Emmy
The Game Boy Emulator

Final Project Report

Author: Oscar Sjöstedt

Supervisor: Ian Kenny

Student ID: 20040078

April 4, 2023

Abstract

This project aims to create a Game Boy emulator web-application, in other words a program capable of receiving Game Boy game files (commonly referred to as ROMs), and interpreting such ROM to play the game, or execute the program, it contains. The emulator will be usable in browsers, for both desktop computers and mobile devices that may not have access to a physical keyboard. The emulator will also contain debugging capacities, to allow other emulator developers to use it when comparing with their emulator and working on it.

The objective of this project is to create a piece of software that could be used by anyone wanting to emulate retro games, without the need for any technical knowledge on emulators or downloading anything (except the ROMs that need to be obtained separately).

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Oscar Sjöstedt

April 4, 2023

Contents

Chapter 1

Introduction

1.1 Motivation

Emulators are an area of computer science widely used today. Either implemented in hardware or software, they allow replicating the behaviour of one system on another. One of its applications is video game emulation, where a computer simulates a game console (usually a retro console). This allows users to play games that either may not be obtainable in stores anymore, or made for consoles that do not function properly anymore. A wide range of emulators already exist for most consoles. Emulation in general is also widely used in developing new systems, and is an active area of computer science.

This project will seek to create a new emulator for the Game Boy allowing users to play retro games on their computer or mobile device, through the browser. This report will document how the original console works and how the emulator imitates this behaviour to the best accuracy possible, as well as comparing the resulting emulator with other existing ones.

1.2 Scope

The scope of this project is creating a new Game Boy and Game Boy Color emulator, working for browsers. Emulation will be as accurate as achievable with the time available – there may be minor inaccuracies in the end product. Extra peripherals and features of the console may be omitted, to allow more focus to be put on the core part of the console.

The emulator will be usable across a range of devices. Debugging tools and additional features may be provided to the user to let them customise their experience to their needs.

1.3 Objectives

The resulting software will allow users to open a game file for the Game Boy (also called a ROM) and play it. They may use the emulator on a computer, controlling the console via the keyboard, or on a touch device, using on-screen buttons. The emulated features of the emulator include proper rendering of the screen, simulating the audio of the console, the different buttons, and support for a variety of chip controllers for game cartridges.

The frontend of the emulator may also contain additional quality of life features, such as custom themes, save states, and debugging options allowing to inspect the state of the Gameboy – a feature vital to emulator developers and retro game developers.

Chapter 2

Background

2.1 Emulation

An emulator is "hardware or software that permits programs written for one computer to be run on another computer" [1]. The imitated computer is the *guest*, and the one that imitates is the *host*. Emulators are nowadays mainly found in the form of software, and have many different uses, from preservation to hardware development.

Emulation was born with the first computers: the very first computer, the Colossus made in 1941, was built to imitate the Enigma machine [2]. However emulation was properly studied in the 1980s, when computing power started to steadily increase. One of the earliest instances of emulation as an actual feature is with the IBM System/360. This computer supported emulation of previous models, such as the IBM 709, 7090, 7094 and 7094 II [3].

Emulation is also vital for preservation: as transistors and motherboards age, old systems become unusable, and with them the software they ran. Companies may also stop producing the hardware to run this software on. Emulating these systems is often the only future-proof and sustainable way to keep this software usable [4].

Finally, another common use for emulation is virtual machines. These programs allow running another Operating System (OS) on a computer, which can be used for instance when developing for other systems, without needing to use the physical device directly, for instance when developing a Windows-compatible app with a Linux computer. In the case where the architectures of the guest and the host are the same, we call this *virtualisation* [5]. Virtualisation is nowadays extremely common, with tools like Docker[1], that allow multiple virtualised instances

[1] See https://www.docker.com/

of computer systems running on the same host. This typically allows for great portability, as the same infrastructure can be copied and ran anywhere.

2.2 Video Game Emulation

Video game emulation is the art of emulation applied to video game hardware systems. This allows the host to run games destined for the original console. This usually requires precise understanding of the console's hardware and functioning, as games may rely on specific behaviours and edge cases to function. This task is rendered harder by the fact that the only legally available source of information on these consoles comes from research and reverse-engineering done by hobbyists, and does not come from an official source, as it is proprietary hardware.

Video game emulation started in the 90s when computers were powerful enough to properly simulate console systems. Although precise dates are hard to get, the first console emulators seem to be either from 1990 or 1993 [6], and were able to run some NES games. The first Gameboy emulators were in the late 90s, with the Virtual GameBoy[2] in 1995 and NO$GMB[3] in 1997 (although its history page[4] seems to indicate development started in 1993) [7].

The original game files and assembled code for video games are copyrighted material, and are referred to as the Read-Only Memory (ROM) of the game. Although distributing these ROMs is usually illegal, there also exist copyright free ROMs: games created by developers that chose to license them under Creative Commons licenses, for instance. Websites such as Retro Veteran[5] host wide collections of legal ROMs.

Official emulators also exist, and are developed by the console manufacturers. These usually only allow playing from a selection of games, limiting options. The reason for this is that they are built to emulate these specific games, rather than the console as a whole, meaning games outside of the catalogue will often not work. They are thus usually less accurate than unofficial emulators – this is the case of the Virtual Console, the official emulator of the Nintendo 3DS, that fails many Game Boy test ROMs [8, Test ROMs].

2.3 Game Boy, Game Boy Color

The Game Boy (GB) is an 8-bit handheld video game console, released in 1989. It has a small 160×144 pixel screen, and has a Sharp LR35902 as its Central Processing Unit (CPU), clocked

[2] See http://fms.komkon.org/VGB/
[3] See https://problemkaputt.de/gmb.htm
[4] See https://problemkaputt.de/gmbhist.htm
[5] See https://www.retroveteran.com/category/nintendo-game-boy-color/

at 4.19MHz [9, Specifications]. In 1998, the Game Boy Color (GBC) was then released. Seen as the successor of the GB, it contains a screen of the same resolution, but supporting colour, from a palette of 32768 options (15 bits per colour). It contains the same CPU as its predecessor, a Sharp LR35902, with now two modes: a 4.19MHz mode and a 8.38MHz mode (double-speed mode). This allows the GBC to be backwards compatible with most GB games – there are a few exceptions to this, games that used hardware bugs of the original GB that were fixed in the GBC [10, STAT IRQ glitches].

From an emulation perspective, the Game Boy Color can thus be seen as an extension of the Game Boy – it has an identical CPU (although with a toggle-able double speed mode), and most of the memory layout is identical. To keep the remaining of this document simple, if not stated, "GB" will refer to both the original Game Boy and the Game Boy Color, as they are very similar. Dot Matrix Game (DMG) refers exclusively to the original Game Boy model.

2.4 Existing Literature

2.4.1 Gameboy Documentation

The Game Boy is one of the best documented consoles for emulation, and a large array of resources exist documenting it. Some useful resources explaining it's behaviour are:

- Pandocs[6] is a technical reference of how the GB works. It is extremely complete and covers a wide range of topics, so it is useful to get a global view of a problem. It is one of the most referenced pieces of literature on the console.
- GB CPU Instructions[7] is a table containing all instructions its CPU has, as well as information on the amount of cycles taken by the instruction, the bytes of memory used, the flags affected by the operation, and a description of the instruction.
- Gameboy Complete Technical Reference[8] (GBCTR) is an unfinished document that contains very detailed information on the CPU and other components of the GB. Although incomplete, it provides a much lower-level view of the details of the GB (compared to Pandocs), making it useful to emulate very specific behaviour like the cycle-by-cycle timing of the CPU.
- GB dev wiki[9] is a wiki containing additional information on the GB, including guides to making games and explanations on some hardware quirks, and in particular a very precise

[6]See https://gbdev.io/pandocs/
[7]See https://meganesu.github.io/generate-gb-opcodes/
[8]See https://gekkio.fi/files/gb-docs/gbctr.pdf
[9]See https://gbdev.gg8.se/wiki/

description of the Audio Processing Unit (APU).

2.4.2 Existing Emulators

A wide range of emulators for the GB and GBC already exist, and many of them are open-source. These are useful when developing a new emulator, to see how they work internally. For performance reasons they're usually written in compiled languages, such as C++ and Rust, but some interpreted language alternatives exist. These emulators include:

- Game Boy Crust[10] is a simple GB emulator written in Rust. It is quite incomplete but has a comprehensive structure, so it's a good project to first figure out how emulators work.
- AccurateBoy[11] is a highly accurate emulator, in particular for its Picture Processing Unit (PPU) that has pixel-perfect accuracy.
- oxideboy[12] is another GB emulator written in Rust, that is much more complete and helpful for some edge cases.
- SameBoy[13] is one of the most accurate open source GB and GBC emulators, written in C. It is much more technically complex but still useful to understand edge cases, especially since it is the emulator used as a reference when developing this project.
- Mooneye GB[14] is a GB research emulator written in Rust. It passes most of the Mooneye test ROMs, making it helpful when encountering issues with these tests.
- GameBoy-Online[15] is a high-accuracy JavaScript emulator, that is particularly useful to understand how to interface the emulator with the browser (notably for the APU).
- Gameboy.js[16] is another JavaScript emulator. It is fairly simple and inaccurate, but is easily hackable, making it useful when starting a new emulator to compare execution traces.
- rboy[17] is an emulator written in Rust that was used when developing the APU, as it passes some complex test ROMs with quite simple code.

The 8th of February 2023, Nintendo announced the release of a Game Boy and Game Boy Color emulator on the Nintendo Switch, via the Nintendo Switch Online subscription [11]. The recent release of official emulators as well as public enthusiasm for the latter prove the relevance

[10] See https://github.com/mattbruv/Gameboy-Crust
[11] See https://github.com/Atem2069/accurateboy
[12] See https://github.com/samcday/oxideboy
[13] See https://github.com/LIJI32/SameBoy
[14] See https://github.com/Gekkio/mooneye-gb
[15] See https://github.com/taisel/GameBoy-Online/
[16] See https://github.com/juchi/gameboy.js/
[17] See https://github.com/mvdnes/rboy

of this kind of emulator.

2.4.3 Gameboy Test ROMs

A core set of resources to develop an emulator are test ROMs. These are ROMs which instead of playing a game will run a series of tests on the console. These tests are first written to pass on the physical console itself, and are then used to ensure they also pass on the emulator. This means issues in specific components can be easily diagnosed (so long as the rest of the emulator responsible for running the test ROM works itself). These test ROMs also have the advantage of being open source, meaning their source code can be referred to in order to understand what they expect of the console.

An other advantage of using test ROMs is that they tend to re-use the same framework across a given test suite to report results. This means testing can easily be automated over multiple tests by inspecting specific registers/memory addresses, rather than having to store an "expected result" image for each test.

The test ROMs used for this project are:

- Blaarg test ROMs[18] are some of the most well-known and used GB test ROMs. They include tests for the CPU, the timings of instructions, hardware bugs and the APU.
- Mooneye test ROMs[19] is a very complete test suite, that verifies most components of the GB: CPU instructions, memory timings of specific instructions, behaviour of Memory Bank Controllers (MBCs), timings of the APU, Direct Memory Access (DMA), PPU and timer.
- Acid Test (DMG[20], GBC[21]) is a test that verifies that the PPU of the GB displays data properly (to line-rendering accuracy), for both Game Boy and Game Boy Color displays.
- SameSuite[22] is a test suite that is valuable for its APU tests: it uses the `PCM12` and `PCM34` registers exclusive to the GBC to inspect the exact output of the APU (whereas other test ROMs tend to inspect the on/off status of the channels, which is much less accurate).

[18] See `https://github.com/retrio/gb-test-roms/`
[19] See `https://github.com/Gekkio/mooneye-test-suite`
[20] See `https://github.com/mattcurrie/dmg-acid2`
[21] See `https://github.com/mattcurrie/cgb-acid2`
[22] See `https://github.com/LIJI32/SameSuite/`

Chapter 3

Requirements and Specification

3.1 Requirements

3.1.1 User Requirements

U1. Run Game Boy games to a satisfiable fidelity, with proper rendering and controls emulation.

U2. Run Game Boy Color games to a satisfiable fidelity, with proper rendering and controls emulation.

U3. Allow the user to run GB and GBC games on both Game Boy and a Game Boy Color.

U4. Allow the user to save the state of the game, to continue their playthrough later. The state can simply be saved as a downloaded file, and re-uploaded later to continue the game.

U5. Allow the user to change the speed at which the game is played: double speed mode, half speed mode, etc.

U6. Have some debug functionality, to inspect the state of the console at any given time.

U7. Allow users to pause the console, and add breakpoints to stop execution at specific moments.

U8. Allow the user to switch between rendering modes (nearest-neighbour, LCD display, Scale2x, etc.)

U9. Allow the user to switch the colour palette of the DMG emulation.

3.1.2 System Requirements

F1. The system can receive a ROM file, construct an instance of the emulated console, and run the code inside said ROM.

F2. The system emulates different components of the GB and GBC, with as much precision as possible (M-cycle precision).

F3. The system renders the output of the emulator to a Web `<canvas />`.

F4. The system creates the required DOM elements for the web-app, and updates them as needed.

F5. The system listens to key presses and releases to emulate controls through the keyboard.

F6. For touch devices, the system may render buttons to simulate the console's controls.

3.1.3 Non-Functional Requirements

N1. The emulator should be accessible on computers through a web browser equipped with a recent version of JavaScript.

N2. The emulator should be accessible on mobile devices through a web browser equipped with a recent version of JavaScript.

N3. The emulator should be accessible on computers through a standalone app.

N4. Maximise the tests passed by the emulator (see Gameboy Test ROMs).

N5. Have the code be well documented, allowing new-comers to the project and to GB emulation to easily understand what is going on – if possible with links to relevant Game Boy emulation resources.

3.2 Specification

Code	Specification	Importance
U1	User can upload a GB ROM file (`.gb`), and the emulator will run the game. The keyboard can be used to control the game, and the output is displayed.	High

Code	Specification	Importance
U2	User can upload a GBC ROM file (`.gbc`), and the emulator will run the game. The keyboard can be used to control the game, and the output is displayed.	Medium
U3	User can upload a GB ROM file (`.gb`). The user can switch between a DMG and a GBC emulator.	Medium
U4	User can press a button to download a save of their game (or, alternatively, the save can be stored inside the browser with a technology like IndexedDB[1]).	Low
U5	User can select the speed of emulation, to dynamically accelerate/decelerate the game.	Medium
U6	User can see debug information of the emulator. This information includes the current tileset, background map, time to draw a frame, and register information.	Low
U7	User can pause the console emulation through a button. They can also input conditions for which the console should break execution.	Low
U8	User can dynamically switch the rendering filter via a dropdown button.	Low
U9	User can dynamically switch the colour palette of the GB via a dropdown button.	Low
F1	A ROM file can be uploaded, is transformed into an `UInt8Array` (because the GB is an 8-bit system), and the appropriate object is created to run the code.	High
F2	Different components exists as different classes, respecting typical OOP principles such as encapsulation and inheritance when relevant.	High
F3	A `<canvas />` element is created, and is updated with the output of the emulator after every frame is drawn (ie. at the start of each VBlank mode).	High
F4	The Preact[2] framework is used to handle the UI of the web-app.	High

[1] See https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API
[2] See https://preactjs.com/

Code	Specification	Importance
F5	Listeners are added to the environment's `window` to listen to all key presses and releases. The emulator can then request for a control update, by reading the state of keys.	High
F6	If a touch device is detected, button are added to the UI and are used by the emulator as inputs.	Medium
N1	A deployed version of the web-app is accessible on a desktop browser and provides full functionality, via keyboard and mouse inputs.	High
N2	A deployed version of the web-app is accessible on a mobile device browser and provides full functionality, via touch controls.	Medium
N3	A downloadable version of the web-app can be used on a computer and provides full functionality, via keyboard and mouse inputs.	Low
N4	As many possible tests as possible should be passed, while ensuring previously passing tests do not fail.	Medium
N5	Main methods and variables must be properly documented, and have links to appropriate online resources to documentation about said element.	High

Chapter 4

Design

In this chapter we will outline the main components of the Gameboy and of this emulator. For the different GB components, we will briefly go over their role, and how they interact with other components and to what end. The emulator's name will be "Emmy", short for emulator[1].

