

CS 2261: Media Device Architecture - Week 3

Discussion Portal

I've gotten feedback that Piazza is better/easier to manage/more familiar to some than the Discussions feature in Canvas.

Thoughts in switching? (I don't want to try to use both)

I can switch the config on Canvas to present only Piazza instead of the Discussions tab if there's a strong consensus to switch.

Overview

- Quick look back at the completed pointer arithmetic example code. (Lecture4LiveCodingExample-PointerArithmetic-Expanded.c)
- Quick “cuddling” note about function macro syntax
 - Also how-to “bake in” a dereference operation
- Basics of animation in Mode3
 - Vertical Synchronization (Vsync)
 - volatile keyword
 - Bouncing Robiclar Rectangle
- Collision Detection
- Input Basics

Function Macros Addendum

- You there can be no space between the function macro name, and the argument list parentheses:
 - `define ADD_V1(x, y) ((x) + (y)) // Good`
 - `define ADD_V2 (x, y) ((x) + (y)) // Bad!`
- `int x = ADD_V1(2, 3);`
`= ((2) + (3)); // looks good`
- `int y = ADD_V2(3, 4);`
`= (x, y) ((x) + (y))(3, 4);`
`// syntax errors and unknown variables`

Baking in a Dereference Within a Macro

```
#define REG_DISPCTL (*(unsigned short *)0x4000000)
#define MODE3 3
#define BG2_ENABLE (1<<10)

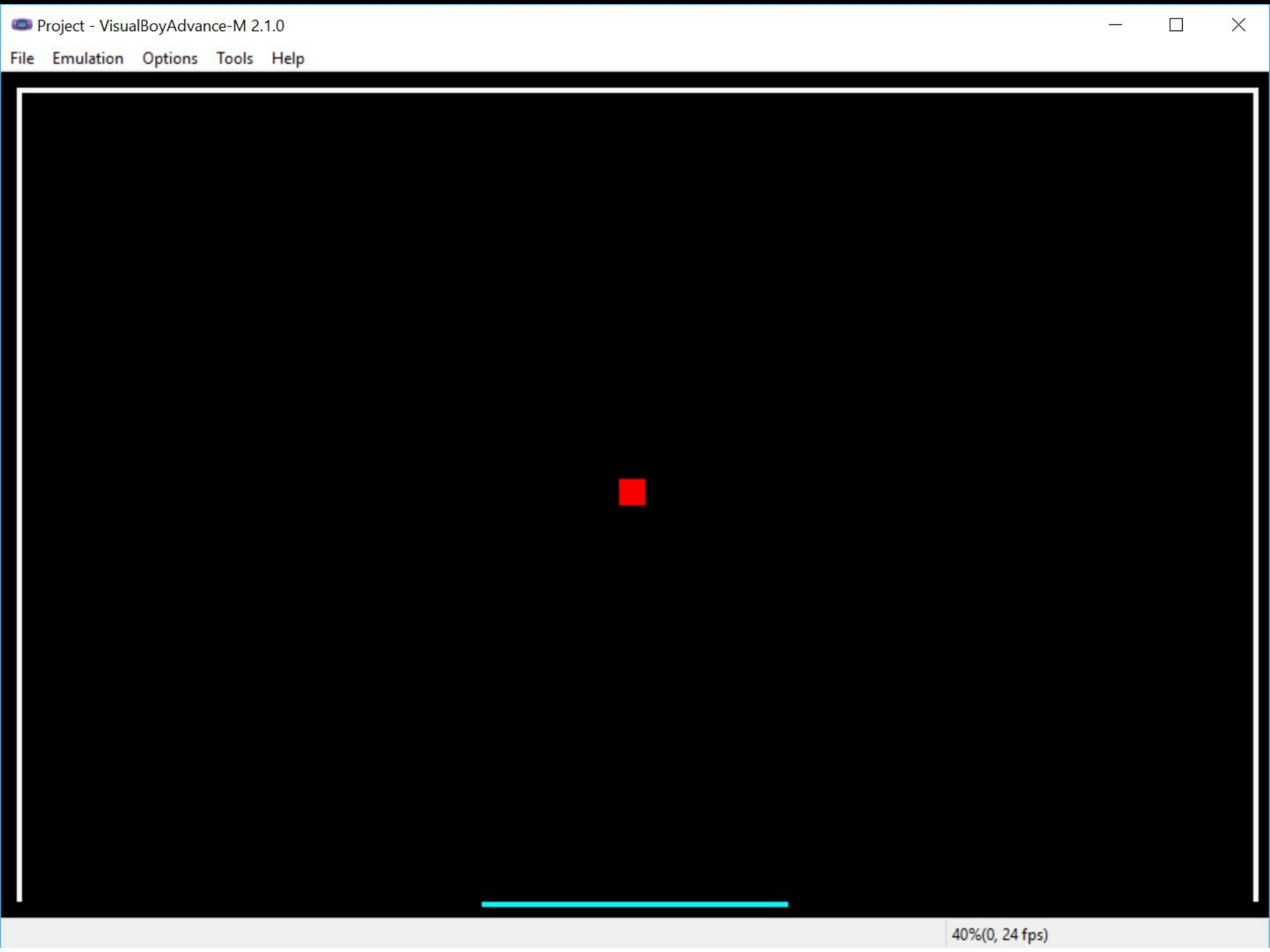
int main() {
    REG_DISPCTL = MODE3 | BG2_ENABLE;
    // same as
    (*(unsigned short *)0x4000000) = 3 | 1024;

    //... do something else?

    while(1){}
    return;
}
```

