VRAM</u>

Set Mode 4

```
#define MODE4 4

int main() {
  REG_DISPCNT = MODE4 | BG2_ENABLE;




  VIDEO_BUFFER[0] = 255; // does nothing...

  /* Mode 4 is active, but we're still not able to
     do anything with it yet. Why not? */

  while(1);
}
```

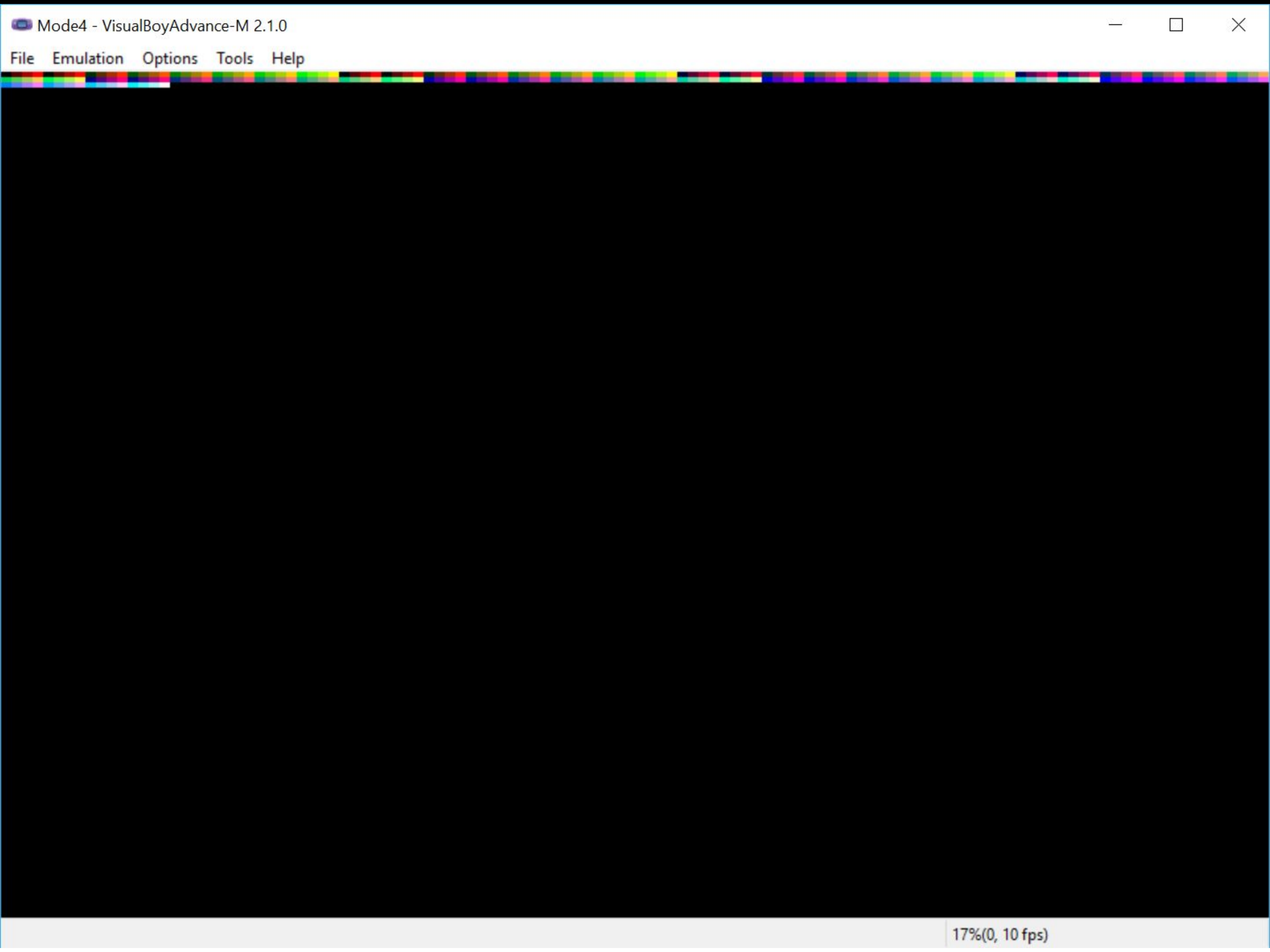
How do I set a color to a pixel?