Because the emulator should not rely on the environment it is running in to work, the functionality of the project can be split into two parts: the emulator core, that is responsible for simulating the Game Boy, and the emulator's frontend, that allows interfacing with the user, and may be changed to work for different platforms (see figure 4.1).

In emulators, the different components are usually split into three groups:

- The CPU, responsible for reading instructions and changing the state of the console. This is what drives the emulation, as other components usually idle unless acted upon.
- Input and output components, such as the APU, the PPU and the joypad. These components interact with the outside user, by either outputting the game state, or reading inputs from the user.
- A memory system, that handles addressing within the console. This part is essential to ensure components are communicating between each other properly, since different addresses may map to different components.

4.1 Emulator Frontend

To allow the user to access the emulator, a frontend needs to be built with it. This frontend is responsible for outputting the graphics and audio data of the emulator, and also for receiving

[1] The emulator was named before the realisation that an emulator-related project called "Emmy" already exists [12]. These two projects are unrelated.

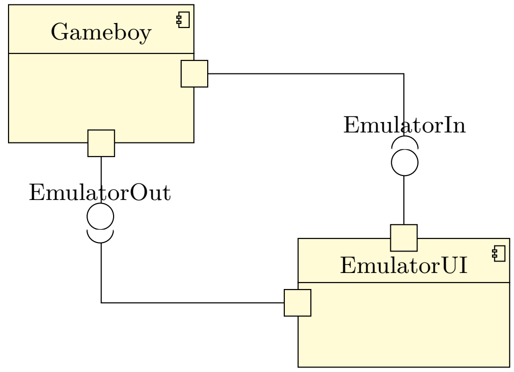

Figure 4.1: Split of emulator core and frontend

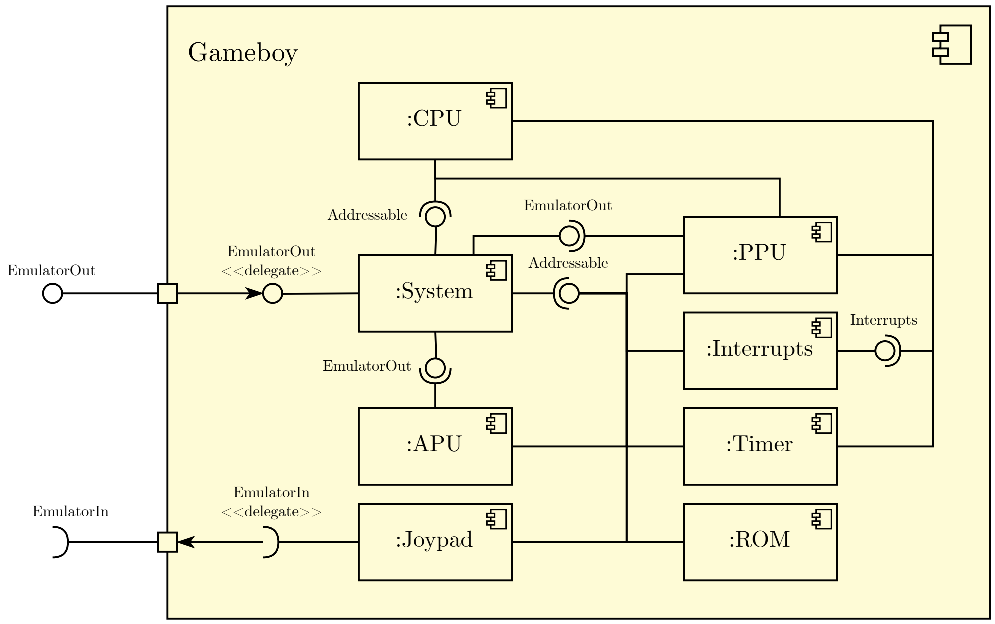

Figure 4.2: Simplified design of the emulator's core
Note utility classes have been excluded for readability purposes

input from the user to pass down to the emulator. Aside from that, extra UI components have been added, to make the usage of the emulator more convenient and add more features: a play and pause button, custom themes, debugging tools, upscaling filters and keybinding options. All of these are kept independent from the emulator, and control transformations of the input and output, or changes in the running of the emulator.

Because this implementation is merely to allow access to the emulator and does not have any outstanding functionality, it's description will be brief – most of the focus will be put on the emulator's core.

4.2 CPU

The Central Processing Unit (CPU) is perhaps the most complex part of the GB. It is however also one of best documented parts of it, making it quite easy to emulate properly. The CPU model is a Sharp LR35902, which is a hybrid of the Intel 8080 chip and the Zilog Z80 [13].

Most emulator are typically "CPU-driven". This means that the CPU is what drives the emulation. If an instruction takes multiple hardware cycles to execute, the CPU is responsible for asking the rest of the system to tick at the appropriate time. In the design chosen for this project however, the emulator will be "System-driven", with the CPU only being part of the overall running of the emulator – it will not be responsible for ticking the rest of the system. This difference will be made obvious by the interfaces exposed to the CPU; it allows for nicer encapsulation and better separation of concerns.

The CPU is thus responsible for stepping through the program it's given. It has 6 different 16-bit registers, with different roles. Some registers' bytes can be accessed independently, operating as two 8-bit registers or one 16-bit register as needed [9, CPU Registers and Flags].

- `AF`: the accumulator-flag register. Most arithmetic operations can be done on `A`. `F` holds the CPU's flags, and can only be altered indirectly, as a consequence of arithmetic operations.
- `PC`: the program counter register. It cannot be accessed directly, and is used to store the current position of the CPU in the program. It can be altered via `CALL`, `RET`, `JP` operations, for instance.
- `SP`: the stack pointer register. It can be modified or incremented, and stores the pointer to the top of the "stack". A typical use of the stack is for handling functions, by pushing the current address to the stack whenever a procedure is called, and popping it back to the stack pointer when it returns.
- `BC`, `DE`: two simple registers that can be used for arithmetic operations.
- `HL`: similar to `BC` and `DE`, it can also be used as an address by some operations, allowing the use of indirect addressing.

Aside from it's registers, the CPU needs to interact with system interrupts, and needs to access memory for reading/writing operations.

4.3 PPU, APU, Joypad

The Picture Processing Unit (PPU) is a complex part of the Gameboy, that handles outputting the game state to a 160×144 screen, that may either be monochrome on the DMG or with colour support on the GBC. It may raise interrupts, and needs to be able to access memory, due to it being responsible for the Object Attribute Memory (OAM) Direct Memory Access (DMA) – a feature allowing transfer of data from an arbitrary location in memory [9, OAM DMA Transfer].

The behaviour of the PPU changes significantly between the DMG and the GBC: it can, for the latter, support colours, and it also supports Video RAM (VRAM) bank switching, a feature the DMG lacks [9, CGB Registers]. It also has an Object Attribute Memory (OAM) DMA mechanism that is quite complex. As such, to make maintenance easier, the PPU is split into different smaller classes. This is not visible to the rest of the system, as only the PPU is exposed to it. See figure 4.3 for a simplified class diagram of its structure.

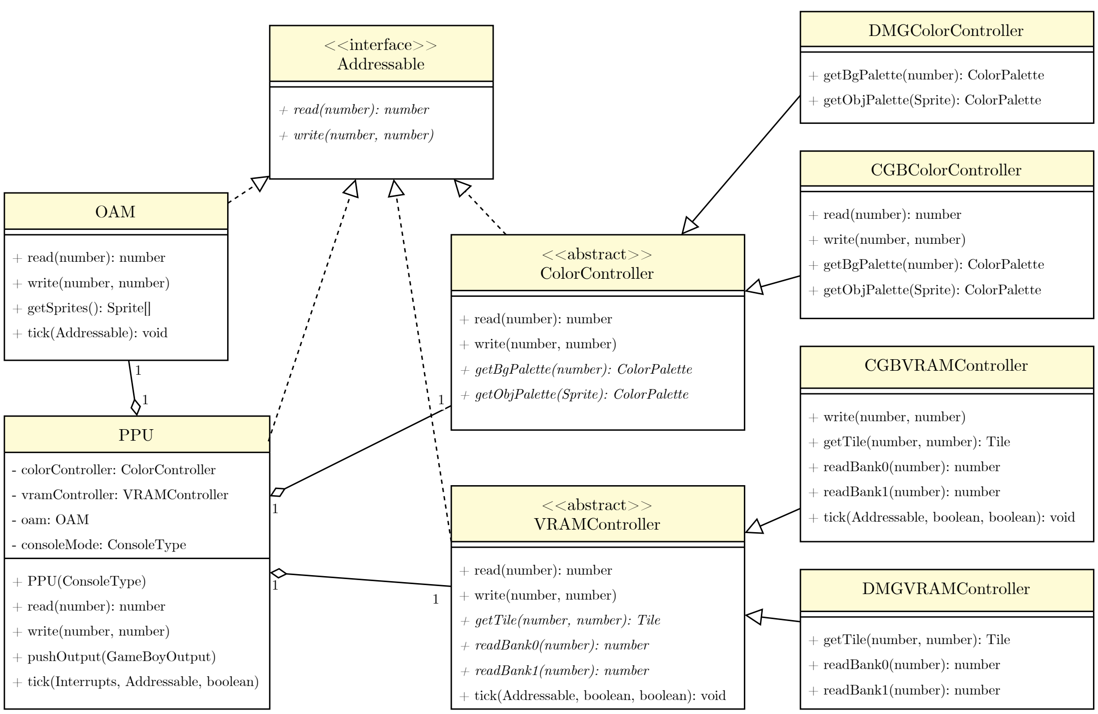

Figure 4.3: Simplified class diagram of the PPU
Note most private fields and methods have been excluded for readability purposes

The Audio Processing Unit (APU) is responsible for playing the audio output of the console. It has a few registers to control it, as well as a short memory area called the wave RAM. It's timings are controlled by the timer, which can be accessed via a memory read of the system.

It's sound output is derived from four separate channels: two square wave channels, a custom wave channel, and a noise channel [9, Audio]. These channels are here modelled as separate entities, but are purely internal (see figure 4.4).

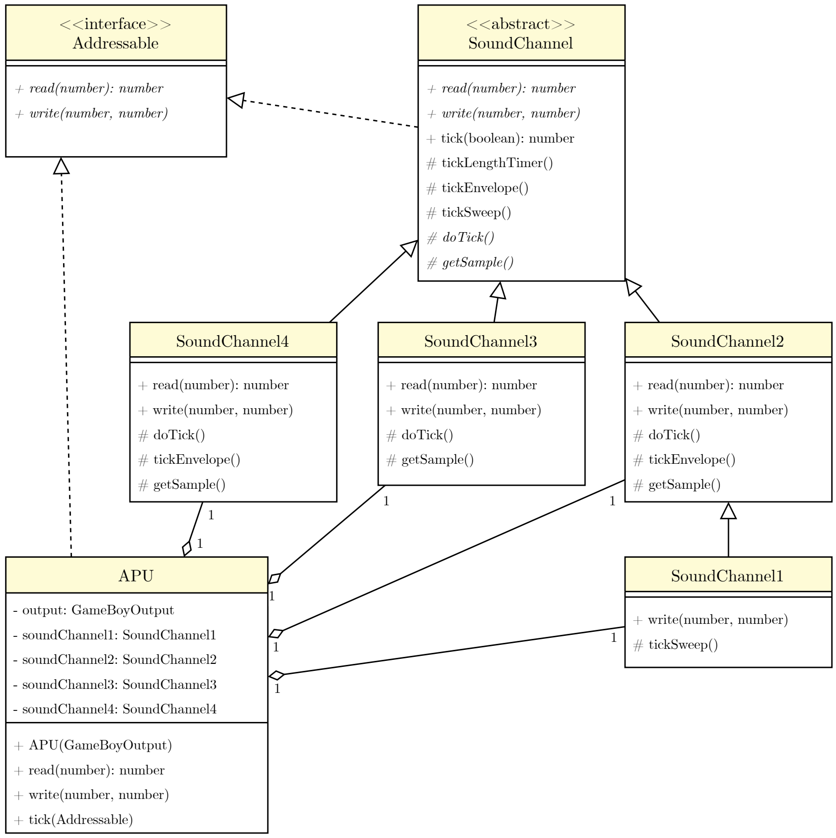

Figure 4.4: Simplified class diagram of the APU
Note most private fields and methods have been excluded for readability purposes

The former two components need to output information to the user, either graphic or audio data. To do this in a portable way, the emulator will expose an interface, that can be implemented by the front-end. This allows the emulator core to be platform-agnostic, and be easily re-usable.

The joypad, on the other hand, is the one input component the GB has. It contains 4 directional arrow buttons, and 4 input buttons: A, B, start and select (see 4.5). To use these inputs, the CPU needs to use two different registers.

Similarly as for the output, the emulator core will expose an interface to read the user's input, ie. the state of all 8 buttons.

Figure 4.5: Picture of a Gameboy, with the screen and joypad visible

4.4 System Bus

Components within the Gameboy need to communicate. To do this, components typically have a set of registers that control how they behave – for instance, the timer has a `TAC` register at `0xFF07` to control it's frequency. To allow this interaction without having all components directly depend on each other, a "system bus" component is created, responsible for handling all components and managing memory-addressing among them.

This component, called `System`, is thus responsible for instantiating the other components, and providing them access to each other, via a read and write method.

4.5 Other Components

Aside from the aforementioned components, the GB has a few components that allow it to run but are not part of the CPU or the input/output components. These will be briefly outlined here, as they have a lesser impact on the emulator's design.

4.5.1 Timer

The timer, responsible for incrementing a counter (the `DIV`) at a set frequency. It can raise interrupts, and owns a few registers.

4.5.2 Interrupts

The interrupt system, that allows components such as the timer or the PPU to interrupt the CPU to run another process. It can also be enabled and disabled by the CPU, with the `EI` and `DI` instructions [9, Interrupts].

This sub-system is designed as a separate component to reduce coupling between other components, to instead have a small object that can be passed to components as needed.

4.5.3 MBC

The Memory Bank Controller (MBC) is an extra chip contained within some game cartridges, to allow access to more ROM data (as well as external RAM in some cases) via banking [9, MBCs]. There are multiple different MBCs, and so it is convenient to define a common interface for all of them, which can then be implemented according to specification for each MBC type.

The type of the MBC is in the header of the ROM [9, The Cartridge Header]. To separate this logic from the base system, a `ROM` class is used. It is responsible for reading the cartridge's header, and creating the appropriate MBC instance.

4.6 Useful Classes

To have similar interfaces over all components, we will also declare specific classes and interfaces to be implemented by them.

4.6.1 Addressable

The `Addressable` interface (see 4.6) provides a read and write method. This allows all components to communicate between each other without needing to be aware of what the component they're communicating with is. Aside from the CPU, all components of the emulator implement this method.

```
interface Addressable {
    read(pos: number): number;
    write(pos: number, data: number): void;
}
```

Figure 4.6: `Addressable` interface

4.6.2 Memory and registers

Simple utility classes can be declared to manage memory in a simple way.

The `RAM` class implements `Addressable`. It can be instantiated with a set size, and can be read and written to. It is used in components that have large blocks of writable data.

The `Register` class implements `Addressable`, but ignores the address parameter of the read and write operations, as it contains only one byte. A `DoubleRegister` class may also be implemented, backed by two `Register`s, to provide support for 16-bit registers, used for instance in the CPU.

