Nintendo Entertainment System (NES) emulator with educational purpose

Emulador de la consola Nintendo Entertainment System (NES) con fines educativos.

Luis Vijande González

Directed by:
José Manuel Redondo López

An End of Degree Project presented for the degree of
Software Engineering.

Universidad de Oviedo

Escuela de Ingeniería del Software
Universidad de Oviedo
24 January 2025

Declaration of originality

According to the ruling regarding End of Degree Projects from BOLETÍN OFICIAL

DEL PRINCIPADO DE ASTURIAS, núm. 62 de 30-III-2020 artículo 8-3, it's declared that this work is completely original, and all sources have been correctly cited.

This whole project and documentation have been developed by the author Luis Vijande González

It's also declared that the author is aware that the jury can use anti plagiarism software on both this document and the project.

28/01/2025

X

Luis Vijande González

Firmado por: VIJANDE GONZALEZ LUIS - 71827154V

Acknowledgment

To my parents Marta González and Guillermo Vijande, for their endless support and love.

To my tutor, José Manuel, for agreeing to direct this mess of a project.

To my friends Nicolás, Miguel, Pelayo, Hugo and Mario for begin amazing people and making my time in university a breeze.

To all the wonderful people I've met through this period of my life.

To the NES homebrew community, for their previous work on this subject, and their efforts to help understand the inner workings of the NES.

Thank you for the bottom of my heart.

Contents

Table Index

Figure Index

1. What is the Purpose of this Work?

1.1. Abstract

The goal of the present work is the implementation of a system capable of basic emulation of the Nintendo Entertainment System (NES in the future) such that if a user does not know how the console works, they can learn something about it, be it what instructions the CPU is executing, what are the current contents in memory or the status of the pixel processing unit. Another goal of this project is to help users understand how the insides of a complex machine like the NES can work with the objective of teaching low level concepts such as CPU registers or the instruction set of the 6502.

The user will be able to run their ROMs through the emulator if they remain within the mapper 0 architecture. Then, the system will show the CPU, PPU and memory status, and will allow the user to control the advancement of the emulation, whether by running it per instruction, per frame or per pixel.

Must be noted that the main objective of this work is to teach what it takes to make an emulator, and how do they work, any real system accuracy is a secondary result of following correct and accurate guides on real hardware.

1.2. Keywords

Emulation, Console, Videogames, Graphics, Education, Retro, programming.

1.3. Resumen

El objetivo de este trabajo es la implementación de un sistema capaz de emulación básica de la consola Nintendo Entertainment System (NES en adelante) de tal forma que si el usuario no entiende cómo funciona la susodicha, pueda aprender algo sobre ella, ya sea las instrucciones que el CPU esté ejecutando, los contenidos actuales de memoria y sus cambios o el estado de la unidad de proceso de pixeles. Otro objetivo de este proyecto es ayudar a usuario a entender como una maquina relativamente compleja como la NES puede funcionar con el resultado de enseñar conceptos de nivel bajo como CPU registros de CPU o el set de instrucciones del 6502.

El usuario será capaz de ejecutar sus propias ROMs a través del emulador siempre y cuando pertenezcan a la arquitectura del mapeador 0. Después, el sistema mostrará el estado del CPU, PPU y la memoria; el sistema también permitirá al usuario controlar el avance de la emulación, ya sea avanzando por instrucción, por fotograma o por píxel.

Debe ser señalado que el objetivo principal de este trabajo es enseñar como se hace y cómo funciona un emulador, y que cualquier símil entre este trabajo y el Sistema real se debe a seguir guías correctas sobre el hardware del sistema.

1.4. Palabras clave

Emulación, Consola, Videojuegos, Gráficos, Educación, Retro, Programación

2. Information System Plan (ISP)

2.1. ISP 1: Start of the ISP

The objective of the present work is to create a system capable of basic NES emulation, that is also capable of teaching the inner workings of the named console. Taking the above into account, it has been decided that implementing a desktop application that won't be replacing more powerful NES emulators such as MESEN, but one that can be used alongside them. An emulator is a piece of software or hardware made to mimic that of a different software or hardware such that the host system can run programs made for the target system, in our case, the host will be the emulator (EMOO in the future), and the target will be the NES. Emulators can be used as a preservation tool, or to use the software of lost hardware with better accessibility. Must be noted that emulators are completely legal and not considered piracy. (1)

The present work will not strive to be the most accurate emulator there is, as discussed further, whether it is by time constraints or technical difficulties.

