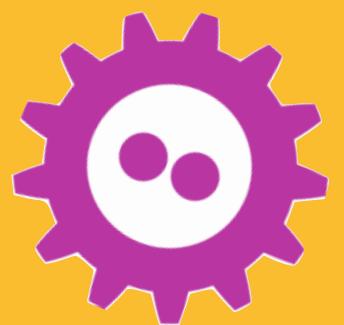


# Rustboy

A Rust journey into Game Boy dev

ffex @ fosdem2026



# 01. Introduction

# My story

## My gameboys



# My story

Me and the gameboy



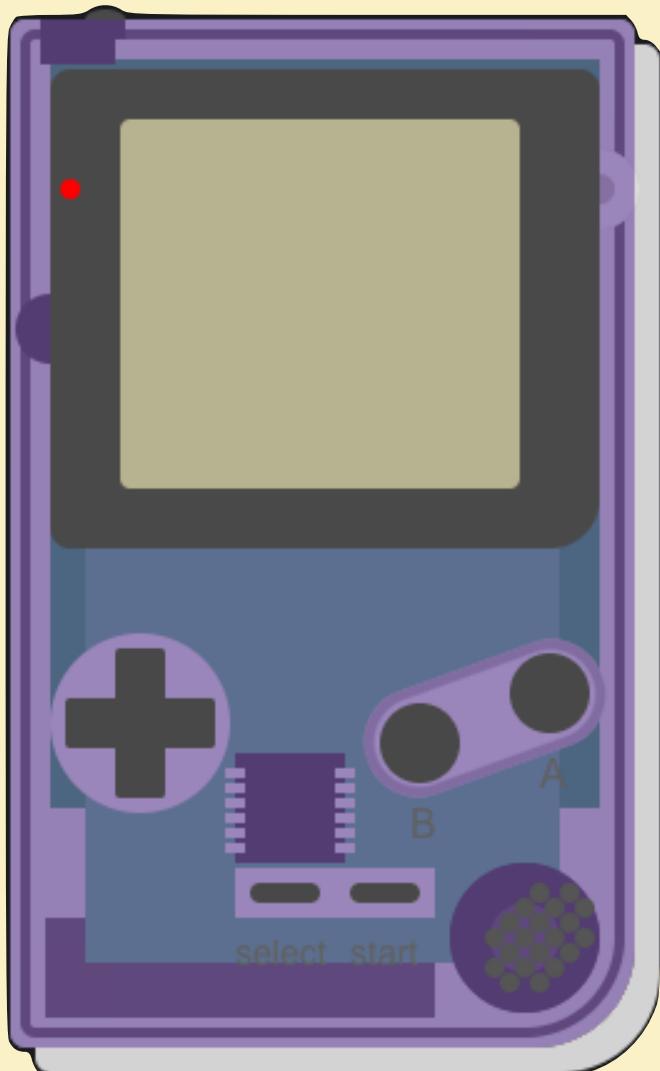
# The not-so-exciting life of a programmer

In the daily routine

- We have a **problem**
- Find and replicate the **problem**
- Search for the **problem** online
- Ask AI about the **problem**
- Copy a solution of the **problem**
- Test the *solution*
- Is it the best *solution*?
- Search for the best *solution* online
- .....

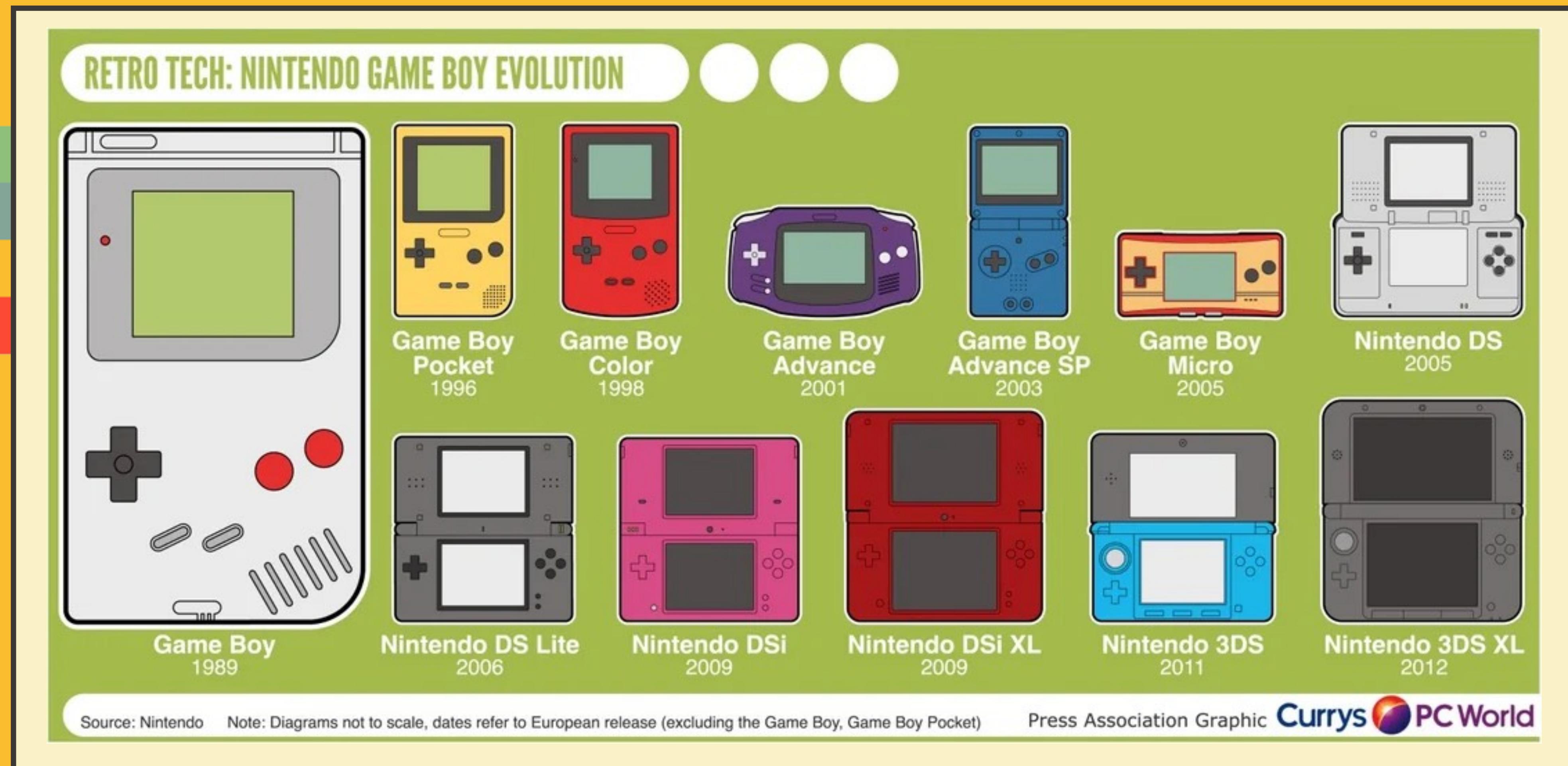
# Fosdem 2025

Boom!

