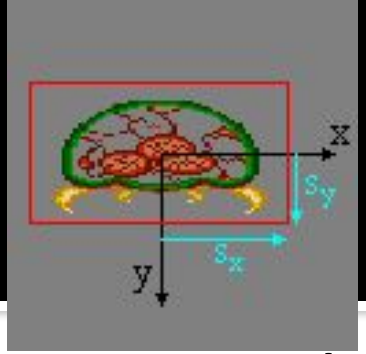
# CS 2261: Media Device Architecture - Week 10, part 2

# Overview

- Affine Sprites
  - Some Math
  - The Affine Matrix
  - 24.8 Fixed Point
  - LUT's
  - Affine Sprite Attributes
- Interrupts
- Function Pointers

# Scaling Transformation




If I have a set of (x,y) coordinates, and I want to scale that shape to a new set of (x',y') coordinates, I need to multiply each x and y by that scaling factor.

x' = x * S

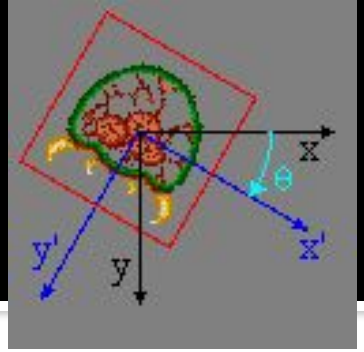
y' = y * S

Or, in matrix form: $\begin{bmatrix} x' \\ y' \end{bmatrix} = S \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$

More generally, you could scale X and Y separately:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} Sx & 0 \\ 0 & Sy \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

# Rotation Transformation




Say, for every (x,y) pair, I want to apply a rotation to some new (x', y'): $\begin{bmatrix} x' \\ y' \end{bmatrix} = R(\theta) \begin{bmatrix} x \\ y \end{bmatrix}$

For a counterclockwise rotation, the given matrix will perform the desired rotation: $\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

For a clockwise rotation, replace $\theta$ with $-\theta$:

$$R(-\theta) = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

# Combining Both (via Matrix Multiplication):

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = R(\theta)\, S(s_x, s_y) \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x \cos\theta & s_y \sin\theta \\ -s_x \sin\theta & s_y \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

# The Affine Matrix

That combination is known as the affine matrix:

$$\begin{bmatrix} Sx*\cos\theta & Sy*\sin\theta \\ -Sx*\sin\theta & Sy*\cos\theta \end{bmatrix}$$

However, the GBA uses the inverse of that matrix:

$$\begin{bmatrix} \cos\theta/Sx & -\sin\theta/Sx \\ \sin\theta/Sy & \cos\theta/Sy \end{bmatrix}$$

Because the GBA maps the screen space to the tiles (and not the other way around)
(this rotation is C.C.W.)

# 24.8 Fixed Point Numbers

Affine matrix values on the GBA are given in 24.8 fixed point encoding.

They are signed integers, but with precision of 1/256, so they are bit-shifted << 8 when encoding in this format.

1 represents 1/256
128 represents .5
256 represents 1.0
etc.

# Look-Up Tables (LUTs)

- Sine and Cosine are ***really*** expensive functions to be running on the GBA.
- So we calculate them all in advance, so we can just look them up when we need them (here, with Python):

```
import math
print("// sin(x) for 0 to 360 deg")
x = 0
dx = 2*math.pi/360
print("const int sin_lut_fixed8[] = {")
for i in range(360):
    print("  " + str(int(256*math.sin(x))) + ",  // " + str(i))
    x += dx
print("  " + str(int(256*math.sin(x))) + "  // " + str(360))
print("};")
```

# Look-Up Tables (LUTs)

- Sine and Cosine are ***really*** expensive functions to be running on the GBA.
- So we calculate them all in advance, so we can just look them up when we need them (here, with Python):

```
import math
print("// sin(x) for 0 to 360 deg")
x = 0
dx = 2*math.pi/360
print("const int sin_lut_fixed8[] = {")
for i in range(360):
    print("   " + str(int(256*math.sin(x))) + ",  // " + str(i))
    x += dx
print("   " + str(int(256*math.sin(x))) + "  // " + str(360))
print("};")
```

