Rust-boy

A Rust journey into Game Boy dev



ffex @ Rustlab

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
- Copy a *solution* of the **problem**
- Test the *solution*
- It works? No loop, yes go ahead
- Is it the best *solution*?
- Search for the best *solution* online
- .....

pre-LLM

- We have a **problem**
- prepare context, skills and ask a coding agent where the **problem** is
- wait for the agent's answer
- read the agent answer
- say "yes, implement it in automode"
- Test the *solution*
- read the solution and ask myself if it is the best one.
- Tell the agent if there are better solutions
- .....

post-LLM

Fosdem 2025

Boom!





Fosdem 2025

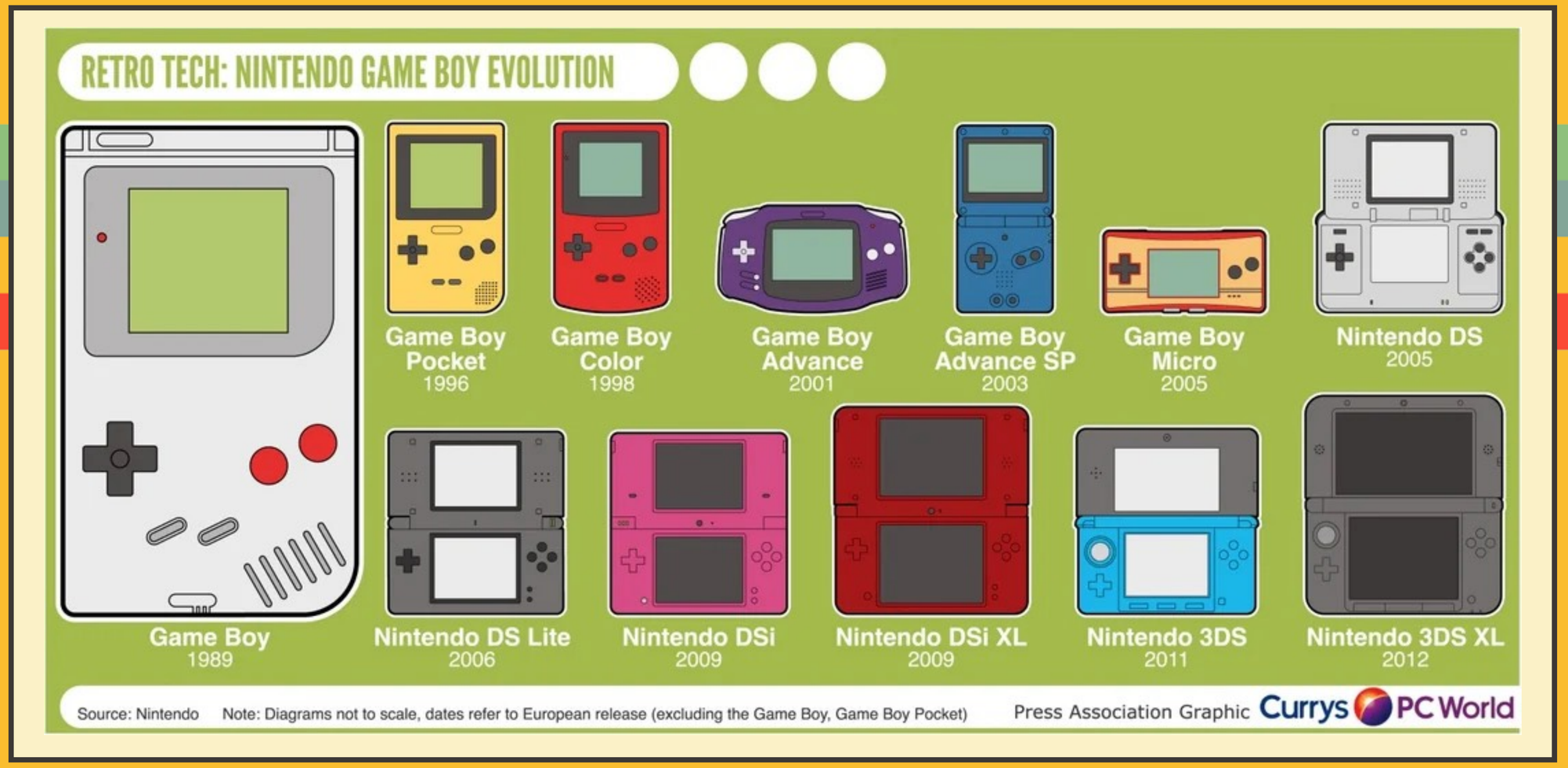
Boom!



