CS 2261: Media Device Architecture - Week 7

HANSEN – Oct. 01, 2018 – Georgia Institute of Technology

Point of Order

- From the syllabus: "Students missing an assignment without an excused absence will receive a grade of zero on that assignment"
  - This applies to Homeworks and Labs as well!
  - Late assignments are a zero, unless you have a legitimate excuse (and let us know in advance!).
- Download and verify your submissions after you submit them!
  - If you can't run them after the download, neither can we (and your grade will reflect that)!
- We've been fairly lenient, but that is coming to an end.

Overview

- Mode 4
- Mode 5
- Fixed Point

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 extra room in VRAM

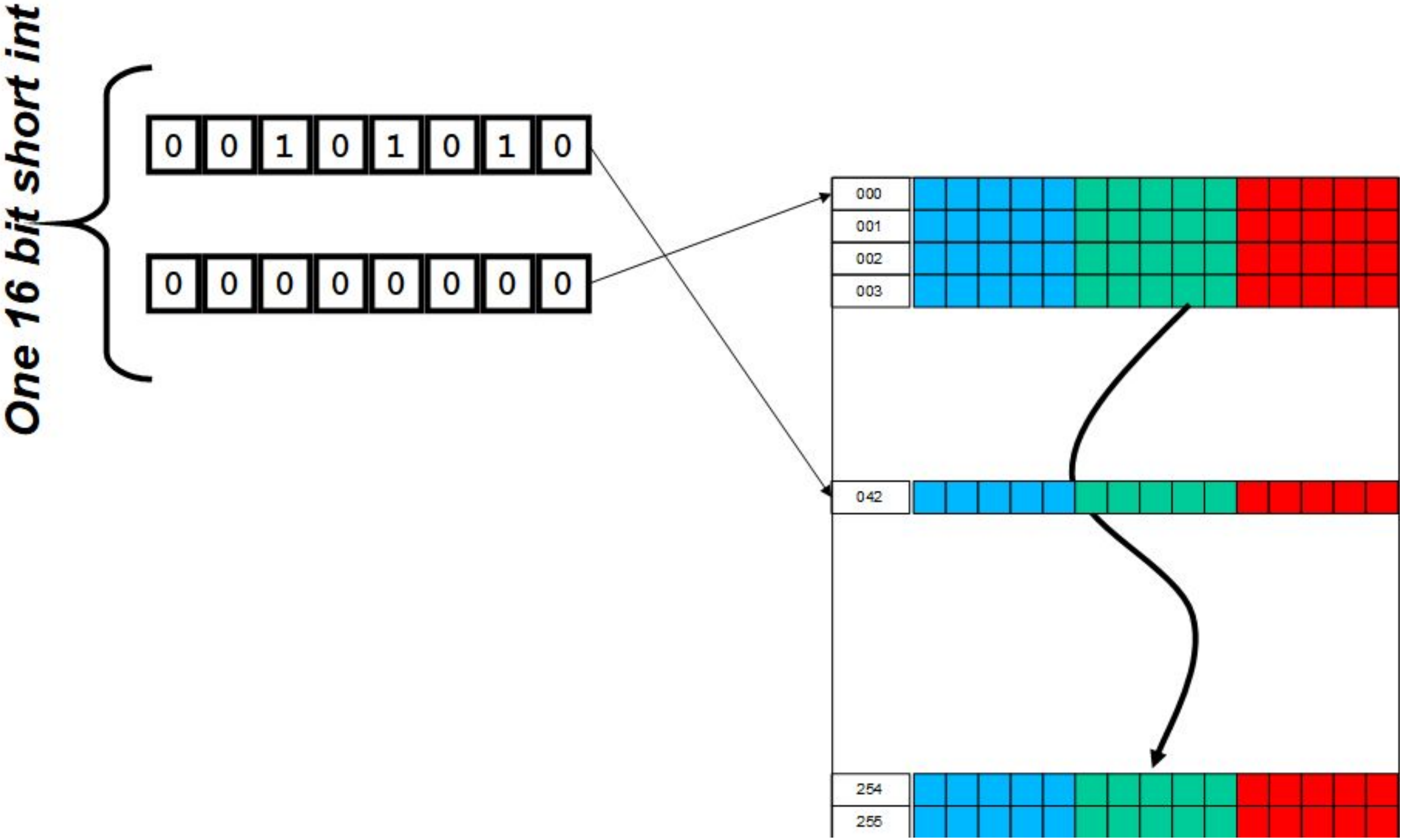
Set Mode 4

```
#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?

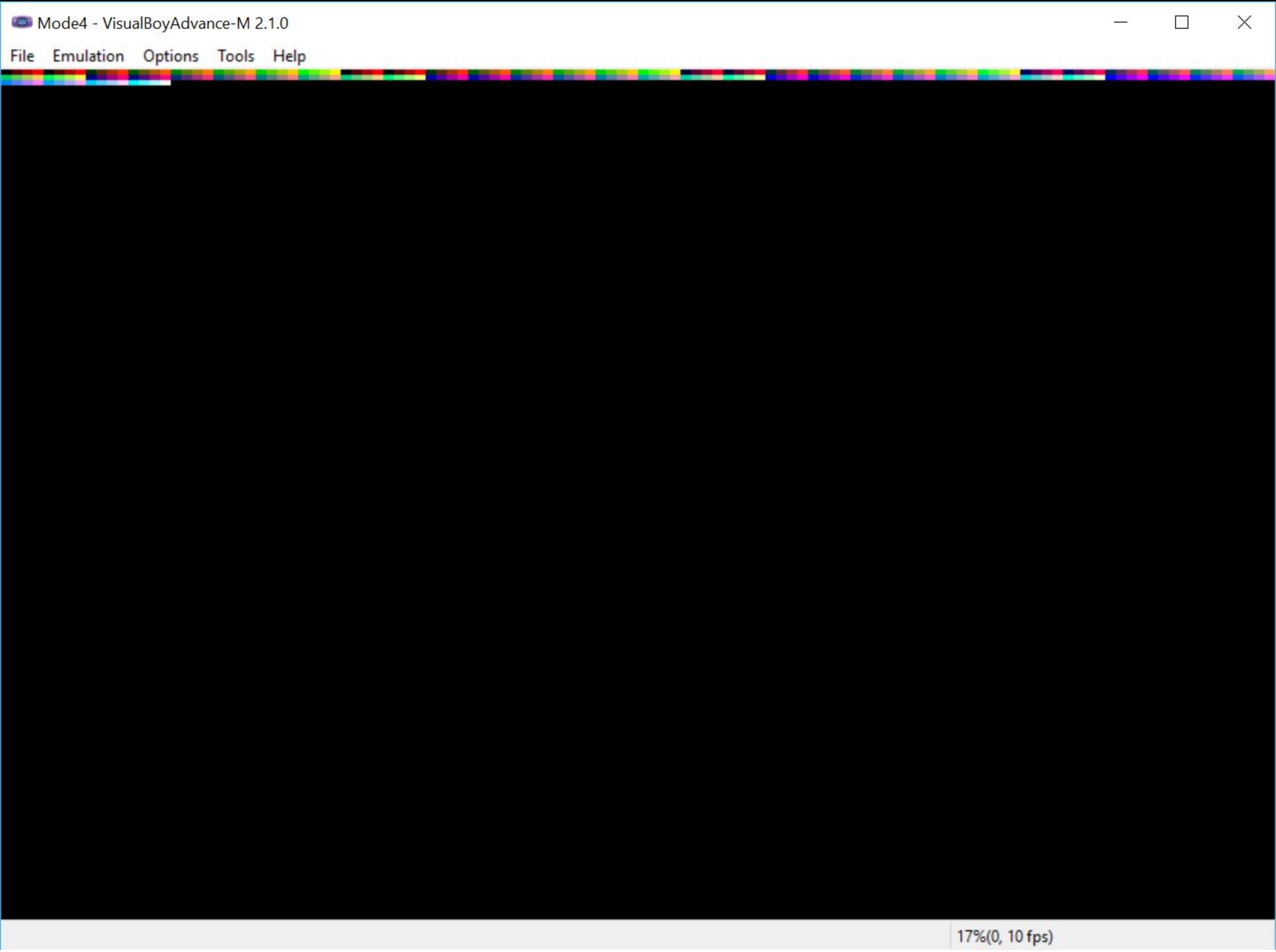
- Pixel values are now chars (0-255).
- Those don't correspond to a predetermined set of 256 colors.
  - You have to set one up!
  - Here's where it goes:
    - `unsigned short* paletteMem = (unsigned short*)0x5000000;`
  - Each color there is the same 15-bit color we know and love from Mode3.

Index Color


One 16 bit short int
0 0 1 0 1 0 1 0
0 0 0 0 0 0 0 0
000
001
002
003
042
254
255

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)

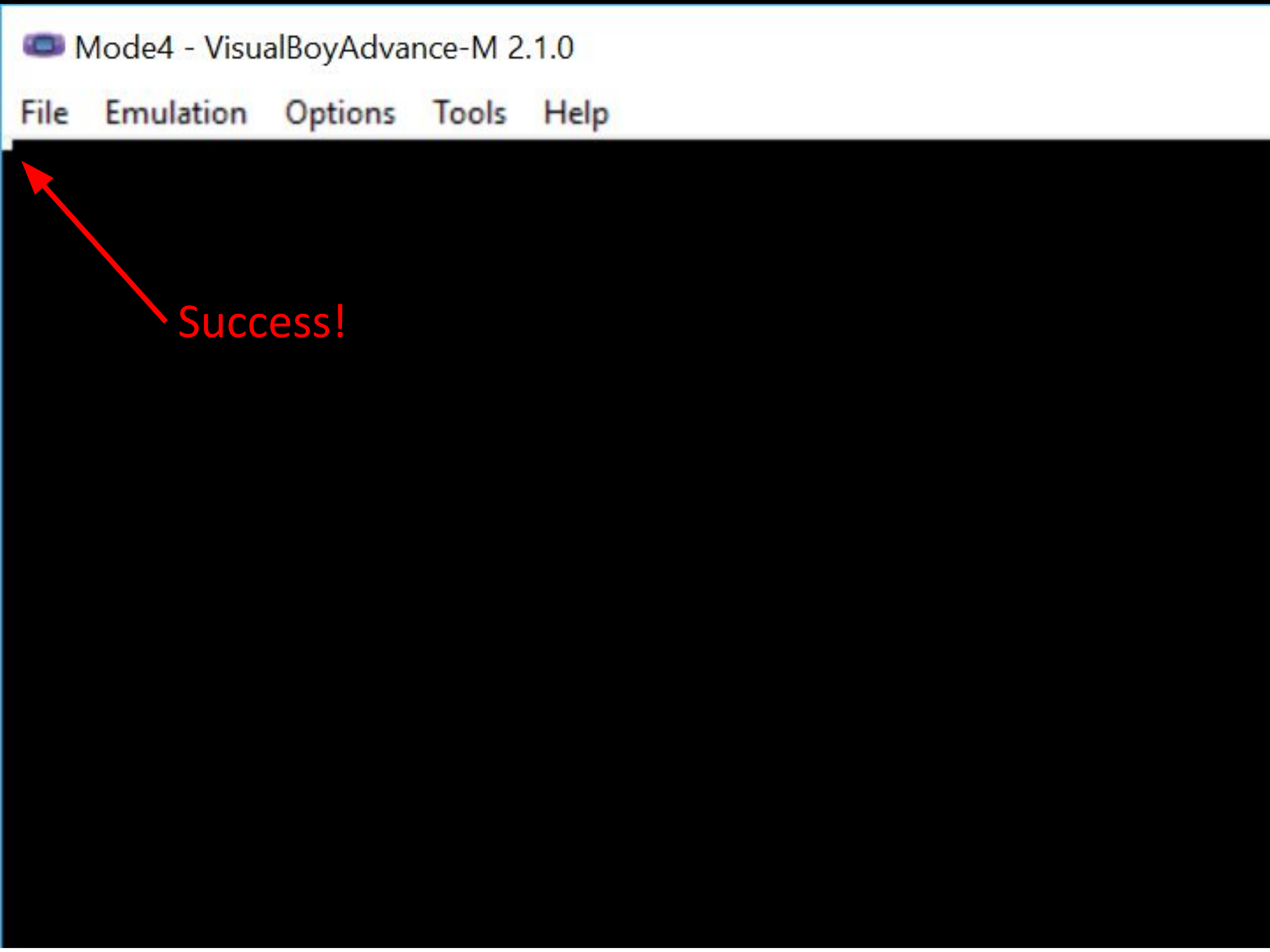
```
volatile unsigned short* paletteMem = (unsigned short*)0x05000000;