We can almost make Brickless BreakOut/BrickOut?

- Graphics
 - 3 lines for walls -- we know how to do that!
 - bouncing square for ball -- we have the square part already!
 - paddle line that moves on input -- oh yes, input!
- Game loop logic
 - Move ball a bit, turn it around if it hits a wall or the paddle
 - Move paddle a bit up if up arrow is pressed, or down if down arrow is pressed
 - Lose if ball goes off bottom of screen



What else does the “game” need?

- Constant update rate, so movement speed is predictable and stable
- Listen to User Input
 - Update state based on it

Vertical Synchronization (VSync) on the GBA -- The easiest way.

- The video controller is friendly, and lets us know what horizontal line it is currently working on drawing.
 - Each horizontal line is 240 pixels, and there are 160 of them.
 - When it is done drawing a line, it pretends to draw a few dozen more pixels (68px) on that line we can't see
 - This is called the HBlank period -- and it's pretty short
 - When it is done drawing the visible lines, it pretends to draw a few dozen more lines (68) past the bottom
 - This is called the VBlank period (and each fake line has the extra fake pixels to the right as well)

Vsync Continued

GBA Framerate:
59.73 Hz

- To prevent updating a line while it is being drawn (which can cause screen tearing), we need to only draw pixels during a VBlank
 - This is a decently useful amount of time (83k cpu cycles, per Tonc).
- To do this, we need to listen to the REG_VCOUNT (0x04000006).
 - This register stores the current line number being drawn to the screen (including the fake lines)



Wait for Vertical Blank

```
// holds the pixel row currently being drawn to the screen  
volatile u16* scanlineCounter = (volatile u16*) 0x04000006;  
// oddly 16 bits, but only uses 8 range: [0, 227]
```

■ Why volatile?

- Value in scanline counter is changed by hardware
- The compiler tries to optimize your code by only re-checking the value in a variable if your code has changed it
- The volatile keyword tells the compiler the value might change, so it always needs to check the value when your code tries to use it.

Aside about volatile syntax

```
volatile int foo; // preferred syntax  
int volatile foo;
```

- Both lead to a volatile int (this case is more common with multithreaded code).

```
volatile unsigned char* p_reg; // preferred  
unsigned char volatile* p_reg;
```

- Both lead to a pointer to a volatile value (usually a register your program does not control).

For completeness:

```
// the pointer itself can change at any time  
short* volatile volatilePtr;  
// volatile pointer to a volatile value  
short volatile * volatile crazyPointer
```

Aside to the aside about pointer syntax

```
int* foo;  
int * foo;  // looks like multiplication -- boo!  
int *foo;
```

These are all the same from the compiler's perspective!