Chapter 5

Implementation

The project uses Preact[1], a light-weight alternative to the more popular React[2] framework. Preact was chosen because the front-end of the web-app is extremely lightweight, so a smaller framework with less features is enough. This also avoids bloating the app with a heavy framework such as React: its GZipped and minified size is around 31.8Kb, while Preact is only 4Kb (87% less) [14].

The language used for the project is TypeScript[3], a typed version of JavaScript. This is essential for the project, as ensuring the correctness of code can be extremely hard without proper typing constraints, in particular when the codebase becomes larger and more complex.

The project is divided into two parts:

- The `frontend/` directory contains the UI for the web-app. The main logic to create the emulator and run it is contained in `app.tsx`.
- The `emulator/` directory contains the actual GB emulator. Although most classes and interfaces used are exported, only three elements are needed to properly interact with the emulator:
  - `GameBoyColor.ts` handles the core loop of the system. It contains the `GameBoyColor` class, the emulator. Instantiating the emulator creates all the necessary sub-components, and calling the `drawFrame()` method runs the emulator for one frame (0.16 seconds).
  - `GameBoyOutput.ts` contains a simple interface, with optional methods to receive any output produced by the emulator (see figure 5.1). The two main methods of this are `receiveGraphics` and `receiveSound`, which use the output of the actual console.

[1] See https://preactjs.com/
[2] See https://reactjs.org/
[3] See https://www.typescriptlang.org/

– `GameBoyInput.ts` contains a simple interface with a required `read()` method that returns an object with the current inputs for the console (see figure 5.2).

```
interface GameBoyOutput {
    receiveGraphics?(data: Uint32Array): void;
    receiveSound?(data: Float32Array): void;

    // Debugging methods:
    debugBackground?(data: Uint32Array): void;
    debugTileset?(data: Uint32Array): void;
    serialOut?(data: number): void;
}
```

Figure 5.1: `GameBoyOutput` interface methods

```
type GameBoyInputRead = {
    up: boolean;
    down: boolean;
    left: boolean;
    right: boolean;

    a: boolean;
    b: boolean;
    start: boolean;
    select: boolean;
};

interface GameBoyInput {
    read(): GameBoyInputRead;
}
```

Figure 5.2: `GameBoyInput` interface method

5.1 Emulator Frontend

The frontend of the emulator is written in Preact, allowing for the creation of a simple, fast and lightweight UI to control it. It contains the emulator title, the control buttons and the emulator video output (see figure 5.3). Along the left of the screen is a sidebar with more options, allowing the user to customise the emulator to their needs and debug the state of the console if needed.

5.1.1 Main Controls

The main controls for the emulator are the 6 buttons above the screen. These are, from left to right:

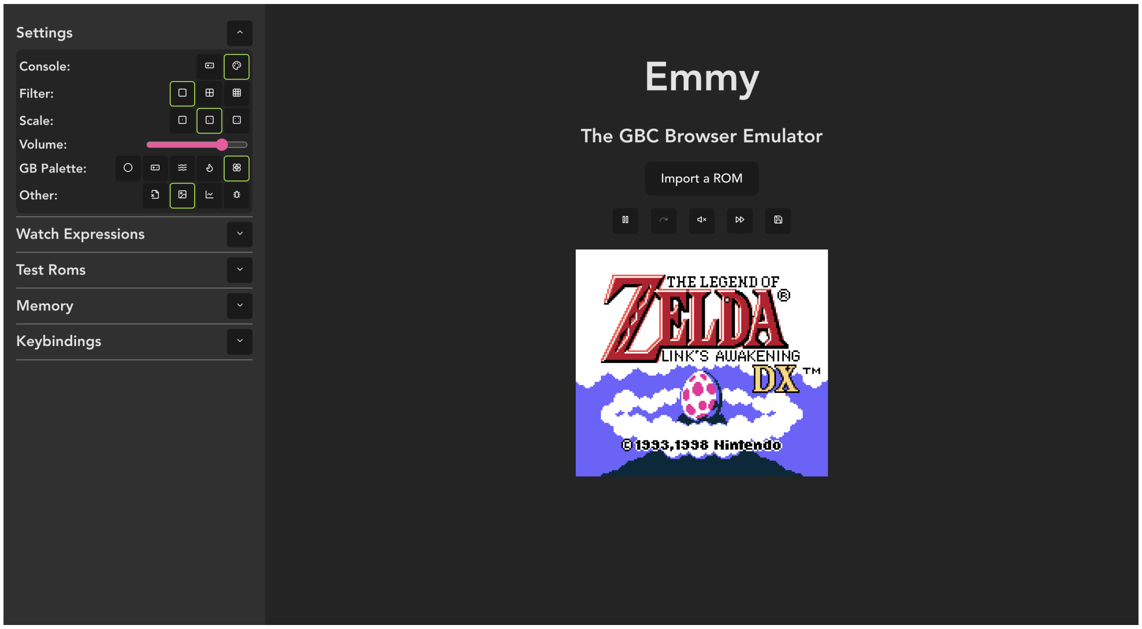

Figure 5.3: Home page of the emulator

- Pause/play: pauses or resumes the emulator.
- Step by one CPU instruction: this is a feature useful for debugging the emulator, when precise information of the state of the emulator is needed.
- Sound toggle: allows enabling or disabling the sound of the emulator. By default the sound is turned off, because modern browsers do noat permit websites playing any sound before the user interacts with the website [15].
- Triple speed toggle: a toggle button that speeds up the emulator to emulate the GB thrice as fast as usual. This is a common feature found in most emulators.
- Save state: saves the current state of the cartridge in the browser's storage, allowing the user to resume playing the game later. Note this does not save the full state of the emulator, but that of the cartridge, making it equivalent to a real-life save on the GB where only the battery-backed storage of the cartridge persists through power-offs.

5.1.2 Settings

In the side drawer, the first tab is "Settings". It contains general settings for the emulator, such as:

- the console used by the emulator. This may either be the DMG or the GBC.
- an extra filter to be applied to the output. This increases the resolution of the screen,

using the Scale2x or Scale4x[4] algorithms (see 5.4).

- the size of the emulator's screen, allowing resizing to two or four times larger.
- a slider to change the volume of the emulator's audio output.
- a palette selector, to change the four hues of the DMG's screen.
- two buttons to upload the boot ROMs of the DMG and GBC. This is needed for users that wish to play with the full start-up screen of the console, as these two ROMs are copyrighted and cannot be distributed. If they're not provided, the emulator simulates their effect on the system.
- miscellaneous toggle-able settings, grouped together:
  - an option to play with or without the initial boot ROM of the emulator (a boot ROM must have been uploaded).
  - a toggle to enable frame-blending, meaning for every new frame the output is mixed with the previous frame. This is a nice addition to have, because certain games made some objects flicker on screen to make them appear translucent (since the flicker was not visible to the eye).
  - a button to show the performance statistics of the emulator. This is mainly useful when developing the emulator to make sure it is still efficient.
  - a toggle to enable the debug view of the emulator, where the currently loaded tileset and background map are displayed (see 5.5).

Figure 5.4: Output of the GBC with, from left to right: no filter, Scale2x and Scale4x

5.1.3 Watch Expressions

The second drawer allows the user to define custom JavaScript functions to inspect the state of the emulator regularly (see 5.6). This requires knowledge of the inner structure of the emulator, as the field names need to be used, but is quite useful when needing to inspect parts of the

[4]See https://www.scale2x.it/

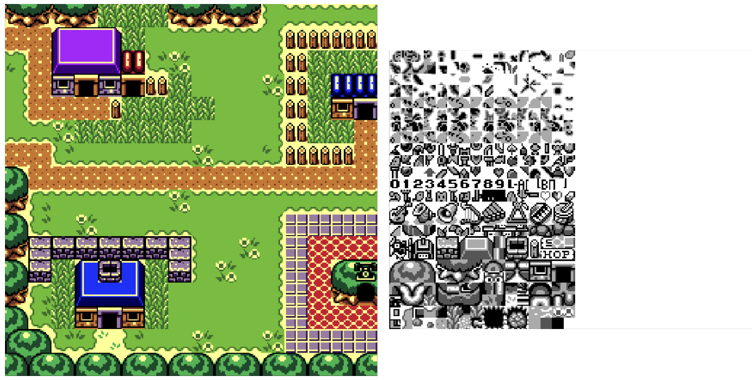

Figure 5.5: Debug view of the emulator

console that are not accessible via memory, like internal counters used for components, or register values.

To implement this, the `Function`[5] constructor is used, which takes in a string with the function's code. This allows the user to dynamically change the expression, and the component will simply update the function, without needing to reload the whole application. These expressions are also automatically saved to `localStorage`[6], meaning they will be kept between sessions.

The user-defined function is then repeatedly invoked, and the result output below the expression, allowing for a live-status of the emulator. If an error is thrown by the function (due to a null value, or invalid expression), the error is caught and '`Error`' is displayed.

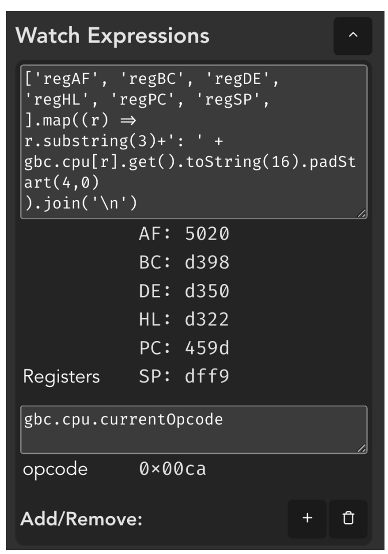

Figure 5.6: Watch expressions drawer

[5] See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/Function

[6] See https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage

5.1.4 Test ROMs

To ensure emulators work properly, a variety of test ROMs have been made, that test most aspects of the GB (see Gameboy Test ROMs). The front-end of the emulator supports running a large number of them in an automated way (see 5.7). The user can select the group of tests desired by ticking the associated checkbox, or select and unselect everything by pressing the top right button. They can then run them by pressing the "Test" button, making all selected tests run internally (without receiving any input our outputting anything directly). The status of each individual test is displayed, and the user can click on the test name to run it on the main emulator, for further debugging or inspection.

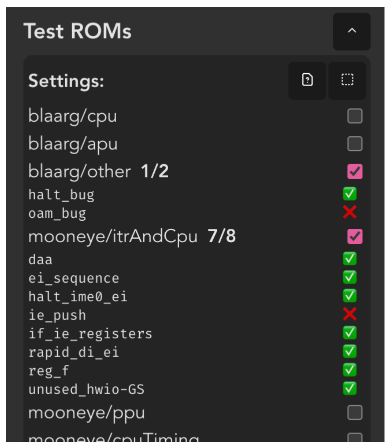

Figure 5.7: Test ROMs drawer

To run the tests, an emulator instance is created, with the test ROM as the input data and "spy" objects provided for input and output – these spies store the output of the emulator, to evaluate the state of the test. To stop execution, the emulator's state is inspected regularly, and the outcome of the test is checked (with execution terminating if the test takes too long to end).

To detect the outcome of each test, these are grouped according to what test suite they belong to. Each test suite is then associated to a *success function*, that can either return `"success"`, `"failure"` or `null` if no outcome has been reached yet. As such, all tests in a test suite have a similar way of reporting success and failure.

The currently automated test suites are:

- The Blaarg test ROMs[7]. The success function of this suite is quite complex, as this suite does not have a standard way of outputting the result. As such multiple parts of the

[7] See `https://github.com/retrio/gb-test-roms`

emulator are inspected simultaneously:

  - `"Passed"`[8] and `"Failed"`[9] may be output to the console's serial port.
  - If the memory at `0xa001-0xa003` is equal to `0xdeb061`, then the byte stored at `0xa000` is the status of the test[10].
  - For the `halt_bug` test, there does not seem to be anywhere where the result is output, so the graphical output of the emulator is verified.

- The Mooneye test ROMs[11] and SameSuite[12], of which all tests are verified the same way. In case of success, the `B`, `C`, `D`, `E`, `H` and `L` registers hold the values 3, 5, 8, 13, 21 and 34 respectively (Fibonacci's sequence). If they instead all hold the value `0x42`, the test failed[13].
- The Acid test ROMs (DMG[14], GBC[15]). These need to be tested graphically, as their purpose is verifying the actual output of the emulator rather than it's behaviour.

Thanks to this, the emulator's frontend supports a total of 191 automated test ROMs, that verify the behaviour of most of the emulator. Of these 191 tests, 101 pass.

5.1.5 Memory Inspect

When debugging an emulator, being able to inspect it's memory is essential, as some bugs may be caused by the wrong mapping of components, or a fault when writing data. To help debugging this, the frontend comes with a basic memory inspection tool, that can show the entirety of the *addressable* data of the emulator (see 5.8). This means data that is unaccessible by the CPU (for instance, because the appropriate ROM bank is not selected) cannot be inspected here. A simple offset can also be indicated, to limit the data to a certain area.

5.1.6 Keybindings

The user can also customise their keybindings for the emulator, by mapping each input to a separate key (see 5.9). This is only relevant on keyboard-equipped devices.

[8] See lines 50-54 of `mem_timing/source/common/testing.s` in `https://github.com/retrio/gb-test-roms`
[9] See lines 112-139 of `mem_timing/source/common/runtime.s` in `https://github.com/retrio/gb-test-roms`
[10] See 'Output to memory' in `dmg_sound/readme.txt` of `https://github.com/retrio/gb-test-roms`
[11] See `https://github.com/Gekkio/mooneye-test-suite`
[12] See `https://github.com/LIJI32/SameSuite/`
[13] See `https://github.com/Gekkio/mooneye-test-suite/#passfail-reporting` for the Mooneye test suite, see lines 265-281 of `include/base.inc` in `https://github.com/LIJI32/SameSuite/` for the SameSuite
[14] See `https://github.com/mattcurrie/dmg-acid2`
[15] See `https://github.com/mattcurrie/cgb-acid2`

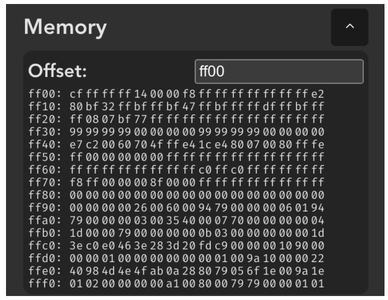

Figure 5.8: The memory inspection drawer

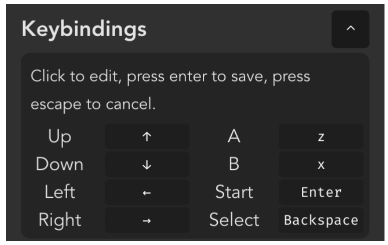

Figure 5.9: The keybindings settings

5.2 Emulator-Frontend Interfaces

To allow the emulator to communicate with the frontend, three components were created, each responsible for part of the output/input.

5.2.1 Screen output handler

To allow easily displaying graphics data from the emulator on the frontend, a `Screen` component was created. This highly configurable screen takes multiple parameters, to allow configuring the screen's size, scale, if it has any upscaling filters applied, if it has any colour palettes, and if it needs to blend frames together (see figure 5.10).

```
type VideoReceiver = (data: Uint32Array) => void;
type ScreenProps = {
    inputRef: MutableRef<VideoReceiver | undefined>;
    width?: number;
    height?: number;
    scale?: number;
    Filter?: ImageFilter;
    blending?: boolean;
    id?: string;
    palette?: Partial<Record<number, number>>;
};
```

Figure 5.10: Parameters of the `Screen` component

All of the parameters are optional, and use sensible defaults when not specified. The only required parameter is `inputRef`, a modifiable "pointer" to the screen input function. This allows passing data to the screen without re-rendering the whole website. Because pointers do not exist in TypeScript, what is passed is actually an object with a modifiable `value` field.