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

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);
}
```

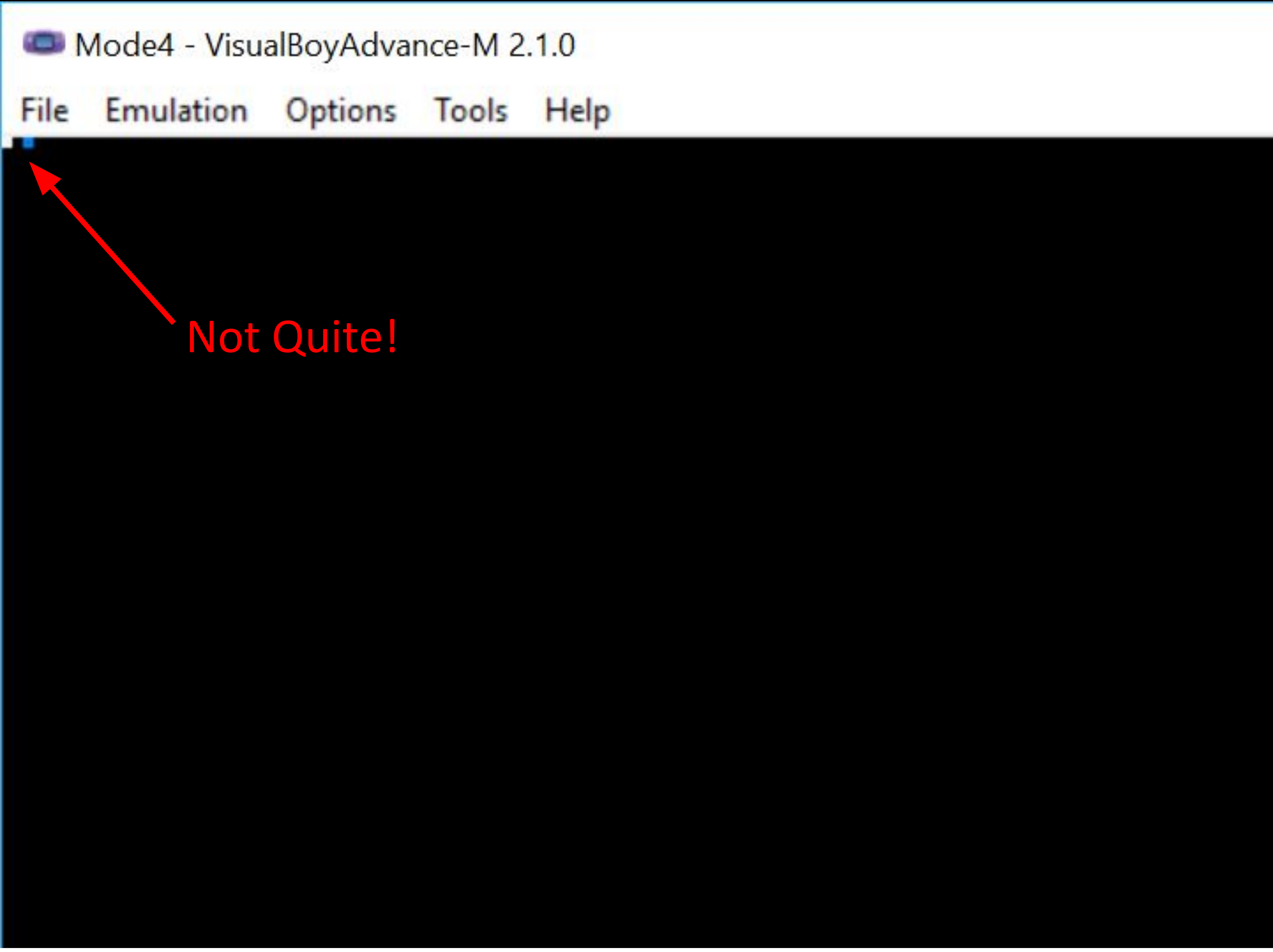


Mode4 - VisualBoyAdvance-M 2.1.0
File
Emulation
Options
Tools
Help

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);
}
```