Generally, just pick one and *try* to stick with it.

void waitForVBlank()

```
// scanlineCounter stores the current row being drawn
// 0-159 is onscreen, 160-227 is the vertical blank
volatile u16* scanlineCounter = (u16*) 0x04000006;

void waitForVBlank() {
    while (*scanlineCounter >= 160); // stall until current VBlank ends
    while (*scanlineCounter < 160);  // stall until next VBlank begins
}

int main() {
    // initialize game state here
    while (1) {
        doGameLogic();    // update and move all mobile game objects
        waitForVBlank();  // wait for current screen to finish drawing
        drawGameData();    // draw the world -- and be fast about it!
    }
    return 0;
}
```

This is a fairly crude (but effective) VSync

- We're wasting a lot of the CPU here doing nothing at all, just waiting until we catch the start of the next VBlank.
- Better VSync involves interrupts, which we'll cover much later.

Bouncing Square

- Basic Algorithm:
 - Draw the square
 - Move it to a new location (in game state)
 - Erase the old square
 - Draw the new square


```

// notQuiteBouncingRectangle.c
#define RGB(R, G, B) ((R) | (G) << 5 | (B) << 10)
#define REG_DISPCNT (*(unsigned short *)0x04000000)
#define MODE3 3
#define BG2_ENABLE (1<<10)
#define VIDEO_BUFFER ((u16*)0x06000000)

typedef unsigned short u16;
typedef unsigned char u8;

#define SetPixel(x, y, val) (VIDEO_BUFFER[(x) + (y)*240] = val)

int time = 0;
u8 ballSize, ballX, ballY, ball_Vx, ball_Vy;
u8 padding, screenWidth, screenHeight;
volatile u16* scanlineCounter = (u16*) 0x04000006;

void drawSquare(u8 x, u8 y, u8 size, u16 color){
    for (u8 i=0; i<size; i++){
        for (u8 j=0; j<size; j++){
            SetPixel(x+i, y+j, color);
        }
    }
}

void waitForVBlank() {
    while (*scanlineCounter >= 160); // wait until current VBlank ends
    while (*scanlineCounter < 160); // wait until next VBlank starts
}

```

```

void updateBallPosition() {
    ballX += ball_Vx;
    ballY += ball_Vy;
}

int main() {
    REG_DISPCNT = MODE3 | BG2_ENABLE;

    time = 0;
    padding = 3;
    ballSize = 5;
    screenWidth = 240 - 2*padding;
    screenHeight = 160 - 2*padding;
    ballX = screenWidth / 2;
    ballY = screenHeight / 2;
    ball_Vx = 1;
    ball_Vy = 2;

    while (1) {
        updateBallPosition();
        drawSquare(ballX, ballY, ballSize, RGB(31, 0, 0));
        time++;
    }
    return 0;
}

```

Sad Demo



```

void updateBallPosition() {
    ballX += ball_Vx;
    ballY += ball_Vy;
}

int main() {
    REG_DISPCNT = MODE3 | BG2_ENABLE;

    time = 0;
    padding = 3;
    ballSize = 5;
    screenWidth = 240 - 2*padding;
    screenHeight = 160 - 2*padding;
    ballX = screenWidth / 2;
    ballY = screenHeight / 2;
    ball_Vx = 1;
    ball_Vy = 2;

    while (1) {
        updateBallPosition();
        waitForVBlank();
        drawSquare(ballX, ballY, ballSize, RGB(31, 0, 0));
        time++;
    }
    return 0;
}

