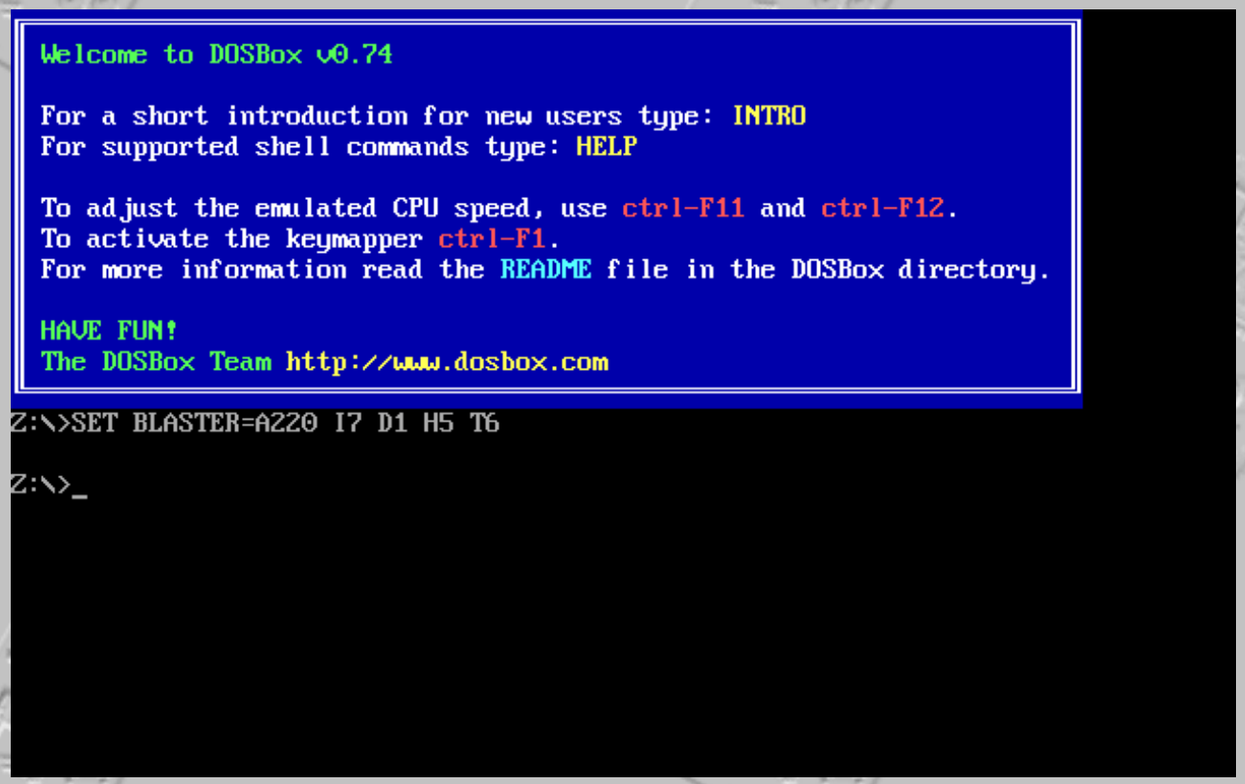
GAME BOY EMULATION

Nicolas Montanaro
nicolas.moe

EMULATION OVERVIEW

“hardware or software that enables one computer system (called the host) to behave like another computer system (called the guest)”



GAME BOY ARCHITECTURE

- Split up software components
- Simulate parts, test individually, "wire" together

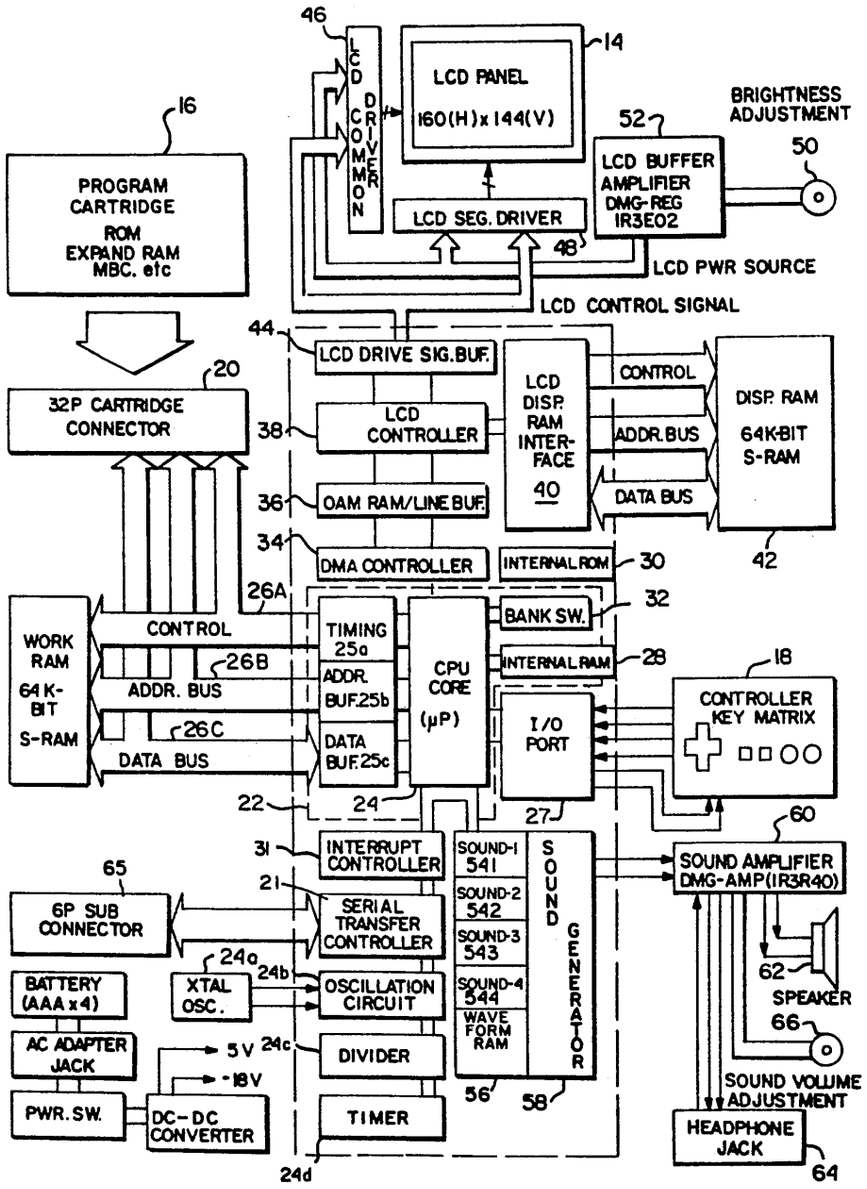
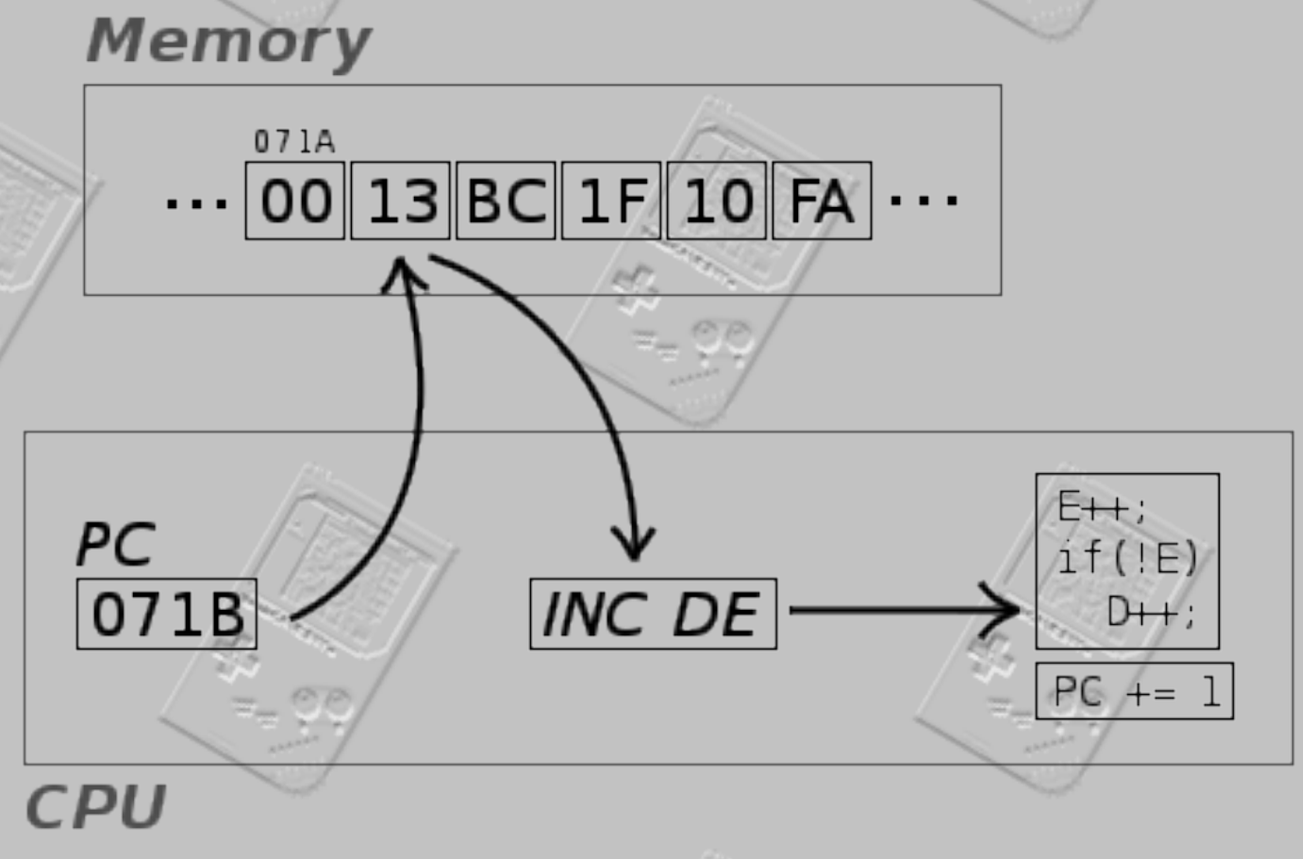