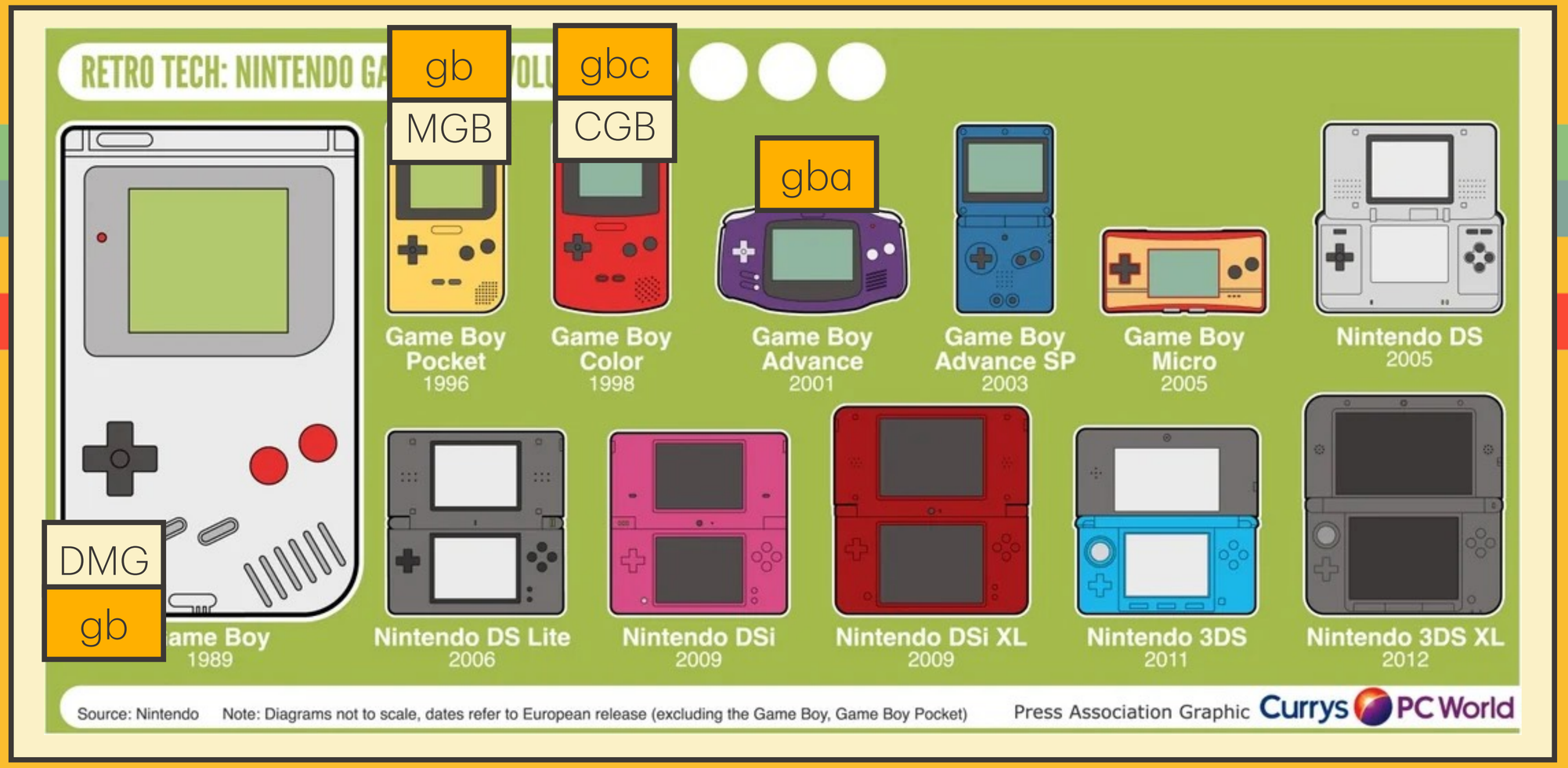


02. Hardware

All the game boys



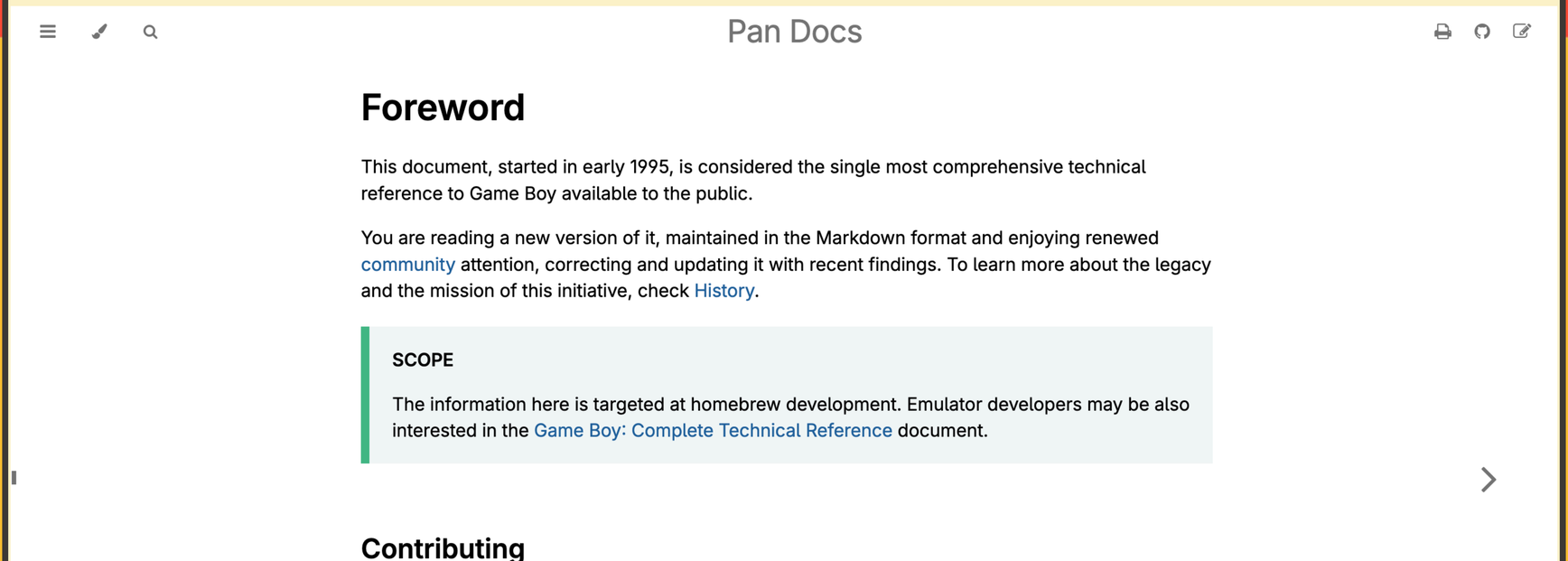
All the game boys



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/**



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 mapper (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	Echo RAM (mirror of C000–DDFF)	Nintendo says use of this area is prohibited.
FE00	FE9F	Object attribute memory (OAM)	
FEA0	FEFF	Not Usable	Nintendo says use of this area is prohibited.
FF00	FF7F	I/O Registers	
FF80	FFFE	High RAM (HRAM)	
FFFF	FFFF	Interrupt Enable register (IE)	