```

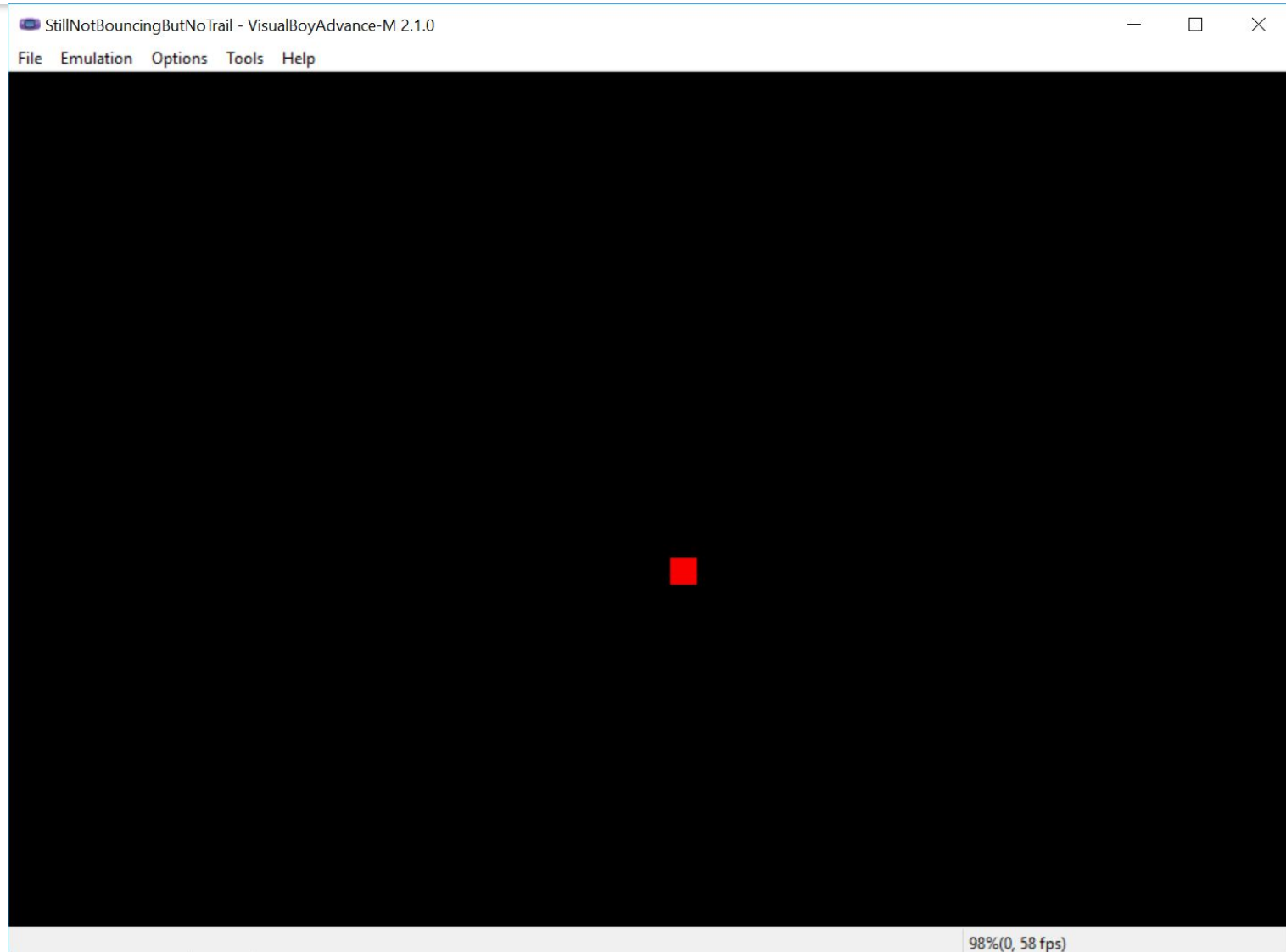
Added VSync Demo



Improvement: Erase the Previous Square

```
while (1) {  
    updateBallPosition(time);  
    waitForVBlank();  
    drawSquare(ballX - ball_Vx, ballY - ball_Vy, ballSize, RGB(0, 0, 0));  
    drawSquare(ballX, ballY, ballSize, RGB(31, 0, 0));  
    time++;  
}
```

Demo again -- one square this time! #progress



Maybe slow it down a little?

```
while (1) {
    updateBallPosition();
    waitForVBlank();
    for(int i=0; i<30000; i++) { /* waste some time */ }
    drawSquare(ballX - ball_Vx, ballY - ball_Vy, ballSize, RGB(0, 0, 0));
    drawSquare(ballX, ballY, ballSize, RGB(31, 0, 0));
    time++;
}
```

Another (better, IMO) option:

```
void updateBallPosition(int time) {
    int timestep = 3; // only do things ever so many frames
    if (time % timestep == 0 && time != 0) {
        ballX += ball_Vx;
        ballY += ball_Vy;
    }
}

/* ... */
while (1) {
    updateBallPosition(time);
    waitForVBlank();
    drawSquare(ballX - ball_Vx, ballY - ball_Vy, ballSize, RGB(0, 0, 0));
    drawSquare(ballX, ballY, ballSize, RGB(31, 0, 0));
    time++;
}
```