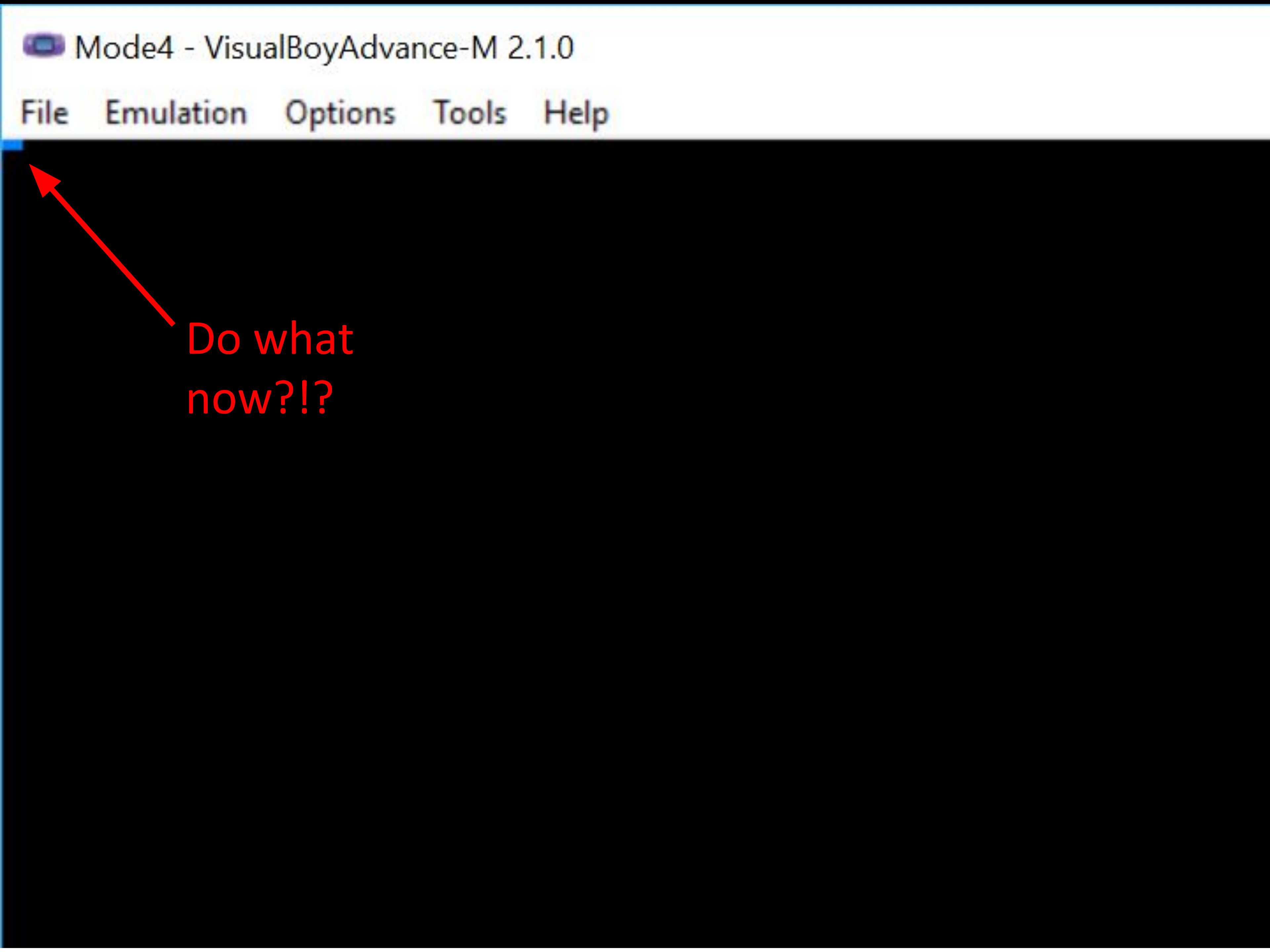
Mode4 - VisualBoyAdvance-M 2.1.0
File
Emulation
Options
Tools
Help

Not Quite!