- Pixel values are now chars (0-255).
- Those don't correspond to a predetermined set of 256 colors.
  - You have to set one up!
  - Here's where it goes:
    - `unsigned short* paletteMem = (unsigned short*)0x5000000;`
  - Each color there is the same 15-bit color we know and love from Mode3.

A Sample Palette (I mapped 3 bits for red, 3 for green, and 2 for blue to the 15 bits we're used to -- crudely)

```
volatile unsigned short* paletteMem = (unsigned short*)0x05000000;

unsigned short palette[] = {
0,11,21,31,32,43,53,63,192,203,213,223,352,363,373,383,512,523,533,543,672,683,693,703
,832,843,853,863,992,1003,1013,1023,1024,1035,1045,1055,1056,1067,1077,1087,1216,1227,
1237,1247,1376,1387,1397,1407,1536,1547,1557,1567,1696,1707,1717,1727,1856,1867,1877,1
887,2016,2027,2037,2047,6144,6155,6165,6175,6176,6187,6197,6207,6336,6347,6357,6367,64
96,6507,6517,6527,6656,6667,6677,6687,6816,6827,6837,6847,6976,6987,6997,7007,7136,714
7,7157,7167,11264,11275,11285,11295,11296,11307,11317,11327,11456,11467,11477,11487,11
616,11627,11637,11647,11776,11787,11797,11807,11936,11947,11957,11967,12096,12107,1211
7,12127,12256,12267,12277,12287,16384,16395,16405,16415,16416,16427,16437,16447,16576,
16587,16597,16607,16736,16747,16757,16767,16896,16907,16917,16927,17056,17067,17077,17
087,17216,17227,17237,17247,17376,17387,17397,17407,21504,21515,21525,21535,21536,2154
7,21557,21567,21696,21707,21717,21727,21856,21867,21877,21887,22016,22027,22037,22047,
22176,22187,22197,22207,22336,22347,22357,22367,22496,22507,22517,22527,26624,26635,26
645,26655,26656,26667,26677,26687,26816,26827,26837,26847,26976,26987,26997,27007,2713
6,27147,27157,27167,27296,27307,27317,27327,27456,27467,27477,27487,27616,27627,27637,
27647,31744,31755,31765,31775,31776,31787,31797,31807,31936,31947,31957,31967,32096,32
107,32117,32127,32256,32267,32277,32287,32416,32427,32437,32447,32576,32587,32597,3260
7,32736,32747,32757,32767 };
```


Mode4 - VisualBoyAdvance-M 2.1.0
File Emulation Options Tools Help
17%(0, 10 fps)

A Sample Palette (I mapped 3 bits for red, 3 for green, and 2 for blue to the 15 bits we're used to -- crudely)