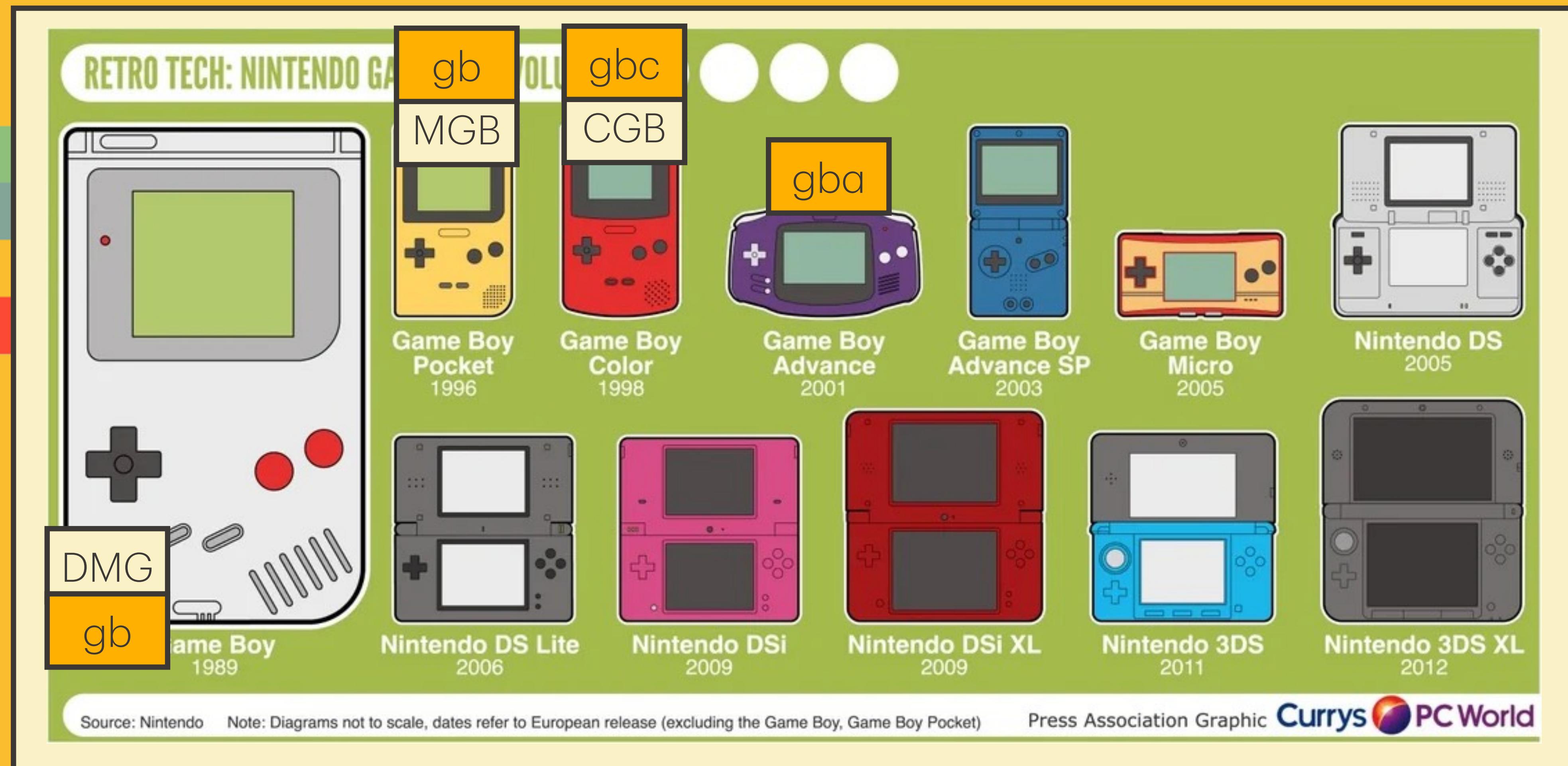


## 02. Hardware

# All the game boys



# All the game boys



# Pandocs

This document, started in early 1995, is considered the single most comprehensive technical reference to Game Boy available to the public.

Link: **<https://gbdev.io/pandocs/>**

The screenshot shows a web-based documentation interface with a light gray header bar. On the left, there are icons for navigation (three horizontal lines, a pen, a magnifying glass). In the center, the title "Pan Docs" is displayed above the "Foreword" section. On the right, there are icons for printing, sharing, and editing. The main content area has a white background. The "Foreword" section contains text about the document's history and current status. Below it, a green-bordered box titled "SCOPE" contains a paragraph about the target audience and a link to another document. At the bottom right of the content area, there is a small right-pointing arrow icon.

Pan Docs

## Foreword

This document, started in early 1995, is considered the single most comprehensive technical reference to Game Boy available to the public.

You are reading a new version of it, maintained in the Markdown format and enjoying renewed [community](#) attention, correcting and updating it with recent findings. To learn more about the legacy and the mission of this initiative, check [History](#).

### SCOPE

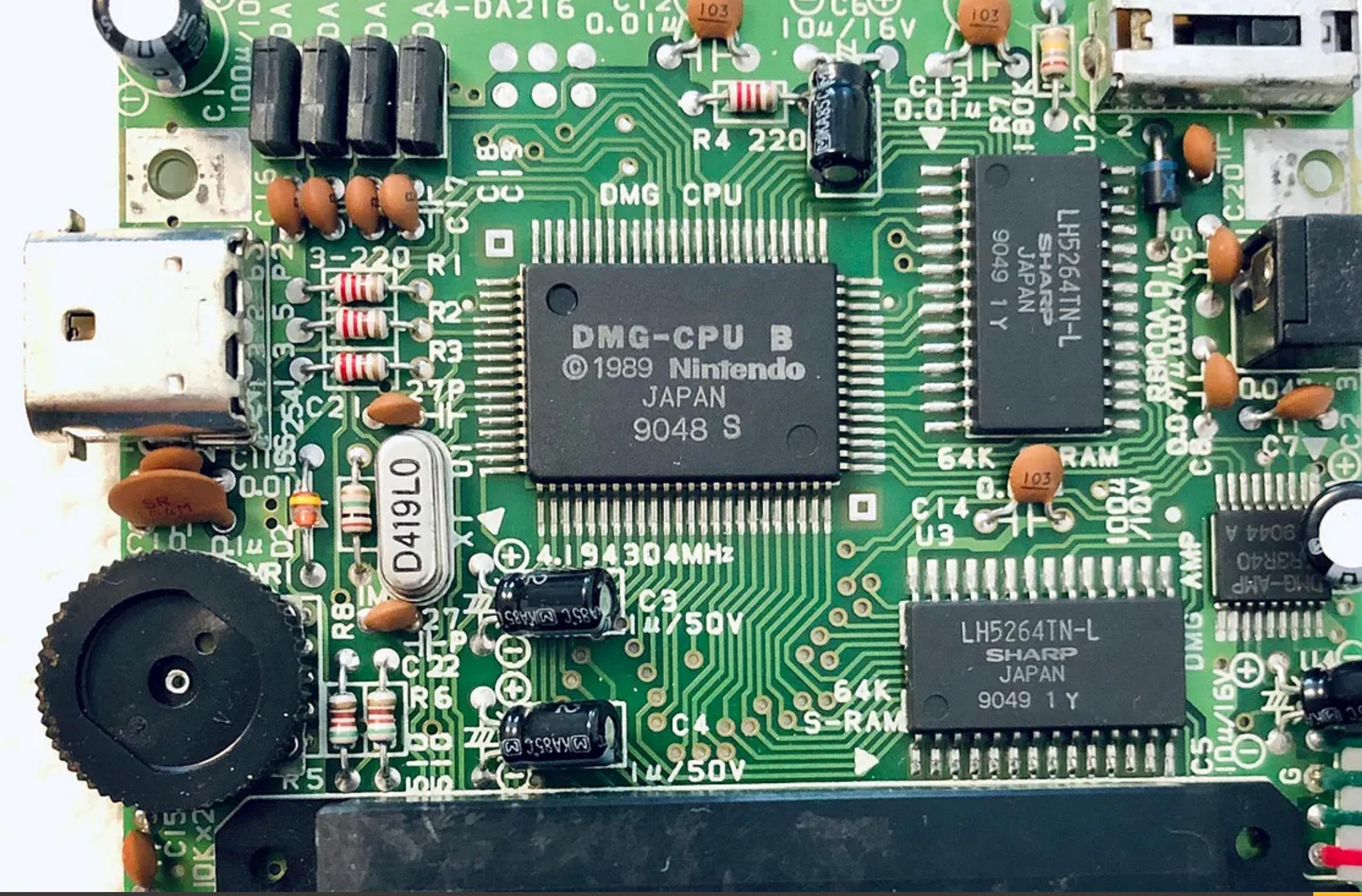
The information here is targeted at homebrew development. Emulator developers may be also interested in the [Game Boy: Complete Technical Reference](#) document.