Whoops

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

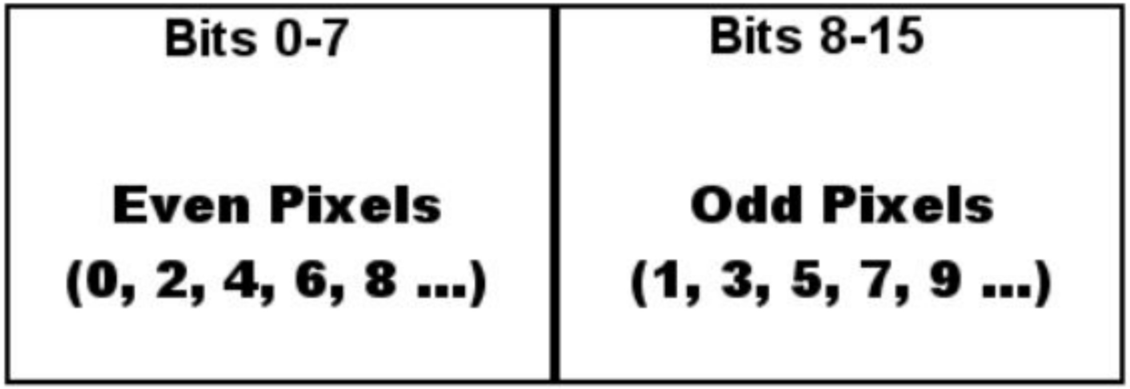
```
int main() {
  REG_DISPCNT = 4 | BG2_ENABLE;
  DMA[3].cnt = 0;
  DMA[3].src = palette; // or &palette, makes no difference
  DMA[3].dst = paletteMem;
  DMA[3].cnt = 1 << 31 | 256;

  ((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?!?

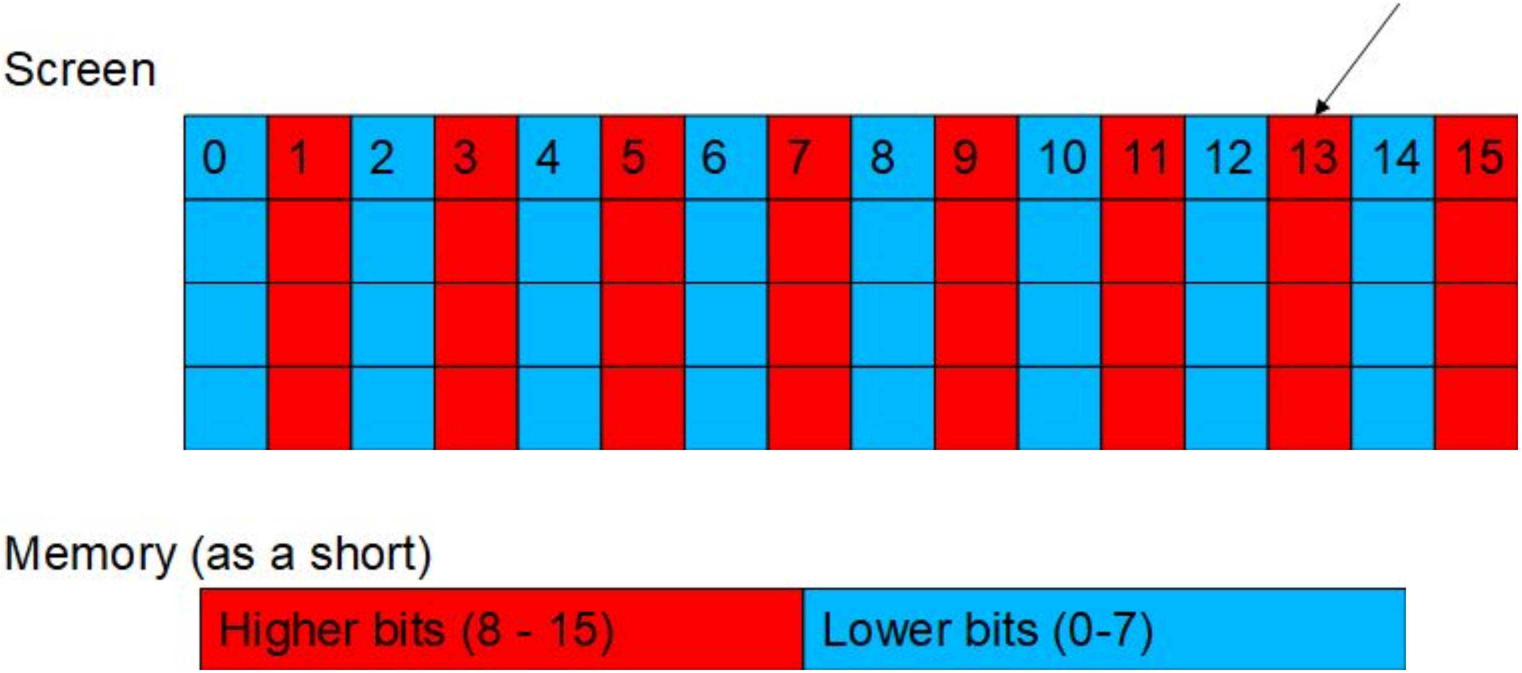
Mode 4 Pixels

- 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 ...)**

The Tricky Part



Screen

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Memory (as a short)

Higher bits (8 - 15) | Lower bits (0-7)

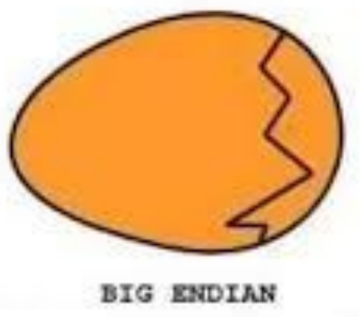
You can only write as a short, so the odd pixels are the "top" 8 bits of the short, representing their palette offset and the lower bits are the even pixel palette offset.

A note on Endianness

Endianness is the order of the bits within a byte.

When we write binary numbers, we do it in Big Endian format.

However, most architectures are actually Little Endian



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)
```

- If x is odd, then copy it to the upper portion of the number, without worrying about bit shifting, like so:
  - `videoBuffer[offset] = (color << 8) | (pixel & 0x00FF);`
- Otherwise x is even, so copy it to the lower 8 bit portion of the number, without worrying about bit shifting, like so:
  - `videoBuffer[offset] = (pixel & 0xFF00) | color;`

setPixel4(int x, int y, u8 pOffset)