Here, we are encoding it into the 24.8 format in advance:

# Look-Up Tables (LUTs)

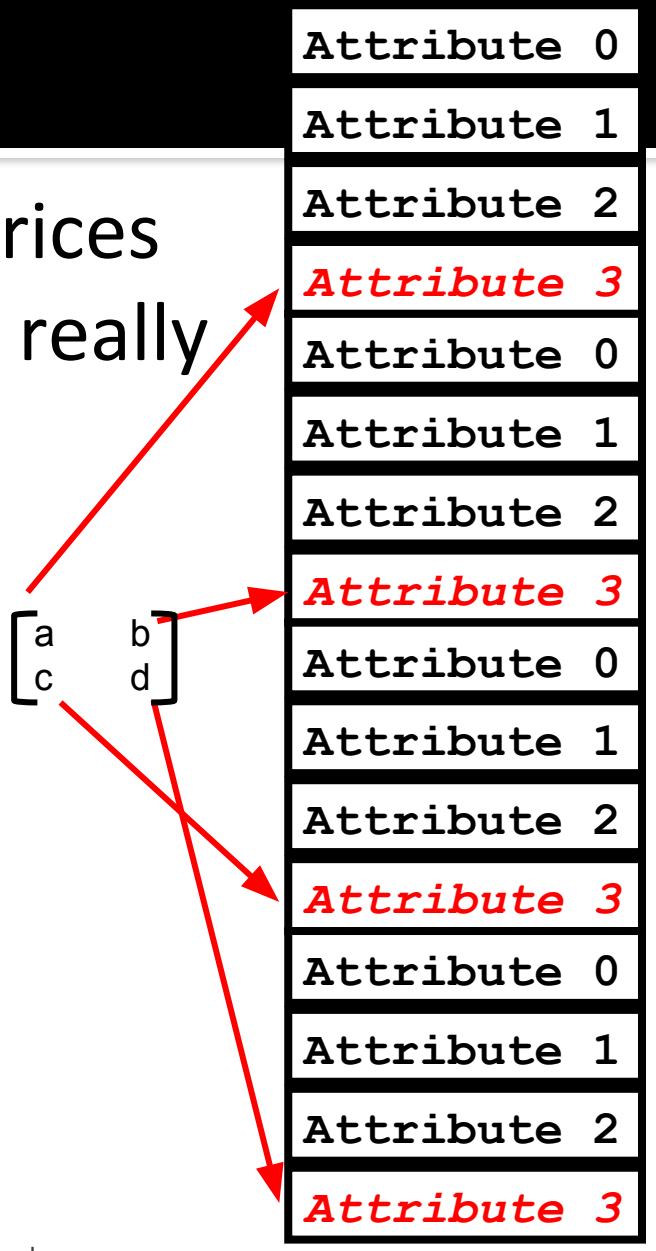
- Now, to get sin(x) in our 24.8 fixed encoding, we just do:
  - `sin_lut_fixed8[deg % 360]`
- We could also make one for cosine, or we could remember the following from Trigonometry class:
  - `cos(x) = sin(x + π/2)  // aka sin(x + 90°)`
- This way, we can use one LUT for both.
  - If you needed every ounce of speed, you might want to go ahead and make one for cosine anyways, to avoid a few minor calculations.