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

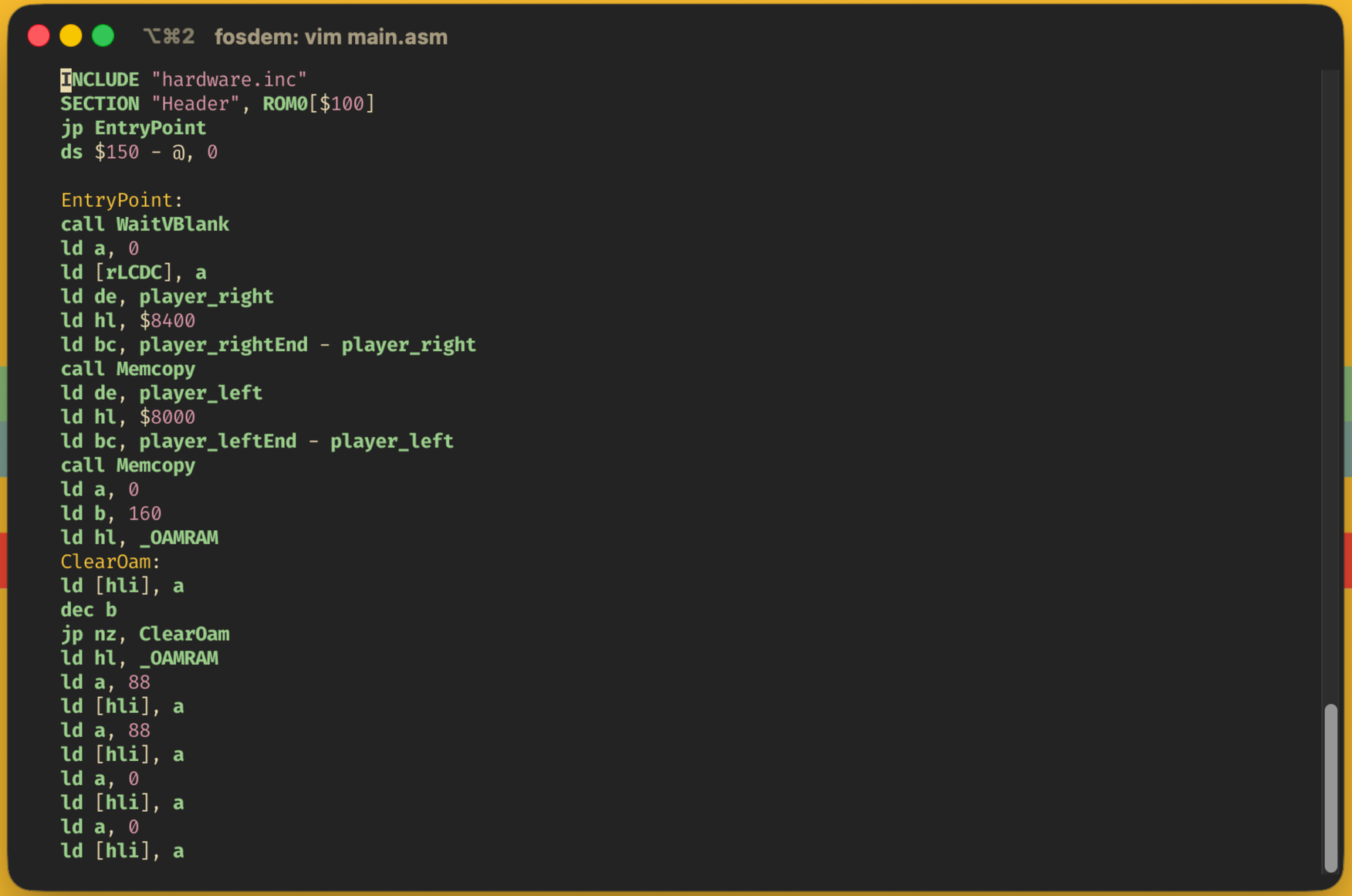
- The CPU core in the Game Boy SoC is a custom Sharp design without a name.
- Some sources claim Game Boy uses a “modified” Zilog Z80 or Intel 8080.
- Using old datasheets and databooks, the core has been identified to be a **Sharp SM83.**



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

03. Make games

ASM



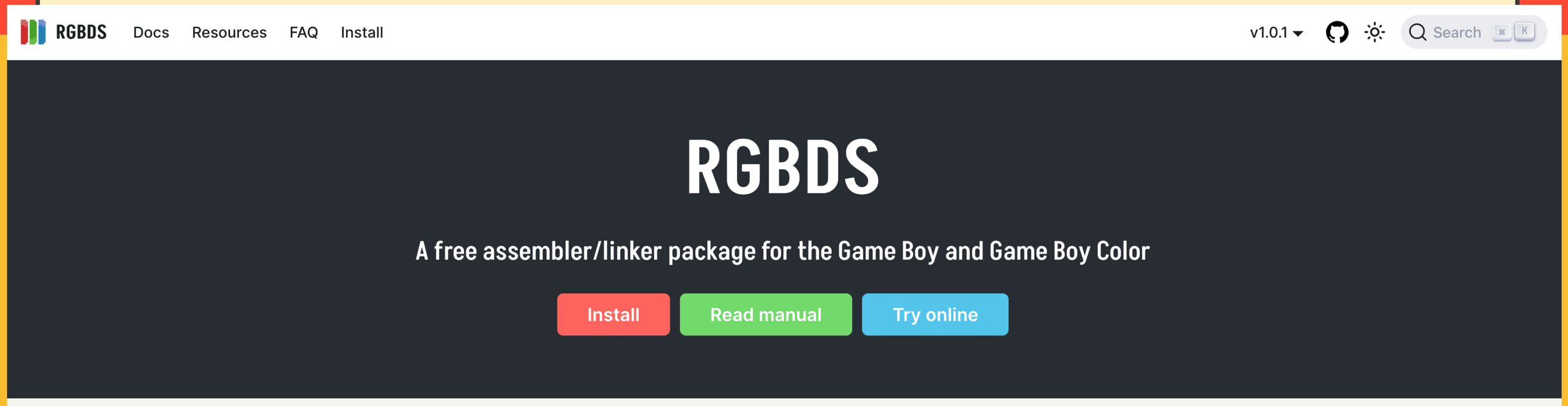
https://gbdev.io/gb-asm-tutorial/

ASM

Rednex Game Boy Development System

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

https://rgbds.gbdev.io/



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)