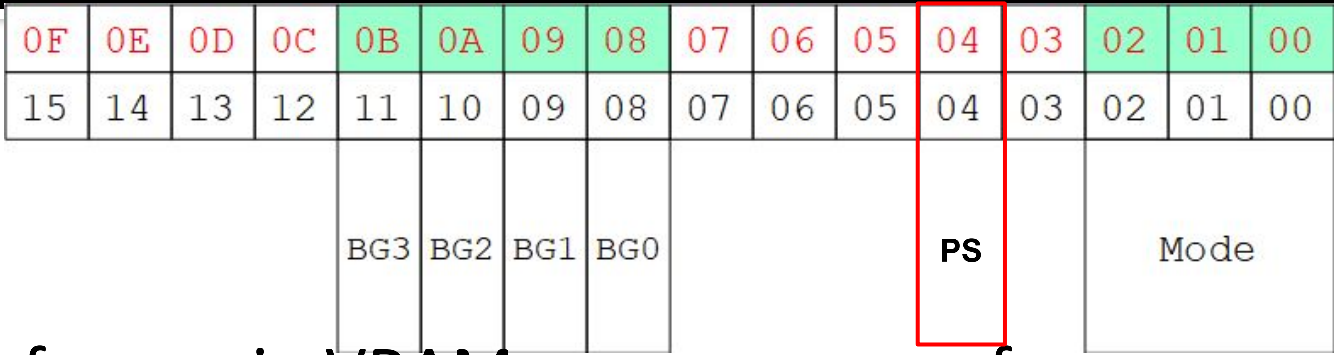
```
void setPixel4(int x, int y, u8 color){
  int offset = (x + y*240) >> 1;
  u16 originalShort = videoBuffer[offset];
  if (x & 1){
    videoBuffer[offset] = color << 8 | (originalShort & 0x00FF);
  } else {
    videoBuffer[offset] = color | (originalShort & 0xFF00);
  }
}
```

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



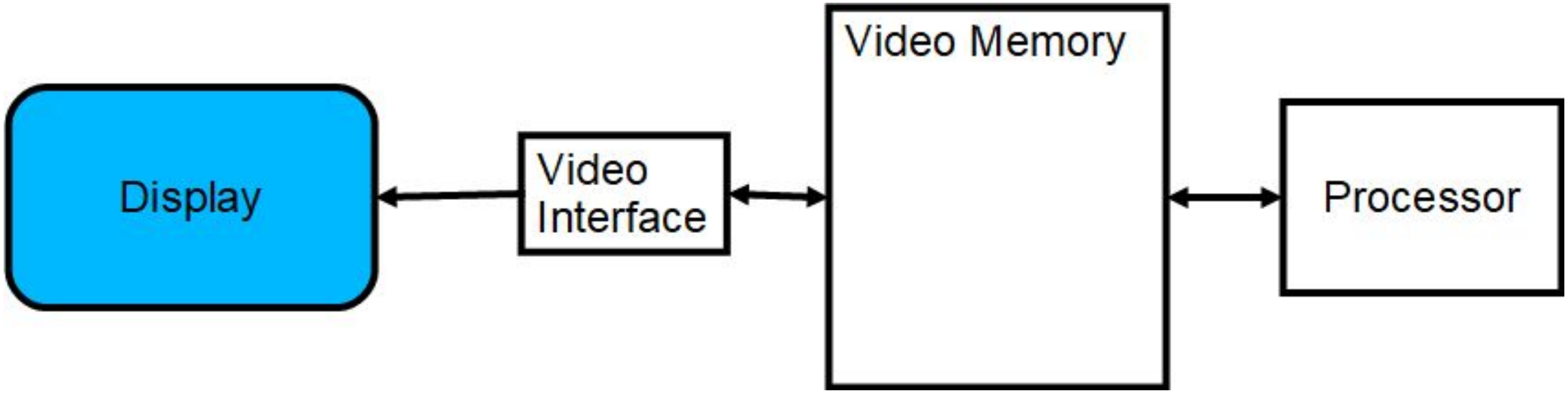
0F	0E	0D	0C	0B	0A	09	08	07	06	05	04	03	02	01	00
15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
				BG3	BG2	BG1	BG0				**PS**		Mode		

- 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, we 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