Whenever any parameter changes, a new *render function* is generated, and is set in the given "pointer". This function takes as input the graphics data (a `Uint32Array`[16], containing ARGB values), and applies the required transformations to it. The function also holds a reference to two backing data buffers, for the previous and the current frame. This allows for blending between frames, and avoids creating new arrays repeatedly, lowering the memory consumption of the screen. The transformations it applies are:

1. Roll the buffers, setting the previous frame to the current frame, and the current frame to the newly received frame.
2. If needed, apply the colour palette to the input. The palette is a simple map, from source colour to target colour.
3. If it's the first frame to be drawn, also set the previous frame to the newly received frame (this avoid blending the new frame with a black frame, since the buffers are initialised with `0x000000` values).
4. If required, blend the previous frame and the current frame.
5. Apply the upscale filter to the image (this can be an identity filter, that simply returns the image, or upscale filters like Scale2x and Scale4x).
6. The image data is created from the `Uint32Array`, and drawn on the `<canvas />`.

This simple pipeline of functions makes the code really easy to understand and potentially modify, and allows for easy extension, as any filter could be given to the screen and it would work seamlessly. It is also of interest to note that the order in which the operations are applied maximises efficiency. For instance, the filter is applied last, to avoid having to do previous operations like blending on much larger images.

5.2.2 Audio output handler

To output the sound generated by the emulator, a utility class `AudioPlayer` was made. It is a simple class, that creates an `AudioContext`[17] instance on creation. This class allows playing

[16] See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Uint32Array

[17] See https://developer.mozilla.org/en-US/docs/Web/API/AudioContext

sound output from the browser.

A difficulty with playing audio data stems from the fact that audio samples are quite short (they are output every frame, so every 16.6 milliseconds), and they must be constantly output, with tolerance for small delays in the output. If there happens to be even a small gap between two consecutive samples, a audible pop might play, which results in a very unpleasant experience.

To avoid this, `AudioPlayer` actually adds a small delay before starting to play audio. This ensures that by the time the first received sample is done playing, a new sample has already been received and queued for output, meaning there are never gaps between samples (except if there is an exceptionally long delay between samples, which should not happen).

A drawback from this solution however is that if samples are received too fast, a delay may create itself, as too many samples are enqueued, and new samples need to wait for all the previous ones to be played before being played itself. To avoid having this delay grow too large, a parameter of `AudioPlayer` allows setting the max size of the enqueued audio samples (the frontend uses a value of 8) – any samples that would make the queue exceed this size are ignored, meaning the max delay of the playback is $8 * \frac{1}{60} = 0.13$ seconds, a reasonable delay.

5.2.3 Input handler

To provide a simple way of handling inputs whether the device has a keyboard or uses touch controls, a `GameInput` component was created. It handles keyboard inputs, via a helper hook[18], `useKeys`. To handle touch screen inputs, this component creates buttons mimicking those of the GB. These buttons all have the `mobile-only` class, which is defined in `main.css` to only be visible when no fine pointer is available on the device (ie. no mouse is detected), as can be seen on figure 5.11.

```css
@media (any-pointer: fine) {
    .mobile-only {
        display: none;
    }
}
```

Figure 5.11: `mobile-only` CSS class

This means that if the user is on a mobile devices, the buttons will show, allowing for input without the need for a keyboard (see figure 5.12).

[18] See https://reactjs.org/docs/hooks-intro.html

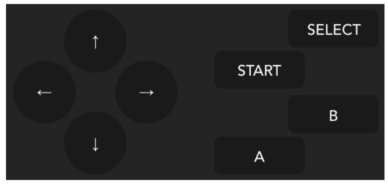

Figure 5.12: Buttons for mobile devices

The component itself then takes as a parameter a callback, that is called with the input function created by this component and returns an up to date object with the currently read inputs `GameBoyInputRead` (see figure 5.2).

5.3 CPU

The CPU is the core of the emulator, and allows running the code from the ROM by reading the operation code (or opcode) and executing the matching action.

The Game Boy's clock ticks at $2^{22} \approx 4.19$MHz, with each of these clock cycles being commonly referred to as T-cycles or T-states [9, CPU Instruction Set]. However, because all of its components tick every four T-cycles, timings of the GB are divided by four and called M-cycles. In other words, four T-cycles make for one M-cycle, and M-cycles have a frequency of $\frac{2^{22}}{4} = 2^{20} \approx 1.05$Mhz [13].

5.3.1 Instruction Set Decoding

Because the Game Boy is an 8-bit system, opcodes are 8 bits (or one byte) long, giving in theory a maximum of $2^8 = 256$ operations. However in the GB the operation `0xCB` gives access to an extended instruction set, meaning that when reading `0xCB` the CPU will read the next byte and use a different logic to execute the operation. This means there are now $2^8 - 1 + 2^8 = 511$ operations. The GB also has 11 'unused' opcodes, that will lock the console when used[9, CPU Comparison with Z80], meaning there are in total $2^8 - 12 + 2^8 = 500$ operations to implement.

Multiple techniques exist to handle this large number of operations:

- Have a large switch-statement for all possible operations. This is the most straightforward option, but can result in quite large switch-statements, especially for CPUs with more opcodes. The GB has comparatively few opcodes as it is an 8-bit system; the Gameboy Advance, the console that followed it, had 32-bit opcodes: way too many for a switch statement to be appropriate.
- Decode the operation by reading specific parts of the opcode, and generate the instructions

dynamically. This method is what is done for larger instruction sets, where opcodes can be split into separate parts to describe the operation's behaviour [16, ARM CPU Reference]. It however comes with the cost of extra processing for each instruction, as it needs to be decoded. This method is applicable to the GB, as many instructions follow a pattern – see, for instance the `LD` instructions, that all follow the same order of registers: `B`, `C`, `D`, `E`, `H`, `L`, `(HL)` (the byte at address `HL`) and `A` [17].

- Create a dispatch table: a map that associates each opcode to a function to execute [18]. This technique is also only viable for small numbers of opcodes, as the map can result quite large. An advantage of this method is that the map does not have to be explicitly written out entirely – generators, or macros, can be used to populate some chunks of it, making it a hybrid of the two previous solutions: there is very little overhead to execute an instruction, as each opcode is associated to a function, but there is also no need to write out every instruction separately, as the map can be generated in parts or in its entirety (inducing a light setup cost).

Initially, the emulator had a large map, with all the opcodes as keys. The functions associated would then execute the instruction and return the number of cycles taken by the instruction (see figure 5.13). This however proved quite repetitive and prone to errors. To solve this, a *generator function* was created. To use it, two parameters must be given: a map of opcodes to values (of an arbitrary type), and a helper function that for each arbitrary value returns a function that executes the instruction (see figure 5.14). This allows generating repetitive instructions more easily, by only specifying what opcodes and objects are used, and not what the whole body of the instruction is (see figure 5.15). This method is used in other emulators – for instance, SameBoy has a map that has an opcode-function mapping for all opcodes, and uses macros to generate the different functions[19].

```
protected instructionSet: Partial<Record<number, InstructionMethod>> = {
    // NOP
    0x00: () => 1,
    // LD BC/DE/HL/SP, d16
    0x01: (s) => { this.regBC.set(this.nextWord(s)); return 3; },
    0x11: (s) => { this.regDE.set(this.nextWord(s)); return 3; },
    0x21: (s) => { this.regHL.set(this.nextWord(s)); return 3; },
    0x31: (s) => { this.regSP.set(this.nextWord(s)); return 3; },
    ...
}
```

Figure 5.13: Initial instruction set implementation

[19] See https://github.com/LIJI32/SameBoy/blob/master/Core/sm83_cpu.c

```
protected generateOperation<K extends number, T>(
    items: Record<K, T>,
    execute: (r: T) => InstructionMethod
): Record<K, InstructionMethod> {
    const obj: Record<K, InstructionMethod> = {};
    for (const [opcode, item] of Object.entries(items)) {
        obj[opcode] = execute(item);
    }
    return obj;
}
```

Figure 5.14: Generator function used for the CPU

```
protected instructionSet: Partial<Record<number, InstructionObject>> = {
    // NOP
    0x00: () => 1,
    // LD BC/DE/HL/SP, d16
    ...generateOperation(
        {
            0x01: this.regBC,
            0x11: this.regDE,
            0x21: this.regHL,
            0x31: this.regSP,
        },
        (register) => (s) => {
            register.set(this.nextWord(s));
            return 3;
        }
    ),
    ...
}
```

Figure 5.15: Improved instruction set implementation

5.3.2 A cycle accurate CPU

The Gameboy is a memory-bound system, meaning that it is limited by it's memory accesses. The CPU can only execute either one read or one write per M-cycle [19, CPU core timing]. It also needs to retrieve the opcode for each instruction, which takes an additional cycle, meaning an instruction performing no memory accesses lasts one cycle, and an instruction performing n memory accesses lasts at least $n + 1$ cycles. Finally, the GB overlaps the last cycle of the execution with the fetching cycle for the next opcode – this will thus be of importance when implementing the fetching of the opcode. See figure 5.16 for the breakdown of an instruction.

The current implementation is not M-cycle accurate: the CPU instruction is executed as one monolithic block, rather than in different smaller parts. This becomes crucial when the timer, OAM and PPU are involved, as they run in parallel with the CPU, so memory accesses

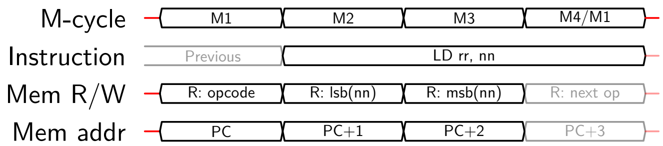

Figure 5.16: Timing of the `LD rr, nn` instruction
Taken from GBCTR [19, Sharp SM83 instruction set]

to these components may return different values depending on the M-cycle.

In all emulators that were found when researching for this project, M-cycle accuracy is reached by making the emulator "CPU-driven". What this means is that inside each instruction, between each cycle, the CPU is responsible for ticking the rest of the system – the main loop is then only responsible for continuously running the CPU, and nothing else. This approach is probably the simplest and most straightforward one, as it is quite simple to implement: all one needs to do is call the system tick method when relevant (see figure 5.17).

Some emulator where this method was found:

- In Mooneye GB, this can be seen by the usage of the `CPUContext.tick_cycle()` method when appropriate[20].
- In accurateboy, the `Bus.tick()` method is called from within the instructions (or it is done implicitly by methods such as `Bus.read()`)[21].
- In Sameboy, the CPU uses functions like `cycle_read` or `cycle_no_access` to tick the rest of the system[22].

```
protected ld_bc_d16(system: System) { // LD BC, d16
    const lower = system.read(this.regPC.inc());
    system.tick();
    const upper = system.read(this.regPC.inc());
    system.tick();
    this.regBC.set(upper << 8 | lower);
}
```

Figure 5.17: CPU-driven `LD BC, (HL)`

This solution however comes with the cost of coupling the CPU to the system, or at least to a way of ticking said system. A particularity of this emulator is that the CPU is almost autonomous, in that it does not interact with any other functionality of the system directly, except the interrupts. The only other interface it requires is an `Addressable`, to allow access to memory (and thus the other components). This also means the emulator is no longer "CPU-

[20] See https://github.com/Gekkio/mooneye-gb/blob/master/core/src/cpu.rs
[21] See https://github.com/Atem2069/accurateboy/blob/master/accurateboy/CPU.cpp
[22] See https://github.com/LIJI32/SameBoy/blob/master/Core/sm83_cpu.c

driven", since the CPU *cannot* tick the rest of the system. It is instead up to the root system to tick all of the components (including the CPU). This is thus a code quality improvement, that is not present in other emulators.

This was done by splitting all the instructions into their respective steps, and have each step return the next step (or `null` if it's the last step). The CPU now must simply store whatever the instruction returns: if it's `null` then it needs to fetch an instruction to prepare for the next cycle, otherwise it's a function and must be executed (and it's result stored for the next step). See figures 5.18 and 5.19 for an example of this.

In the final version of the CPU's tick method, a method `loadNextOp` is implemented. It not only handles getting an instruction associated with the desired opcode, but it also is responsible for checking if an interrupt was raised (in which case the interrupt handling procedure occurs [9, Interrupts]), and for ensuring the CPU is not halted before running an instruction.

```
protected ld_bc_d16(system: Addressable) { // LD BC, d16
    const lower = system.read(this.regPC.inc());
    return () => {
        const upper = system.read(this.regPC.inc());
        return () => {
            this.regBC.set(upper << 8 | lower);
            return null;
        }
    }
}
```

Figure 5.18: System-driven `LD BC, d16`

```
step(system: Addressable, interrupts: Interrupts) {
    if (this.nextStep === null) {
        // looks up opcode in instruction table
        const nextStep = this.loadNextOp(system, interrupts);
        if (nextStep === "halted") return;
        this.nextStep = nextStep;
    }
    this.nextStep = this.nextStep(system, interrupts);
    if (this.nextStep === null) {
        // opcode is fetched on the last cycle of execution
        this.currentOpcode = system.read(this.regPC.inc());
    }
}
```

Figure 5.19: System-driven step of the CPU. Note how the opcode is fetched directly when the next instruction is over, to emulate the overlap between the fetch and the execute steps.

5.4 System

The `System` class implements the system bus, as well as the ticking of all other components. It handles the ticking of the PPU, timer, APU and interrupt logic. Furthermore, it is the component that links all of the data together: whenever a component is ticked (any of the above or the CPU), the `System` instance is passed, so that the components can read and write to the rest of the system. It implements `Addressable` (see 4.6), and internally has a `getAddress` method that returns the `Addressable` at this specific address, to be accessed in the `read` and `write` methods. This ensures that the logic used to determine what component is accessed depending on the address is not duplicated.

All components of the Game Boy exist is the same address space. The *memory map* is what describes the allocation of different components to different areas of memory.

5.4.1 `getAddress` optimisation

Because `System` is a highly used component and is accessed for almost every read and write, the `getAddress` method is under a lot of pressure: for a game like "The Legend of Zelda: Link's Awakening DX", the method is called around 650,000 times per second. For a more complex and resource-intensive game such as "Alone in the Dark: The New Nightmare", around 1,900,000 calls are made. This intensity in usage can easily be explained, as the GB is a memory-bound system: almost all interactions between components occur through memory reads and write. `getAddress` must thus be optimised as much as possible, as it is one of the main bottle-necks of the emulator.

An initial implementation of `getAddress` used the combination of a list of if-statements for different ranges of the memory map, as well as a map where all register addresses where mapped to the component responsible for them. The code would first check if the key exists in this map, and if not it would then go through a series of if-conditions (see figure 5.20). It would return a tuple, containing both `Addressable` to use and the address within in – this was needed as some components had a particular mapping to memory. This is the case for the Work RAM (WRAM), who's memory starts at `0xC000` and ends at `0xFDFF` (spanning over 15.5KB bytes) despite it only being 8KB long, because the last 7.5KB of its address range map back to the beginning of it (this is called the Echo RAM [20]).

This proved quite costly:

- This code creates a new map every time it is called, when checking for the registers'

```
protected getAddress(pos: number): AddressData {
    const register = {
        0xff00: this.joypad,
        ...
        0xffff: this.intEnable,
    }[pos];
    if (register !== undefined) return [register, pos];

    if (0x0000 <= pos && pos <= 0x7fff) return [this.rom, pos]; // ROM Bank
    ...
    if (0xfe00 <= pos && pos <= 0xfe9f) return [this.oam, pos]; // OAM

    // Unmapped area, return 'fake' register
    return [{ read: () => 0xff, write: () => {} }, 0];
}
```

Figure 5.20: Initial implementation of `getAddress`

addresses.

- Having to return both an `Addressable` and an address is quite costly, as a new array with two items must be created every time. It also adds unnecessary complexity to the method, as the system bus should not be responsible for handling the details of address mapping, and instead simply delegate the task to the appropriate component.
- The if-conditions for the ranges of the biggest areas of memory (everything between `0x0000` and `0xEFFF`) happened after the register checks, which delayed response for these reads and writes. This is all the more important that this range represents $\frac{\texttt{0xEFFF}}{\texttt{0xFFFF}+1} \approx 93\%$ of the total address space.
- Chaining if-conditions is inefficient, as the JS engine must step through all conditions and check the values each time. Furthermore, although having both the lower and upper bound of the memory section indicated in the condition (e.g. `0x0000 <= pos && pos <= 0x7FFF` for $[\texttt{0x0000}; \texttt{0x7FFF}]$) makes the translation from memory map to code easier, it is slower, since only the upper bound of the range is needed for the condition if all the ranges below have already been handled.