Demo Again

Maybe it should *Bounce*?

```
void updateBallPosition(int time) {
    int timestep = 3;
    if (time % timestep == 0 && time != 0) {
        ballX += ball_Vx;
        ballY += ball_Vy;

        if (ballX < 0){ // we should have to switch everything back to ints!
            ballX = -ballX;
            ball_Vx = -ball_Vx;
        }
        if (ballY < 0){
            ballY = -ballY;
            ball_Vy = -ball_Vy;
        }
        if (ballX + ballSize >= 240) {
            ballX -= ballX + ballSize - 240;
            ball_Vx = -ball_Vx;
        }
        if (ballY + ballSize >= 160) {
            ballY -= ballY + ballSize - 160;
            ball_Vy = -ball_Vy;
        }
    }
}
```

Maybe it should *Bounce*?

I need to be able to check for `ballX < 0` -- u8 (unsigned char) won't cut it! [0-255]

```
int ballSize, ballX, ballY, ball_Vx, ball_Vy; // change these to ints and try now!
```

```
int padding, screenWidth, screenHeight;
```

Almost!



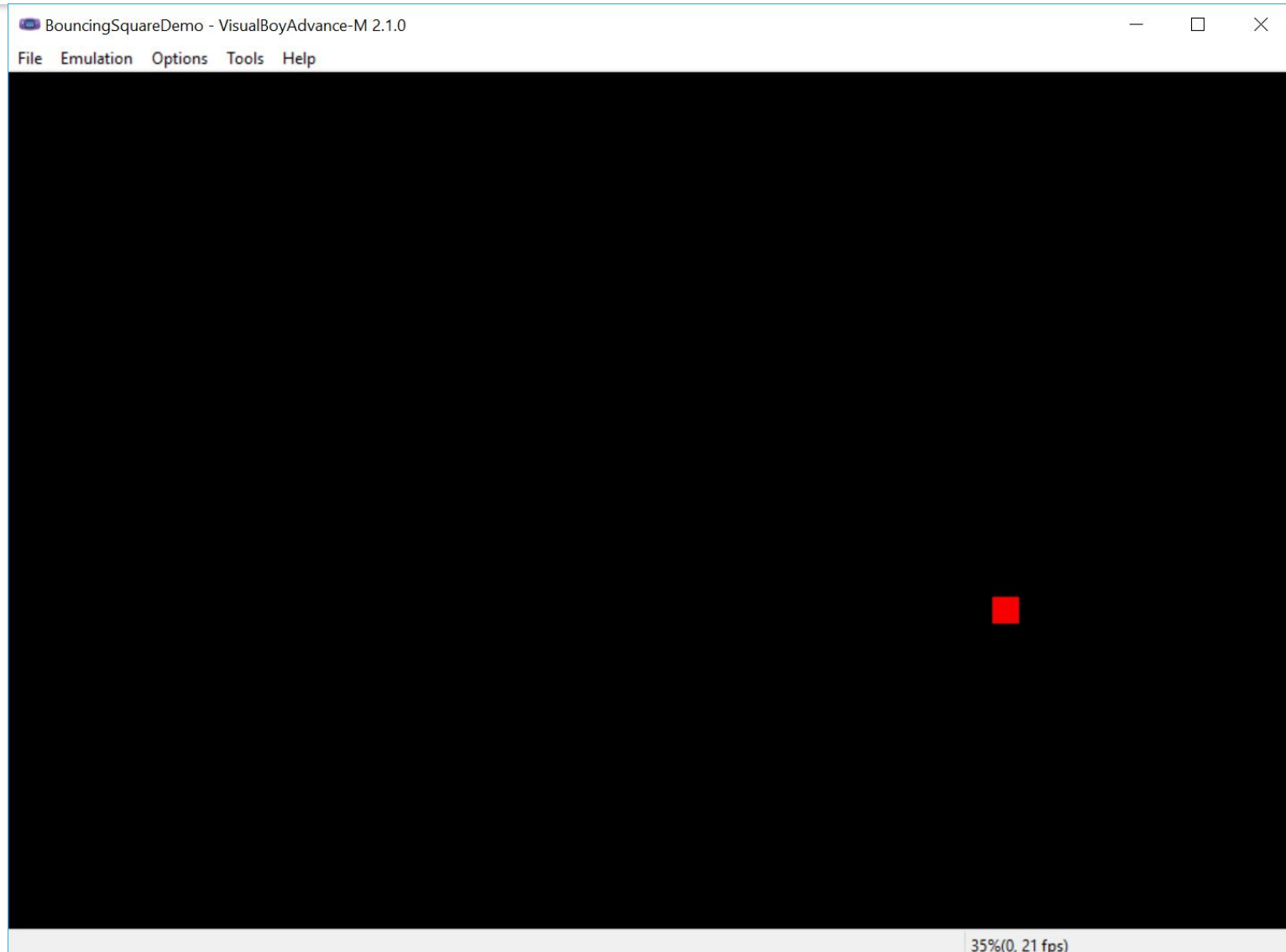
Be a little less clever

```
int padding, screenWidth, screenHeight;
int prevBallX, prevBallY;

void updateBallPosition(int time) {
    int timestep = 3;
    prevBallX = ballX;
    prevBallY = ballY;
    if (time % timestep == 0 && time != 0) {
        /* ... */
    }

    while (1) {
        updateBallPosition(time);
        waitForVBlank();
        drawSquare(prevBallX, prevBallY, ballSize, RGB(0, 0, 0));
        drawSquare(ballX, ballY, ballSize, RGB(31, 0, 0));
        time++;
    }
}
```