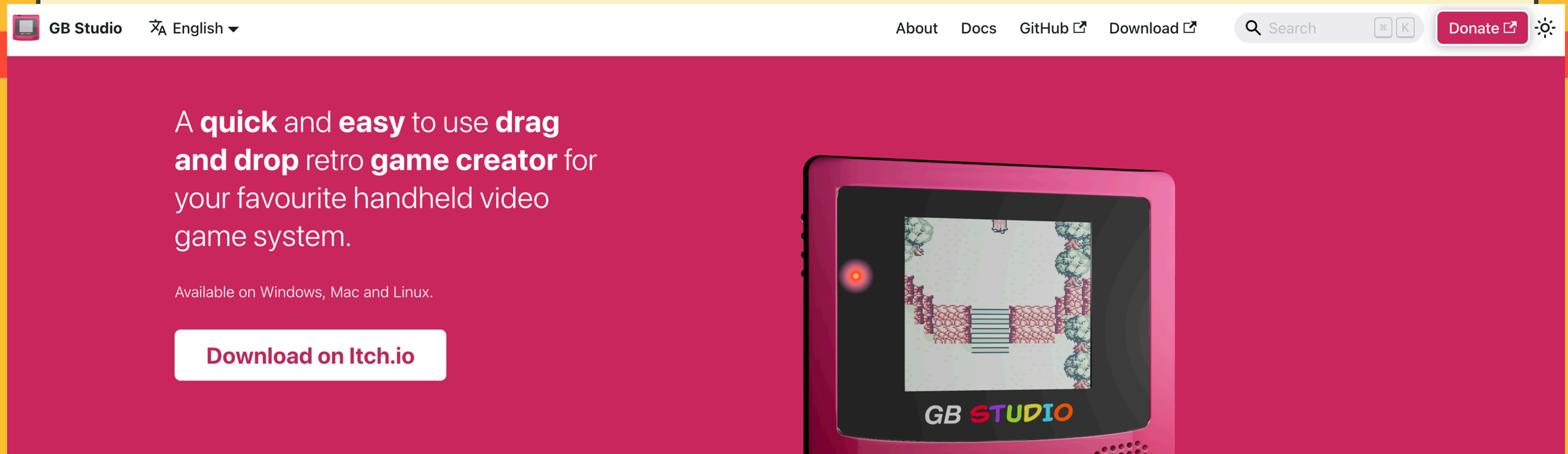
- MSXDOS

https://gbdk.org/

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/



04. ...and Rust?

What we have

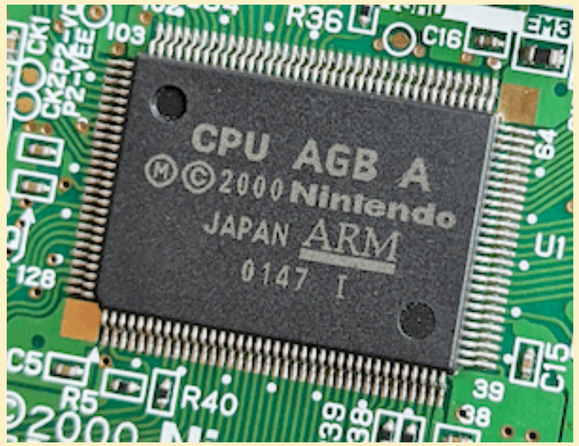
Emulators! Emulators everywhere!

- **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**
  - Other educational projects and lists of utilities



- The gba has a different architecture and a different processor it has an ARM CPU.

Rust-GB

By zlfn

- Is a project work in progress...
- Right now use this toolchain to obtain results:
  1. Forked **Rust** compiler to compile a **LLVM-IR** for the **SM83**
  2. A fork of **LLVM** with an SM83 backend produces SM83 object files (.o)
  3. A **runtime library**: an assembly entry point that sets up the Game Boy hardware and calls main
  4. A **linker and fixer** to obtain the final **ROM**



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

05. Rust-boy

The idea

- We have in front of us only one CPU...
- We have to manage in a specific way memory and register
- Maybe is create a compiler for the **SM83**?
- **Great idea!** I have always wanted to develop a compiler

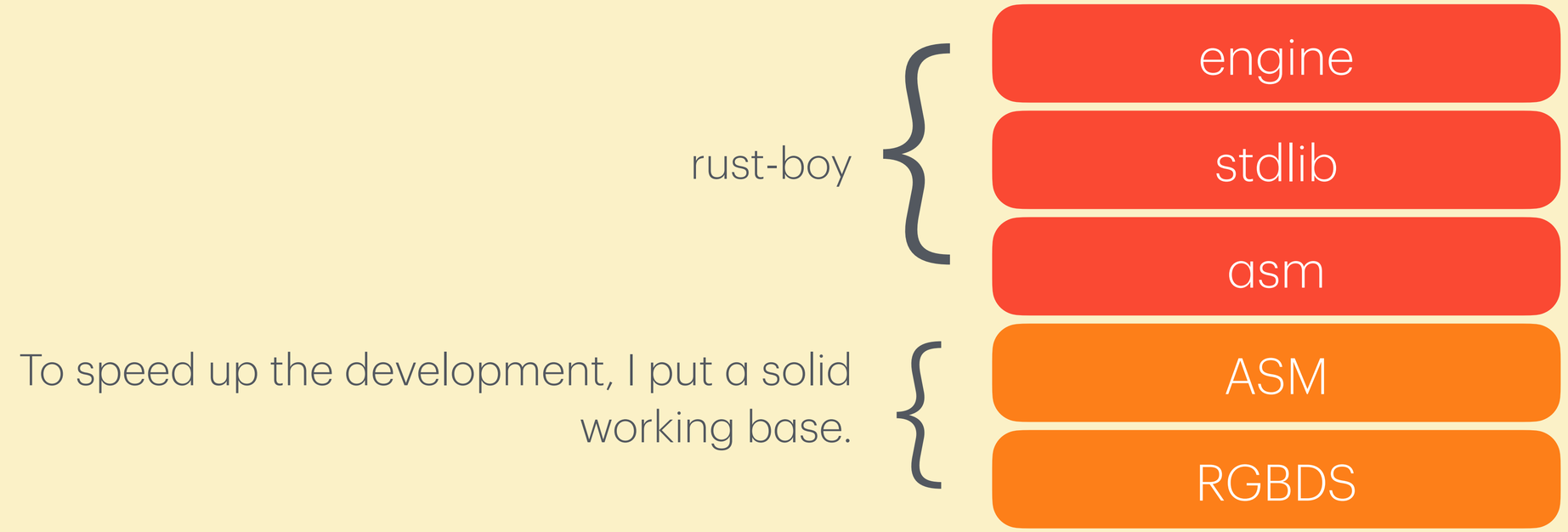
https://github.com/ffex/rust-boy



PoC

Proof of concept

A compiler is a **big goal**. The PoC is a Rust library that **writes the assembly** for you: your Rust runs on your PC and prints main.asm, and RGBDS builds the ROM.