This work is motivated by the my desire to understand low level systems and the possibility to emulate simple systems.

2.1.1. ISP 1.1: Analysing why the ISP is needed

As mentioned in the above excerpt, the project to be developed is similar to a common emulator. The main requirements of the system are:

- Being able to run Mapper 0 NTSC ROMs.
- Design a user interface capable of showing the guts and inner workings of the NES.
- The emulator won't have an audio system.
- During the emulation, the user will be able to stop and continue the emulation as they please, they will also be able to control the advancement of the emulation, that means running it per CPU cycle, per CPU instruction, per PPU cycle, per PPU pixel or per PPU scanline.
- The user should be able to change ROM at runtime.
- The user should be able to control the emulated game with a common controller such as an Xbox or play station controller.

2.2. ISP 2: Definition and Organization of the ISP

2.2.1. ISP 2.1: Scope and Context Specification

To better tackle this project, it will be divided into three distinct parts, a rendering backend, the emulator, and an interface.

2.2.1.1. Renderer

The renderer will oversee putting the pixel data provided by the emulator in our screen, it will be an extremely simple renderer that will put the pixel data into a modifiable texture.

The renderer will also provide a backend for the interface system.

2.2.1.2. Emulator

The emulator will be the 'brain' of the system, it will be the component that loads the ROM, and executes it, providing the renderer and the interface the necessary data to function. However, the emulator will not run by itself, as the frontend or the application will be the one to run it, as it will need to be told when to run a frame, or a pixel, or an instruction…

The emulator will also come with a disassembler that will be used by the interface.

2.2.1.3. Interface

The frontend will be the 'body' of the application, the user will load a ROM to be sent to the emulator through the interface, and the interface will make the emulator run per frame until the user decides to either stop it, or to change ROM. The interface will allow the user to show additional information such as internal status of the CPU, PPU or a memory visualizer.

The system will also be implemented in stages or 'sprints', the stages will be as follow. First, the system will be a working emulator of NES's processor, the Ricoh RP2A03, a chip based in the extremely popular 6502; after the processor is fully functional and tested, the second stage will be a semi working PPU and renderer, this semi working PPU will only render the backgrounds of the games; once the backgrounds are properly rendered and the test ROM passes, the second to last stage will be rendering sprites, and the last stage will be the interface and user controls.

2.3. ISP 3: Study of Relevant Information

2.3.1. ISP 3.1: Background Selection and Analysis

To correctly design and develop the system, some guides have been followed. These guides are:

- **Emulator source code**. The source code of other emulators like the MESEN source code, which might be the premier emulator of the NES, other source code studied includes olcNES or mos6502 which is a 6502 emulator.
- **The NES Development Wiki.** The NesDev Wiki is an incredibly powerful resource, it includes a reference guide for the NES, a programming guide for NES and a lot of useful information.
- **Other Guides.** Like YouTube videos explaining the NES rendering process by NesHacker (2) or the emulator series by OneLoneCoder (3).

3. ISP 7: Definition of the Technological Architecture

3.1. ISP 7.1: Identifying Technological Infrastructure Needs

As we get closer to defining the structure of the project, some technologies must be studied to make the development as smooth as possible. Up next some pondered technologies will be discussed with pros and cons. First, this will be developed with C++20, this is because I'm accustomed to C++, and like some of the more modern features.

3.1.1. Build Tool

Sadly, C++ is not like other programming languages, that come with a nice build tool that works of the box for you, so a build tool must be chosen.

- **Cmake**: Cmake is the de-facto standard for building C++. Rather than saying that it is a build tool, it's a build tool for the build tool, since it can generate a gnu make or visual studio solution for you to it with them. It has a scripting language that can be incredibly cumbersome to use (4).
- **PreMake**: PreMake is an alternative to cmake made in Lua, it works like cmake, but has very nice documentation, and the all-powerful Lua as a scripting language. Sadly, it's not feature complete (5).

3.1.2. Tests in the Emulator

The emulator will need to be tested, as it's the most hard and important part of the system; and in the C++ world choosing a testing tool is not as easy as it looks, as there are very powerful options.