# Memory Map

Start	End	Description	Notes
0000	3FFF	16 KiB ROM bank 00	From cartridge, usually a fixed bank
4000	7FFF	16 KiB ROM Bank 01–NN	From cartridge, switchable bank via <a href="#">mapper</a> (if any)
8000	9FFF	8 KiB Video RAM (VRAM)	In CGB mode, switchable bank 0/1
A000	BFFF	8 KiB External RAM	From cartridge, switchable bank if any
C000	CFFF	4 KiB Work RAM (WRAM)	
D000	DFFF	4 KiB Work RAM (WRAM)	In CGB mode, switchable bank 1–7
E000	FDFF	<a href="#">Echo RAM</a> (mirror of C000–DDFF)	Nintendo says use of this area is prohibited.
FE00	FE9F	<a href="#">Object attribute memory (OAM)</a>	
FEA0	FEFF	<a href="#">Not Usable</a>	Nintendo says use of this area is prohibited.
FF00	FF7F	<a href="#">I/O Registers</a>	
FF80	FFFE	High RAM (HRAM)	
FFFF	FFFF	<a href="#">Interrupt Enable register (IE)</a>	

# SoC

gb, gbc, gba



- Game Boys use only a single integrated System-on-a-Chip (SoC)
- SoC includes the processor (CPU) core, some memories, and various peripherals
- The Game Boy SoC is sometimes called the “CPU”

More about CPU: <https://gekkio.fi/files/gb-docs/gbctr.pdf>  
Photos: <https://raphaelstaebler.medium.com/>

# CPU

gb, gbc

- The CPU core in the Game Boy SoC is a custom Sharp design without a name.
- Some sources claim Game Boy uses a “modified” Zilog Z80 or Intel 8080.
- Using old datasheets and databooks, the core has been identified to be a **Sharp SM83**.

More about CPU: <https://gekkio.fi/files/gb-docs/gbctr.pdf>  
Photos: <https://www.copetti.org/writings/consoles/game-boy/>



03. Make games

# ASM

fosdem: vim main.asm

```
INCLUDE "hardware.inc"
SECTION "Header", ROM0[$100]
jp EntryPoint
ds $150 - @, 0

EntryPoint:
call WaitVBlank
ld a, 0
ld [rLCDC], a
ld de, player_right
ld hl, $8400
ld bc, player_rightEnd - player_right
call Memcopy
ld de, player_left
ld hl, $8000
ld bc, player_leftEnd - player_left
call Memcopy
ld a, 0
ld b, 160
ld hl, _OAMRAM
ClearOam:
ld [hli], a
dec b
jp nz, ClearOam
ld hl, _OAMRAM
ld a, 88
ld [hli], a
ld a, 88
ld [hli], a
ld a, 0
ld [hli], a
ld a, 0
ld [hli], a
```

# ASM

Rednex Game Boy Development System

- Four programs to cover the whole compilation pipeline:
- Image converter / Assembler / Linker / Fixer

**<https://rgbds.gbdev.io/>**

# RGBDS

A free assembler/linker package for the Game Boy and Game Boy Color

Install

Read manual

Try online

C

# GBDK-2020 - Game Boy Development Kit

## gbdk-2020

An updated version of GBDK, C compiler, assembler, linker and set of libraries for the Nintendo Gameboy, Nintendo Entertainment System, Sega Master System, Sega Game Gear.

[View the Project on GitHub](#)  
gbdk-2020/gbdk-2020

## GBDK-2020

GBDK is a cross-platform development kit for sm83, z80 and 6502 based gaming consoles. It includes libraries, toolchain utilities and the [SDCC C](#) compiler suite.

### Supported Consoles: ([see docs](#))

- Nintendo Game Boy / Game Boy Color
- Analogue Pocket
- Sega Master System & Game Gear
- Mega Duck / Cougar Boy
- NES

Experimental consoles (not yet fully functional)

- MSXDOS

# GB Studio

- It is the most advanced retro game creator. It is a complete engine to create complete games. It is based on **GBDK** and **GBVM**
- 

**<https://www.gbstudio.dev/>**



The screenshot shows the top navigation bar of the GB Studio website. It includes a logo with a handheld device icon, the text "GB Studio", a language dropdown set to "English", and links for "About", "Docs", "GitHub", "Download", "Search", "Keyboard Shortcuts", "Donate", and a brightness slider.

A **quick** and **easy** to use **drag and drop** retro **game creator** for your favourite handheld video game system.

Available on Windows, Mac and Linux.

**Download on Itch.io**



## 04. ...and Rust?

# What we have

Emulators! Emulators everywhere!

- **Mooneye GB** – A Game Boy research project and emulator written in Rust
  - Code: <https://github.com/Gekkio/mooneye-gb>
- **Boytacean** – Full-featured Rust emulator with Web, SDL & Libretro frontends
  - Code: <https://github.com/joamag/boytacean>
- **Retro Boy** – Cycle-accurate emulator compiled to WebAssembly
  - Code: <https://github.com/smparsons/retroboy>
- **Wasm-GB** – Game Boy emulator in WebAssembly + WebGL 2.0 (Rust)
  - Code: <https://github.com/andrewimm/wasm-gb>
- **gameboy** – Game Boy emulator written in Rust
  - Code: <https://github.com/raphamorim/gameboy>

# What we have

Crates for gba

- There are some crates to make games for gba:
  - **gba**
  - **agb**
  - Others project educational / list of utilities
  - As said, the gba has a different architecture and a different processor. gba have an ARM CPU.



# Rust-GB

By zlfn

- Is a project work in progress...
- And try to obtain the build with a “workaround.”
  1. The **Rust** compiler can generate **LLVM-IR** for the **ATMega328**
  2. **LLVM-IR** can be converted in **C** with *llvm-cbe*
  3. **C** compiled to **Z80 assembly** with *sdasgb*
- 4. **Z80 Assembly** can be assembled into **GBZ80** with *sdasgb*
- 5. **GBZ80 object code** can be linked in a **ROM gb** with *GBDK*
- There is another project by zlfn: cranelift-z80



<https://github.com/zlfn/rust-gb>

<https://github.com/zlfn/cranelift-z80>

05. Rustboy

# The idea

- We have in front of us only one CPU...
- Can we do a specific Rust compiler for the **SM83**?
- **Great idea!** I always develop a compiler!