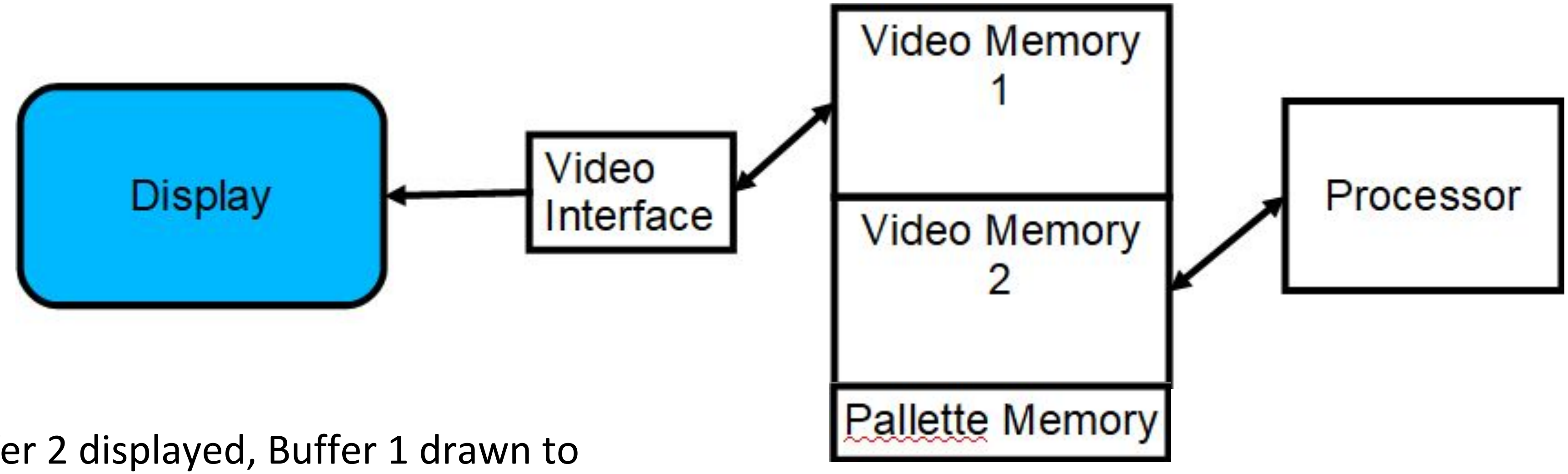
- This is technically "Page Flipping", but it's very similar to "Double Buffering"
    - TONC rants a bit about the difference, but basically double-buffering involves a quick copy via something like DMA

Mode 3

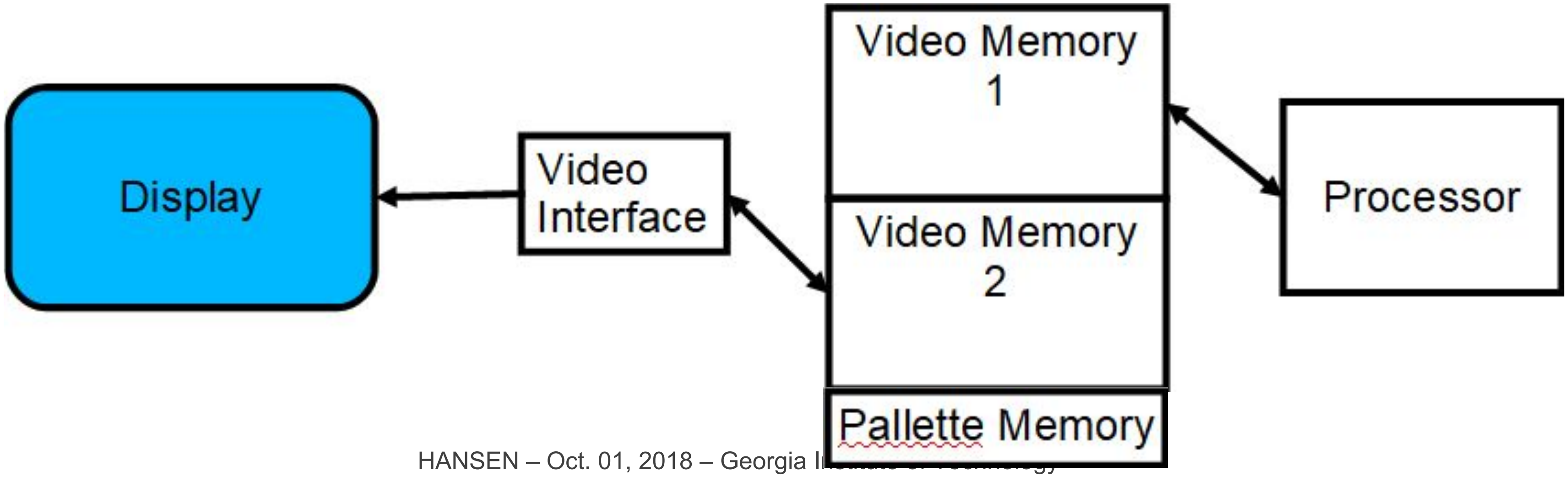


Mode 4

Buffer 1 displayed, Buffer 2 drawn to:



Buffer 2 displayed, Buffer 1 drawn to



Mode 3 vs Mode 4

Mode 3:

```
while (1) {
  calculate_postions();
  waitForVblank();
  drawStuff();
}
```

Mode 4:

```
while (1) {
  calculate_postions();
  drawStuff();
  waitForVblank();
  flipPages();
}
```

Page Flipping in Practice

```
unsigned short *FrameBuffer1 = (unsigned short*)0x06000000;
unsigned short *FrameBuffer2 = (unsigned short*)0x0600a000;
#define PS 16

void flipPages(){
  if(REG_DISPCNT & PS){
    videoBuffer = FrameBuffer2;  // videoBuffer needs to be a variable
  } else {
    videoBuffer = FrameBuffer1;
  }
  REG_DISPCNT ^= PS; // flip this bit every time
}
// Better yet!:
void flipPages(){
  videoBuffer = (u16*)(((int)videoBuffer) ^ 0xa000);
  // now I don't need the FrameBuffer1/2 ptrs
  REG_DISPCNT ^= PS;
}
```