- **GoogleTest**. Gtest is a very popular testing tool, it includes testing a mocking functionality and integrates nicely with Visual Studio, also has very nice libraries. On the other hand, is a very bulky library (6).
- **BoostTest**. A very popular C++ library that integrates with other Boost libraries. I do not like developing with Boost (7).
- **Catch2.** A very nice tool that also includes benchmarking. Not as widely adapted as the other two (8).

3.1.3. Windowing

There are a lot of good windowing tools for C++, but we will mainly look at SDL and GLFW.

- **SDL**. Simple DirectMedia Layer, a cross-platform development library that provides access to audio, controller, keyboard, mouse and graphics hardware, can be used with OpenGL, Vulkan, or Direct3D; also has its own rendering library. It's written in C, so developing for it can be annoying. In my own experience, its documentation can be lacking. SDL is mainly a monolith, this means that even if you only want the windowing system and the controller subsystem, you will also get the rendering subsystem and audio subsystems (9).
- **GLFW**. An open source, multi-platform library mainly designed for OpenGL but can also be used with other rendering APIs like Vulkan. It's also written in C so it can be cumbersome to use, has support for controller access and very good documentation since it's been widely used in the world of game engines and application development (10).

3.1.4. Renderer

To present the emulator and interface to screen, a way to access graphics hardware is needed. In the multi-platform world there are two main rendering APIs, OpenGL and Vulkan, but there are also other libraries layered on top of them, like SDL.

- **Vulkan**. Vulkan is a cross-platform rendering specification developed by Khronos Group, it's very low level and can be very powerful when used right, thanks to the level of control it offers, and the possible performance gain, it has taken the shape of a industry standard, sadly, that level of control comes with a price, in this case, it's a very steep learning curve, and some difficult concepts to grasp, like synchronization and memory management; it can be very verbose, taking as much as 1200 lines of code to display a triangle, but there are add-ons like C++ language bindings in the form of vulkan-hpp or ways to bootstrap Vulkan and device instantiation in the form of vkbootstrap that make this problem take a relative back seat. It also has extensive documentation and very helpful guides in the shape of vkguide or vulkan-tutorial. Also, it has a lot of community made plugins that make things easier, like completely removing render passes with the dynamic rendering layer, or making texture slots handling a non-issue with bindless resources (11) (12) (13).

- **OpenGL**. OpenGL is another rendering specification originally developed by Silicon Graphics Inc. but now maintained by the Khronos group, it's higher level than Vulkan and easier to use, especially in tandem with GLFW. It has been widely acclaimed as the industry's foundation ever since its conception. Its documentation is widely available and very good, and it has been used by every single developer that dabbles in graphics. Modern OpenGL has made some very good quality of life changes to its API that make development easier (14).
- **SDLRenderer**. SDLRenderer comes with SDL out of the box, which makes it a good choice if you are using SDL as a windowing system already, as it's not a graphics specification like OpenGL or Vulkan, it comes with far less control. It also comes with the downsides of SDL, lacking documentation and monolith design (9).

3.1.5. Interface

To develop the interface some libraries have been selected to be discussed.

- **Dear ImGui**. Dear Imgui is an immediate mode bloat-free graphical user interface library for C++, it can be slotted very easily in a very wide net of window-renderer configurations that makes it using it in 2D and 3D applications a breeze. It's developed in C++ so no need to interface with it using C. It follows the immediate mode UI paradigm, which means that it has no state (15). It has two main branches, docking and main, the docking branch provides window docking and window viewports, allowing easy user customization of the interface. It's currently in development which can make appear some extremely tough bugs, and the immediate mode paradigm can be hard to grasp to developers unfamiliar with it (16).
- **Qt**. Qt is a cross-platform software development framework, it makes developing beautiful applications easier, but has a very weird license thing going on which makes it unappealing (17) (18).

Other Ui libraries and oddities considered are WxWidgets, but frankly it looks awful, and doing the interface with WPF in .NET and type marshalling it to the rest of the application.

3.1.6. Input Handling

As one of the requirements of this project is being able to hand a modern games controller, a robust and easy to use input handling technology is needed. Some of the options that were considered are:

- **XInput.** A cross-platform API that enables Windows applications to interact with modern controllers such as Xbox or third-party ones (19).
- **SDLController.** As stated before, SDLController is part of SDL, that comes with all the good and the not so good of SDL (9).

3.1.7. File managing and Serialization

I did not find a lot of options to manage different files, but I did find that boost has a serialization component.