Where to start

Unbricked - an Arkanoid clone

- 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 parts of this game in asm



Unbricked - asm

Inits

```
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
```

Variables & Sprites

```
Paddle:
        dw `13333331
        dw `30000003
        dw `13333331
        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
```

Functions

```
    jp Main
; Copy bytes from one area to another
; @param de: source
; @param hl: destination
; @param bc: lenght
Memcopy:
    ld a, [de]
    ld [hli], a
    inc de
    dec bc
    ld a, b
    or a, c
    jp nz, Memcopy
    ret
UpdateKeys:
    ; poll hald the controller
    ld a, P1F_GET_BTN
    call .onenibble
    ld b, a ; B7-4 = 1; B3-0 = unpressed bu

    ; poll the other half
    ld a, P1F_GET_DPAD
    call .onenibble
    swap a ; A7-4 upressed direction; a3-0
```

Tiles & Tilemap

```
        dw `33300333
        dw `33000333
        dw `33000333
        dw `33333333
    ; 8
        dw `33333333
        dw `33000033
        dw `30333003
        dw `33000033
        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
0,0,0,0,0,0,0,0,0,0,0
```

Unbricked - asm

Main loop

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
```

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
```

rust-boy | asm

- The **lowest**-layer
- Almost **one-to-one** with the asm but in Rust!

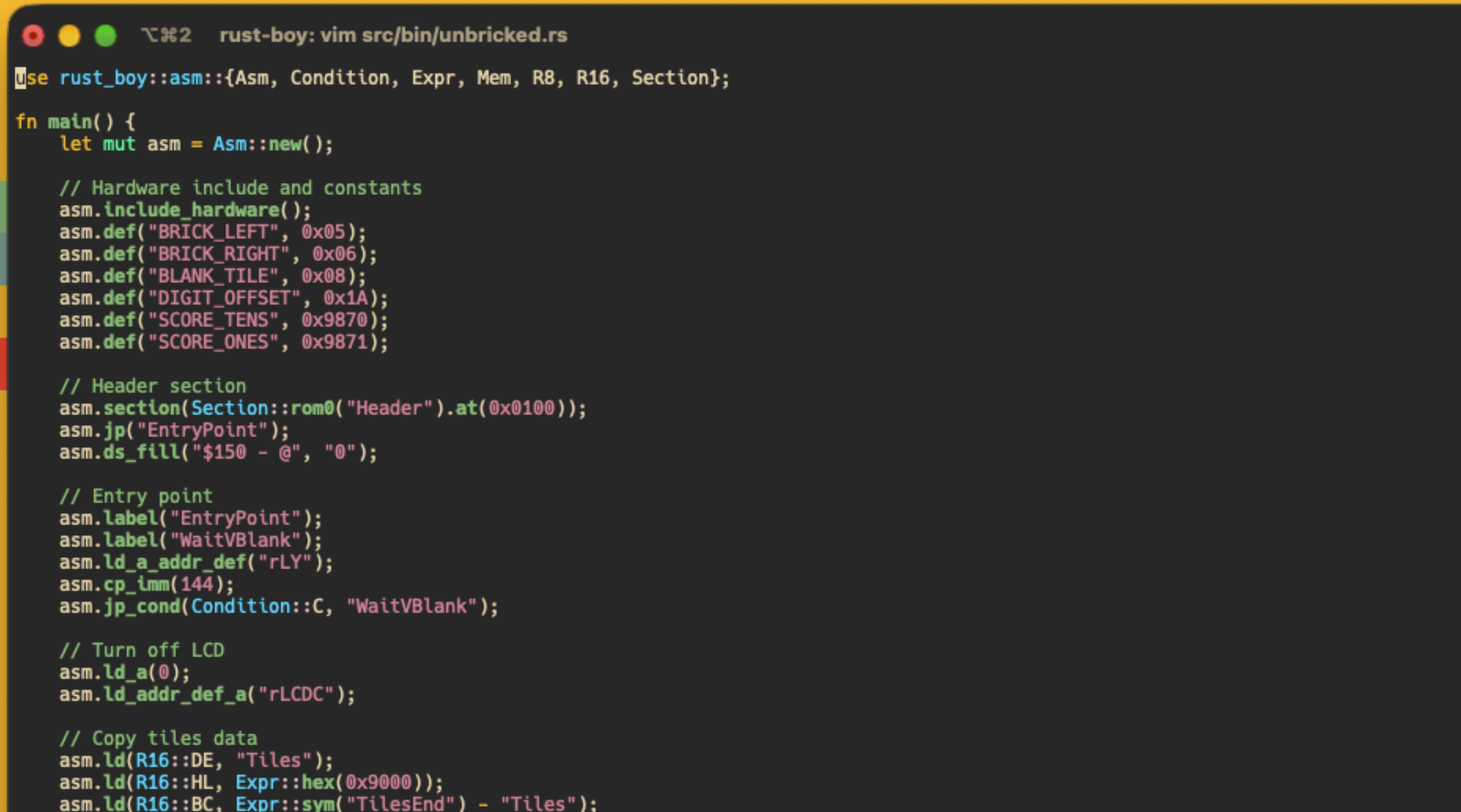
```
asm/
├── program.rs    Asm: the whole program
├── instr.rs      Instr: every SM83 instruction
├── expr.rs       Expr: values and labels
├── builders.rs   asm.ld( .. ), asm.cp( .. ), …
├── block.rs      Block: a piece of code
├── section.rs    typed SECTIONs
├── labels.rs     unique local labels
└── relax.rs      jr → jp when out of range
```

- Covers the whole SM83 **instruction set**, with **typed operands**.
- Link to an article that inspired me:

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

rust-boy | asm

unbricked.rs

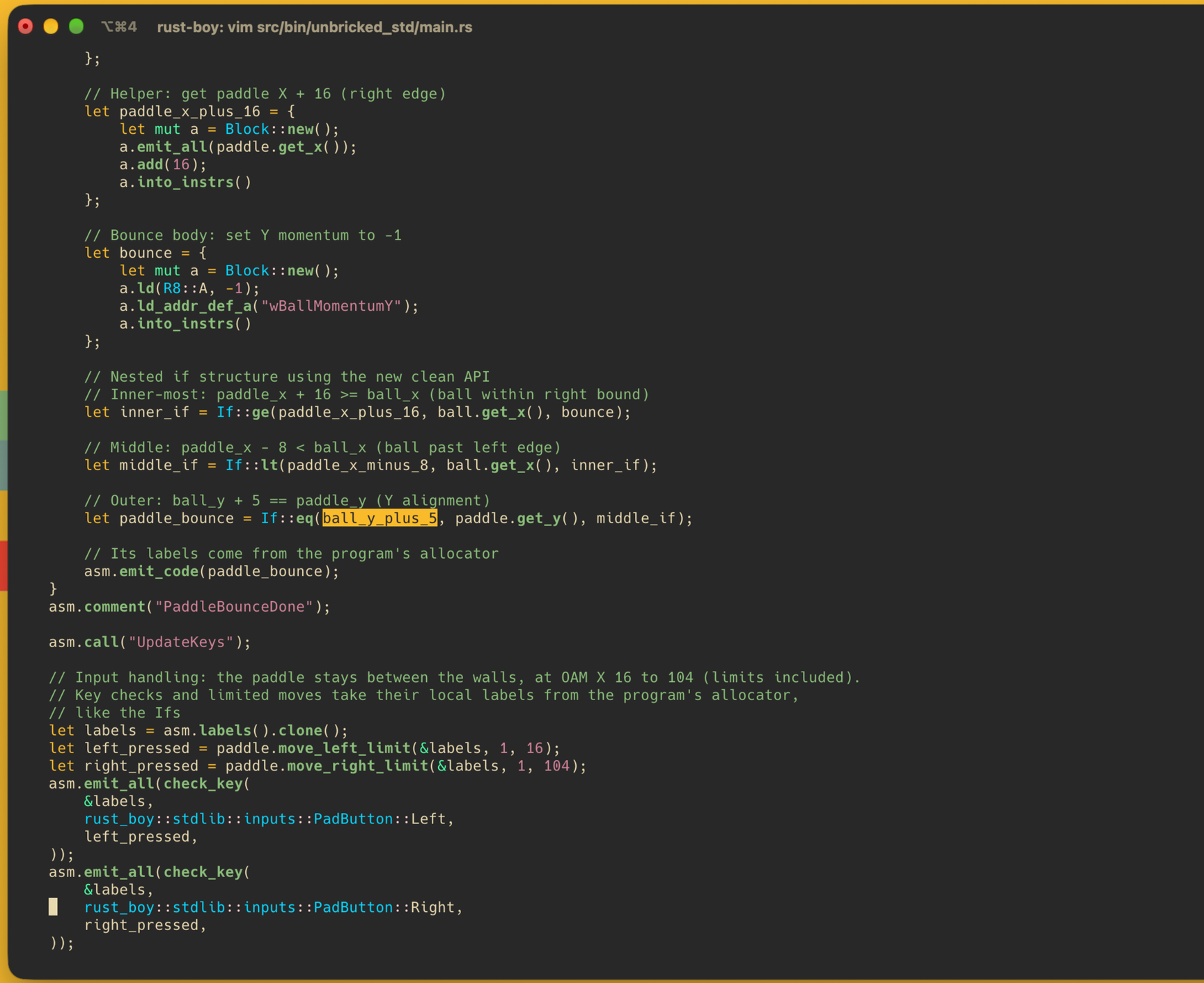


rust-boy | stdlib

- The middle layer: it joins the asm instructions into **reusable pieces**.
- **Stateless**: each helper just returns instructions, and I decide where they go.
- **Routines** written once (Memcopy, WaitVBlank, UpdateKeys...).
- **Tiles** and tilemaps helpers.
- Control flow: **If, While, For, SwitchA**. No more writing labels and jumps by hand.

rust-boy

stdlib



rust-boy | engine

With stdlib I understood what to **level up**:

- Write code anywhere: the engine puts it in the right section

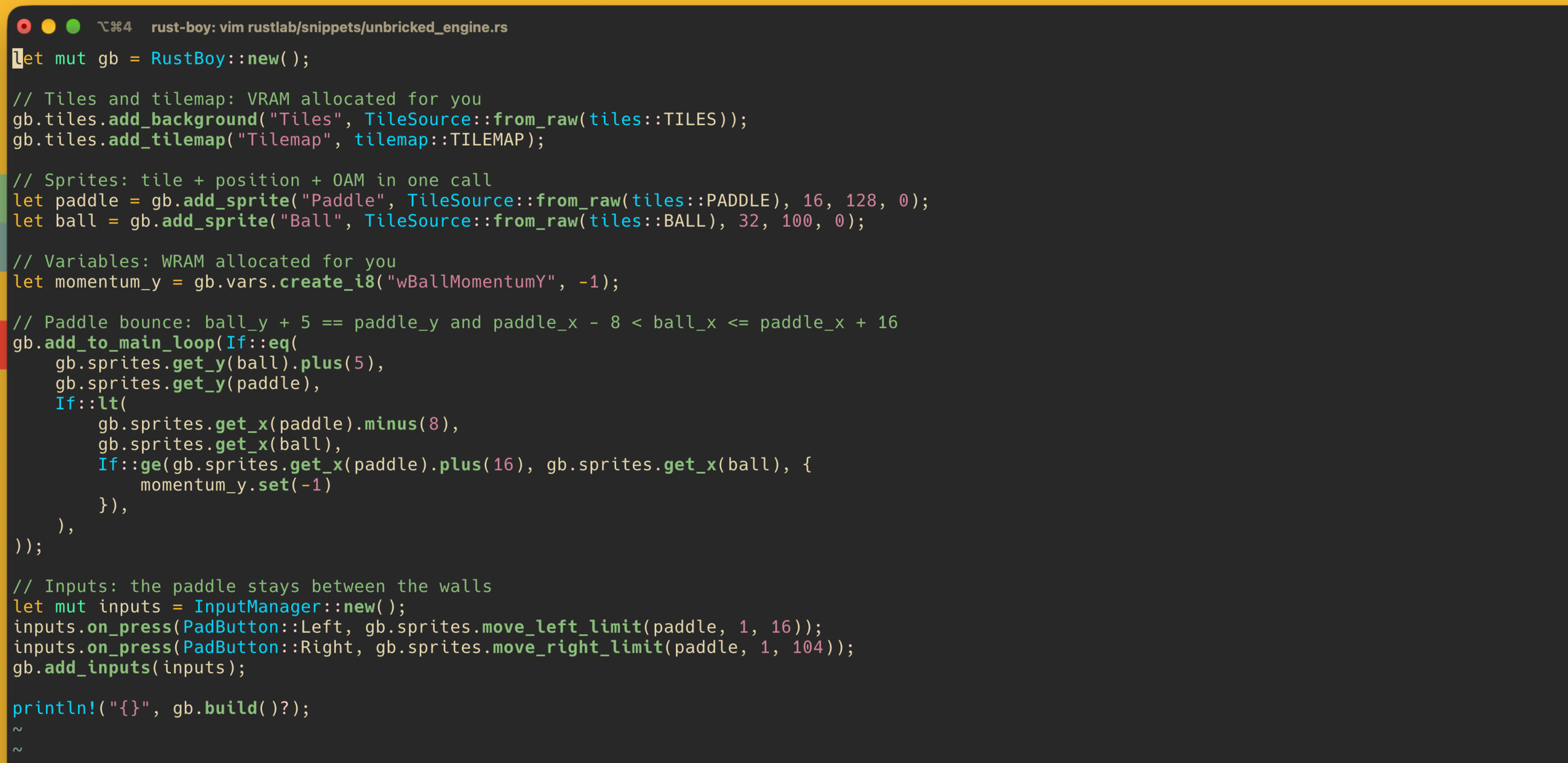
- **Init code generated** for you

- **"Built-in" routines** (UpdateKeys, WaitVBlank, Memcopy...), **included only when used**

- A **manager** for each **game concept** (tiles, sprites, variables, inputs, animations...)

- **No memory addresses**: VRAM, WRAM and OAM are allocated for you

rust-boy | engine



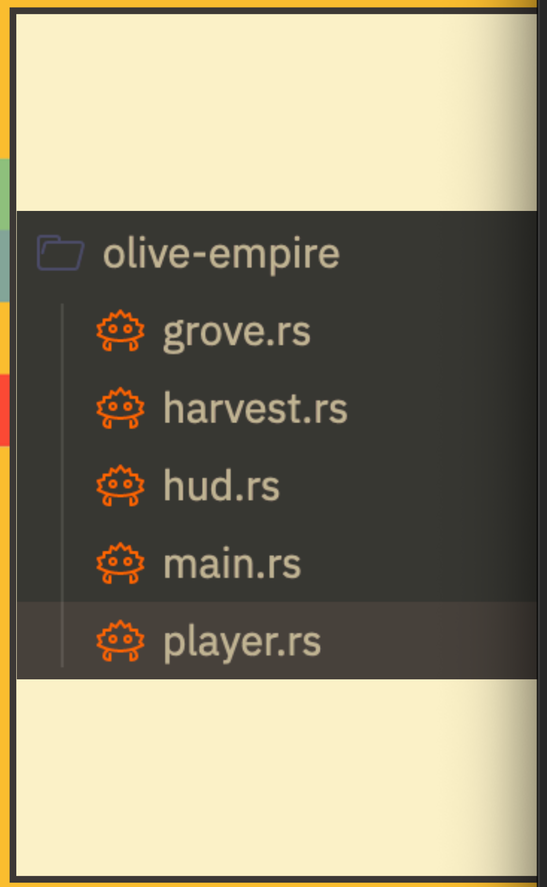
Try rust-boy!

The RustLab example | **Olive Empire Simulator**





Olive Empire



⌥⌘4 rust-boy: vim src/bin/olive-empire/main.rs

```rust
//! Olive Empire: walk a 16x16 farmer through an olive grove and harvest its trees. One module
//! per feature; the engine puts each one's tiles, variables, start-up and main-loop code where
//! they belong in the ROM

mod grove;
mod harvest;
mod hud;
mod player;

use rust_boy::engine::{RustBoy, RustBoyConfig, SpriteSize};

fn main() -> Result<(), Box<dyn std::error::Error>> {
    let mut gb = RustBoy::with_config(
        RustBoyConfig::default()
            .sprite_size(SpriteSize::Size8x16)
            .window(hud::window()),
    );

    let grove = grove::add(&mut gb)?;
    let hud = hud::add(&mut gb)?;
    let player = player::add(&mut gb, &grove)?;
    harvest::add(&mut gb, &player, &grove, &hud);

    println!("{}", gb.build()?);
    Ok(())
}
~
~
~
~
~
~
~
~
~
~
~
~
```

Olive Empire

⌥⌘4 rust-boy: vim src/bin/olive-empire/harvest.rs

```rust
//! Harvest: A next to the trunk of a tree with olives turns it bare and counts it on the HUD

use rust_boy::asm::{Block, R8, R16};
use rust_boy::engine::{InputManager, RustBoy};
use rust_boy::stdlib::inputs::PadButton;

use crate::grove::Grove;
use crate::hud::Hud;
use crate::player::Player;

/// Add the harvest counter and its action on A
pub fn add(gb: &mut RustBoy, player: &Player, grove: &Grove, hud: &Hud) {
    let harvested = gb.vars.create_u16("wOlivesHarvested", 0);
    let bare_tile = grove.bare_tile;
    let mut harvest = Block::new();
    // hl = the touched trunk cell. Raw asm (no API to align a map address yet): every tree
    // starts on an even column, so clearing bit 0 gives its left cell, and the row above
    // (-32 cells) its top-left one
    harvest.res(0, R8::L).ld(R16::BC, -32).add_hl(R16::BC);
    harvest.emit_all(
        gb.queue_tile_block_write(2, &[bare_tile, bare_tile + 1, bare_tile + 2, bare_tile + 3]),
    );
    harvest.emit_all(harvested.increment());
    // Raw asm: the number printer takes its value in de, increment leaves it in hl
    harvest.ld(R8::D, R8::H).ld(R8::E, R8::L);
    harvest.emit_all(hud.olives.set_value(gb.labels()));

    let mut inputs = InputManager::new();
    inputs.on_just_pressed(
        PadButton::A,
        gb.sprites.when_composite_touches(
            player.sprite,
            player.feet,
            &grove.full_trunks,
            harvest.into_instrs(),
        ),
    );
    gb.add_inputs(inputs);
}
```