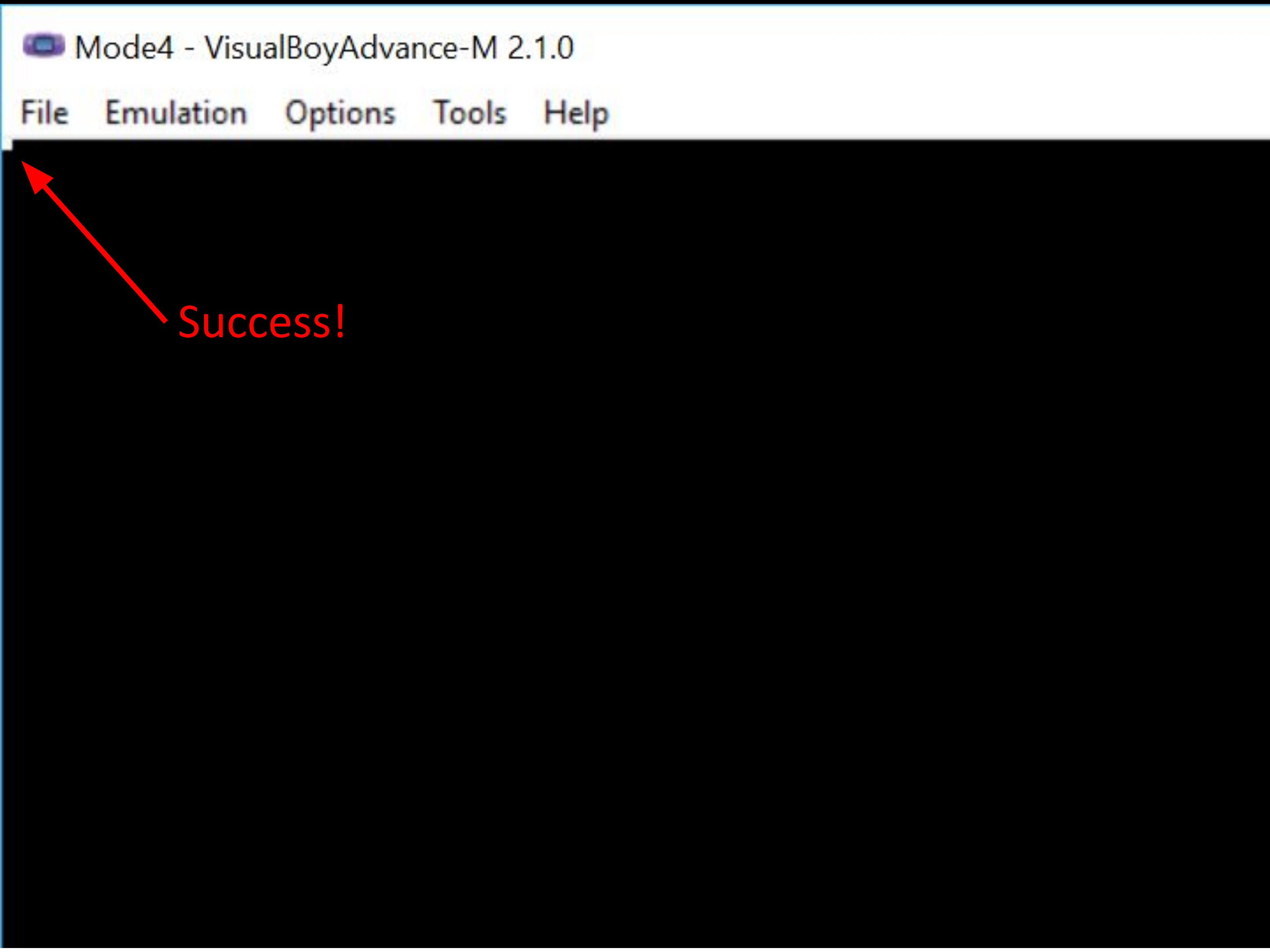
```
// These belong in a lib somewhere.
#define DMA ((volatile DMAREC*)0x040000b0)
typedef struct
{
  const volatile void *src;
  volatile void *dst;
  volatile u32 cnt;
} DMAREC;

volatile unsigned short* paletteMem = (unsigned short*)0x05000000;

unsigned short palette[] = {
0,11,21,31,32,43,53,63,192,203,213,223,352,363,373,383,512,523,533,543,672,683,693,703,832,843,853,863,992,1003,1013,1023,1024,10
35,1045,1055,1056,1067,1077,1087,1216,1227,1237,1247,1376,1387,1397,1407,1536,1547,1557,1567,1696,1707,1717,1727,1856,1867,1877,1
887,2016,2027,2037,2047,6144,6155,6165,6175,6176,6187,6197,6207,6336,6347,6357,6367,6496,6507,6517,6527,6656,6667,6677,6687,6816,
6827,6837,6847,6976,6987,6997,7007,7136,7147,7157,7167,11264,11275,11285,11295,11296,11307,11317,11327,11456,11467,11477,11487,11
616,11627,11637,11647,11776,11787,11797,11807,11936,11947,11957,11967,12096,12107,12117,12127,12256,12267,12277,12287,16384,16395
,16405,16415,16416,16427,16437,16447,16576,16587,16597,16607,16736,16747,16757,16767,16896,16907,16917,16927,17056,17067,17077,17
087,17216,17227,17237,17247,17376,17387,17397,17407,21504,21515,21525,21535,21536,21547,21557,21567,21696,21707,21717,21727,21856
,21867,21877,21887,22016,22027,22037,22047,22176,22187,22197,22207,22336,22347,22357,22367,22496,22507,22517,22527,26624,26635,26
645,26655,26656,26667,26677,26687,26816,26827,26837,26847,26976,26987,26997,27007,27136,27147,27157,27167,27296,27307,27317,27327
,27456,27467,27477,27487,27616,27627,27637,27647,31744,31755,31765,31775,31776,31787,31797,31807,31936,31947,31957,31967,32096,32
107,32117,32127,32256,32267,32277,32287,32416,32427,32437,32447,32576,32587,32597,32607,32736,32747,32757,32767 };

int main() {
  REG_DISPCNT = 4 | BG2_ENABLE;
  DMA[3].cnt = 0;
  DMA[3].src = palette; // or &palette, makes no difference
  DMA[3].dst = paletteMem;
  DMA[3].cnt = 1 << 31 | 256;

  VIDEO_BUFFER[0] = 255; // first pixel white
  while(1);
}
```