# Using the Affine Matrix on the GBA

There are 32 places for affine matrices to go on the GBA, and they're in a really odd place:

Remember this?:

```
typedef struct
{
  u16 attr0;
  u16 attr1;
  u16 attr2;
  short fill;
} ALIGN(4) OBJ_ATTR;
```



# The affine parameters are in the Attribute 3 slots of the OAM

- Since there are 128 sprite locations, and each one has ¼ of an affine matrix, there are 32 affine matrix "locations" (if we're a little clever about it)
  - ```
    typedef struct
    {
        u16 fill0[3];  // leave gaps for the OAM attrs
        s16 a;         // make sure they're signed (-sin(x), etc.)!
        u16 fill1[3];
        s16 b;
        u16 fill2[3];
        s16 c;
        u16 fill3[3];
        s16 d;
    } ALIGN4 OBJ_AFFINE;

    OBJ_AFFINE *SHADOW_OAM_AFF=(OBJ_AFFINE*)SHADOW_OAM;
    ```
    ```

# Setting up your Affine Sprite

OAM Attribute 0:

| 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 |
| Shape | | 256 | | | α | OM | | Row Location | | | | | | | |

- Bits 0-7: Row location (Top of Sprite)
- Bits 8-9: Object Mode:
  - 00: Regular, 01: Affine, 10: Hide, 11: "Double" Affine
- Bit 10: Enable alpha blending
- Bit 13: 16 colors if cleared, 256 colors if set
- Bits 14-15: Sprite shape:
  - 00: Square, 01: Wide, 10: Tall

# Setting up your Affine Sprite (continued)

OAM Attribute 1:

| 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 |
| Size | | Flip | | | | | Column Location | | | | | | | | |
| | | Affine Index | | | | | | | | | | | | | |

- Bits 0-8: Column location (Left side of sprite)
- Bits 12-13: Horizontal/Vertical Flip (regular mode)
  - 00: No Flip, 01: Horizontal Flip, 10: Vertical Flip
- Bits 9-13: Affine Index (0-31) (affine mode)
- Bits 14-15: Sprite Size
  - 00: 8 pixels, 01: 16, 10: 32, 11: 64

# Set the Affine Matrix to *Something*

- If the affine matrix you're using is all zeros, the entire sprite on the screen will be filled with the top-left pixel of your sprite.

- Setting it to the identity matrix is a good start
  - ```
    SHADOW_OAM_AFF[0].a = 256; // 1 in .8 fixed
    SHADOW_OAM_AFF[0].d = 256;
    ```
    ```

# Demo (with some gotchas)

# Demo Takeaways

- Forgetting to set Affine values is not a good idea.
- If rotating, you might want the "double" affine, but getting it aligned properly might take some tweaking (because if your sprite is off-centered, it will make it even worse).

# Interrupts

- An interrupt causes the CPU to stop what it is currently doing and execute another piece of code (process the interrupt)
- There are two kinds of interrupts
  - Software interrupts – used to call low-level code that is typically provided by the OS or BIOS – called “software interrupt” because the interrupt is initiated in software. These are also sometimes called traps or system calls.
  - Hardware interrupts – triggered by hardware events – used to let the CPU respond to asynchronous hardware events
- We’re only looking at hardware interrupts

# Processing Interrupts

- The CPU saves the current value of registers
- The CPU starts executing code at a special location, the interrupt entry point
- When the interrupt code is finished, the CPU restores the registers and goes back to the code it was in the middle of working on when it was interrupted
- We'll only worry about processing one interrupt at a time (not processing interrupts while we're interrupted), because we disable interrupts while one is being handled.

# GBA Interrupts

- You use `REG_IE` (interrupt enable register) - `0x04000200` to to tell the GBA which events you want to be interrupted by. The you read the bits of `REG_IF` - `0x04000202` (the interrupt fired register), to see which fired.