- **Boost.** Part of the boost libraries, I do not like boost.
- **FMan.** A library of my own design (not published anywhere, as it was made to my own needs for different projects), based on the architecture of ImGui, lacking in features and exists as a wrapper to C++ fstream.

3.2. ISP 7.2: Selection of Technological Architecture

3.2.1. Build Tool

Since I have been using PreMake for all my other projects, I believe that changing to Cmake will be a liability, so the build tool will be PreMake with some python scripting here and there to provide the uncompleted features needed.

3.2.2. Tests in the Emulator

Taking all the above into account, it was decided to use GoogleTest since it has very good documentation, and I have used it before.

3.2.3. Windowing

SDL was chosen since I have already used it in university so I was familiar with it.

3.2.4. Renderer

Taking the above and some things that will be discussed in the interface section, I have chosen Vulkan, that may look like a brave choice taking into account the little graphics experience I have, but, since Vulkan is the successor of OpenGL and considered the industry standard, has excellent documentation and guides, and an incredibly powerful debug layer, I believe that this will be a great learning experience.

3.2.5. Interface

After careful consideration, Dear Imgui on the docking branch was chosen, it was chosen since it could be used with a wide net of render-window configurations, and since the idea of immediate mode was more attractive to the more classic interface design; choosing Imgui meant using Vulkan an even more enticing offer, since the Imgui implementation for it already existed, so the performance impact of rendering the interface would be nil.

3.2.6. Input Handling

I chose SDL since I had already used it before.

3.2.7. File managing and Serialization

As stated before, I do not like boost, so I decided to use my own previously developed library.

3.2.8. Other Libraries

This project makes use of other libraries in less important parts of the system, these libraries are:

- **Compile Time Regular Expressions**. Used in the toy assembler made to write tests for the cpu. Everyone that has used C++'s native regex library understands why I chose to not use it.
- **Portable File Dialogs**. Used to have a native window when loading ROMs.
- **Freetype**. A very bloated font rendering system used to replace Imguis default one.
- **GLM**. OpenGL Math, a graphics Math library, can be used with Vulkan after changing some configurations.
- **cppicons**. A set of font icon bindings to type icons easier.
- **VMA**: Vulkan memory allocator, to make allocate memory in Vulkan easier.
- **vkBootstrap**: Helps initialise Vulkan structures

4. TFG Planning and Management

4.1. Project Planning

4.1.1. Identification of Stakeholders

Identified stakeholders are:

- Users interested in emulation.
- Users interested in the NES.
- Users that want to learn about the internals of the NES.
- The author of the project.
- The director of the project.

4.1.2. Organization

This planning was created taking into consideration the following. The team behind the development of this work is made of a single developer that works 8hrs from Monday to Friday, the standard calendar.

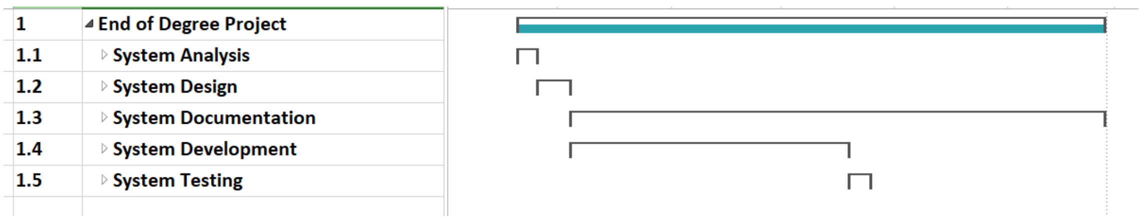
This project starts the 20th of September 2023 and ends the 27th of February 2024.

This project is defined in such a way that the developer will work part time in documentation and part time in development after the system analysis and design are done.