Mode4 - VisualBoyAdvance-M 2.1.0
File Emulation Options Tools Help
Success!

A Sample Palette (I mapped 3 bits for red, 3 for green, and 2 for blue to the 15 bits we're used to -- crudely)

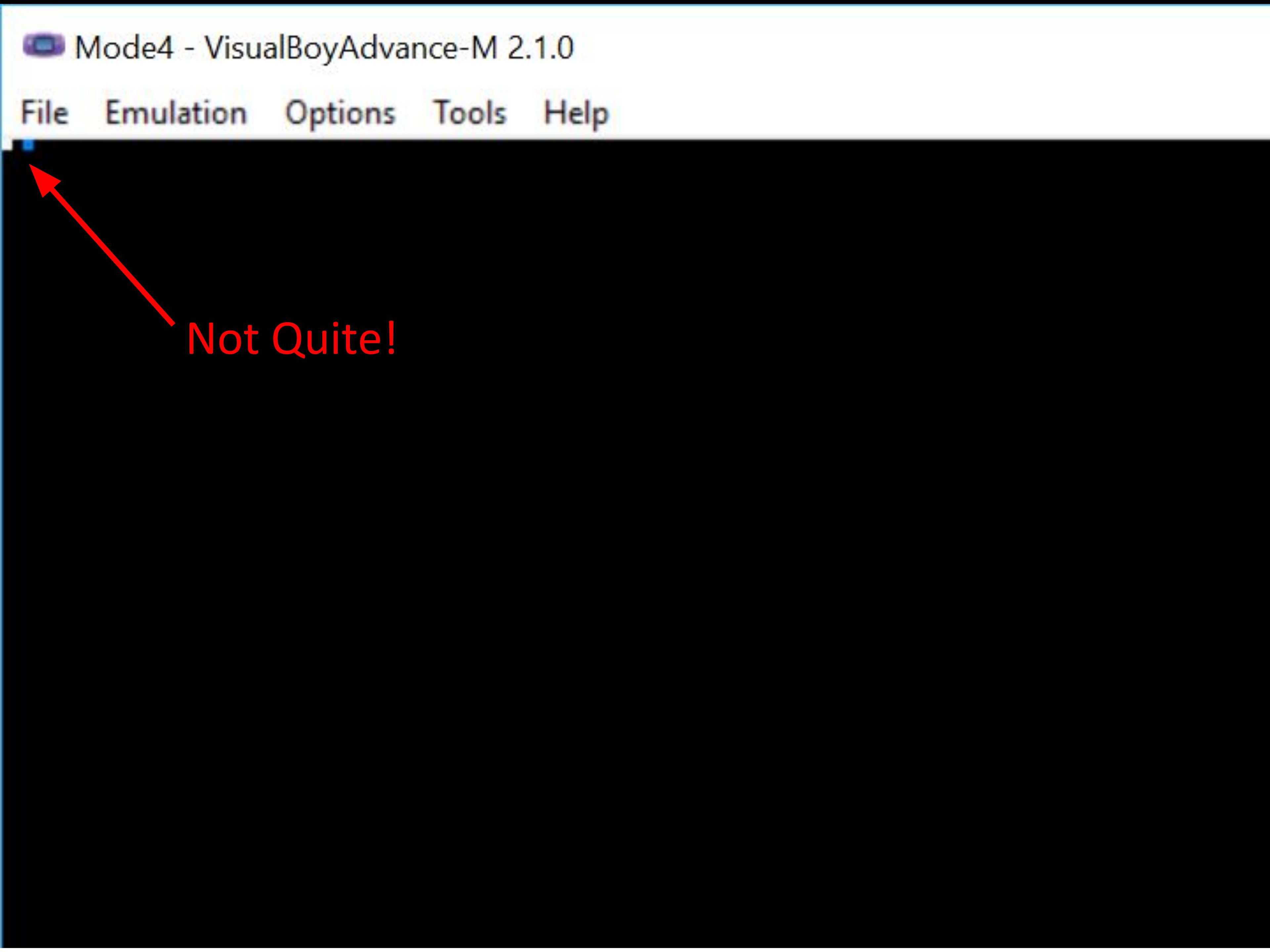
```
// These belong in a lib somewhere.
#define DMA ((volatile DMAREC*)0x040000b0)
typedef struct
{
  const volatile void *src;
  volatile void *dst;
  volatile u32 cnt;
} DMAREC;

volatile unsigned short* paletteMem = (unsigned short*)0x05000000;

unsigned short palette[] = {
0,11,21,31,32,43,53,63,192,203,213,223,352,363,373,383,512,523,533,543,672,683,693,703,832,843,853,863,992,1003,1013,1023,1024,10
35,1045,1055,1056,1067,1077,1087,1216,1227,1237,1247,1376,1387,1397,1407,1536,1547,1557,1567,1696,1707,1717,1727,1856,1867,1877,1
887,2016,2027,2037,2047,6144,6155,6165,6175,6176,6187,6197,6207,6336,6347,6357,6367,6496,6507,6517,6527,6656,6667,6677,6687,6816,
6827,6837,6847,6976,6987,6997,7007,7136,7147,7157,7167,11264,11275,11285,11295,11296,11307,11317,11327,11456,11467,11477,11487,11
616,11627,11637,11647,11776,11787,11797,11807,11936,11947,11957,11967,12096,12107,12117,12127,12256,12267,12277,12287,16384,16395