To fix these issues, we may first move the fine-grained addressing logic to sub-components like the WRAM. This removes the need to return an address from the method: only an `Addressable` is enough, as that component will then be responsible for decoding the address further.

The last two points may then be addressed, by removing the use of if-conditions, and moving this part of the code to the beginning of the function.

With the memory map of the console (see figure 5.21) we can notice how the largest chunks of memory all end with the last three nibbles of the address as `0xFFF`. This means that the component mapped to an address can be determined by simply looking at the first nibble of said address. This is probably done to simplify the circuitry responsible for addressing, as only the most significant 4 bits need to be compared. The emulator can take advantage of this.

Start	End	Description
`0000`	`3FFF`	16 KB ROM bank 00
`4000`	`7FFF`	16 KB ROM Bank 01∼NN
`8000`	`9FFF`	8 KB Video RAM (VRAM)
`A000`	`BFFF`	8 KB External RAM
`C000`	`CFFF`	4 KB Work RAM (WRAM)
`D000`	`DFFF`	4 KB Work RAM (WRAM)
`E000`	`FDFF`	Mirror of `0xC000`∼`0xDDFF`

Figure 5.21: Memory map for the largest chunks of memory [20]

For this area of memory (`0x000-0xEFFF`) the system bus may simply isolate the most significant nibble, and then use a map to associate each address block to a component. Only if the address corresponds to `0xFF--` do we try matching it to a register address. This removes the need for the if-conditions, and is also faster as it is evaluated much earlier on (see figure 5.22). The map can be created on instantiation and kept for later use, to avoid unnecessary memory allocations.

```
protected getAddress(pos: number): Addressable {
    // Checking last nibble
    let addressable = this.addressesLastNibble[pos >> 12];
    if (addressable) return addressable;

    // Registers
    if((address & 0xff00) === 0xff00) {
        addressable = this.addressesRegisters[pos & 0xff];
        if (addressable) return addressable;
    }

    if (pos <= 0xfdff) return this.wram;  // Echo RAM
    if (pos <= 0xfe9f) return this.ppu;   // OAM
    if (pos <= 0xfeff) return Register00; // Illegal Area

    return RegisterFF; // fake register
}
```

Figure 5.22: Optimised implementation of `getAddress`

To ensure placing the “main block” first was the best choice, measurements have been taken of four different GB games. The emulator would log all memory accesses by groups of $100\,000\,000 = 10^8$, and categorise them by address “blocks”: the main block, the OAM block, the illegal block (called like this because this area of memory is restricted by Nintendo, and only returns `0x00`) and the register block. This separation is justified by the fact these are the five chunks of memory that must be checked separately, due to their irregular boundaries. The *second* set of 10^8 accesses was then used to gather the statistics of what blocks are used the most. The first 10^8 accesses are not used, as they may include setup operations that only happen when the game loads, and as such do not represent what the average execution will look like. see figure 5.23 for the results.

Name	**Memory Area**	**Game 1**	**Game 2**	**Game 3**	**Game 4**
Main Block	`0x0000-0xEFFF`	88.869%	95.679%	84.289%	95.740%
Echo RAM	`0xF000-0xFDFF`	0.000%	0.000%	0.000%	0.000%
OAM Block	`0xFE00-0xFE9F`	0.001%	0.000%	0.000%	0.000%
Illegal Block	`0xFEA0-0xFEFF`	0.001%	0.000%	0.000%	0.000%
Register Block	`0xFF00-0xFFFF`	11.129%	4.321%	15.711%	4.260%

Figure 5.23: Access rate per memory blocks, for the second set of 10,000,000 accesses of 4 different games.
Games are, respectively, "Alone in the Dark: The New Nightmare", "The Legend of Zelda: Link's Awakening DX", "Tetris" and "Pokémon Silver".

As we can see, the vast majority of memory accesses go to the main part of memory, with almost all of the rest going to the "register area", the last 256 bytes of the address space. This can easily be explained by the fact all registers that allow interaction with other components (the PPU, the APU, the timer, etc.) are in this narrow range, so it is bound to have a high usage. It is also quite interesting to note that neither the Echo RAM or the OAM block are used at all, except very rarely for one of the tested games. It is thus safe to assume that the conditions responsible for mapping these areas can be left at the end of the function, as they will rarely match an address.

5.4.2 Evaluating the `getAddress` optimisation

A simple experiment was then run, to verify the performance improvement. The first 25 million instructions of the `cpu_instrs`[23] test ROM were run. This sample was chosen because it is considerably large and because the test itself requires around 25 million instructions to complete. For the measurement, `window.performance.now()`[24] was used before and after each drawn frame, and the values were then summed.

The result was the following: 33 955.9ms before the change, and 20 039.1ms after the change. The relative difference is thus $\frac{20\,039.1-33\,955.9}{33\,955.9} = -0.4098$, thus reducing time taken by 40.98%. By measuring the time spent within `getAddress` for these 25 million instructions, we get a total

[23] See `https://github.com/retrio/gb-test-roms/tree/master/cpu_instrs`
[24] See `https://developer.mozilla.org/en-US/docs/Web/API/Performance/now`

spent time in the method of 0.000 151ms on average, showing that this method is no longer a bottleneck for the system, as its impact on performance is minimal.

5.5 PPU

The Picture Processing Unit (PPU) is the component responsible for rendering the game onto the Gameboy's screen. It is one of the most complex components of the GB, with intricate timings, and a behaviour that changes between the DMG and the GBC, due to the addition of colour-support, as well as Video RAM (VRAM) banking.

5.5.1 Presentation

The screen rendering is divided intro three layers (see figure 5.24), drawn on top of each other. From bottom to top, these are:

- The background, a 256×256 image loaded into memory that support scrolling on both axis.
- The window, similar to the background, is a 256×256 image that can be moved around the screen, and is toggle-able. It however does not support scrolling.
- The objects, that are drawn at the very top, are smaller 8×8 tiles. These can be moved freely, and support some transformations, such as horizontal or vertical flipping. Their data is stored in the Object Attribute Memory (OAM).

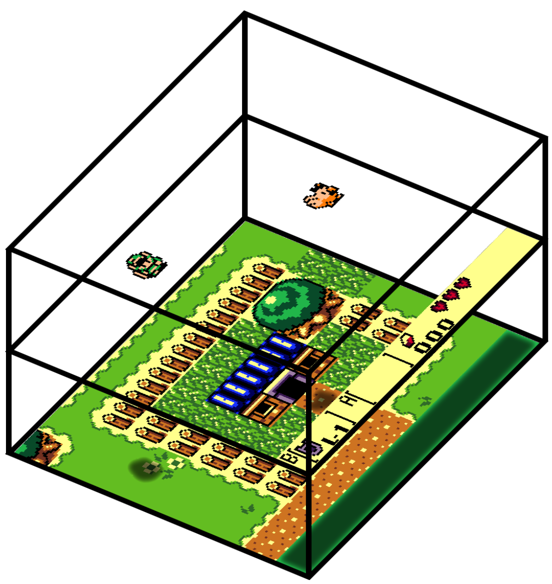

Figure 5.24: Layers of the PPU
From top to bottom, the objects, the window and the background

It renders onto the screen line by line, meaning it will draw a 1-pixel wide line (the scanline) from left to right, and then proceed to the line below. Drawing an entire line takes 114 M-cycles, and there is an 1140 M-cycle delay when the bottom of the screen is reached called the VBlank. The advantage of having per-line rendering is that by modifying the position of the background between each drawn line, complex visual effects can be obtained quite easily [21], see figure 5.25 for an example.

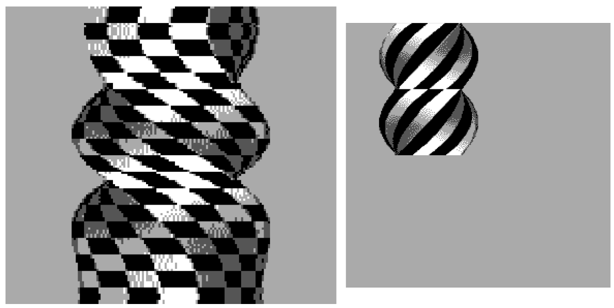

Figure 5.25: Visual effects made by scrolling the background
Gameboy output (right), and loaded background data (right)

One of the main characteristics of the PPU is that it operates in modes. During each mode, it perform a different set of operations, and can only be interacted with in certain way [9, LCD Status Registers]. These four modes are, in order:

- Mode 2: the PPU looks through the OAM, to look for all the sprites to draw. During this time, the OAM is not accessible.
- Mode 3: the PPU reads both the VRAM and OAM, and draws the line. None of the video data is accessible here.
- Mode 0: the PPU does nothing for a short length of time after each line. An interrupt is raised at the start of this period, allowing the CPU to be notified that now is the time to update what's rendered if necessary. This is called the horizontal blank, or HBlank.
- Mode 1: the PPU does nothing for a long length of time after the bottom of the screen is loaded. This is usually where the bulk of the graphics update go, as it lasts a considerable amount of time (1140 M-cycles). An interrupt is also raised here, to notify the CPU. This is called the vertical blank, or VBlank.

Rendering of the line is thus done during mode 3, when no graphics data is accessible, using a FIFO queue (First In First Out) of pixels that can be pushed in and out as the data is read. Some of the registers are still writable, but most games do not use them during this mode because they would require very precise timing. An example of a game relying on this is "Prehistorik Man", that changes the colour palette registers during the scanline in its intro-scene [8, Tricky-to-emulate games].

Since this is only the case for a minority of games, we can go past this inaccuracy. We'll thus take advantage of this to simplify greatly the rendering logic and make the renderer "scanline based", drawing an entire line all at once. This shortcut is commonly used by emulators that are not too accuracy-oriented.

To handle each mode separately, we have four distinct "mode" objects of type `PPUMode`, that hold some basic information on the operation of the mode: it's length, it's flag for the `STAT` register and the name of the method in `PPU` responsible for ticking said mode (see figure 5.26). Note `KeyForType<T, V>` is the union of all keys `k` of `T` such that `T[k]` is of type `V`.

```
type PPUMode = {
    doTick: KeyForType<PPU, (interrupts: Interrupts) => void>;
    flag: number;
    cycles: number;
};
```

Figure 5.26: Type definition of `PPUMode`

5.5.2 Rendering Logic

The first step needed to render the game is to select and order all the sprites to be shown on screen. One of the limitations of the GB is that despite having enough space in memory for 40 sprites, only 10 may be rendered at a time in a scanline [9, OAM]. The sprites are selected during mode 2 – as such, during the last cycle of mode 2, the emulator will look through the available sprites in the OAM, and select the appropriate ones. This can be done quite elegantly in a functional manner, using an array (see figure 5.27).

```
this.readSprites = this.oam
    .getSprites()
    .filter((sprite) => sprite.y <= y && y < sprite.y + objHeight)
    .slice(0, 10)
    .map((sprite, index) => [sprite, index])
    .sort(objPrioritySort)
    .map(([sprite]) => sprite);
```

Figure 5.27: Retrieve the selected sprites for the scanline

First, we retrieve the sprites from the OAM. Internally, sprite data is cached between scanlines and invalidated when written to – `getSprites()` updates the dirty tiles in the cache and returns them. This avoids decoding the sprite every line if that tile has not received any changes, while also ensuring excess decoding is not done if the sprite is modified twice between scanlines. We then select the first 10 sprites to be part of the scanline – this selection is done based on index: the PPU looks through the OAM sequentially to find matching sprites [9, OAM]. Finally, a re-ordering of the sprites needs to be done, to determine which sprite goes above which – this may depend on the sprite's position in the OAM, or on it's X coordinate. This is handled in `objPrioritySort`. To allow sorting based on both attributes, the sprites must be briefly remapped to a tuple with the sprite and it's index (as JavaScript does not expose the indices of elements when ordering them). These sprites are then stored in the PPU object, to be used when rendering at the end of mode 3.

The PPU uses a form of indirect addressing to handle data (see figure 5.28). It has an area in VRAM that stores tile data, a tile being an 8×8 image. Whenever one of these tiles needs to be used, for the background, or for an object, the identifier of the tile needs to be used, this identifier being derived from the tile's address. This allows for the very simple re-use of tiles, which is particularly relevant for backgrounds where they may be repeated a lot. This background data is stored in a *tile map*, a 256×256 map of 1-byte indices to the actual tile data. The GB has two such maps, and both the background and window can display either of them – this is controlled via a flag in the `LCDC` register.

To render the scanline, the layers need to be drawn on top of each other: background, then window, then objects. Due to the similarities between background and window, a `drawLayer` method can be shared to handle the bulk of the work.

This method will loop over all tiles it needs to draw in the current scanline. It first needs to determine the tile's index. This index can be used to the retrieve the tile's address in VRAM. It can also be used to fetch the attributes of the tile; this is a GBC-exclusive feature, that allows transforming tiles (for instance, flipping them, or selecting a different palette for the tile). These attributes are stored in the second bank of VRAM, at the same address, in a single byte of data. The DMG and GBC distinction is however not needed, because the attributes for a `0x00` value match the attributes used by the DMG. This means that instead of having two different methods split on the console versions, we can setup the DMG emulator to always return `0x00` for the second bank of VRAM, and keep the GBC way of handling of attributes.

Once the tile index has been acquired, the PPU may retrieve the tile data address from VRAM. This one-byte address needs to be converted to a valid address in the tile data range of VRAM, as it contains two partially overlapping tile data areas – a flag in the `LCDC` register controls this. Once the address where the tile data (ie. it's texture) is obtained, the PPU may finally retrieve said data, and draw it in the scanline.

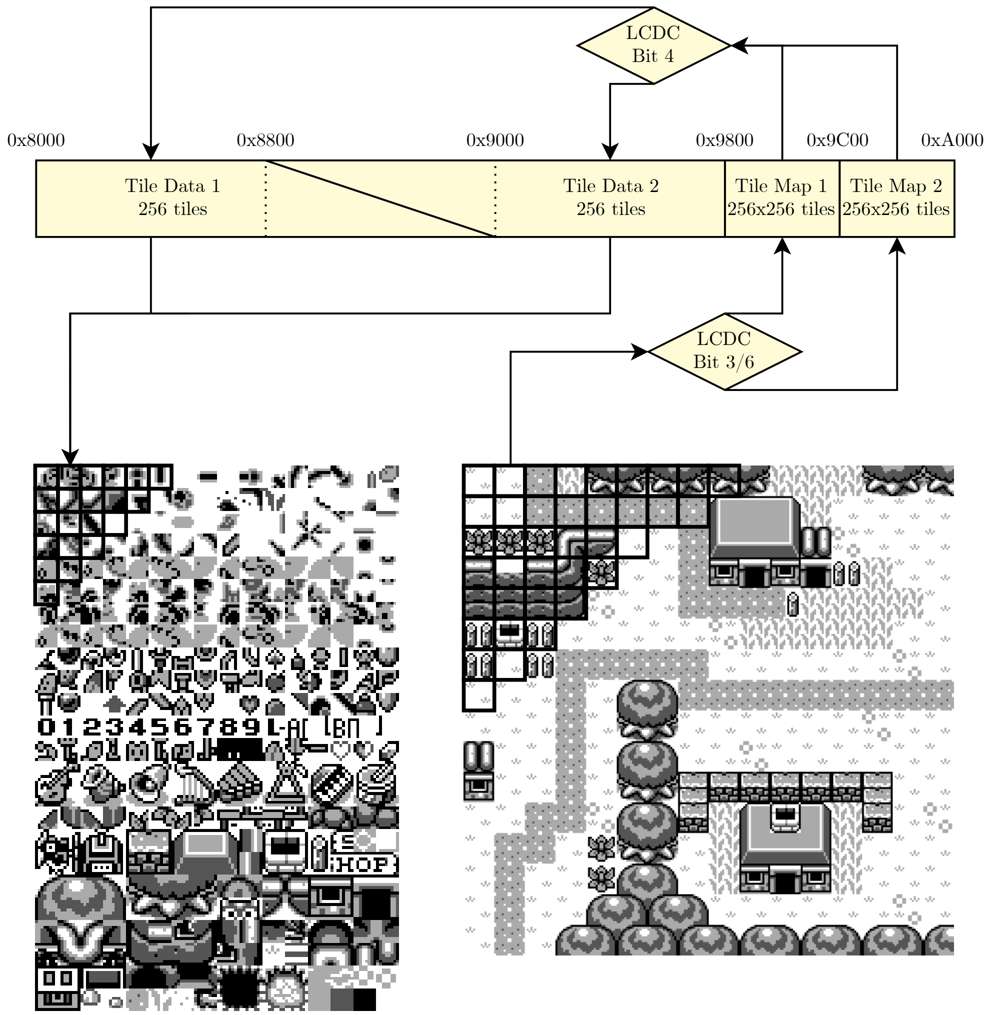

Figure 5.28: PPU logic to obtain tile data
This applies to the DMG – the CGB has two VRAM banks, thus having twice as many tile textures, and an additional tile attribute map.