FIG. 4

WHAT'S A ROM?

"I've been downloading them 100% legally for years!"

```
00000100:□00c3□5001□ceed□6666□cc0d□000b□0373□0083□□..P...ff.....s..□
00000110:□000c□000d□0008□111f□8889□000e□dccc□6ee6□□................n.□
00000120:□dddd□d999□bbbb□6763□6e0e□eccc□dddc□999f□□......gcn.......□
00000130:□bbb9□333e□5445□5452□4953□0000□0000□0000□□..3>TETRIS......□
00000140:□0000□0000□0000□0000□0000□0001□000b□89b5□□................
```



SHARP LR35902

- Frankenstein Zilog Z80 / Intel 8080

- 8 8-bit registers, 2 16-bit

- Whopping 4.19MHz clock speed

- 245 non-prefix instructions, 255 CB-prefixed

CPU IMPLEMENTATION

- CPU
- registers
- instructions
- CB-prefixed instructions
- executors

REGISTERS

```
type Registers struct {
    a byte // Accumulator
    // Flags
    // ZNHC 0000
    // Z = zero, N = subtract, H = half carry, C = carry
    f byte

    b byte
    c byte

    d byte
    e byte

    h byte
    l byte

    sp []byte // Stack pointer
    PC []byte // Program counter
}
```

Also contains methods to modify flags

INSTRUCTIONS

```
type Instruction struct {
    Mnemonic string
    // Number of T cycles instruction takes to execute
    // Divide by 4 to get number of M cycles
    TCycles uint16
    NumOperands uint16
    Executor    func() int // Executes appropriate function
}
```

```
0x00: Instruction{"NOP", 4, 1, func() int { return 0 }},
0x01: Instruction{"LD BC,i16", 12, 3, func() int { gbcpu.LDrrnn(&gbcpu.Regs.b, &gbcpu.Regs.c); return 0 }},
0x02: Instruction{"LD (BC),A", 8, 1, func() int { gbcpu.LDaar(&gbcpu.Regs.b, &gbcpu.Regs.c, &gbcpu.Regs.a); return 0 }},
0x03: Instruction{"INC BC", 8, 1, func() int { gbcpu.INCrr(&gbcpu.Regs.b, &gbcpu.Regs.c); return 0 }},
0x04: Instruction{"INC B", 4, 1, func() int { gbcpu.INCr(&gbcpu.Regs.b); return 0 }},
```

EXECUTORS

```
// XORr -> e.g. XOR B
// Bitwise XOR of reg into A
// Flags: Z000
func (gbcpu *GBCPU) XORr(reg *byte) {
    gbcpu.Regs.a ^= *reg

    // Check for zero
    if gbcpu.Regs.a == 0x00 {
        gbcpu.Regs.setZero()
    } else {
        gbcpu.Regs.clearZero()
    }

    // Set flags
    gbcpu.Regs.clearSubtract()
    gbcpu.Regs.clearHalfCarry()
    gbcpu.Regs.clearCarry()
}
```

```
// LDSPnn -> e.g. LD SP,i16
// Loads 16 bit value from next 2 bytes into SP
// Flags: none
func (gbcpu *GBCPU) LDSPnn() {
    operands := gbcpu.getOperands(2)
    gbcpu.Regs.sp = operands
}
```

```
// CALLaa -> e.g. CALL $028B
// Pushes the addr at PC+3 to the stack
// Jumps to the address specified by next 2 bytes
func (gbcpu *GBCPU) CALLaa() {
    operands := gbcpu.getOperands(2)
    nextInstr := gbcpu.sliceToInt(gbcpu.Regs.PC) + 3
    nextInstrBytes := make([]byte, 2)
    binary.LittleEndian.PutUint16(nextInstrBytes, nextInstr)
    gbcpu.pushByteToStack(nextInstrBytes[1])
    gbcpu.pushByteToStack(nextInstrBytes[0])
    gbcpu.Regs.PC = operands

    gbcpu.Jumped = true
}
```