4.1.3. Initial Planning. WBS

WBS	Task Name	Work	Duration	Start	Finish
1	End of Degree Project	729 hrs	114.38 days	Wed 20/09/23	Tue 27/02/24
1.1	System Analysis	25 hrs	3.38 days	Wed 20/09/23	Mon 25/09/23
1.2	System Design	56 hrs	7 days	Mon 25/09/23	Wed 04/10/23
1.3	System Documentation	416 hrs	104 days	Wed 04/10/23	Tue 27/02/24
1.4	System Development	216 hrs	54 days	Wed 04/10/23	Tue 19/12/23
1.5	System Testing	16 hrs	4 days	Tue 19/12/23	Mon 25/12/23

Table 1. Initial Planning



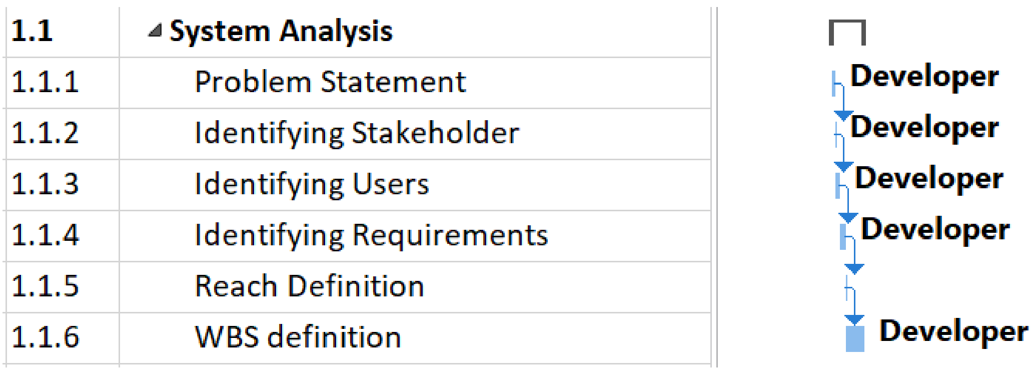
1	End of Degree Project
1.1	System Analysis
1.2	System Design
1.3	System Documentation
1.4	System Development
1.5	System Testing

Figure 1. Initial Planning Gantt Diagram

4.1.3.1. System Analysis

WBS	Task Name	Work	Duration	Start	Finish
1.1	System Analysis	25 hrs	3.38 days	Wed 20/09/23	Mon 25/09/23
1.1.1	Problem Statement	4 hrs	4 hrs	Wed 20/09/23	Wed 20/09/23
1.1.2	Identifying Stakeholder	3 hrs	3 hrs	Wed 20/09/23	Wed 20/09/23
1.1.3	Identifying Users	2 hrs	2 hrs	Wed 20/09/23	Thu 21/09/23
1.1.4	Identifying Requirements	8 hrs	1 day	Thu 21/09/23	Fri 22/09/23
1.1.5	Scope Definition	0 hrs	2 hrs	Fri 22/09/23	Fri 22/09/23
1.1.6	WBS definition	8 hrs	1 day	Fri 22/09/23	Mon 25/09/23

Table 2. System Analysis



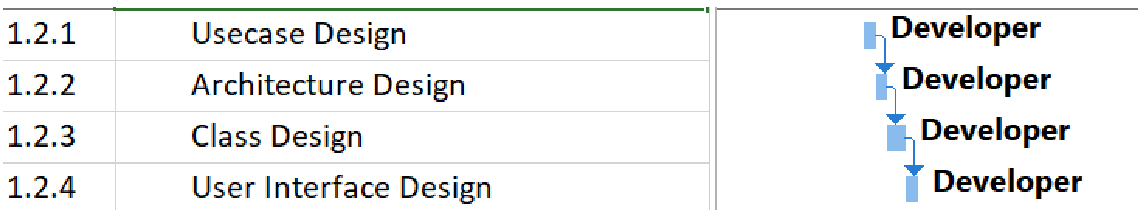
1.1	System Analysis
1.1.1	Problem Statement
1.1.2	Identifying Stakeholder
1.1.3	Identifying Users
1.1.4	Identifying Requirements
1.1.5	Reach Definition
1.1.6	WBS definition

Figure 2. System Analysis Gantt Diagram

4.1.3.2. System Design

WBS	Task Name	Work	Duration	Start	Finish
1.2	System Design	56 hrs	7 days	Mon 25/09/23	Wed 04/10/23
1.2.1	Usecase Design	16 hrs	16 hrs	Mon 25/09/23	Wed 27/09/23
1.2.2	Architecture Design	16 hrs	2 days	Wed 27/09/23	Fri 29/09/23
1.2.3	Class Design	8 hrs	1 day	Fri 29/09/23	Mon 02/10/23
1.2.4	User Interface Design	16 hrs	2 days	Mon 02/10/23	Wed 04/10/23

Table 3. System Design



1.2.1	Usecase Design
1.2.2	Architecture Design
1.2.3	Class Design
1.2.4	User Interface Design

Figure 3. System Design Gantt Diagram

4.1.3.3. System Documentation

System documentation is the longest part of the project, this is in thanks to the study section, that includes investigating other emulators, and concepts related to this work.