<https://github.com/ffex/rust-boy>

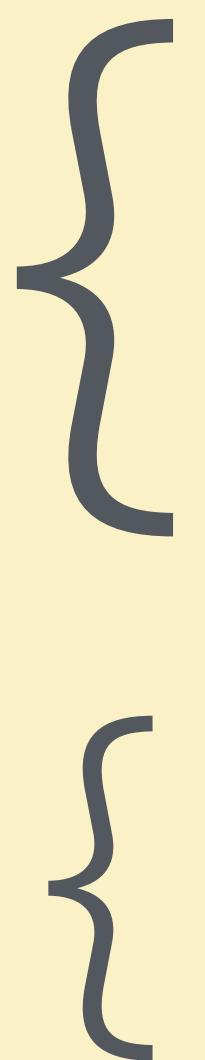


# PoC

Proof of concept

To speed up the development, I put a solid working base.

RustBoy



rust\_boy

gb\_std

gb\_asm

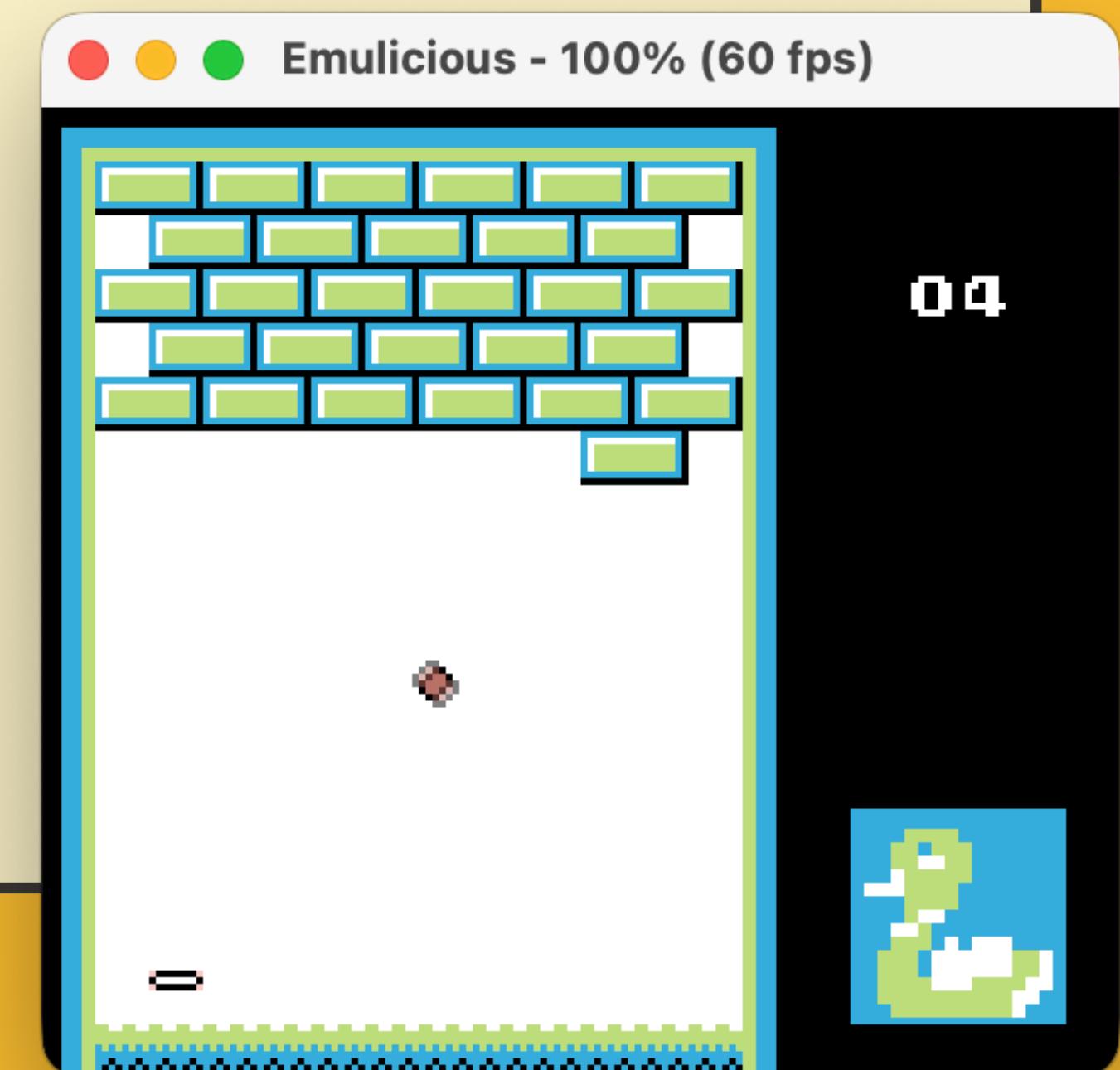
ASM

RGBDS

# How show results?

Unbricked - an Arkanoid copy

- In [gbdev.io](#), as an example to illustrate how to create games in asm, the initial example is a copy of the famous Arkanoid.
- This example is important to see what happens when we go up to the high level to the code
- So let me explain some part of this game in asm



# Unbricked

Inits

● ● ● 2 originals: vim main.asm

```
INCLUDE "hardware.inc"

DEF BRICK_LEFT EQU $05
DEF BRICK_RIGHT EQU $06
DEF BLANK_TILE EQU $08
DEF DIGIT_OFFSET EQU $1A
DEF SCORE_TENS EQU $9870
DEF SCORE_ONES EQU $9871

SECTION "Header", ROM0[$100]

jp EntryPoint

ds $150 - @, 0 ; room for header

EntryPoint:

WaitVBlank:
    ld a, [rLY]
    cp 144
    jp c, WaitVBlank

    ; Turn off LCD
    ld a, 0
    ld [rLCDC], a

    ; Copy tiles data
    ld de, Tiles
    ld hl, $9000
    ld bc, TilesEnd - Tiles
    call Memcopy

    ; Copy the tilemap
"main.asm" 757L, 15183B
```

# Unbricked

## Variables

● ○ ● ↵ 2 originals: vim main.asm

Paddle:

```
dw `13333331  
dw `30000003  
dw `13333331  
dw `00000000  
dw `00000000  
dw `00000000  
dw `00000000  
dw `00000000
```

PaddleEnd:

Ball:

```
dw `00033000  
dw `00322300  
dw `03222230  
dw `03222230  
dw `00322300  
dw `00033000  
dw `00000000  
dw `00000000
```

BallEnd:

```
SECTION "Counter", WRAM0  
wFrameCounter: db
```

```
SECTION "Input Variables", WRAM0
```

```
wCurKeys: db  
wNewKeys: db
```

```
SECTION "Ball Data", WRAM0
```

```
wBallMomentumX: db  
wBallMomentumY: db
```

```
SECTION "Score", WRAM0
```

```
wScore: db
```