,16405,16415,16416,16427,16437,16447,16576,16587,16597,16607,16736,16747,16757,16767,16896,16907,16917,16927,17056,17067,17077,17
087,17216,17227,17237,17247,17376,17387,17397,17407,21504,21515,21525,21535,21536,21547,21557,21567,21696,21707,21717,21727,21856
,21867,21877,21887,22016,22027,22037,22047,22176,22187,22197,22207,22336,22347,22357,22367,22496,22507,22517,22527,26624,26635,26
645,26655,26656,26667,26677,26687,26816,26827,26837,26847,26976,26987,26997,27007,27136,27147,27157,27167,27296,27307,27317,27327
,27456,27467,27477,27487,27616,27627,27637,27647,31744,31755,31765,31775,31776,31787,31797,31807,31936,31947,31957,31967,32096,32
107,32117,32127,32256,32267,32277,32287,32416,32427,32437,32447,32576,32587,32597,32607,32736,32747,32757,32767 };

int main() {
  REG_DISPCNT = 4 | BG2_ENABLE;
  DMA[3].cnt = 0;
  DMA[3].src = palette; // or &palette, makes no difference
  DMA[3].dst = paletteMem;
  DMA[3].cnt = 1 << 31 | 256;

  VIDEO_BUFFER[0] = 255;
  VIDEO_BUFFER[1] = 240; // the next pixel
  while(1);
}
```