WBS	*Task Name*	*Work*	*Duration*	*Start*	*Finish*
1.3	***System Documentation***	***416 hrs***	***104 days***	***Wed 04/10/23***	***Tue 27/02/24***
1.3.8	*System Study*	*100 hrs*	*25 days*	*Wed 04/10/23*	*Wed 08/11/23*
1.3.1	*Information System Plan*	*32 hrs*	*8 days*	*Wed 08/11/23*	*Mon 20/11/23*
1.3.2	*System Feasibility Study*	*32 hrs*	*8 days*	*Mon 20/11/23*	*Thu 30/11/23*
1.3.3	*Analysis of the Information System*	*52 hrs*	*13 days*	*Thu 30/11/23*	*Tue 19/12/23*
1.3.4	*Information System Design*	*80 hrs*	*20 days*	*Tue 19/12/23*	*Tue 16/01/24*
1.3.5	***Manuals***	***40 hrs***	***10 days***	***Tue 16/01/24***	***Tue 30/01/24***
1.3.5.1	*User Manuals*	*40 hrs*	*10 days*	*Tue 16/01/24*	*Tue 30/01/24*
1.3.6	*Conclusions*	*40 hrs*	*10 days*	*Tue 30/01/24*	*Tue 13/02/24*
1.3.7	*Annexes*	*40 hrs*	*10 days*	*Tue 13/02/24*	*Tue 27/02/24*

Table 4. System Documentation

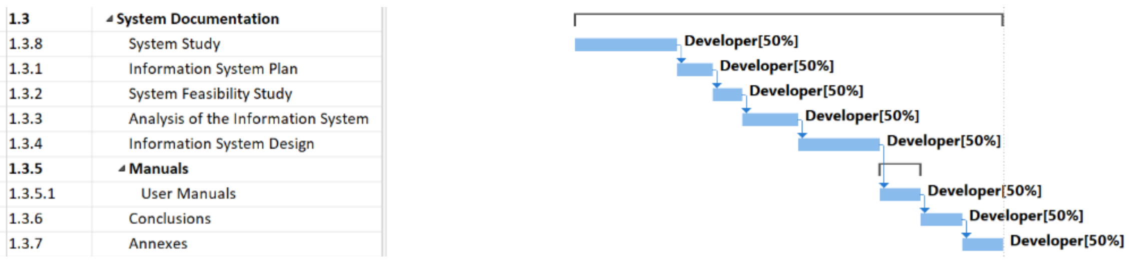
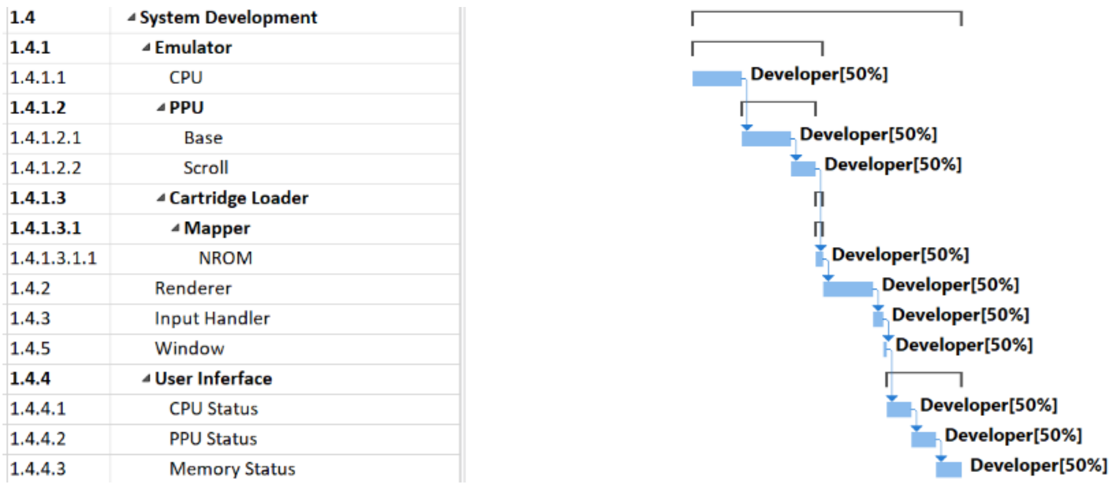