# Unbricked

## Tiles and Tilemap

⌚ 2 originals: vim main.asm

```
dw `33300333
dw `33000333
dw `33000333
dw `33333333
; 8
dw `33333333
dw `33000033
dw `30333003
dw `33000033
dw `30333003
dw `30333003
dw `30333003
dw `33000033
dw `33333333
; 9
dw `33333333
dw `33000033
dw `30330003
dw `30330003
dw `33000003
dw `33330003
dw `33000033
dw `33333333

TilesEnd:

Tilemap:
    db $00, $01, $01, $01, $01, $01, $01, $01, $01, $01, $01, $01, $01, $02, $03, $03, $03, $03,
0,0,0,0,0,0,0,0,0,0
    db $04, $05, $06, $05, $06, $05, $06, $05, $06, $05, $06, $05, $06, $07, $03, $03, $03, $03, $03,
0,0,0,0,0,0,0,0,0,0
    db $04, $08, $05, $06, $05, $06, $05, $06, $05, $06, $05, $06, $08, $07, $03, $03, $03, $03, $03,
0,0,0,0,0,0,0,0,0,0
    db $04, $05, $06, $05, $06, $05, $06, $05, $06, $05, $06, $05, $06, $07, $03, $03, $03, $03, $03,
0,0,0,0,0,0,0,0,0,0
```

# Unbricked

## Memcpy

● ● ● 2 originals: vim main.asm

```
ld a, 0
ld [rLCDC], a

; Copy tiles data
ld de, Tiles
ld hl, $9000
ld bc, TilesEnd - Tiles
call Memcopy

; Copy the tilemap
ld de, Tilemap
ld hl, $9800
ld bc, TilemapEnd - Tilemap
call Memcopy

; Copy the paddle tile
ld de, Paddle
ld hl, $8000
ld bc, PaddleEnd - Paddle
call Memcopy

; Copy the balltile
ld de, Ball
ld hl, $8010
ld bc, BallEnd - Ball
call Memcopy

; initialize OAM
ld a, 0
ld b, 160
ld hl, _OAMRAM
```

ClearOam:

# Unbricked

## Functions

```
○ ○ ○  ↵⌘2 originals: vim main.asm

jp Main
; Copy bytes from one area to another
; @param de: source
; @param hl: destination
; @param bc: lenght
Memcpy:
    ld a, [de]
    ld [hl], a
    inc de
    dec bc
    ld a, b
    or a, c
    jp nz, Memcopy
    ret

UpdateKeys:
    ; poll half the controller
    ld a, P1F_GET_BTN
    call .onenibble
    ld b, a ; B7-4 = 1; B3-0 = unpressed button

    ; poll the other half
    ld a, P1F_GET_DPAD
    call .onenibble
    swap a ; A7-4 upressed direction; a3-0 =1
    xor a, b ; A= pressed button + directions
    ld b,a ;B = pressed buttons + directions

    ; And release the controller
    ld a, P1F_GET_NONE
    ldh [rP1], a

    ; Combine with previous wCurKeys to make wNew Keys
    ld a, [wCurKeys]
```

# Unbricked

## Main loop - Input

```
● ○ ●  ↵⌘2 originals: vim main.asm

PaddleBounceDone:

    call UpdateKeys

    ; First check if the left button is pressed
CheckLeft:
    ld a, [wCurKeys]
    and a, PADF_LEFT
    jp z, CheckRight

Left:
    ; move the paddle one pixel to the left
    ld a, [_OAMRAM+1]
    dec a
    cp a, 15
    jp z, Main
    ld [_OAMRAM+1], a
    jp Main

CheckRight:
    ld a, [wCurKeys]
    and a, PADF_RIGHT
    jp z, Main

Right:
    ; move the paddle one pixel to the left
    ld a, [_OAMRAM+1]
    inc a
    cp a, 105
    jp z, Main
    ld [_OAMRAM+1], a
    jp Main

; Copy bytes from one area to another
; @param de: source
; @param hl: destination
; @param bc: lenght
```

# Unbricked

## Main loop - Movement

● ○ ● ↵2 originals: vim main.asm

```
; Wait until it's *not* VBlank
ld a, [rLY]
cp 144
jp nc, Main
WaitVBlank2:
ld a, [rLY]
cp 144
jp c, WaitVBlank2

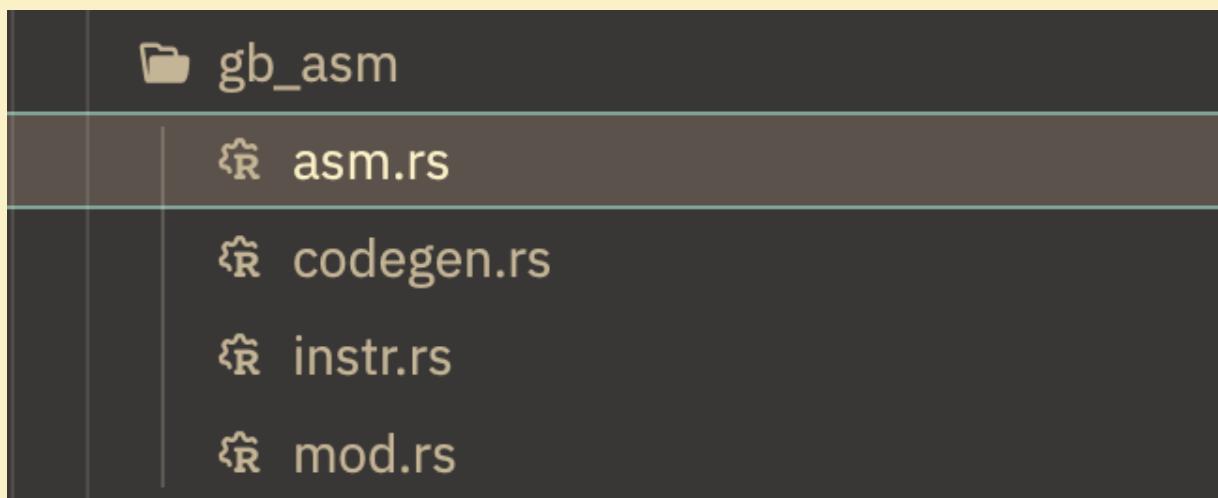
;Add the ball's momentum to its position in OAM
ld a, [wBallMomentumX]
ld b, a
ld a, [_OAMRAM +5]
add a, b
ld [_OAMRAM +5], a

ld a, [wBallMomentumY]
ld b, a
ld a, [_OAMRAM +4]
add a, b
ld [_OAMRAM +4], a

BounceOnTop:
; Remember to offset the OAM position!
; (8, 16) in OAM coordinates is (0, 0) on the screen.
ld a, [_OAMRAM + 4]
sub a, 16 + 1
ld c, a
ld a, [_OAMRAM + 5]
sub a, 8
ld b, a
call GetTileByPixel ; Returns tile address in hl
ld a, [hl]
```

# gb\_asm

- It is the most low-level library.
- Almost one-to-one with the asm but in Rust!