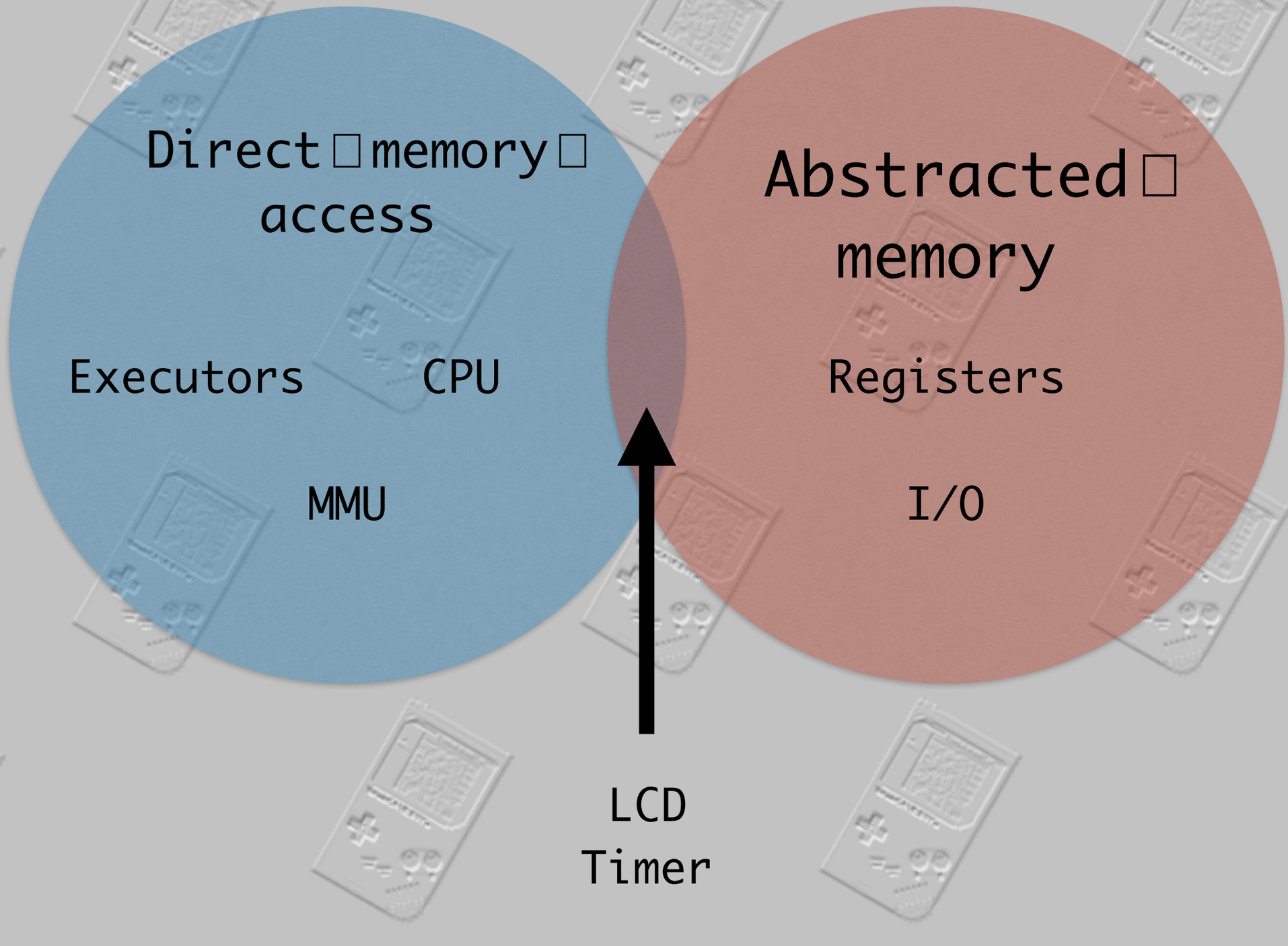
MEMORY

- Straightforward representation as a flat byte array
- Load ROM into correct location, begin fetch-decode-dispatch loop
- Can ignore all memory dealing with video & I/O

GameBoy Memory Areas

$FFFF	Interrupt Enable Flag
$FF80-$FFFE	Zero Page - 127 bytes
$FF00-$FF7F	Hardware I/O Registers
$FEA0-$FEFF	Unusable Memory
$FE00-$FE9F	OAM - Object Attribute Memory
$E000-$FDFF	Echo RAM - Reserved, Do Not Use
$D000-$DFFF	Internal RAM - Bank 1-7 (switchable - CGB only)
$C000-$CFFF	Internal RAM - Bank 0 (fixed)
$A000-$BFFF	Cartridge RAM (If Available)
$9C00-$9FFF	BG Map Data 2
$9800-$9BFF	BG Map Data 1
$8000-$97FF	Character RAM
$4000-$7FFF	Cartridge ROM - Switchable Banks 1-xx
$0150-$3FFF	Cartridge ROM - Bank 0 (fixed)
$0100-$014F	Cartridge Header Area
$0000-$00FF	Restart and Interrupt Vectors