This tile data, an 8×8 images where each pixel can be one of four shades is encoded in 16 bytes [9, VRAM Tile Data]. Because tile data is rarely changed and the procedure to extract the image data from these 16 bytes is quite complex and requires some bit manipulation, tiles are cached.

5.5.3 Subcomponents

Although it is presented to the system as a unit, the PPU is implemented as multiple smaller components, that handle different parts of the screen logic. This is all the more important that the PPU is extremely complex, so care is needed to ensure the code remains maintainable.

The Object Attribute Memory (OAM) is the memory area in the PPU that stores object data. This small 160-byte long area has a feature called OAM Direct Memory Access (DMA), allowing for direct transfers from ROM or RAM to the OAM. Because this transfer runs in parallel to the other components, the OAM itself is its own component, contained inside the PPU. This allows for a better separation of concerns between the two: the PPU handles rendering, the OAM handles OAM DMA and sprite storing and decoding.

The colour management of the PPU also requires some extra logic that can be extracted into its own class. Although on the DMG this is limited to splitting a byte into 4 shades of 2 bits (white, light grey, dark grey and black), the GBC has 16 palettes of 4 colours, 8 palettes for the background and window, and 8 for sprites [9, Palettes]. As such, a `ColorController` class was created: it implements `Addressable`, and has a method to retrieve the palette for the background, and a method to retrieve the palette of a sprite. This class is extended by a `DMGColorController`, and a `CGBColorController`. The `PPU` picks one of the two on instantiation, based on the emulated console, and can then use both, without needing to know if the screen is monochrome or coloured.

Similarly, the VRAM behaves slightly different between the DMG and the GBC: the latter's VRAM has two banks it can switch between, via an additional register, and also supports VRAM DMA transfers, similar to what is possible with the OAM. A `VRAMController` class is created: it also implements `Addressable`, and has a few extra methods, to allow getting tiles from it, and reading each bank individually (see figure 5.29).

```
abstract class VRAMController implements Addressable {
    read(pos: number): number;
    write(address: number, value: number): void;

    tick(system: Addressable, isInHblank: boolean, isLcdOn: boolean): boolean;

    getTile(tileAddress: number, bankId: 0 | 1): Int2[][];
    readBank0(pos: number): number;
    readBank1(pos: number): number;
}
```

Figure 5.29: Interface of `VRAMController`

5.6 APU

The Audio Processing Unit (APU) is the component responsible for producing the sounds and music of the GB. It's output is the combination of four different channels, each with their own properties: two square channels, a wave channel and a noise channel [9, Audio]. These channels run independently, can be turned on or off individually, and are merged to form the output.

Most features of the channels can be derived to a counter, that ticks down from a value and has an effect when reaching 0. Although they differ in output, all channels have some attributes in common. For example, they all have a *length timer*, that turns off the channel when it reaches 0. Channels 1, 2 and 4 also have an *envelope*, that allows updating the volume of the channel at a set frequency. To have these behaviours work across all channels, an abstract class `SoundChannel` was created. It holds the registers all channels have in common, as well as methods like `tickLengthTimer` to handle common mechanisms across all channels.

To return their output, all channels have a `getSample` method that returns the current output value of the channel, a value between 0 and 15 (4 bits). This output can then be combined, and output to the frontend to handle. It is of interest to note that the GB is always outputting sound, with a resolution of 1 M-cycle, ie. $2^{20}\text{Hz} \approx 1.05\text{MHz}$. This is significantly more than the sample rate computers usually use, 44.1Hz. To fit to this standard, the emulator's APU has an internal counter that ticks down and only produces a sample every $\frac{2^{20}}{44.1 * 10^3} = 23.8$ cycles. This is not the most accurate way of producing the audio data, but is far simpler and faster than downsampling the audio. An example of very accurate emulator that does downsampling is SameBoy[25].

5.7 MBCs and ROMs

A majority of GB cartridges came shipped with a Memory Bank Controller (MBC). This was done to circumvent the memory limit of the GB: due to it's 16-bit addresses, only memory in `0x0000`–`0x7FFF` is mapped to the ROM, limiting it's size to 32KB. MBCs provided a way of extending this memory limit, by doing *bank switching* [9, MBCs]. The cartridge holds more data than it can address, and the CPU can modify which part of the memory it's accessing by writing to a register in the MBC. Because ROMs are read-only, the write-operations can be safely re-used to instead write to these built-in registers without losing any functionality.

On top of more memory space, MBCs can sometimes provide additional features, such as external RAM, a battery (to be able to keep the state of the RAM between sessions, which allows saving the game) [9, MBCs], a rumble motor [9, MBC5], or a battery-backed real time clock [9, MBC3], for example. The most common MBCs found were the MBC1, with space for a maximum of 2MB [9, MBC1], and the MBC5 – the only MBC to officially support the GBC's double speed mode [22]. see figure 5.30 for statistics of the usage of different MBCs.

[25]See Accuracy, `https://sameboy.github.io/features/`

Name	ROM Count	Percentage
No MBC	2150	23.0%
MBC1	4010	43.0%
MBC2	227	2.4%
MBC3	367	3.9%
MBC5	2532	27.1%
Others	53	0.6%

Figure 5.30: Statistics of different MBCs [23]

To ensure the emulator can run as many games as possible, the main MBCs have been implemented: MBC1, MBC2, MBC3 and MBC5 (although support for the MBC3's real time clock has not been added). Since the GB interacts with the cartridge in the exact same way whether there is an MBC or not, the choice of making a `MBC` abstract class came quite naturally: it's interface is that of `Addressable`, and it also supports a `save` and `load` method, for save files.

To decide which MBC to use, we have a `GameCartridge` class, that wraps around `MBC`. It's role is decoding the cartridge's header, contained in `0x0100-0x014F` [9, The Cartridge Header], to decide on which MBC to use, as well as some of the properties of the cartridge: does it have RAM, is it battery backed (in which case it supports saving), what's the game's title and identifier.

The `read` and `write` methods received an optimisation similar to what was done with `System.getAddress`: because the internal addressing logic of the MBCs only relies on the most significant nibble, a simple switch statement can be made (see figure 5.31).

Internally, the `MBC`s have a `ROM` instance to store the cartridge data (this is obtained from the ROM uploaded by the user), as well as an optional `RAM` instance. This RAM is what the game can edit. If it is battery backed, memory is kept when the GB is turned off, allowing data like scoreboards and progress to be saved. To replicate this saving behaviour, the emulator exposes a `save` and `load` method, that respectively return and set the data in the RAM. The emulator's core is thus not responsible for handling these saves, and the frontend can decide freely how to store them.

In the implemented frontend, this is done by saving the RAM data in the browser's available storage, using the "localForage" library[26]. When a ROM is loaded, the frontend checks if a save for this game exists, by using the ROM's identifier decoded in the `GameCartridge` class. If it does, the save is then loaded. Similarly, when changing ROMs, closing the window, or pressing the "Save" button, the frontend retrieves the RAM data from the emulator and saves

[26] See https://github.com/localForage/localForage

```
read(pos: number): number {
    switch (pos >> 12) {
        case 0x0: // ROM bank 00
        case 0x1:
        case 0x2:
        case 0x3:
            return this.data[pos & addressMask];
        case 0x4: // ROM bank 01-ff
        case 0x5:
        case 0x6:
        case 0x7: {
            const address =
                (pos & ((1 << 14) - 1)) |
                (this.romBankLower8.get() << 14) |
                (this.romBankUpper1.get() << 22);
            return this.data[address & addressMask];
        }
        case 0xa: // ERAM
        case 0xb: {
            if (this.ramEnable.get() !== RAM_ENABLED) return 0xff;
            const address = this.resolveERAMAddress(pos);
            return this.ram.read(address);
        }
    }
    throw new Error(`Invalid address`);
}
```

Figure 5.31: `read` method of `MBC5` [9, MBC5]

it, by setting the key of the entry in the local storage to the identifier of the ROM.

5.8 Timer

The timer is a component in the GB that ticks regularly. It allows the control of two independent mechanisms [9, Timer and Divider Registers].

First is the *divider counter*, accessed and controlled via the `DIV` register. This counter is internally 16-bits, although only the upper 8-bits can be accessed. It is incremented every clock cycle (ie. increased by 4 every M-cycle), can be read, and writing to the divider resets the counter (see figure 5.32).

The second, more complex part of the timer is a *customisable timer*. It is made of three registers:

- `TIMA`: the timer counter. Every time a falling edge is detected on one of the bits of `DIV` (ie. the bit goes 1 to 0), this register is incremented. What bit is inspected can be controlled via the `TAC` register, effectively changing the frequency of the timer. When this register

```
protected divider = new DoubleRegister(0xab00);
protected addresses: Record<number, Register> = {
    0xff04: this.divider.h, // we only ever read the upper 8 bits
    ...
};

tick(interrupts: Interrupts): void {
    const newDivider = wrap16(this.divider.get() + 4);
    this.divider.set(newDivider);
    ...
}

write(pos: number, data: number): void {
    if (pos === 0xff04) { // Writing anything to DIV clears it.
        this.divider.set(0);
        return;
    }
    ...
}
```

Figure 5.32: Implementation of divider counter

overflows (is incremented when equal to `0xFF`), an interrupt is raised.

- `TAC`: the timer control register. It allows enabling or disabling the timer, and changing the inspected bit of `DIV`.
- `TMA`: the timer modulo. It defines what value `TIMA` is reset to when overflowing.

Although it is a seemingly simple system made of 3 registers, the behaviour of this system is quite complex, as it has multiple edge cases [9, Timer obscure behaviour]. The code that handles the increase of `TIMA` is thus quite long, and required some re-factoring to be easily readable (see figure 5.33). As can be seen from this implementation, the `TIMA` register can actually be incremented for two reasons (line 16). The first reason is the regular falling edge on the inspected bit. However, if the timer is disabled then the bit of `DIV` read is going to be 0, regardless of its actual value. This means that if the timer gets disabled while the read bit of `DIV` is 1, the timer detects a falling edge and increases `TIMA`. It is also of interest to note that `TIMA` is not set to the value of `TMA` when overflowing – this is actually done the following cycle (lines 2–9). This logic requires some extra fields to be present, recording the values of the registers on the previous tick. Some additional logic is required in the `write` method, because writes to `TIMA` are ignored when the timer overflows.

```
tick(interrupts: Interrupts): void {
    this.previousTimerOverflowed = false;
    if (this.timerOverflowed) {
        const modulo = this.timerModulo.get();
        this.timerCounter.set(modulo);
        interrupts.requestInterrupt(IFLAG_TIMER);
        this.timerOverflowed = false;
        this.previousTimerOverflowed = true;
    }

    const timerControl = this.timerControl.get();
    const timerIsEnabled = timerControl & TIMER_ENABLE_FLAG;
    const speedMode = (timerControl & 0b11);
    const checkedBit = TIMER_CONTROLS[speedMode];

    const currentBitState = timerIsEnabled && (newDivider >> checkedBit) & 1;

    if (this.previousBitState && !currentBitState) {
        const result = (this.timerCounter.get() + 1) & 0xff;
        this.timerCounter.set(result);
        if (result === 0) this.timerOverflowed = true;
    }

    this.previousBitState = currentBitState;
}
```

Figure 5.33: Code to manage the `TIMA` increments

5.9 Helpful Components

To avoid repeating logic throughout the main components, small classes have been written. One of these is the `Register` class: it holds a single integer, that can be read or written. It also comes with a `flag` method, that returns whether a given flag is set in the register, as this is a frequent operation.

This class is extended by `MaskRegister`, a class to support registers of which all bits may not be used. This is the case for instance for `TAC`: bits 0–2 control the timer, and bits 3–7 are hardwired to 1, and cannot be reset. A `MaskRegister` allows defining this behaviour directly in the register, avoiding the need for additional logical in the class using the register (here, `Timer`). On instantiation, a *mask* is passed as an argument, and is applied whenever the value of the register needs to be changed (see figure 5.34).

To accommodate the 16-bit registers of the CPU, a `DoubleRegister` class was implemented. It is made of two `Register` instances, and provides methods to get and set its value as if it held a 16-bit number (despite being implemented as two 8-bit numbers). This provides flexibility to the CPU, enabling both 16-bit and 8-bit arithmetic, without having to manage the way the

```
class MaskRegister extends Register {
    protected mask: number;

    constructor(mask: number, value: number = 0) {
        super(value | mask);
        this.mask = mask;
    }

    override set(value: number): void {
        super.set(value | this.mask);
    }
}
```

Figure 5.34: Implementation of `MaskRegister`

register is implemented.

Classes to manage memory were also created, to avoid having high-level components access data structures directly. For instance, the `ROM` class is backed by a `UInt8Array`[27], and implements `Addressable`. It is extended by `RAM`, to allow `write` operations.

To simplify the operation of certain components, a `CircularRAM` class was also created. It extends `RAM`, and must be provided an *offset* on creation. When reading or writing to it, the offset is subtracted from the address, and the modulo of the RAM's length is then applied to it (see figure 5.35). This allows this part of memory to handle addressing in an independent way: the high-level component simply has to provide it with it's address on creation.

```
class CircularRAM extends RAM {
    protected offset: number;

    constructor(size: number, offset: number, data?: Uint8Array) {
        super(size, data);
        this.offset = offset;
    }
    override read(pos: number): number {
        return super.read((pos - this.offset) % this.size);
    }
    override write(pos: number, data: number): void {
        super.write((pos - this.offset) % this.size, data);
    }
}
```

Figure 5.35: Implementation of `CircularRAM`

This can be used to implement the WRAM. It can be addressed from `0xC000` to `0xFDFF`, but `0xE000`–`0xFDFF` maps back to `0xC000`–`DDFF`. As such, a `CircularRAM` with an offset of

[27] See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Uint8Array

`0xC000` and a length of `0x2000` will properly wrap address in the `0xE000`–`0xFDFF` range back to the beginning of its memory. This avoids handling this logic in high-level components, and provides a generic solution to this kind of memory component. Other places where this class is used include the wave RAM of the APU, and the VRAM of the PPU.

Chapter 6

Evaluation

Evaluation of this emulator consists in verifying it simulates the guest console (the Game Boy) properly. This can be done via test ROMs, that verify the behaviour of parts of the console. In this chapter, we will both look at how the emulator performs by itself, and how it performs compared to other GB emulators. We will then also ensure that it is performant, and that it complies with the specification.