- Instructions are something like: `asm.ld(...)`, `asm.cp(...)`, ...
- Link to an article that inspire me:

<https://tinycomputers.io/posts/building-z80-roms-with-rust-a-modern-approach-to-retro-computing.html>

# gb\_asm

## unbricked.rs

```
● ● ●  ↵2 bin: vim unbricked.rs

use rust_boy::gb_asm::{Asm, Condition, Operand, Register};

fn main() {
    let mut asm = Asm::new();

    // Hardware include and constants
    asm.include_hardware();
    asm.def("BRICK_LEFT", 0x05);
    asm.def("BRICK_RIGHT", 0x06);
    asm.def("BLANK_TILE", 0x08);
    asm.def("DIGIT_OFFSET", 0x1A);
    asm.def("SCORE_TENS", 0x9870);
    asm.def("SCORE_ONES", 0x9871);

    // Header section
    asm.section("Header", "ROM0[$100]");
    asm.jp("EntryPoint");
    asm.ds("$150 - @", "0");

    // Entry point
    asm.label("EntryPoint");
    asm.label("WaitVBlank");
    asm.ld_a_addr_def("rLY");
    asm.cp_imm(144);
    asm.jp_cond(Condition::C, "WaitVBlank");

    // Turn off LCD
    asm.ld_a(0);
    asm.ld_addr_def_a("rLCDC");

    // Copy tiles data
    asm.ld_de_label("Tiles");
    asm.ld_hl_label("$9000");
"unbricked.rs" 879L, 24318B
```

# gb\_std

- This lib is more of a high-level and implements:
  - A chunk system (Main, Functions, Tiles, etc.) so you can put code from everywhere, and at the time of the generation are put in the right section.
  - Tile and tilemap utilities
  - A sprite manager
  - The initial attempt at an if statement.

# gb\_std

● ● ● 7%2 rust-boy: vim src/bin/unbrick\_std/main.rs

```
// add("paddle" in WRAM)
// automatically manage the address ($8000 and after $8010)
asm.chunk(rust_boy::gb_asm::Chunk::Tiles);

asm.emit_all(add_tiles("Tiles", tiles::TILES));
asm.emit_all(add_tiles("Ball", tiles::BALL));
asm.emit_all(add_tiles("Paddle", tiles::PADDLE));

asm.chunk(rust_boy::gb_asm::Chunk::Main);
asm.emit_all(cp_in_memory("Tiles", "$9000"));
asm.emit_all(cp_in_memory("Ball", "$8010"));
asm.emit_all(cp_in_memory("Paddle", "$8000"));
asm.emit_all(cp_in_memory("Tilemap", "$9800"));

//FLOW1 we continue with the main
asm.emit_all(initialize_objects_screen());
asm.emit_all(clear_objects_screen());

//Sprite managment
let mut sprite_manager = SpriteManager::new();
sprite_manager.add_sprite(16, 128, 0, 0);
sprite_manager.add_sprite(32, 100, 1, 0);
asm.ld_a(1);
asm.ld_addr_def_a("wBallMomentumX");
asm.ld_a_label("-1");
asm.ld_addr_def_a("wBallMomentumY");
asm.emit_all(sprite_manager.draw());

asm.emit_all(turn_on_screen());
asm.ld_a(0b11100100);
asm.ld_addr_def_a("rBGP");
asm.ld_a(0b11100100);
asm.ld_addr_def_a("rOBP0");
```

# gb\_std

## Utilities

```
● ● ●  ↵⌘2  rust-boy: vim src/gb_std/graphics/utility.rs

use crate::gb_asm::{Asm, Condition, Instr, Operand, Register};

//TODO
// refactor code:
// - punt in the form of builder (like cp_in_memory)

pub fn add_tiles(label: &str, tiles: &[&[&str; 8]]) -> Vec<Instr> {
    let mut asm = Asm::new();
    asm.label(label);
    for tile in tiles {
        for line in tile {
            asm.dw(line);
        }
    }
    asm.label(&format!("{}End", label));
    asm.get_main_instrs()
}

pub fn add_tiles_2bpp(label: &str, path: &str) -> Vec<Instr> {
    let mut asm = Asm::new();
    asm.label(label);
    asm.incbin(path);
    asm.label(&format!("{}End", label));
    asm.get_main_instrs()
}

pub fn add_tiles_tilemap(label: &str, path: &str) -> Vec<Instr> {
    let mut asm = Asm::new();
    asm.label(label);
    asm.incbin(path);
    asm.label(&format!("{}End", label));
    asm.get_main_instrs()
}

"src/gb_std/graphics/utility.rs" 163L, 5212B
```

# gb\_std

If statement

● ● ● 2 rust-boy: vim src/bin/unbricked\_std/main.rs

```
//TODO refactorBounceDone
asm.comment("TESTBOUNCEDONCE");
asm.emit_all(sprite_manager.get_sprite(0).unwrap().get_y(Register::B));
asm.emit_all(sprite_manager.get_sprite(1).unwrap().get_y(Register::A));
asm.add(Operand::Reg(Register::A), Operand::Imm(5));
let if__ball_y_check = If::new(
    IfCondition::new(
        ConditionOperand::Register(Register::A),
        ConditionOperand::Register(Register::B),
        rust_boy::gb_std::flow::ComparisonOp::E,
    ),
    {
        let mut bounce_x_check = Asm::new();
        bounce_x_check.emit_all(sprite_manager.get_sprite(1).unwrap().get_x(Register::B));
        bounce_x_check.emit_all(sprite_manager.get_sprite(0).unwrap().get_x(Register::A));
        bounce_x_check.sub(Operand::Reg(Register::A), Operand::Imm(8));
        let if__ball_y_check = If::new(
            IfCondition::new(
                ConditionOperand::Register(Register::A),
                ConditionOperand::Register(Register::B),
                rust_boy::gb_std::flow::ComparisonOp::LT,
            ),
            {
                let mut bounce_x_check_2 = Asm::new();
                bounce_x_check_2.add(Operand::Reg(Register::A), Operand::Imm(8 + 16));
                let if__ball_x_check_2 = If::new(
                    IfCondition::new(
                        ConditionOperand::Register(Register::A),
                        ConditionOperand::Register(Register::B),
                        rust_boy::gb_std::flow::ComparisonOp::GE,
                    ),
                    {
                        let mut bounce = Asm::new();
```

# rust\_boy

- With gb\_std we understood what we can level up:
  - If statements have to be simpler
  - The init instructions must be written in an automatic way
  - There are some functions that can be considered “BuiltIn (UpdateKeys, WaitVBlank, Memcopy, etc.)
  - Make more managers (Tiles, Inputs, etc.)
  - Hide every reference to the memory address

# rust\_boy

## unbrickeds.rs

```
rust-boy: vim src/bin/unbrickeds_rustboy/main.rs (-zsh) #1
rust-boy: vim src/rust_boy/rustboy.rs (-zsh)

fn main() {
    let mut gb = RustBoy::new();

    // =====
    // CONSTANTS - No more manual DEF statements!
    // =====
    gb.define_const("BRICK_LEFT", "0x05")
        .define_const("BRICK_RIGHT", "0x06")
        .define_const("BLANK_TILE", "0x08")
        .define_const("DIGIT_OFFSET", "0x1A")
        .define_const_hex("SCORE_TENS", 0x9870)
        .define_const_hex("SCORE_ONES", 0x9871);

    // =====
    // TILES - Auto VRAM allocation!
    // =====
    // Background tiles go to $9000
    gb.tiles
        .add_background("Tiles", TileSource::from_raw(tiles::TILES));

    // Tilemap goes to $9800
    gb.tiles.add_tilemap("Tilemap", tilemap::TILEMAP);

    // Sprites: tile + position + OAM in one call!
    let paddle = gb.add_sprite("Paddle", TileSource::from_raw(tiles::PADDLE), 16, 128, 0);
    let ball = gb.add_sprite("Ball", TileSource::from_raw(tiles::BALL), 32, 100, 0);

    // =====
    // VARIABLES - Auto WRAM allocation!
    // =====
    let _frame_counter = gb.vars.create_u8("wFrameCounter", 0);
    let _cur_keys = gb.vars.create_u8("wCurKeys", 0);
    let _new_keys = gb.vars.create_u8("wNewKeys", 0);
```