REPRESENTATION

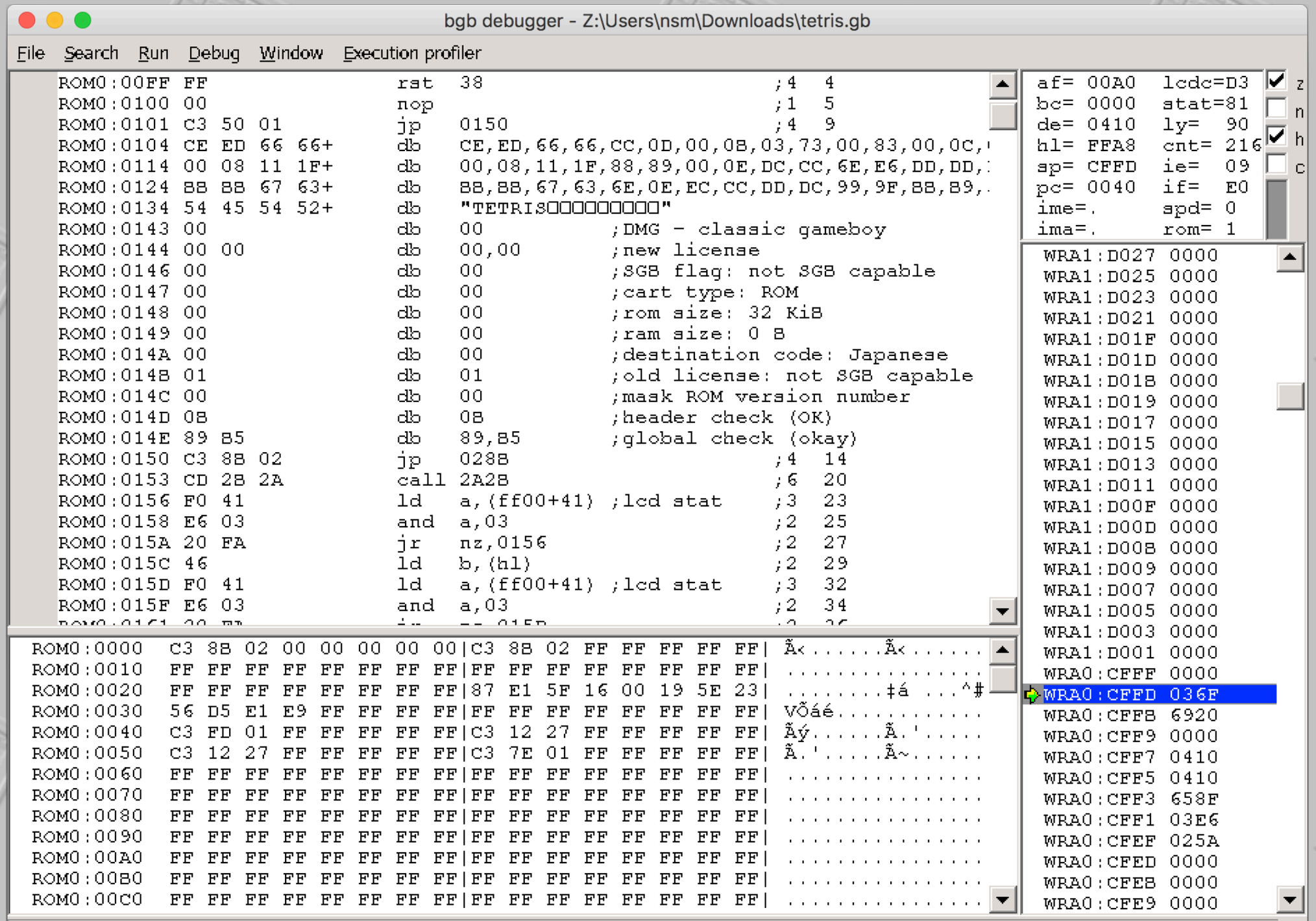


BENEFITS

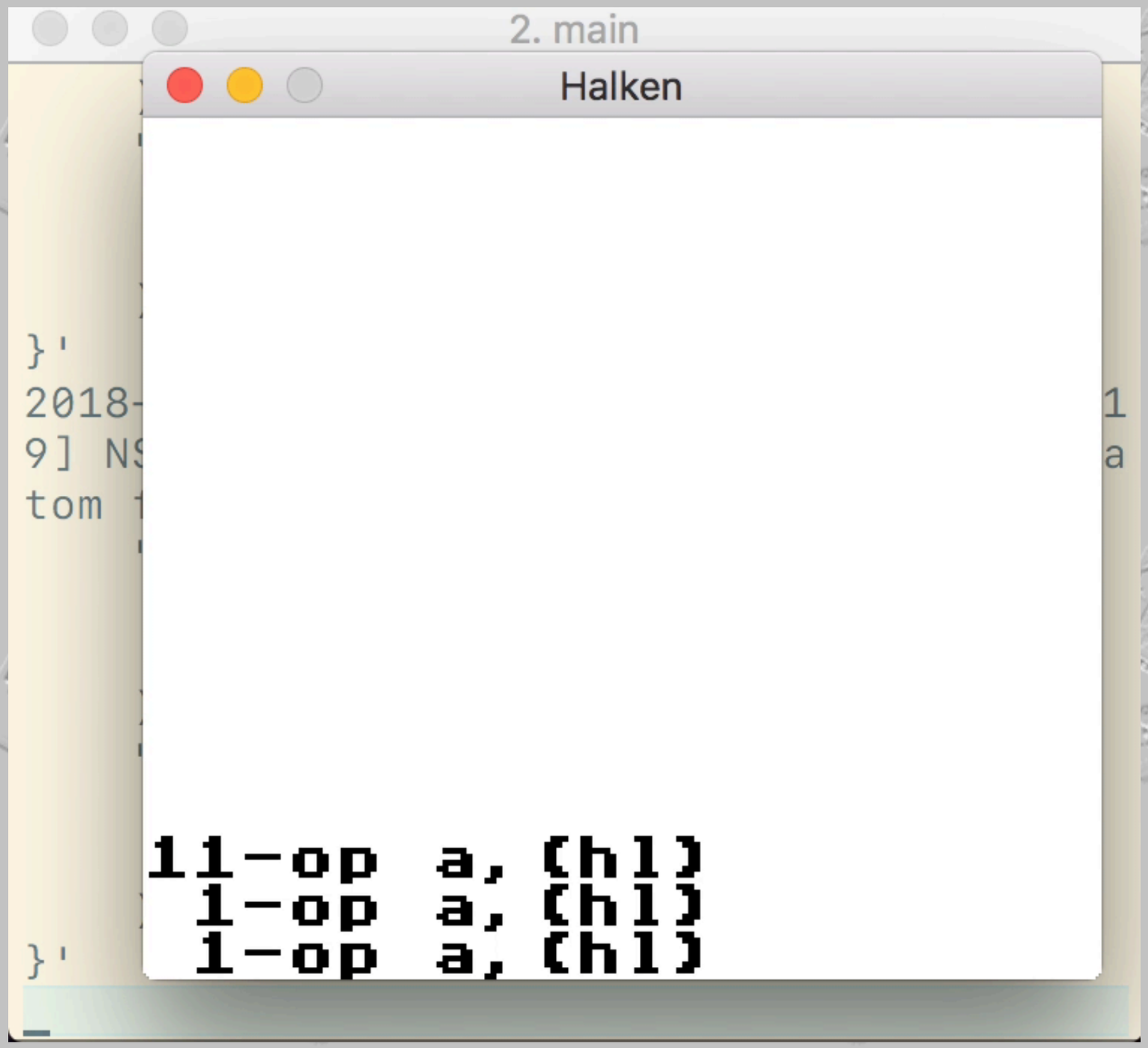
```
wb: function(addr,val) {
  switch(addr&0xF000)
  {
    // ROM bank 0
    // MBC1: Turn external RAM on
    case 0x0000: case 0x1000:
      switch(MMU._carttype)
      {
        case 1:
          MMU._mbc[1].ramon = ((val&0xF)==0xA)?1:0;
          break;
      }
    break;

  // MBC1: ROM bank switch
  case 0x2000: case 0x3000:
    switch(MMU._carttype)
    {
      case 1:
        MMU._mbc[1].rombank &= 0x60;
        val &= 0x1F;
        if(!val) val=1;
        MMU._mbc[1].rombank |= val;
        MMU._romoffs = MMU._mbc[1].rombank * 0x4000;
        break;
    }
    break;
```

```
func (gbmmu *GBMMU) WriteData(addr uint16, data byte) {
    if addr == 0xFF00 {
        GbIO.SetCol(data)
    } else if addr == 0xFF0F {
        // TODO What do writes here really do? Ignored or bit set?
        // gbmmu.Memory[0xFF0F] |= (1 << 0)
    } else if addr == 0xFF41 {
        // TODO Same as above
    } else if addr >= 0x0000 && addr <= 0x150 {
        // Don't allow writes to invalid locations
    } else if addr == 0xFF46 {
        spriteAddr := int(data) * 256
        for i := range gbmmu.Memory[0xFE00:0xFEA0] {
            gbmmu.Memory[0xFE00+i] = gbmmu.Memory[spriteAddr+i]
        }
    } else if addr == 0xFF07 {
        gbmmu.Memory[addr] += data
    } else {
        gbmmu.Memory[addr] = data
    }
}
```

PAINFUL DEBUGGING



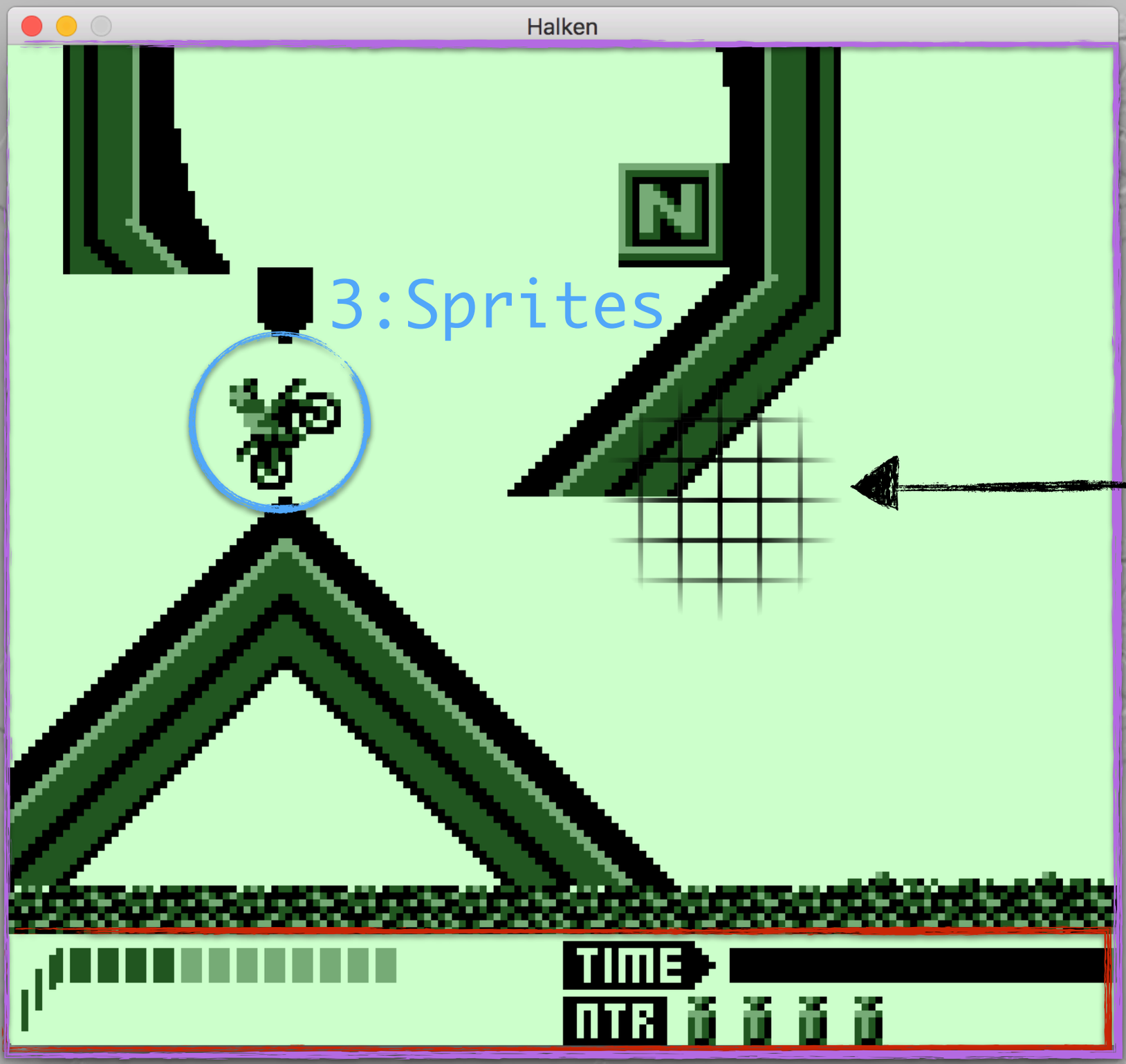
NICE DEBUGGING



THE LCD

- **160x144** pixels
- **20x18** tiles, each **8x8** pixels
- **256x256** “background”
- **40** sprites at once
- **4** colors, shades of gray