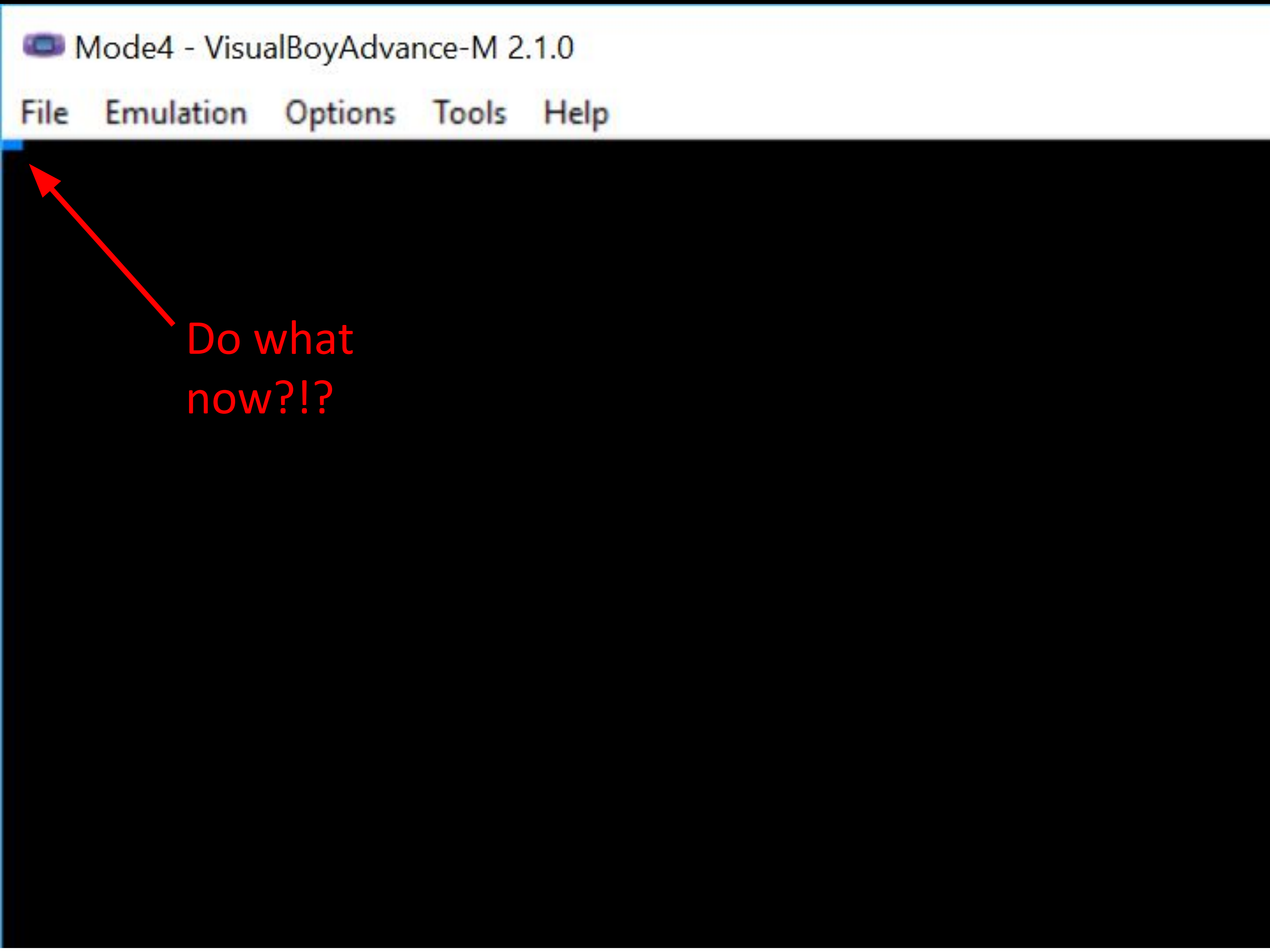

Mode4 - VisualBoyAdvance-M 2.1.0
File Emulation Options Tools Help
Not Quite!