# rust\_boy

## unbrickeds.rs



The image shows a terminal window with three tabs. The active tab on the left contains the title 'rust\_boy' and 'unbrickeds.rs'. The other two tabs are labeled 'rust-boy: vim src/bin/unbrickeds\_rustboy/main.rs (-zsh)' and 'rust-boy: vim src/rust\_boy/rustboy.rs (-zsh)'. The main pane displays the following Rust code:

```
// =====
// FUNCTIONS - Auto WRAM allocation!
// =====
gb.define_function(
    "IsWallTile",
    is_specific_tile(
        "IsWallTile",
        &["$00", "$01", "$02", "$04", "$05", "$06", "$07"],
    ),
);
gb.define_function_from(
    "CheckAndHandleBrick",
    vec![
        boxed(IfConst::eq(
            Call::with_args("GetTileByPixel", gb.sprites.get_pivot(ball, 0, 1)),
            "BRICK_LEFT",
            vec![
                TileRef::set_tile_label("BLANK_TILE"),
                TileRef::next_tile(),
                TileRef::set_tile_label("BLANK_TILE"),
            ],
        )),
        boxed(IfA::eq(
            "BRICK_RIGHT",
            vec![
                TileRef::set_tile_label("BLANK_TILE"),
                TileRef::prev_tile(),
                TileRef::set_tile_label("BLANK_TILE"),
            ],
        )),
    ],
);
```

# rust\_boy

## unbrickeds.rs

```
rust-boy: vim src/bin/unbrickeds_rustboy/main.rs (-zsh) #1 rust-boy: vim src/rust_boy/rustboy.rs (-zsh)

// =====
// MAIN LOOP - Game logic
// =====

// Ball movement
gb.add_to_main_loop(gb.sprites.move_x_var(ball, "wBallMomentumX"));
gb.add_to_main_loop(gb.sprites.move_y_var(ball, "wBallMomentumY"));

// Bounce on top
gb.call_args("GetTileByPixel", gb.sprites.get_pivot(ball, 0, 1));
gb.add_to_main_loop>IfCall::is_true(
    "IsWallTile",
    vec![
        boxed(Call::new("CheckAndHandleBrick")),
        boxed(_ball_momentum_y.set(1)),
    ],
);

// Bounce on right
gb.call_args("GetTileByPixel", gb.sprites.get_pivot(ball, -1, 0));
gb.add_to_main_loop>IfCall::is_true("IsWallTile", _ball_momentum_x.set(-1));

// Bounce on left
gb.call_args("GetTileByPixel", gb.sprites.get_pivot(ball, 1, 0));
gb.add_to_main_loop>IfCall::is_true("IsWallTile", _ball_momentum_x.set(1));

// Bounce on bottom
gb.call_args("GetTileByPixel", gb.sprites.get_pivot(ball, 0, -1));
gb.add_to_main_loop>IfCall::is_true("IsWallTile", _ball_momentum_y.set(-1));
gb.add_to_main_loop{
    // make a debug label in as API
    let mut lbl_debug = Asm::new();
    lbl_debug.label("PaddleBounce");
}
```

# rust\_boy

## unbricked.rs

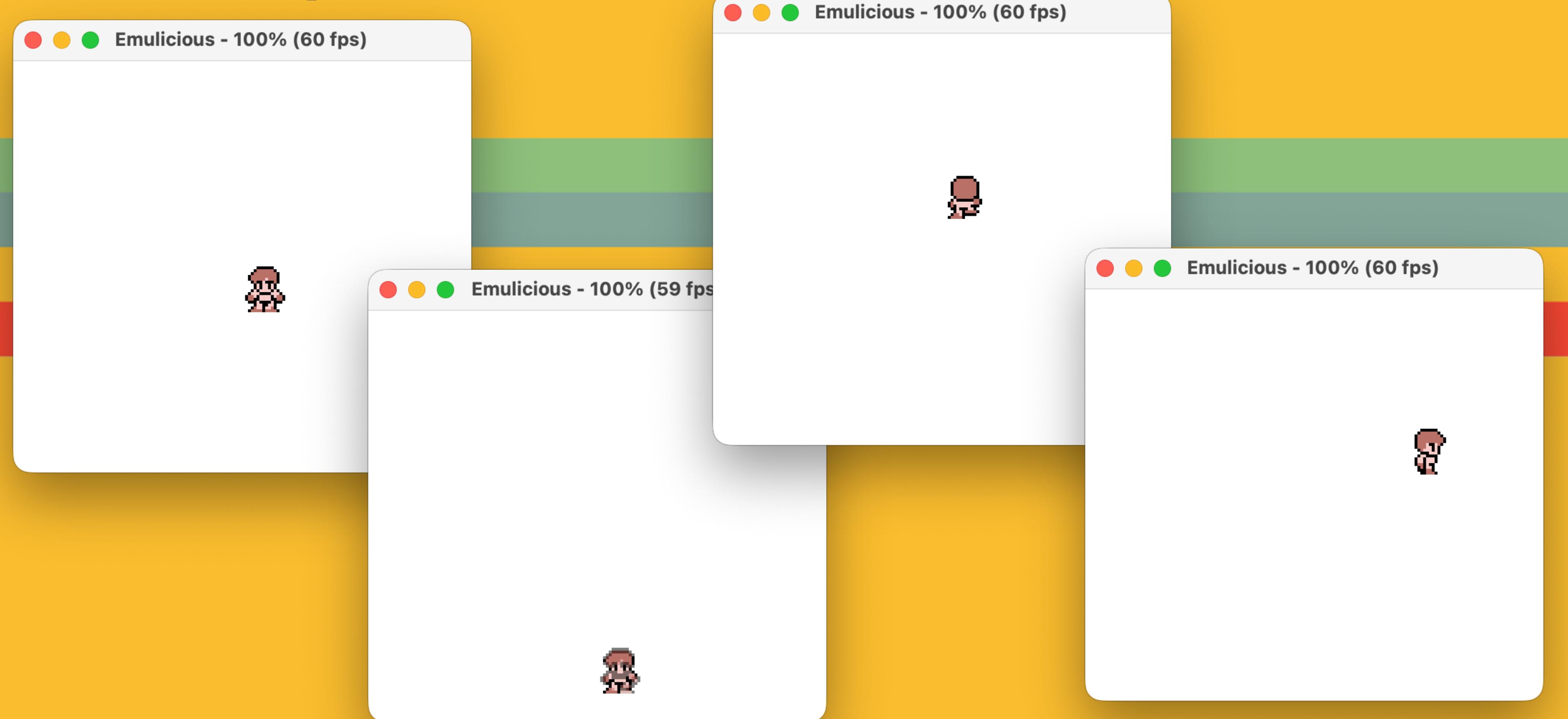
```
...boy: vim src/bin/unbricked_rustboy/main.rs (-zsh) ⌘1 rust-boy: vim src/rust_boy/rustboy.rs (-zsh)

// Paddle bounce
let paddle_bounce = If::eq(
    gb.sprites.get_y(paddle),
    gb.sprites.get_y(ball).plus(5),
If::lt(
    gb.sprites.get_x(ball),
    gb.sprites.get_x(paddle).minus(8),
    If::ge(gb.sprites.get_x(ball), gb.sprites.get_x(paddle).plus(16), {
        _ball_momentum_y.set(-1)
    }),
),
);
gb.add_to_main_loop(paddle_bounce);
gb.add_to_main_loop({
    // make a debug label in as API
    let mut lbl_debug = Asm::new();
    lbl_debug.label("PaddleBounceEND");
    lbl_debug.get_main_instrs()
});
// Input handling
let mut inputs = InputManager::new();
inputs.on_press(Button::Left, gb.sprites.move_left_limit(paddle, 1, 15));
inputs.on_press(
    Button::Right,
    gb.sprites.move_right_limit(paddle, 1, 105),
);
gb.add_inputs(inputs);

// =====
// BUILD AND OUTPUT
// =====
println!("{}", gb.build());
}
```