GRAPHICS "LAYERS"



TILE MAPS

=

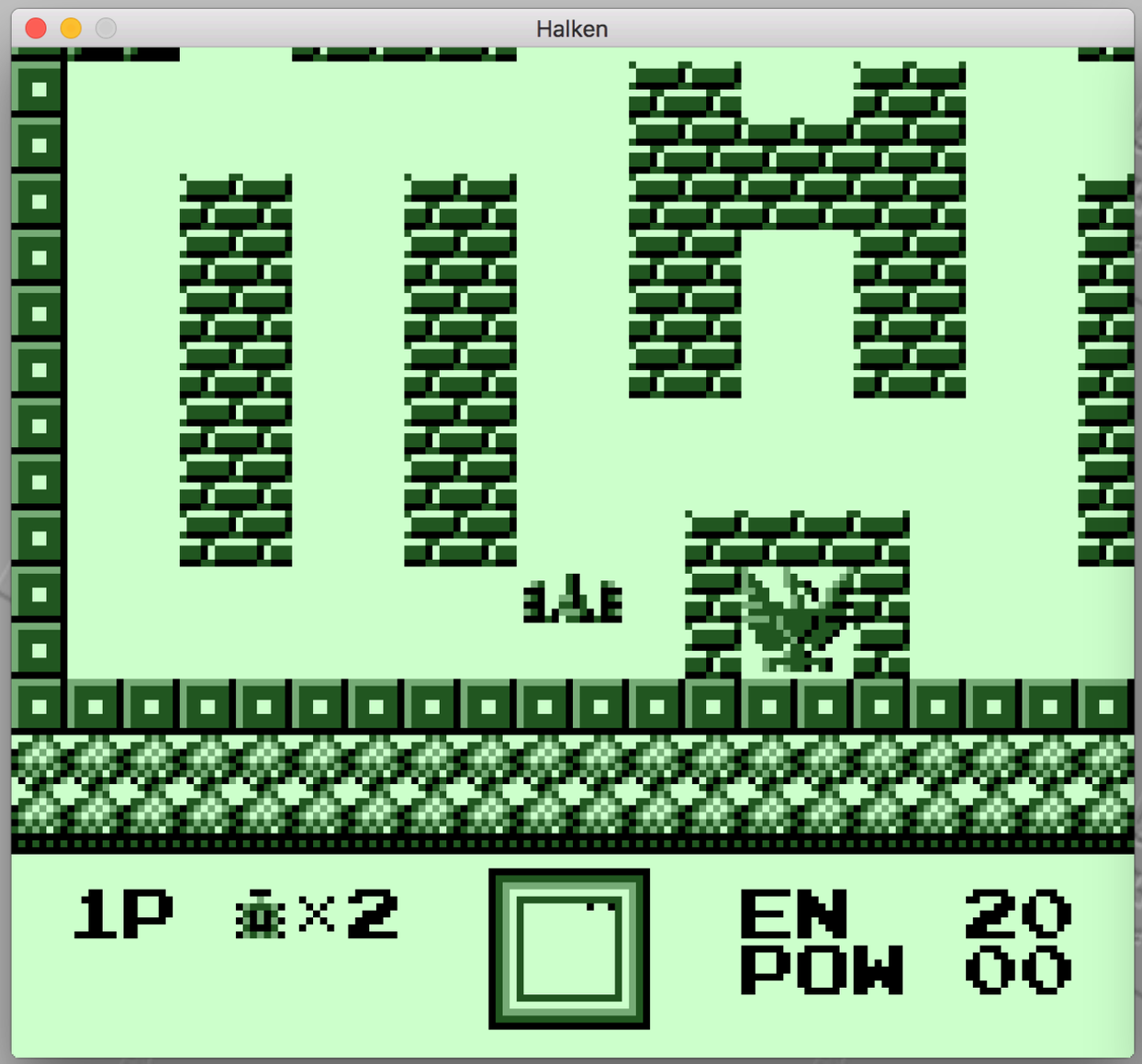
```
8E,8E,8E,8E,8E,8E,8E,8E,8E,8E,8E,8E,8E,8E,8E,8E,8E,8E
8E,8E,8E,8E,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F
5A,5B,5B,5B,5B,5B,5B,5B,5B,5B,5B,5B,5B,5B,5B,5B,5B,5B
5B,5B,5B,5C,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F
5D,80,81,82,83,90,91,92,81,82,83,90,6C,6D,6E,6F
70,71,72,5E,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F
5D,84,85,86,87,93,94,95,85,86,87,93,73,74,75,76
77,78,2F,5E,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F
5D,2F,88,89,2F,96,97,98,88,89,2F,96,79,7A,7B,7C
7D,7E,2F,5E,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F
5D,2F,8A,8B,2F,8E,8F,6B,8A,8B,2F,8E,7F,66,67,68
69,6A,2F,5E,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F
5F,60,60,60,60,60,60,60,60,60,60,60,60,60,60,60,60,60
60,60,60,61,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F
8E,3C,3C,3C,3C,3C,3C,3C,3C,3C,3C,3C,3C,3C,3D,3E
3C,3C,3C,8E,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F
8E,8C,8C,62,63,8C,8C,3A,8C,8C,8C,8C,8C,3A,42,43
3B,8C,8C,8E,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F
8E,3A,8C,64,65,8C,8C,8C,8C,3B,8C,8C,8C,8C,44,45
8C,8C,8C,8E,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F
8E,8C,8C,8C,8C,8C,8C,8C,8C,8C,8C,8C,8C,8C,46,47,48
49,3F,40,8E,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F
8E,8C,8C,8C,8C,3A,8C,8C,8C,8C,53,54,8C,4A,4B,4C
4D,42,43,8E,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F
8E,8C,8C,8C,8C,8C,8C,8C,8C,54,55,56,57,4E,4F,50
51,52,45,8E,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F
41,41,41,41,41,41,41,41,41,41,41,41,41,41,41,41
41,41,41,41,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F
2F,2F,59,19,15,0A,22,0E,1B,2F,2F,2F,99,19,15,0A
22,0E,1B,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F
2F,2F,9A,9A,9A,9A,9A,9A,9A,2F,2F,2F,9A,9A,9A,9A
9A,9A,9A,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F
2F,2F,2F,2F,33,30,31,32,31,2F,34,35,36,37,38,39
2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F,2F
```