6.1 Absolute Accuracy

To measure the *absolute accuracy* of the emulator (how accurate it is, independently of other emulators), we will run different test ROMs and see what aspects of the emulator pass which tests. This will help us guarantee that specific parts are functional, and can be relied on.

Note that although on a functioning system test ROMs help diagnose issues, if the system is faulty then there may be false positives. Indeed, the test ROM is still a ROM that runs on the system, and relies on it somewhat working. For instance, a faulty CPU might make a test seem like it passes, despite it actually being wrong. As such, results should be taken with a grain of salt. Furthermore, test ROMs can only test known behaviour of the GB: if there exists a bug or behaviour that is unknown of, there is no way of writing a test ROM to verify it.

Using the "Test ROMs" panel from the emulator frontend, we can run a total of 191 tests automatically. Because each test focuses on a specific characteristic of a component, we can group them, to see the *success rate* this particular component has; see figure 6.1.

We first may notice from this table that the accuracy of the CPU is quite good, the only test failing being `ie_push`[1], a test verifying an edge-case of interrupt handling. The success-rate

[1] See `https://github.com/Gekkio/mooneye-test-suite/blob/main/acceptance/interrupts/ie_push.s`

Component	Category	Test count	Tests passed	Success rate
CPU	Instructions	20	19	95.0%
	Timing	23	23	100.0%
	Sum	43	42	97.7%
PPU	Rendering	2	2	100.0%
	Timing	12	5	41.6%
	DMA	12	7	58.3%
	Sum	26	14	53.8%
Timer		13	13	100.0%
MBCs	MBC1	12	12	100.0%
	MBC2	7	7	100.0%
	MBC5	8	8	100.0%
	Sum	27	27	100.0%
APU	General	17	3	17.6%
	Channel 1	21	0	0.0%
	Channel 2	15	0	0.0%
	Channel 3	16	2	12.5%
	Channel 4	13	0	0.0%
	Sum	82	5	6.1%
All		191	101	52.8%

Figure 6.1: Results of the test ROMs on different components

of these tests being high is essential, since of course all tests run on the CPU and rely on it functioning to operate properly.

The PPU's accuracy is good in terms of rendering, but the timings of it are wrong. When analysing the test ROMs further to understand what goes wrong, we can notice that some of the modes of the PPU are one cycle too long. This can be seen by inspecting the test's output (see figure 6.2), and noticing the error is in the `D` register, that has a value of `0x02` instead of `0x01`. The part of the code responsible for this is in lines 35–40 of the test's source[2]. Although the timing of the emulator could be modified to make this test pass, the change makes other PPU timing tests fail; as such, it is reasonable to suspect the fault is in the logic itself of the `PPU` class and not the number of cycles only. Many modifications have been attempted to improve the accuracy of the timing, to no avail – more research would thus be needed to fix this.

The timer and MBCs have great accuracy, and do not have any problems as far as the tests can tell. This is important, in particular for the timer, because multiple timing tests rely on the timer functioning. As such, having this part of the emulator work reliably is vital to ensuring the other tests work properly.

Finally, the APU performs very poorly. This is firstly due to the complexity of the different channels: they all have several mechanisms working separately, and if only one of them behaves wrong then the whole set of tests will fail. Furthermore, the majority of these tests come from

[2] See https://github.com/Gekkio/mooneye-test-suite/blob/main/acceptance/ppu/intr_2_mode3_timing.s

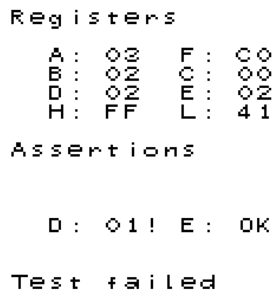

Figure 6.2: Output of PPU timing test `ppu_intr_2_mode3_timing`

the SameSuite[3] test suite. These tests differ from other because they use the `PCM12` and `PCM34` registers of the GBC, to directly inspect the output of each channel [9, Audio Details]. These allows the tests to check with much more precision that the emulator works, making the test significantly more stringent. Although this is an obvious indicator that the APU needs to be reworked and fixed, this does not impact the user experience, or at least not that of a casual user, as the audio of the game does not sound off when being used.

Seeing the total of tests passed is also of interest. The total success rate is 52.8%, but this value is strongly influenced by the fact almost half of the tests are for the faulty APU (82 tests out of 191). If these tests are filtered out, we get a total of 109 tests, of which 96 pass, giving a success rate of 88%. A really good result, given the emulator was written in a short period of time with no prior experience.

6.2 Relative Accuracy

After verifying the accuracy of Emmy by itself, using automated tests, we may also want to compare it to other existing GB emulators, and see how it does in contrast with them.

Thankfully, a tool for this already exists: "GBEmulatorShootout"[4]. It automatically tests GB emulators, and displays their results in a large table, allowing for the comparison of emulators' accuracy.

The way the tool works is that each emulator it supports must have a matching file, to allow downloading the emulator's executable and running it with a given test ROM. The program then regularly screenshots the emulator's window, and compares it with the expected visual output of the test, meaning it expects the emulator's window to only contain the graphical output of the GB.

[3] See https://github.com/LIJI32/SameSuite
[4] See https://github.com/daid/GBEmulatorShootout

This means that adding this emulator to the tool required changes to the tool's source code. Indeed, it has two notable distinctions to other emulators tested by the tool. First and foremost, this emulator is not based on an executable – it is instead *browser based*, meaning that simply running the emulator's file is not possible. The second difference is that the emulator's output does not cover the whole window, as there is an additional UI around it. We must thus instead find a way to extract the emulator's output from the window.

To do this, we will use web development techniques to automate the tests. Selenium[5] is a tool that allows programmatically opening a browser and running actions on it, mimicking those of a user. In this case, these actions would be opening the emulator's website, and uploading the ROM to the emulator, as a user would do. Because the rest of the tool is written in Python[6], we will use the `selenium` Python package[7]. See figure 6.3 for the code to setup the emulator, and the code to run a ROM on it. Opening the window and accessing the emulator is quite simple (see lines 2–6), as we must just open the window at a given URL. We may note the script also does two additional actions: the first is enabling "triple speed mode" (line 5), to make the tests run faster. We also open the "Settings" tab of the sidebar (line 6), as this is where the console selection (DMG or GBC) is done. This is relevant because some tests are designed for a console in particular, and the only way to switch the console on this emulator is via this UI.

```
1  class Emmy(Emulator):
2      def setup(self):
3          self.driver = webdriver.Chrome("emu/chromedriver_win32/chromedriver.exe")
4          self.driver.get("https://emmy-gbc.vercel.app/")
5          self.driver.find_element(value="emu-speed").click()
6          self.driver.find_element(value="drawer-section-settings").click()
7  
8      def startProcess(self, rom, *, model, required_features):
9          model_btn_id = {DMG: "dmg-mode", CGB: "cgb-mode"}.get(model)
10         if model_btn_id is None: # console not supported
11             return None
12         self.driver.find_element(value=model_btn_id).click()
13         rom_path = os.path.abspath(rom)
14         self.driver.find_element(value="rom-input").send_keys(rom_path)
15         try: # if an alert appeared, it means the rom is incompatible
16             self.driver.switch_to.alert.accept()
17             return None
18         except: # no alert, so error thrown, so the rom is compatible
19             return self.driver
```

Figure 6.3: Setup and run code to automate emulator testing

[5] See https://www.selenium.dev/
[6] See https://www.python.org/
[7] See https://pypi.org/project/selenium/

Once the emulator is setup, the test ROM must be handled. First, the desired console model is checked, and the appropriate console is selected in the UI (lines 9–12). This is done my matching the model ID to the ID of the button that selects said console, and clicking the button with that ID. The test ROM may then be uploaded, by pressing the "Import a ROM" button, and entering the ROM's path. We must also check if an alert is raised in the browser (lines 15–19) – this occurs if an error occurs when reading a ROM, usually because the MBC is not supported.

The second part of automated testing requires monitoring the emulator, and verifying if the test succeeded or failed. GBEmulatorShootout requires taking screenshots of the emulator frequently, and comparing the screenshot to an image of the expected result. This method implies storing an additional image for each test ROM, but comes with the advantage that no complex code is required to look into the emulator's state for other indicators of success (like what was done in this project's frontend, see subsection Test ROMs). This makes it the ideal solution for this kind of tool, as the additional code to add an emulator to the test suite is minimal (all emulators will output the Game Boy's screen, whereas not all emulators have the same internal structure).

To do this, we must extract the image from the <canvas/> element the emulator's frontend draws in, and convert it to an appropriate Python object (see figure 6.4). We first fetch the output canvas by it's ID, and execute a small JavaScript function to retrieve the image inside the canvas, in a PNG format. This data is then decoded, and used to create an image, that is resized to the correct size (as the emulator's screen may be upscaled by the browser).

```python
def getScreenshot(self):
    canvas = self.driver.find_element(value="emulator-frame")
    canvas_base64 = self.driver.execute_script(
        "return arguments[0].toDataURL('image/png').substring(21);",
        canvas
    )
    canvas_png = base64.b64decode(canvas_base64)
    large_image = PIL.Image.open(io.BytesIO(canvas_png))
    small_image = large_image.resize((160, 144), PIL.Image.NEAREST)
    return small_image
```

Figure 6.4: Code to get the emulator's output with Selenium

After other minor tweaks to GBEmulatorShootout's code to support web-based emulators, the test script can be run to compile the results and be able to finally compare our emulator to others. The results are accessible online at https://n1ark.github.io/GBEmulatorShootout/ (see figure 6.5). The website to access the results is made of a large table, with on the X axis the emulator running the test ROM, and on the Y axis the name of the test ROM. Emulators are ordered from most accurate (most tests passed) to least accurate, from left to right. The tool currently supports 239 test ROMs, for 14 emulators.

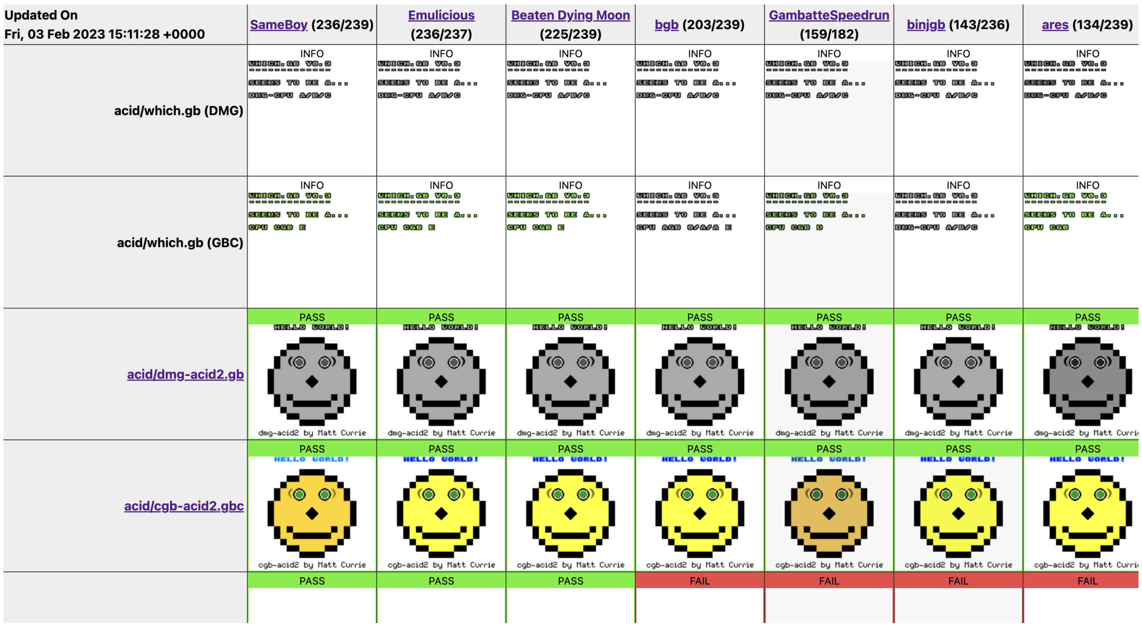

Figure 6.5: Screenshot of the UI of `GBEmulatorShootout`

We may first notice that Emmy passes 118 out of 236 tests (3 tests are skipped because the emulator does not support the "Super Game Boy"), ranking it 9th of the 14 tested emulators. It is relevant here to note that most of these other emulators have existed for several years now, and have been worked on by many people.

We may also use this table to compare the accuracy of this emulator to others, by looking at the success rate for tests of different components (see figure 6.6). Note that if an emulator cannot run a test (because it lacks support for a feature needed by the test), the test is skipped, rather than counted as failed. This explains why "VisualBoyAdvance-M" has a success rate of 57.1% but is ranked lower than Emmy.

This table is a good indicator of where our emulator, Emmy, does and does not perform well, compared to other GB emulators. For instance, it has a really accurate timer, compared to emulators ranked around it. This also emphasises the fact that its CPU is of an accuracy similar to that of the higher accuracy emulators, passing almost 80% of tests. It also helps see that only very high accuracy emulators manage to pass the APU tests. The only outlier seems to be VisualBoyAdvance-M, with an APU success rate of 60.7%. However, when inspecting the test results, we notice that most of the APU tests were not run on it – this is because it lacks

Emulator	Tests Run	CPU	PPU	Timer	MBCs	APU	All
SameBoy	239	100.0%	91.9%	100.0%	100.0%	100.0%	98.7%
Emulicious	237	98.5%	100.0%	100.0%	100.0%	100.0%	99.6%
Beaten Dying Moon	239	100.0%	97.3%	100.0%	97.1%	85.9%	94.1%
bgb	239	74.6%	91.9%	100.0%	97.1%	82.4%	84.9%
GambatteSpeedrun	182	98.5%	81.1%	100.0%	57.1%	100.0%	87.4%
binjgb	236	84.6%	63.9%	92.3%	74.3%	29.4%	60.6%
ares	239	82.1%	51.4%	38.5%	80.0%	29.4%	56.1%
mGBA	233	83.1%	52.8%	92.3%	90.6%	20.0%	57.1%
Emmy	236	76.9%	45.9%	100.0%	79.4%	10.6%	50.0%
VisualBoyAdvance-M	182	53.7%	45.9%	76.9%	62.9%	60.7%	57.1%
PyBoy	143	32.1%	17.2%	38.5%	90.6%	0.0%	40.6%
Goomba	232	30.8%	8.6%	7.7%	59.4%	3.5%	20.7%
no$gmb	236	35.4%	22.2%	0.0%	20.0%	2.4%	17.8%
KiGB	231	29.2%	20.0%	0.0%	12.9%	2.4%	14.7%

Figure 6.6: Success rate of emulators for different components, sorted by total passed tests

support for the `PCM12` and `PCM34` registers these tests use. If we add the un-run tests to the percentage, we end with 17 tests passed out of 85, or a 20% success rate, in line with the success rate of similar emulators. This further shows that creating an accurate APU is challenging, especially given the standards of the used test ROMs.

6.3 Performance

Aside from accuracy, we may want to measure how performant the emulator is. This is important, because most emulators offer a "turbo mode", allowing the emulator to run the GB faster. Furthermore, this emulator is designed to run on mobile devices, that may have less computing power – as such, the emulator being efficient with the resources it has is vital for these platforms.

To measure performance, the most relevant metric is frame time: the time needed in milliseconds to draw a frame (this, of course, includes the rendering of the frame but also all the processing that goes before). The emulator's frontend already comes with a measure of frame time – we thus just need to select what ROM the test will be performed on, and how the measure will be taken.

For the ROM, the performance will be measured on 4 distinct ROMs, all outlining different uses of the emulator. These are:

- "The Legend of Zelda: Link's Awakening", a DMG game. It is quite simple and should not be too memory intensive. It will help outline the average performance on the DMG.
- The `cpu_instrs` test ROM. It's advantage is that it is quite long, and it's purpose is to

test the entirety of the CPU, meaning it will be very processor-intensive.