Whoops

They're chars, not shorts… better use VIDEO_BUFFER as (unsigned char *) instead of (unsigned short *).

```
int main() {
  REG_DISPCNT = 4 | BG2_ENABLE;
  DMA[3].cnt = 0;
  DMA[3].src = palette; // or &palette, makes no difference
  DMA[3].dst = paletteMem;
  DMA[3].cnt = 1 << 31 | 256;

  ((unsigned char *)VIDEO_BUFFER)[0] = 255;
  ((unsigned char *)VIDEO_BUFFER)[1] = 240;
  while(1);
}
```


Mode4 - VisualBoyAdvance-M 2.1.0
File Emulation Options Tools Help
Do what now?!?

Mode 4 Pixels

- You can't write to the videoBuffer section of memory (VRAM) a byte at a time (you have to write 2 or 4 at a time).
  - You *can* read a single byte at a time, though.
- Mode 4 pixels are 8 bits each, so you have to pack two of them together into a 16 bit video buffer entry
- To set a pixel you read existing 16 bit value, combine it with a new 8 bit half, and write the 16 bits back to memory
  - So let's go back to the old `unsigned short *videoBuffer`

Bits 0-7	Bits 8-15
Even Pixels (0, 2, 4, 6, 8 ...)	**Odd Pixels (1, 3, 5, 7, 9 ...)**

Writing a Single Mode4 Pixel

- First, read the existing unsigned short value, dividing the x value by 2

```
unsigned short offset = (y * 240 + x) >> 1;
pixel = videoBuffer[offset];
```

- Next, determine whether x is even or odd and AND'ing x with 1:

```
if (x & 1)
```

- Finally, if x is even, then copy the pixel to the lower portion of the unsigned short.
  - In order to do this, you must shift the color bits left by 8 bits so they can be combined with a number that is right-aligned, like this:

    ```
    videoBuffer[offset] = (color<<8) + (pixel&0x00FF);
    ```

  - If x is odd, then copy it to the upper portion of the number, without worrying about bit shifting, like so:

    ```
    videoBuffer[offset] = (pixel & 0xFF00) + color;
    ```

Why go to all this trouble to use half the space in memory?

Surely, this Mode4 stuff is more trouble than it's worth…

Mr. Hansen sure does ask a lot of rhetorical questions...

Page Flipping / Double Buffering

0F	0E	0D	0C	0B	0A	09	08	07	06	05	04	03	02	01	00
15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
				BG3	BG2	BG1	BG0				**PS**		Mode		

- We saved a lot of space in VRAM, so we can use for something else:
  - Two frames at once!
  - One is actively being displayed
  - We draw to the other
  - When we're done drawing, with flip frames by updating REG_DISPCNT bit 4, "PS"
    - When PS is 0, the video controller uses 0x06000000 as the actively drawn frame;
    - When PS is 1, the video controller uses 0x0600a000; as the actively drawn frame;

Page Flipping / Double Buffering

- This is technically "Page Flipping", but it's very similar to "Double Buffering"
  - TONC rants a bit about the difference, but basically double-buffering involves a quick copy via something like DMA

Page Flip in Practice

```
unsigned short *FrameBuffer1 = (unsigned short*)0x06000000;
unsigned short *FrameBuffer2 = (unsigned short*)0x0600a000;

#define PS 16

void FlipPage(){
  if(REG_DISPCNT & PS){
    videoBuffer = FrameBuffer2;
  } else {
    videoBuffer = FrameBuffer1;
  }
  REG_DISPCNT ^= PS; // flip this bit every time
}
```

Mode4 Gotchas

- You can't just naïvely erase where things were last frame.
  - Because the last frame is in a different buffer entirely
  - You'd need to erase from the old buffer and draw in the new buffer (or just keep prev_x,prev_y AND prev_prev_x,prev_prev_y) and do some bookkeeping.