⌥⌘4 rust-boy: vim src/bin/olive-empire/player.rs

```rust
use crate::grove::Grove;

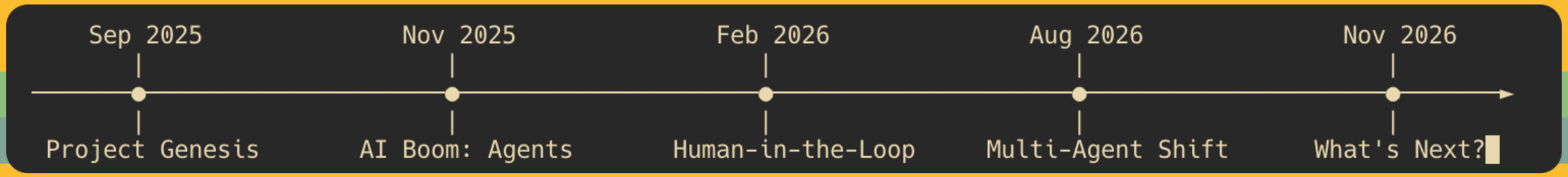
/// What the other features need of the player
pub struct Player {
    pub sprite: CompositeSpriteId,
    /// The part of the character that collides: its feet
    pub feet: Hitbox,
}

/// Add the player at the centre of the screen, standing, with its walk on the D-pad
pub fn add(gb: &mut RustBoy, grove: &Grove) -> Result<Player, AssetError> {
    let sprite = gb.add_sprite_16x16(
        "player",
        TileSource::from_png(
            concat!(
                env!("CARGO_MANIFEST_DIR"),
                "/examples/olive-empire/char-l.edited.png"
            ),
            &PngOptions::default().tile_order(TileOrder::Sprite8x16),
        )?,
        TileSource::from_png(
            concat!(
                env!("CARGO_MANIFEST_DIR"),
                "/examples/olive-empire/char-r.edited.png"
            ),
            &PngOptions::default().tile_order(TileOrder::Sprite8x16),
        )?,
        80,
        72,
        0,
    );
    let feet = Hitbox::new(2, 8, 12, 8);

    let walk = WalkAnimations {
        front: gb.sprites.add_composite_animation(
            sprite,
            "playerWalkFront",
            0,
            3,
            AnimationType::Loop,
        ),
        back: gb.sprites.add_composite_animation(
            sprite,
            "playerWalkBack",
            4,
            7,
            AnimationType::Loop,
        ),
        left: gb.sprites.add_composite_animation(
            sprite,
            "playerWalkLeft",
            8,
            11
```

What's next?

Timeline



Some aspects to consider

- I like TUI
- A new compiler?
- Coding can be automated (and still fun)
- Just for fun
- Why? Why-not?
- I like designing games
- Try an idea is more simple than before
- Can an agent implement a game mechanic?

I milestone

- Cover the whole SM83 **instruction set**
- Implement all **common routines** in standard library level
- Cover all system with **helpers / managers / utilities** (audio system, graphic system, etc...)
- Cover every kind of control **flow** a game needs
- Try to cover all the **common game mechanics** possible
- Eliminate the asm from code



II milestone

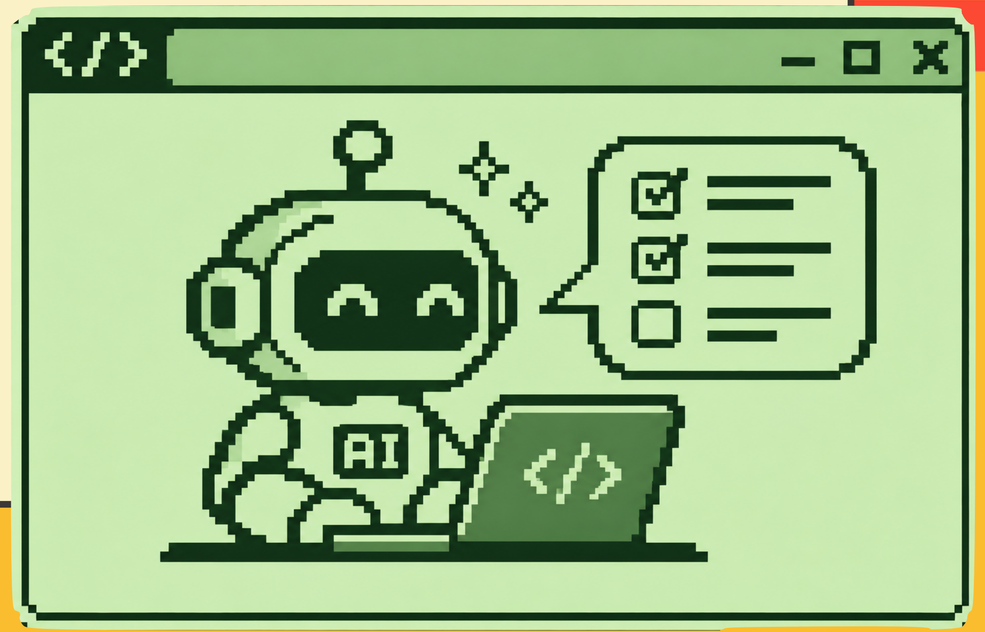
IDE

- Implement an **IDE** to simplify the development (in a **TUI** maybe?)
- Add all sorts of utilities to convert / adjust / manage the **image**
- Utility for **sound**
- An **editor for sprites** with a automatic tile management



III milestone

- Above the **IDE** implement an AI coding agent specialized in GameBoy games
- AGENTS.md, skills, common snippets, etc...



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/