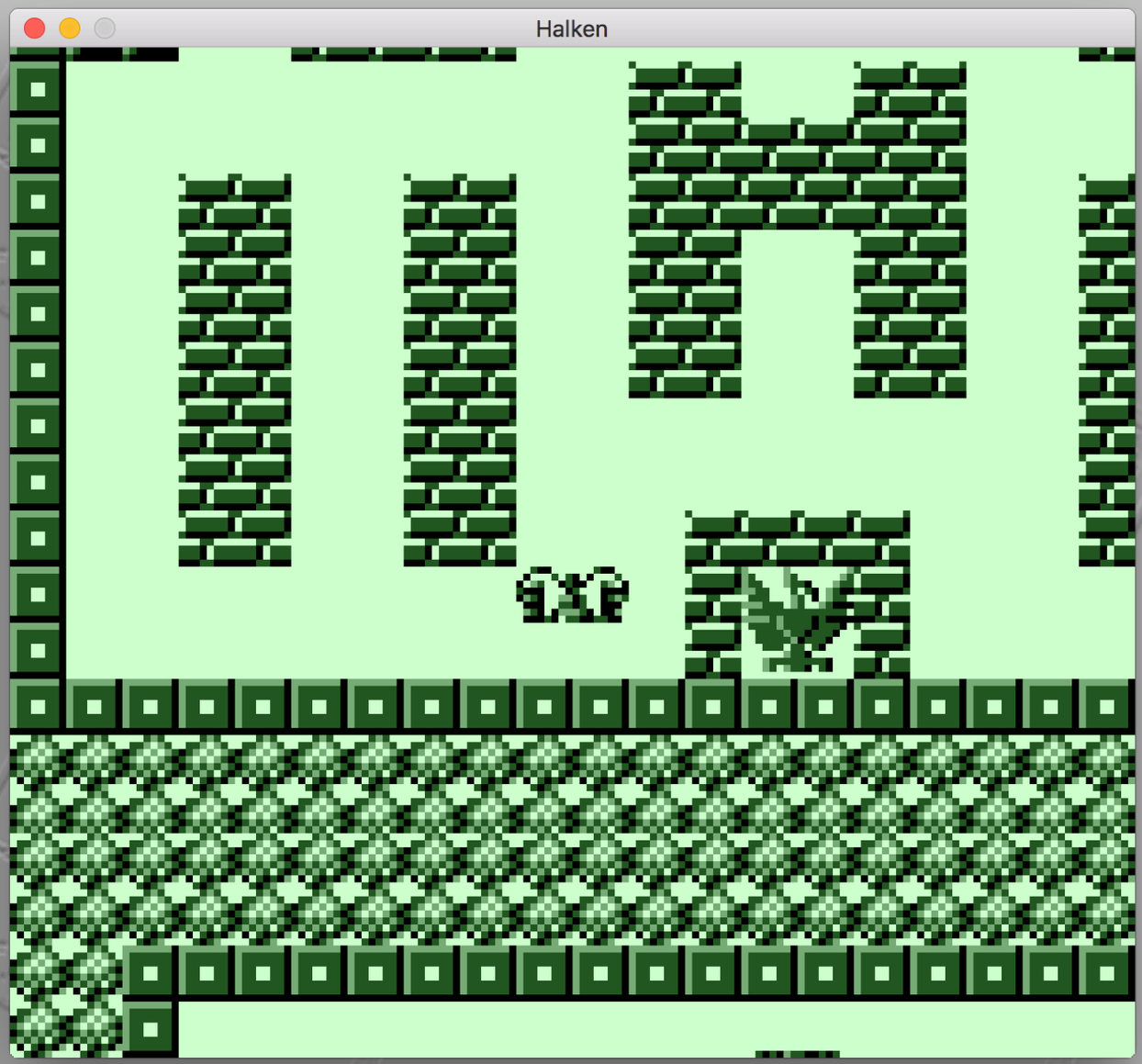
TILE RENDERING

```
Tile:                                  Image:

.33333..                    .33333.. -> 01111100 -> 7Ch
22...22.                                01111100 -> 7Ch
11...11.                    22...22. -> 00000000 -> 00h
2222222. <-- digits represent           11000110 -> C6h
33...33.     color numbers  11...11. -> 11000110 -> C6h
22...22.                                00000000 -> 00h
11...11.                    2222222. -> 00000000 -> 00h
........                                11111110 -> FEh
                            33...33. -> 11000110 -> C6h
                                        11000110 -> C6h
                            22...22. -> 00000000 -> 00h
                                        11000110 -> C6h
                            11...11. -> 11000110 -> C6h
                                        00000000 -> 00h
                            ........ -> 00000000 -> 00h
                                        00000000 -> 00h
```

ASCII from: https://fms.komkon.org/GameBoy/Tech/Software.html

WINDOW





Win.□X□offset->$FE4B,□Win.□Y□offset->$FE4A

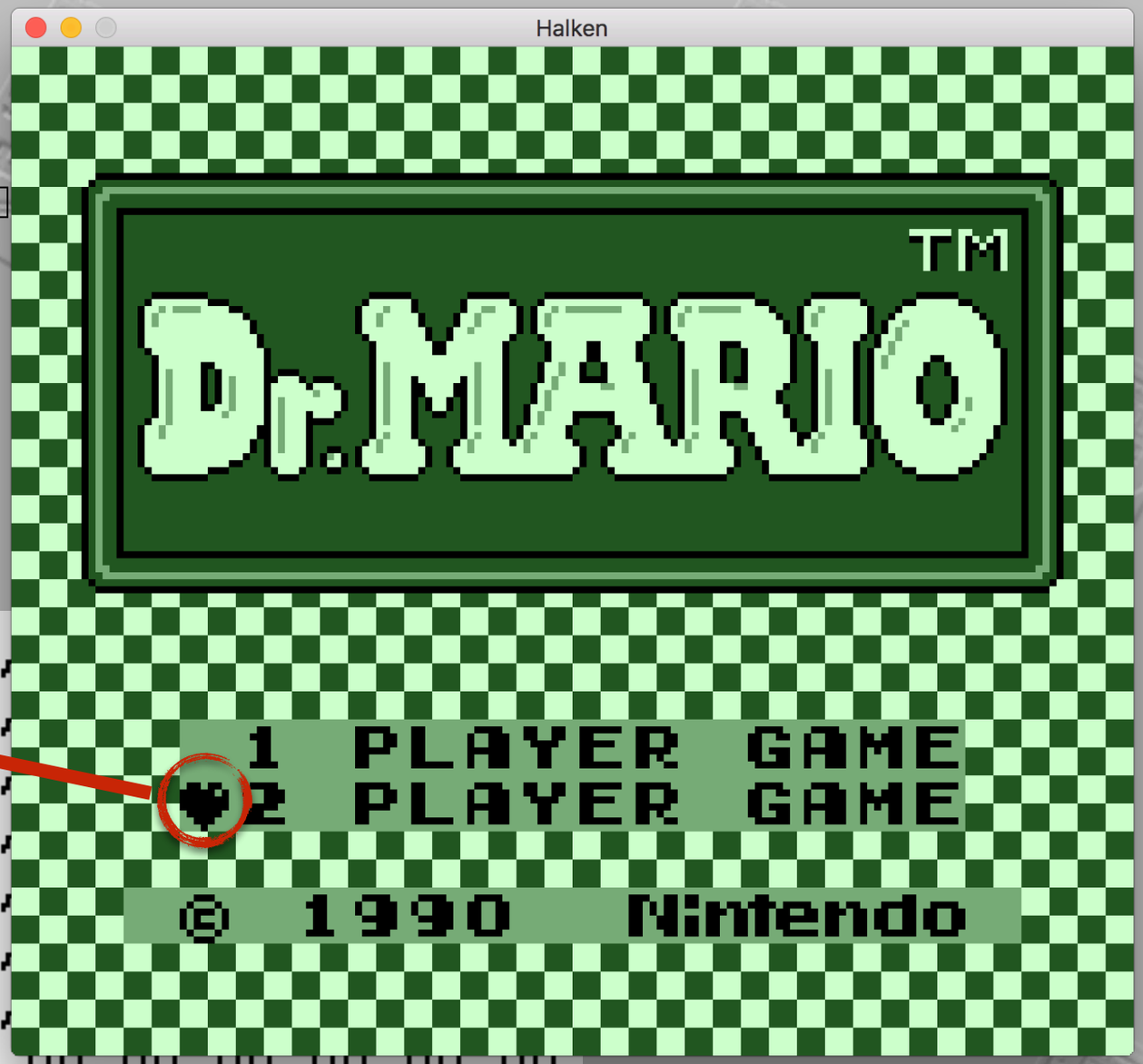
SPRITES

Attributes

```
0->Palette□#□(CGB)□□
1->Palette□#□(CGB)□
2->Palette□#□(CGB)□
3->VRAM□bank□(CGB)□
4->Palette□#□(DMG)□
5->X□flip□
6->Y□flip□
7->Above□BG
```