| Bit | Description | Note |
|---|---|---|
| 0 | VB – vertical blank interrupt | Also requires REG_DISPSTAT bit 3 |
| 1 | HB – horizontal blank interrupt | Also requires REG_DISPSTAT bit 4 |
| 2 | VC – vertical scanline count interrupt | Also requires REG_DISPSTAT bit 5 |
| 3 | T0 – timer 0 interrupt | Also requires REG_TM0CNT bit 6 |
| 4 | T1 – timer 1 interrupt | Also requires REG_TM1CNT bit 6 |
| 5 | T2 – timer 2 interrupt | Also requires REG_TM2CNT bit 6 |
| 6 | T3 – timer 3 interrupt | Also requires REG_TM3CNT bit 6 |
| 7 | COM – serial communication interrupt | // we're not using this at all |
| 8 | DMA0 – DMA0 finished interrupt | Also requires REG_DMA0CNT bit 30 |
| 9 | DMA1 – DMA1 finished interrupt | Also requires REG_DMA1CNT bit 30 |
| 10 | DMA2 – DMA2 finished interrupt | Also requires REG_DMA2CNT bit 30 |
| 11 | DMA3 – DMA3 finished interrupt | Also requires REG_DMA3CNT bit 30 |
| 12 | KEYPAD – keypad interrupt | Also requires REG_KEYCNT bit 14 |
| 13 | CART – game cartridge interrupt | Raise when cartridge is REMOVED! |
| 14-15 | unused | |

# Steps for setting up an interrupt

1. Disable all interrupts
2. Set the interrupts you want to pay attention to
3. Set the subordinate interrupt control register for your specific interrupts
4. Set up the entry point for your interrupt handling routine
5. Re-enable interrupts

# REG_IME

- The interrupt master enable register – the master switch for all interrupts
- When its value is 0, all interrupts are turned off
- When its value is 1, interrupts can happen (but only the ones you've specifically enabled)
- `#define REG_IME *(u16*)0x4000208`

# REG_IE

- The interrupt enable register
- Set specific bits in this register to enable specific interrupts
- The meaning of the different bits is defined a couple of slides before
- `#define REG_IE *(u16*)0x4000200`

# Specific interrupt control registers

- Each specific interrupt type has its own control register
- Display interrupts (VB, HB, VC) use the DISPSTAT (display status) register
- Each DMA channel as a REG_DMAxCNT (e.g. DMA0 control) register
- Each timer has a REG_TMxCNT (e.g. timer 0 control) register
- There are also control registers for the button (REG_KEYCNT) and serial communication interrupts

# REG_DISPSTAT

- The REG_DISPSTAT register is used both to control display related interrupts and indicates display status

| Bit | Description |
|---|---|
| 0 | VB – vertical blank is occurring |
| 1 | HB – horizontal blank is occurring |
| 2 | VC – vertical count reached |
| 3 | VBE – enables vblank interrupt |
| 4 | HBE – enables hblank interrupt |
| 5 | VCE – enables vcount interrupt |
| 6-15 | VCOUNT – vertical count value (0-159) |

# Interrupt Handler

- When an interrupt happens, the GBA starts executing the code whose address is stored at location 0x03007FFC (REG_INTERRUPT)
- That means that, 0x03007FFC is the location of a *pointer* that points to *code*

```
#define REG_INTERRUPT *(u32*)0x03007FFC
REG_INTERRUPT = (u32)interruptHandler;
```

# Point to code?: Function Pointers

In C symbols often represent addresses

```
int a;
short b[10];
struct {
    int x;
    int y;
} point1;
// etc.
```

# Function Pointers

```
int someFunction(int arg1, int arg2)
{
  return arg1 + arg2;
}
```

What is "someFunction"?

# Function Pointers

- In C the name of a function is a symbol whose value is the memory address.
- We are "cheating" when we say:

```
#define REG_INTERRUPT *(u32*)0x03007FFC
REG_INTERRUPT = (u32)interruptHandler;
```

- It works and for our purposes, but if you are going to be a C programmer you should know there is a right way™.