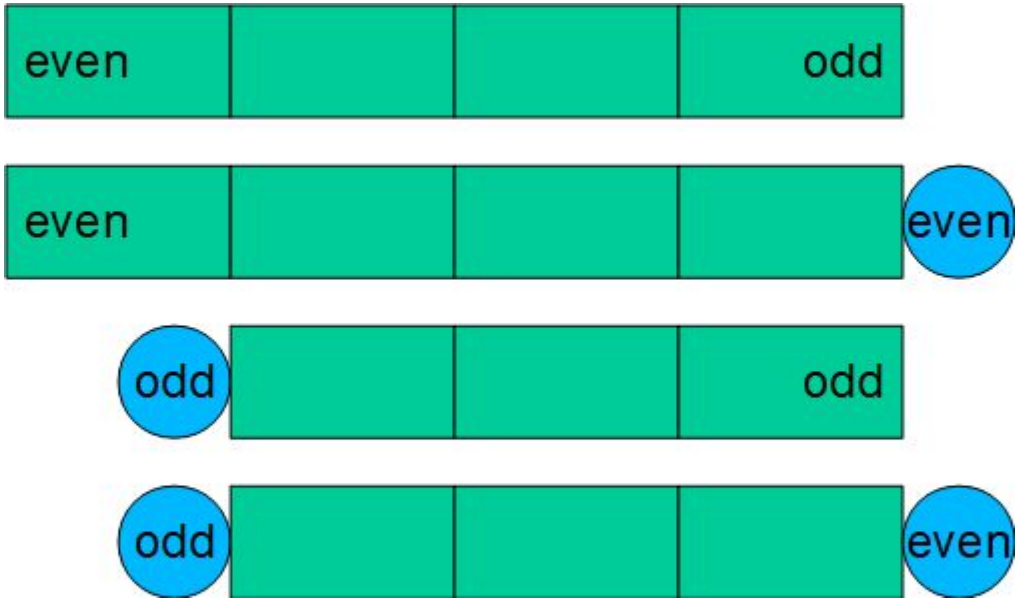
Mode4 Gotchas

- 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.

- Drawing rectangles is going to be a little more complicated (at least, doing it quickly -- see next slide)

Mode 4 and DMA

- Recall that DMA works with 16 or 32 bit chunks
- How does that impact using DMA to fill rectangles in Mode 4
- Now have 4 possibilities when drawing some pixels in a row



Modifying the palette on the fly

- Since you're working with a palette, and not a fixed color space per-pixel. You can change all the pixels using the same pallet color all at once.

- Degenerate example: setting the whole palette back to black is a quick way to blank out the screen.
  - 256 changes instead of thousands!

- It can also be useful to have two instances of the same color in your palette, to be able to change groups of pixels together, logically.

Mode 5

- Color depth of mode 3 plus the double buffering of mode 4
- Smaller screen size of 160x128 [40kiB per frame]

```
drawPixel(int x, int y, u16 color)
   videoBuffer[x + y * 160] = color
```

- Page flipping works exactly the same way (and at the same two locations in VRAM) as Mode4

GBA Bitmap Video Modes Summary

- 3 Bitmap modes: 3-5
  - 3: 240x160 resolution, 15 bit color
    - 32k colors on screen at a time
  - 4: 240x160, palette, double buffer
    - 256 colors at a time, smoother drawing
    - palette at 0x05000000
    - second frame starts at 0x0600A000
  - 5: 160x128, 15 bit color, double buffer
    - 32k colors, smoother drawing, smaller screen
    - second frame starts at 0x0600A000

A Dilemma

- Typical Object Movement:

```
int x = 120;
int dx = 1;
while(TRUE) {
  if(button(right)) {
    x = x + dx;      // plus limit check...
  }
  if(button(left)) {
    x = x - dx;      // plus limit check...
  }
}
```

  - But what if this is too slow?
    - int dx = 2;
  - But what if that ends up too fast?!

Splitting the Difference

- How about 1.5?

```
float x = 120.0;
float dx = 1.5;
while(TRUE) {
  if(button(right)) {
    x = x + dx;     // plus limit check...
  }
  if(button(left)) {
    x = x - dx;     // plus limit check...
  }
  // Draw object at (int)x
}
```

  - This might work out at first
    - Or it might bring your whole game to a crawl (depending on how much you're doing this).

Floating Point is generally too slow on the GBA

- There's no hardware support for it, so when you use it on the GBA, it is implemented in software.
  - That's ***much*** slower than dedicated hardware!

- So what if we multiplied everything by 10, and then just divided out every time we went to use it?
  - So to represent 121.5, we'd use 1215.
  - Simple multiplying and dividing by 10, no?

```
int x = 120 * 10;
int dx = (int)(1.5 * 10);
while (1){
  // x + dx, y - dx -- same as before
  // Draw object at x/10
}
```

But division can be kinda slow too, plus we like powers of two

- Instead of multiplying and dividing by 10 we will use 8
  - Old system a 1 represented 0.1
  - New system a 1 represents 0.125 (1/8th)

- Now we replace *10 and /10 by *8 and /8 except we write: <<3 and >>3
  - (though compiler optimization would also do this for us)

```
int x = 120 << 3;
int dx = (int)(1.5 * 8); // don't bit-shift here (it won't even compile)!
while (1){
  // x + dx, y - dx -- same as before
  // Draw object at x >> 3
}
```

This is an example of a Fixed Point encoding

- We do all our calculations using the internal encoding.
- If we need to use the number for something like drawing we convert to external encoding.

Our encoding is free to do any amount of shifting necessary

- We chose shifting by 3, because it was close to dividing/multiplying by 10.
  - Something like: 1111 1111 1111 1.111 (so the higher bits are whole numbers, and the lower bits are ½, ¼, ⅛.

- We could choose other options:
  - Ex: 1111 1111.1111 1111 (shifting everything by 8 bits)
    - Now the lower bits go all the way down to 1/256!

- What fixed-point encoding you choose is all about the level of precision you need, and the highest value you want to represent.