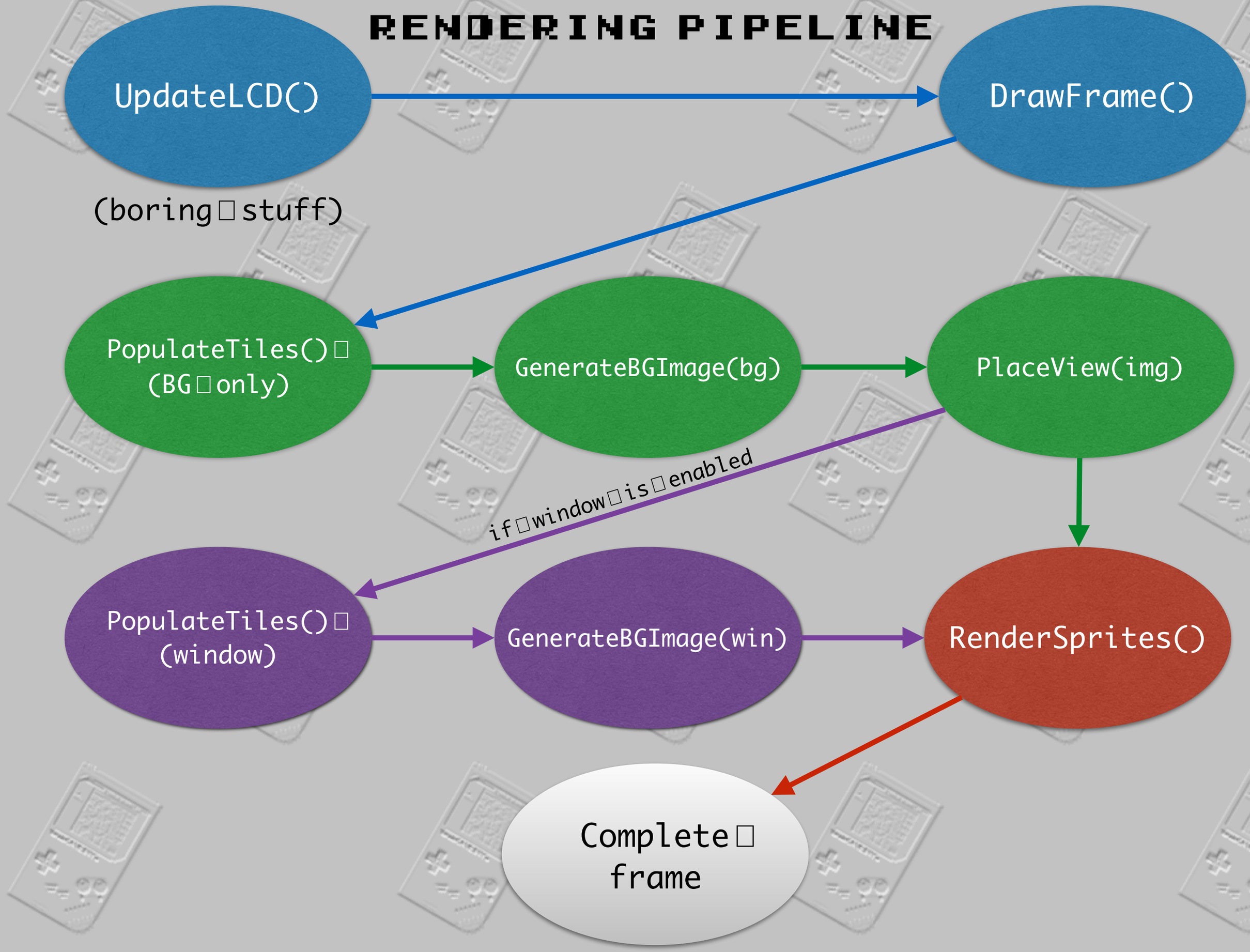
ID X Y

```
78,20,9B,00 00,00,00,00,00,
00,00,00,00,00,00,00,00,00,
00,00,00,00,00,00,00,00,00,
00,00,00,00,00,00,00,00,00,
00,00,00,00,00,00,00,00,00,
00,00,00,00,00,00,00,00,00,
00,00,00,00,00,00,00,00,00,
00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00
00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00
00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00
```



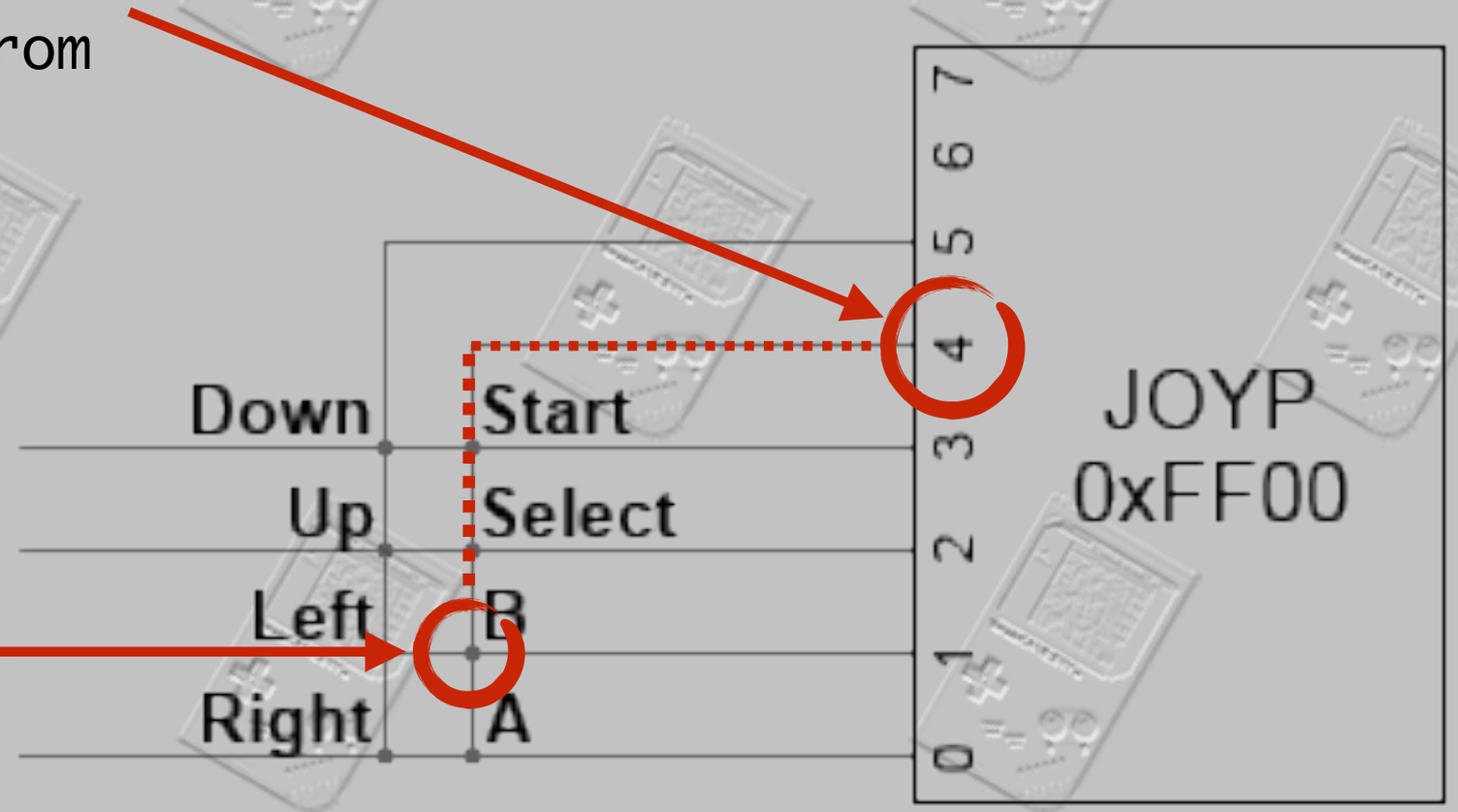
Sprite□OAM:□$FE00-$FEA0

RENDERING PIPELINE



INPUTS

Hardware selects which column it wants input from



Last button pressed
updated 60 times
per second

cols = [0x0F, 0x0F]
Values altered by reading inputs
Selected column = index 0 or 1
Return value