# Function Pointers

```
typedef void (*ihp_t)(void);

#define REG_INTERRUPT *((ihp_t *)0x03007FFC)
REG_INTERRUPT = interruptHandler;
```


# Constants

```
//primary interrupt locations
#define REG_IME *(u16*)0x4000208
#define REG_IE *(u16*)0x4000200
#define REG_IF *(volatile u16*)0x4000202
#define REG_INTERRUPT *(u32*)0x3007FFC
#define REG_DISPSTAT *(u16*)0x4000004


//interrupt constants for turning them on
#define INT_VBLANK_ENABLE 1 << 3


//interrupt constants for checking which type of interrupt
  happened
#define INT_VB 1 << 0
#define INT_BUTTON 1 << 12
```

```
//interrupt constants for checking which type of interrupt happened
#define INT_VB     1 <<  0 // VB - vertical blank interrupt
#define INT_HB     1 <<  1  // HB - horizontal blank interrupt
#define INT_VC     1 <<  2 // VC - vertical scanline count interrupt
#define INT_T0     1 <<  3 // T0 - timer 0 interrupt
#define INT_T1     1 <<  4 // T1 - timer 1 interrupt
#define INT_T2     1 <<  5 // T2 - timer 2 interrupt
#define INT_T3     1 <<  6 // T3 - timer 3 interrupt
#define INT_COM    1 <<  7 // COM - serial communication interrupt
#define INT_DMA0   1 <<  8 // DMA0 - DMA0 finished interrupt
#define INT_DMA1   1 <<  9 // DMA1 - DMA1 finished interrupt
#define INT_DMA2   1 << 10 // DMA2 - DMA2 finished interrupt
#define INT_DMA3   1 << 11 // DMA3 - DMA3 finished interrupt
#define INT_BUTTON 1 << 12 // BUTTON - button interrupt
#define INT_CART   1 << 13 // CART - game cartridge interrupt
```

# Setup

```
void setupInterrupts(void) {
  REG_IME = 0x0; //disable interrupts
  REG_INTERRUPT = (u32)interruptHandler; //set int handler
  enableVBlankInterrupt();
  REG_IME = 0x1; //enable interrupts
}
void enableVBlankInterrupt() {
 REG_IE |= INT_VB; // Enable vblank interrupt detection
 REG_DISPSTAT |= INT_VBLANK_ENABLE; // Make display generate
                                    // vblank interrupts
}
```

# Interrupts for a Precise Update Rate

```
void interruptHandler(void) {
  REG_IME = 0; //disable interrupts
  // Check which event happened, and do something if
  // you care about it
  if (REG_IF == INT_VBLANK) {
    // A vblank happened, call appropriate response
    // function (one update)
    gameLoop();
  }
  REG_IF = REG_IF;  // Tell GBA that interrupt has
                    // been handled
  REG_IME = 1;      //enable interrupts
}
```

# MyHandler (draw pixel on vertical blank interrupt)

```
void MyHandler() {
   u16 x, y; // declare 2 unsigned shorts for x,y location

   REG_IME = 0x00;              //disable interrupts

   sprintf(str, "%d", i++);    // Print out count of MyHandler calls
   Print(0, 0, str, 0xFFFF);

   if(REG_IF == INT_VBLANK)  { //look for vertical refresh interrupt
       x = rand() % 240;         //draw a random pixel
           y = rand() % 160;
       DrawPixel3(x, y, RGB(rand()%31, rand()%31, rand()%31));
   }

   REG_IF = REG_IF; // What's this line for?

   //enable interrupts
   REG_IME = 0x01;
}
```

# REG_IF

- REG_IF has a bit set for each of the interrupts that have occurred and have not yet been handled
- You tell the hardware that you've handled an interrupt by setting the bit in REG_IF corresponding to the interrupt you've handled
- Assuming there's only one outstanding interrupt (we won't let ourselves be interrupted while handling an interrupt), this is equivalent to setting REG_IF to itself!
  - From the point of view of changing variable values, a = a does nothing
  - Setting REG_IF = REG_IF writes to REG_IF which clears the bit acknowledging the interrupt has been handled
  - Warning: The comments in the book imply that what's going in is you're backing up and restoring REG_IF. No!
- If you don't set REG_IF to itself, the interrupt handler will be called endlessly for the same event

# Some things to think about

- How would we use the button interrupt to look at button values?
  - Input via interrupts instead of polling.
- How could we use the vblank interrupt with page flipping?
- How could we use the hblank interrupt to display more than 256 colors on the screen at a time in mode 4?
  - What are the limitations of this approach?