# Try `rust_boy!`

The fosdem example



# rust\_boy

at Fosdem!

```
rust-boy: vim src/bin/fosdem/main.rs (-zsh) ⌘1 rust-boy: vim src/rust_boy/rustboy.rs (-zsh)
```

```
fn main() {
    let mut gb = RustBoy::new();

    // Add 16x16 composite sprite (two 8x16 sprites side by side)
    let player = gb.add_sprite_16x16(
        "player",
        TileSource::from_file("char.2bpp", 64),
        TileSource::from_file("char-dx.2bpp", 64),
        80,
        72,
        0,
    );

    // Add looping animations to the composite sprite (applies to both halves)
    // add_composite_animation returns the animation index
    // Animation order: front, back, left, right (frames 0-3, 4-7, 8-11, 12-15)
    let anim_walk_front =
        gb.sprites
            .add_composite_animation(player, "playerWalkFront", 0, 3, AnimationType::Loop);
    let anim_walk_back =
        gb.sprites
            .add_composite_animation(player, "playerWalkBack", 4, 7, AnimationType::Loop);
    let anim_walk_left =
        gb.sprites
            .add_composite_animation(player, "playerWalkLeft", 8, 11, AnimationType::Loop);
    let anim_walk_right =
        gb.sprites
            .add_composite_animation(player, "playerWalkRight", 12, 15, AnimationType::Loop);

    // Start with no animation (disabled)
    gb.sprites
        .set_composite_initial_animation(player, ANIM_DISABLED);
```



# rust\_boy

at Fosdem!

```
rust-boy: vim src/bin/fosdem/main.rs (-zsh) ⌘1 rust-boy: vim src/rust_boy/rustboy.rs (-zsh)  
]  
.concat(),  
);  
inputs.on_press(  
    PadButton::Right,  
    [  
        gb.sprites.move_composite_right_limit(player, 1, 150),  
        gb.sprites  
            .enable_composite_animation(player, anim_walk_right),  
    ]  
.concat(),  
);  
inputs.on_press(  
    PadButton::Up,  
    [  
        gb.sprites.move_composite_up_limit(player, 1, 0),  
        gb.sprites  
            .enable_composite_animation(player, anim_walk_back),  
    ]  
.concat(),  
);  
inputs.on_press(  
    PadButton::Down,  
    [  
        gb.sprites.move_composite_down_limit(player, 1, 150),  
        gb.sprites  
            .enable_composite_animation(player, anim_walk_front),  
    ]  
.concat(),  
);  
gb.add_inputs(inputs);  
println!("{}", gb.build());  
}
```

# What's next?

Improvement

- Refactor the code:
  - It is written in a cumulative way so there are some optimization to do.
  - Try to write new examples to find new built-in functions
    - Example: implement the sound manager.

# What's next?

At low-level

An assembler and linker to generate the  
gb ROM

RustBoy

rust\_boy

gb\_std

gb\_asm

Assembler/linker  
KODDS

# What's next?

High-level

An assembler and linker to generate the  
gb ROM

RustBoy

Rust parser

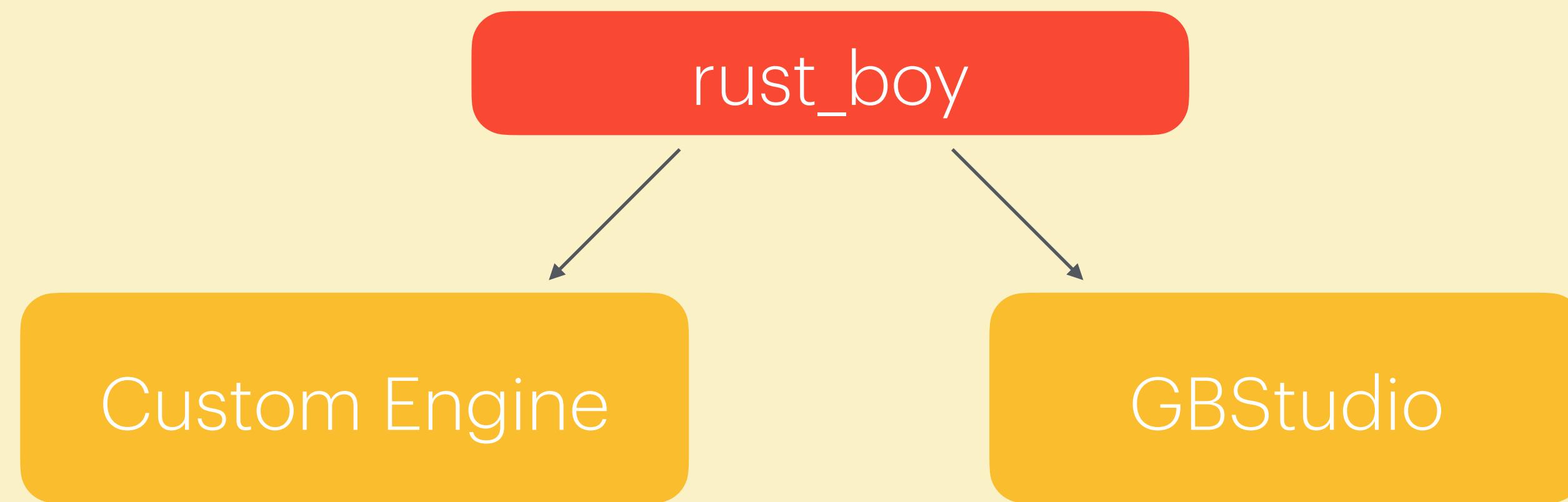
rust\_boy

gb\_std

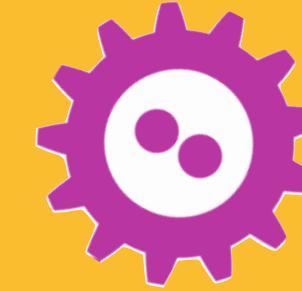
gb\_asm

Assembler/linker  
ROBDS

# One more thing...



# Thank you



<https://github.com/ffex/rust-boy>



<https://github.com/ffex>



<https://www.linkedin.com/in/federico-bassini/>



<https://mastodon.social/@ffex>



[https://www.instagram.com/ffex\\_tech/](https://www.instagram.com/ffex_tech/)