- "The Legend of Zelda: Oracle of Ages", a GBC game. It is similar to the previous Zelda ROM, but was built for the GBC, meaning it likely requires more resources to run.
- "Alone in the Dark: The New Nightmare", a GBC game. This is a very complex game released for the GBC, that supports "3D-scenes", and very regularly changes the colour palette registers [8, Tricky-to-emulate games]. Both of these properties combined should make this ROM slower, making it a hypothetical "upper-bound" on frame time.

As for how to measure the performance, the frame time of every frame will be summed and averaged, over the first 10 million cycles, to give a more accurate measure. See figure 6.7 for the results.

Console	ROM	**Frame time** (ms)	**Speed increase**
DMG	The Legend of Zelda: Link's Awakening	4.39	279%
	`cpu_instrs`	5.43	207%
GBC	The Legend of Zelda: Oracle of Ages	4.89	241%
	Alone in the Dark: The New Nightmare	5.78	188%

Figure 6.7: Average frame time of different ROMs

From this table we can notice two things. First, DMG games run faster than GBC games. This can be due to the double speed mode, that forces the emulator to run twice as many instructions, or to the fact that the PPU logic with colour is more complex and needs to be optimised. Secondly we may look at the average frame times of these ROMs.

The GB runs at 60 frames per second, meaning a frame must last at most $\frac{1}{60} = 16.6$ms. We may calculate the maximum speed increase attainable for a frame time via the formula $\frac{1000}{\text{frame time}*60}$, or $(\frac{1000}{\text{frame time}*60} - 1) * 100$ to get an increase percentage. For instance, for "The Legend of Zelda: Link's Awakening", the maximum speed increase could be of around 279%.

This is a satisfiable value: a speed increase of 200% is good enough for most uses. Many emulators provide speed increases much superior to this, however this is because they are often written in compiled languages like C++ or Rust, and will thus run much faster than an interpreted language like JavaScript.

WebAssembly[8] is a low-level language made for browsers, resembling assembly language. It is designed for efficiency, and would thus be a great pick for a browser-based emulator as it runs faster than JavaScript [24]. Languages like C, C++ and Rust can be compiled to WebAssembly via tools like emscripten[9], allowing for great performance with portable code. Examples of such

[8] See `https://webassembly.org/`
[9] See `https://emscripten.org/`

emulators playable from the browser include GBEmu[10], an emulator written in Rust.

This would however require a full rewrite of the code, since TypeScript and C++ have little in common in terms of syntax. Another simpler alternative to convert the emulator's code to a WebAssembly-compilable language is using AssemblyScript[11], a language similar to TypeScript.

Research has been done to convert the project's code to AssemblyScript, and the progress so far can be seen on the `assembly-script` branch[12]. However, because AssemblyScript targets such a low-level language, many TypeScript constructs are not (yet) available in it, such as arrow functions, or dynamic objects. Although the implemented emulator does not use these features extensively and most components were successfully migrated to AssemblyScript, the CPU uses arrow functions for all instructions – a successful conversion of the entire emulator would thus require a lengthy rewrite of the CPU. This project was thus abandoned in favour of additional features in the emulator's core and frontend. An example of GB emulator in AssemblyScript is wasmboy[13].

6.4 Compliance to Specification

The presented emulator fulfils most of the specification, with some specifications of lower importance having been left out. These include:

- U4: User can press a button to download a save of their game (or, alternatively, the save can be stored inside the browser with a technology like IndexedDB.

  The user cannot, currently, download the save of their game. The save file is instead stored in the browser. Some research was done to allow saving the emulator's state to the BESS[14] save format, however this required a lot more effort so priority was shifted on other features, as the existing save system was deemed sufficient.

- U7: User can pause the console emulation through a button. They can also input conditions for which the console should break execution.

  The emulator's frontend does not provide a way to add breakpoints to the emulation. This feature was initially supported from the browser's console, but it's implementation was unsatisfactory. A possible improvement to the emulator would thus be support for such breakpoints, maybe in a format similar to that of the current "Watch Expressions"

[10]See `https://github.com/BlueBlazin/gbemu`
[11]See `https://www.assemblyscript.org/`
[12]See `https://github.com/N1ark/gbc-emulator/tree/assembly-script`
[13]See `https://github.com/torch2424/wasmboy/`
[14]See `https://github.com/LIJI32/SameBoy/blob/master/BESS.md`

menu.

- N3: A downloadable version of the web-app can be used on a computer and provides full functionality, via keyboard and mouse inputs.

  The implemented emulator cannot currently be downloaded as a local app. This could however be added quite easily, using a tool like Electron[15] that allows converting web applications to desktop apps. Another option is converting the web-app into a Progressive Web App (PWA)[16], allowing users to "download" the web-app and use it while offline.

All of these specifications are however of low importance, and are mainly small quality of life features. Other possible improvements could be adding more debugging tools (such as a way to inspect the APU's raw output), or more features added to the emulator's core, like support for other MBCs (like the MBC6, MBC7, or MMM01 [9, MBCs]), the real time clock of the MBC3, additional outputs of the GB like the Game Boy Printer or the Game Boy Camera [9], and support for the Link Cable, allowing multiple Gameboys to play together [9, Serial Data Transfer].

[15] See `https://www.electronjs.org/`

[16] See `https://web.dev/progressive-web-apps/`

Chapter 7

Legal, Social, Ethical and Professional Issues

7.1 Privacy

This piece of software is safe to use in terms of privacy – it runs entirely locally, with no data ever being sent from the user to the server. There are no cookies, and the only forms of storage used are localStorage[1] and localForage[2], both of which are local and offline.

localStorage is handled by the browser, so it is the user's responsibility to ensure that they use a secure browser.

localForage is an open source library, allowing for more transparency – one could go through the source code to verify that it is safe as well. Whenever a new update is released, we may simply verify that the modified code is still safe, and then configure the project to use the new version in the `package.json` file.

7.2 Legality

The Game Boy is a copyrighted console, owned by Nintendo, but making an emulator for it is deemed legal as long as it is done following a "clean room" design [25]. Overall, it is legal to make emulators, as long as they follow this method. This was ruled via a series of court appeals between console manufacturers and groups that produce emulators, with the latter consistently winning the appeal. This was the case for Sony Computer Entertainment America

[1] See https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage
[2] See https://github.com/localForage/localForage

v. Connectix Corporation trial, that deemed that the "Virtual Game Station" PlayStation emulator was not a copyright infringement on Sony, and that Connectix's reverse engineering of the original PlayStation's BIOS was fair use [26]. Emmy was developed using online resources about the GB compiled by people who reverse engineered the GB, and as such was developed with a clean room design too.

Although distributing the emulator is legal, distributing the BIOS (Basic Input/Output System) of the console with the emulator is not, as it is still copyrighted software [25]. Because acquiring such BIOS is however not illegal as long as the user owns the console, a user is free to use their own BIOS in the emulator (see subsection Settings). The emulator otherwise simulates the effect of the BIOS on the system, by setting up all the necessary memory and registers.

Distributing copyrighted game ROMs is also not allowed – this is why the user must upload the ROM they want to use themselves. The software does not upload the ROM, and simply stores it in local storage.

7.3 Integrity

This piece of software was developed with integrity, always thinking critically on what was developed, and with no intention to harm others. This report has also been written with honesty, without attempting to withhold information on the resulting software. No acquired data was falsified or modified to fit a narrative.

The produced software is open source, and available at `https://github.com/N1ark/gbc-emulator`. This means it can be used by others to create similar emulators, or for educational purposes. Other users may also contribute to the project, or fix issues it has.

Overall, this software was developed with the British Computer Society's Code of Conduct & Code of Good Practice[3] in mind, and all of its relevant rules were followed.

[3] See `https://www.bcs.org/media/2211/bcs-code-of-conduct.pdf`

Chapter 8

Conclusion and Future Work

8.1 Conclusion

This project helped outline the typical structure of a an emulator, and in particular that of a Game Boy emulator. It describes how the console works in detail, what components it is made of, and how they interact with each other. It showcased optimisation methods for emulating different components of an emulator, like the CPU or the system bus. The resulting software is a fully playable Game Boy and Game Boy Color emulator, that has multiple quality of life features to make the experience more pleasant while also providing debugging tools for retro game and emulator developers. This emulator can be played on both computer and mobile devices, and is of good accuracy compared to other existing emulators.

8.2 Future Work

As it stands the emulator has three main properties that could be improved, each independently one of the other.

The first is improving the accuracy of the emulator. As seen in the evaluation of the project, its APU should be reworked. It is currently lacklustre, and represents a part of the system that could be improved without requiring a full re-write, as it is self contained. Other small improvements could be done to improve the accuracy of other components like the MBCs or the PPU, or to add currently unsupported MBCs and accessories. All of these improvements can be done separately and in small increments, and do not entail any breaking changes outside of the emulator's core.

A second important point to be improved in the emulator is its performance. As seen in its evaluation, an average maximum of a 200% speed increase is reachable. To improve the performance of the emulator (which will be needed if more features are to be added), its performance must be improved substantially. Although code optimisations and caching could improve some of the performance issues, a significant bottleneck is the language used itself. If the emulator were to be entirely rewritten using a language such as AssemblyScript, it would likely run much faster. This would be a much more time-consuming change, and would require updating the front-end to work properly with it, but could yield great performance improvements if done properly.

Finally, the third point to improve on this project is it's frontend. Although it has a good range of available features, it could still be improved to be up to par with other technical emulators. More and better debugging tools could be provided to the user, as well as more customisation options, such as new screen filters, a full-screen mode or support for input macros.

Acronyms

APU Audio Processing Unit.

CPU Central Processing Unit.

DMA Direct Memory Access.

DMG Dot Matrix Game.

GB Game Boy.

GBC Game Boy Color.

MBC Memory Bank Controller.

OAM Object Attribute Memory.

OS Operating System.

PPU Picture Processing Unit.

ROM Read-Only Memory.

VRAM Video RAM.

WRAM Work RAM.

Bibliography

[1] Merriam-Webster. “Emulator definition and meaning.” (Mar. 2022), [Online]. Available: https://www.merriam-webster.com/dictionary/emulator (visited on 03/02/2023).

[2] A. Kaluszka. “Computer emulation, history.” (Dec. 2001), [Online]. Available: https://kaluszka.com/vt/emulation/history.html (visited on 03/03/2023).

[3] IBM. “709/7090/7094/7094 II compatibility feature for IBM System/370 models 165, 165 II, and 168.” (1973), [Online]. Available: http://bitsavers.org/pdf/ibm/370/compatibility_feature/GA22-6955-1_709x_Compatibility_Feature_for_IBM-370_165_168.pdf (visited on 03/03/2023).

[4] S. Granger, “Emulation as a digital preservation strategy,” *D Lib Mag.*, vol. 6, no. 10, Oct. 2000. DOI: 10.1045/october2000-granger. [Online]. Available: https://doi.org/10.1045/october2000-granger (visited on 04/02/2023).

[5] L. Boley. “Emulation or virtualization: What’s the difference?” (Mar. 2014), [Online]. Available: https://www.dell.com/en-us/blog/emulation-or-virtualization-what-s-the-difference/ (visited on 04/02/2023).

[6] MyaMyaMya. “First Famicom/NES emulator? - Zophar’s Domain.” (Jan. 2009), [Online]. Available: https://www.zophar.net/forums/index.php?threads/first-famicom-nes-emulator.10169/ (visited on 03/07/2023).

[7] Emulation General Wiki. “History of emulation.” (Jan. 2023), [Online]. Available: https://emulation.gametechwiki.com/index.php/History_of_emulation#Game_Boy.2FColor (visited on 03/07/2023).

[8] “Gameboy development wiki.” (2023), [Online]. Available: https://gbdev.gg8.se/wiki/ (visited on 03/28/2023).

[9] “Pandocs.” (2023), [Online]. Available: https://gbdev.io/pandocs/ (visited on 03/07/2023).

[10] endrift. “"holy grail" bugs in emulation, part 1.” (May 2017), [Online]. Available: https://mgba.io/2017/05/29/holy-grail-bugs/ (visited on 04/03/2023).

[11] T. Phillips, “Game Boy and Game Boy Color titles headed to Nintendo Switch Online,” *Eurogamer*, Feb. 2023. [Online]. Available: https://www.eurogamer.net/game-boy-and-game-boy-color-titles-headed-to-nintendo-switch-online-1 (visited on 04/02/2023).

[12] M. J. Flynn, C. Neuhauser, and R. M. McClure, “EMMY: An emulation system for user microprogramming,” in *Proceedings of the May 19-22, 1975, National Computer Conference and Exposition*, ser. AFIPS ’75, Anaheim, California: Association for Computing Machinery, 1975, pp. 85–89, ISBN: 9781450379199. DOI: 10.1145/1499949.1499968. [Online]. Available: https://doi.org/10.1145/1499949.1499968.

[13] Pan of Anthrox, GABY, M. Fayzullin, *et al.* “Game Boy CPU manual.” (May 2008), [Online]. Available: http://marc.rawer.de/Gameboy/Docs/GBCPUman.pdf (visited on 03/22/2023).

[14] A. Vynogradenko. “Framework-sizes.md.” (2023), [Online]. Available: https://gist.github.com/Restuta/cda69e50a853aa64912d (visited on 04/03/2023).

[15] MDN Web Docs. “Autoplay guide for media and web audio APIs.” (Mar. 2023), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Media/Autoplay%5C_guide (visited on 03/23/2023).

[16] M. Korth. “GBATEK.” (2014), [Online]. Available: https://problemkaputt.de/gbatek.htm (visited on 03/26/2023).

[17] M. Sullivan. “Game boy CPU intructions.” (Oct. 2022), [Online]. Available: https://meganesu.github.io/generate-gb-opcodes/ (visited on 03/22/2023).

[18] A. Goldfuss. “Function dispatch tables in C.” (Mar. 2019), [Online]. Available: https://blog.alicegoldfuss.com/function-dispatch-tables/ (visited on 03/29/2023).

[19] Joonas “Gekkio” Javanainen. “Game Boy: Complete technical reference.” (Mar. 2023), [Online]. Available: https://gekkio.fi/files/gb-docs/gbctr.pdf (visited on 03/07/2023).

[20] R. Mongenel. “GameBoy memory map.” (2010), [Online]. Available: http://gameboy.mongenel.com/dmg/asmmemmap.html (visited on 03/27/2023).

[21] M. Steil. “The ultimate game boy talk (33c3).” (Dec. 2016), [Online]. Available: https://youtu.be/HyzD8pNlpwI?t=2460 (visited on 03/27/2023).

[22] Great Hierophant. “MBC5 backwards compatibility.” (Nov. 2009), [Online]. Available: https://forums.nesdev.org/viewtopic.php?p=52575#post_content52603 (visited on 03/29/2023).

[23] M. Achibet. “Game boy/game boy color roms.” (Jul. 2015), [Online]. Available: http://merwanachibet.net/gameboy-roms (visited on 03/29/2023).

[24] B. Velghe. “Is WebAssembly really faster than JavaScript? a hands-on experiment!” (May 2022), [Online]. Available: https://www.tmssoftware.com/site/blog.asp?post=933 (visited on 04/01/2023).

[25] J. Conley, E. Andros, P. Chinai, E. Lipkowitz, and D. Perez, “Use of a game over: Emulation and the video game industry, a white paper,” Northwestern Journal of Technology and Intellectual Property, Tech. Rep., 2004. [Online]. Available: https://scholarlycommons.law.northwestern.edu/njtip/vol2/iss2/3/ (visited on 03/31/2023).

[26] *Sony Computer Entertainment America, Inc. v. Connectix Corporation*, 203 F.3d 596, 2000. [Online]. Available: https://web.archive.org/web/20070228070634/http://www.ca9.uscourts.gov/ca9/newopinions.nsf/0/66b3a352ea33712988256952007578c2?OpenDocument (visited on 03/31/2023).