Real Bouncing Demo



Basics of Collision Detection

- We handled colliding with a barrier (simple out of bounds checking)
- How do we detect a collision with another square, for example?
 - $x1 + s1 > x2 \ \&\& \ x1 < x2 + s2 \ \&\&$
 $y1 + s1 > y2 \ \&\& \ y1 < y2 + s2$
 - Pretty straightforward

What about Collisions with the paddle?

- Y check is degenerate, same as checking hitting the bottom.
 - `ballY + ballSize >= paddleY`
- X check is only slightly more interesting
 - `ballX + ballSize >= paddleX && ballX < paddleX + paddleWidth`
- When X and Y checks are both true, collision!
 - Note: you could also allow for side collisions like real brick games do as well

Input Basics

- 10 buttons
 - Start
 - Select
 - A
 - B
 - Left
 - Right
 - Up
 - Down
 - Left shoulder
 - Right shoulder

Button Register REG_KEYINPUT

0x04000130

REG_KEYINPUT (REG_P1) @ 0400:0130h

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
-	-	-	-	-	-	L	R	down	up	left	right	start	select	B	A

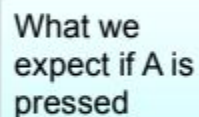
- Only bits 0-9 are used here.
 - 1 is not pressed
 - 0 is pressed ???
 - It's a trick of the wiring that zeros out the bit when you ground it by completing the circuit.
- When nothing is pressed, all the bits are 1
 - To make things more intuitive, flip them before checking

```
#define REG_KEYINPUT (*(volatile u16*)0x04000130)
```

```
#define KEY_A      0x0001  
#define KEY_B      0x0002  
#define KEY_SELECT 0x0004  
#define KEY_START  0x0008  
#define KEY_RIGHT   0x0010  
#define KEY_LEFT    0x0020  
#define KEY_UP      0x0040  
#define KEY_DOWN    0x0080  
#define KEY_R       0x0100  
#define KEY_L       0x0200
```

```
// is KEY_A pressed?  
KEY_A & ~REG_KEYINPUT
```

What we
expect if A is
pressed



&

Button register:

1111101111111111

~(Button register):

0000010000000000

0000000000000001

????????????????

```
#define KEY_DOWN_NOW(key) (~(REG_KEYINPUT) & key)
```

Paste all those defines in my previous project

```
// lots of defines above!
volatile u16* scanlineCounter = (u16*) 0x04000006;
u16 color;

u16 colorFromInputs(){
    u16 color = RGB(31, 0, 0);
    if (KEY_DOWN_NOW(KEY_A)) {
        color = color | RGB(0, 0, 31);
    }
    if (KEY_DOWN_NOW(KEY_B)) {
        color = color | RGB(0, 31, 0);
    }
    return color;
}

while (1) {
    updateBallPosition(time);
    waitForVBlank();
    color = colorFromInputs();
    drawSquare(prevBallX, prevBallY, ballSize, RGB(0, 0, 0));
    drawSquare(ballX, ballY, ballSize, color);
    time++;
}
```

Demo color changing bouncing ball