Image: http://imrannazar.com/GameBoy-Emulation-in-JavaScript:-Input

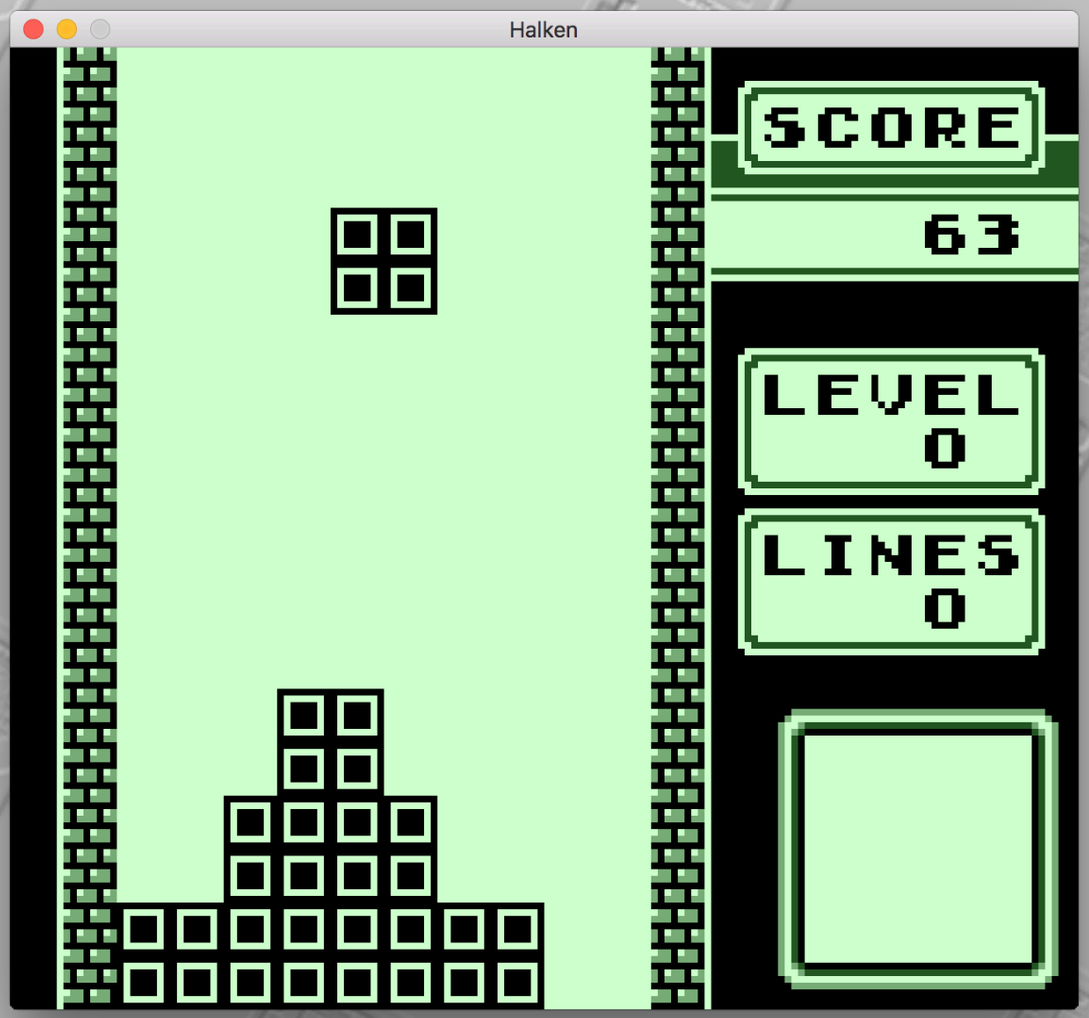
TIMER

- $FF04 = divider, acts as PRNG

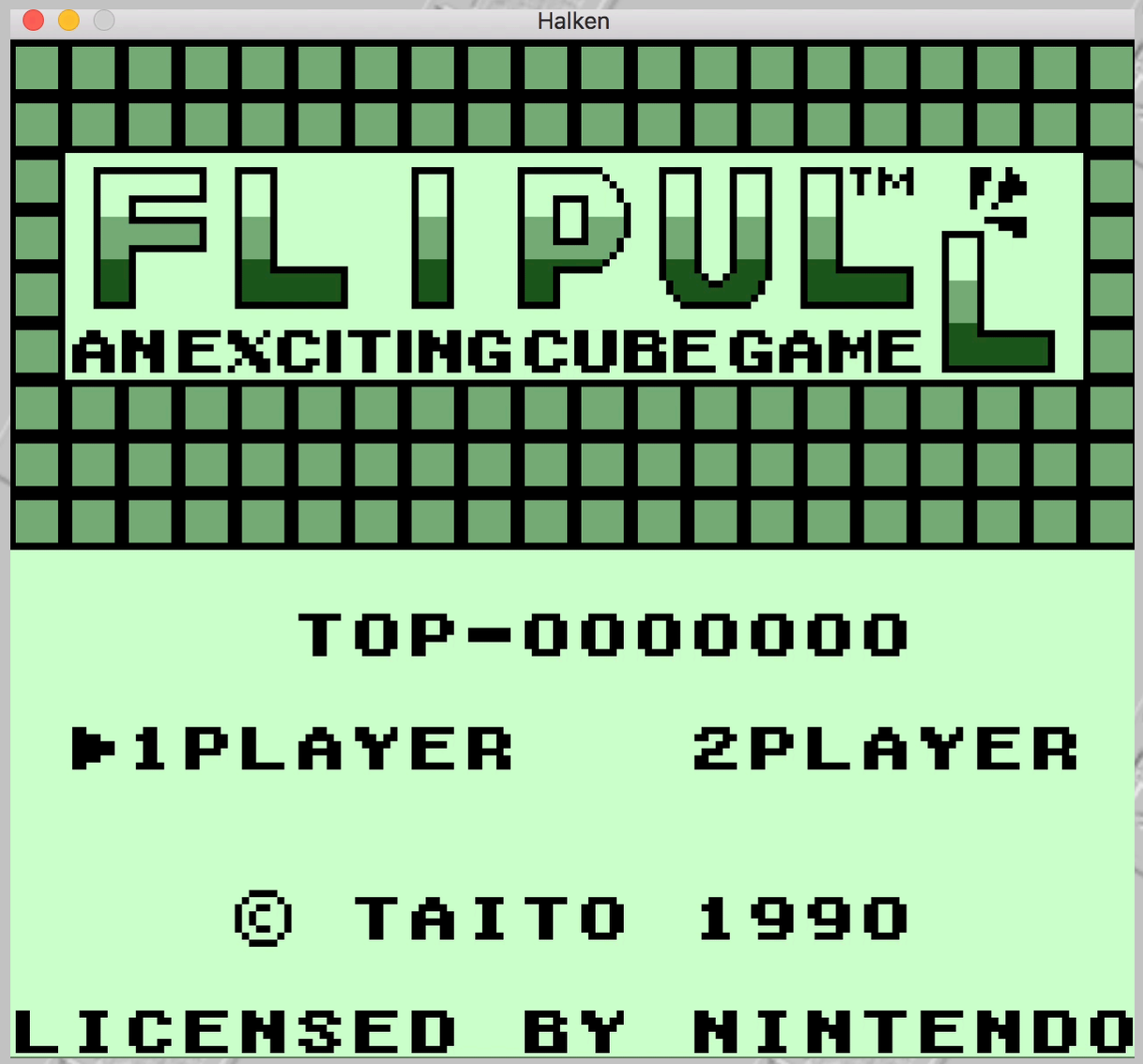
- $FF05 = counter, triggers timer interrupt when overflows

- $FF06 = modulo, when counter overflows, resets. Used to generate PRNG in divider

- $FF07 = control, bits 0 & 1 set speed, bit 2 sets timer on/off



WHAT'S DONE



- CPU
- Memory
- (most) Graphics
- (most) Interrupts
- Timer
- I/O

WHAT'S NOT

- Sound
- LCD STAT interrupt not entirely correct
- Sprites only use 1 color palette
- 8x16 sprites aren't implemented, get cut off
- ROM banking

WHAT I LEARNED

- a lot
- Excellent exercise in lower-level programming, computer organization/architecture
- Some software projects take a LONG time to write
- Testing is extremely helpful, especially when someone else wrote the tests
- Hardware restrictions lead to creative programming