Figure 4. System Documentation Gantt Diagram

4.1.3.4. System Development

WBS	Task Name	Work	Duration	Start	Finish
1.4	**System Development**	**216 hrs**	**54 days**	**Wed 04/10/23**	**Tue 19/12/23**
1.4.1	**Emulator**	**108 hrs**	**27 days**	**Wed 04/10/23**	**Fri 10/11/23**
1.4.1.1	CPU	40 hrs	10 days	Wed 04/10/23	Wed 18/10/23
1.4.1.2	**PPU**	**60 hrs**	**15 days**	**Wed 18/10/23**	**Wed 08/11/23**
1.4.1.2.1	Base	40 hrs	10 days	Wed 18/10/23	Wed 01/11/23
1.4.1.2.2	Scroll	20 hrs	5 days	Wed 01/11/23	Wed 08/11/23
1.4.1.3	**Cartridge Loader**	**8 hrs**	**2 days**	**Wed 08/11/23**	**Fri 10/11/23**
1.4.1.3.1	**Mapper**	**8 hrs**	**2 days**	**Wed 08/11/23**	**Fri 10/11/23**
1.4.1.3.1.1	NROM	8 hrs	2 days	Wed 08/11/23	Fri 10/11/23
1.4.2	Renderer	40 hrs	10 days	Fri 10/11/23	Fri 24/11/23
1.4.3	Input Handler	4 hrs	1 day	Fri 24/11/23	Mon 27/11/23
1.4.5	Window	4 hrs	1 day	Mon 27/11/23	Tue 28/11/23
1.4.4	**User Inferface**	**60 hrs**	**15 days**	**Tue 28/11/23**	**Tue 19/12/23**
1.4.4.1	CPU Status	20 hrs	5 days	Tue 28/11/23	Tue 05/12/23
1.4.4.2	PPU Status	20 hrs	5 days	Tue 05/12/23	Tue 12/12/23
1.4.4.3	Memory Status	20 hrs	5 days	Tue 12/12/23	Tue 19/12/23

Table 5. System Development



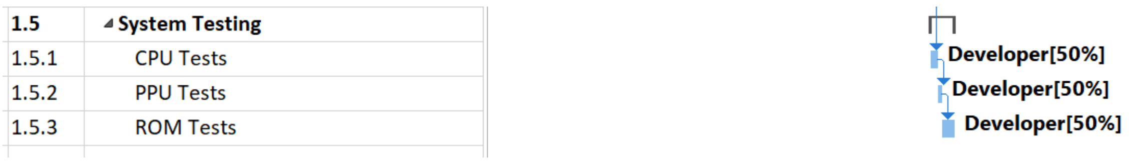
1.4	**System Development**
1.4.1	**Emulator**
1.4.1.1	CPU
1.4.1.2	**PPU**
1.4.1.2.1	Base
1.4.1.2.2	Scroll
1.4.1.3	**Cartridge Loader**
1.4.1.3.1	**Mapper**
1.4.1.3.1.1	NROM
1.4.2	Renderer
1.4.3	Input Handler
1.4.5	Window
1.4.4	**User Inerface**
1.4.4.1	CPU Status
1.4.4.2	PPU Status
1.4.4.3	Memory Status

Figure 5. System Development Gantt Diagram

4.1.3.5. System Testing

WBS	Task Name	Work	Duration	Start	Finish
1.5	System Testing	16 hrs	4 days	Tue 19/12/23	Mon 25/12/23
1.5.1	CPU Tests	8 hrs	2 days	Tue 19/12/23	Thu 21/12/23
1.5.2	PPU Tests	4 hrs	1 day	Thu 21/12/23	Fri 22/12/23
1.5.3	ROM Tests	4 hrs	1 day	Fri 22/12/23	Mon 25/12/23

Table 6. System Testing



1.5	**System Testing**
1.5.1	CPU Tests
1.5.2	PPU Tests
1.5.3	ROM Tests

Figure 6. System Testing Gantt Diagram

The file manager and serialization library does not appear in planification since it was developed long before this project.

4.1.4. Risks

In this section a brief risk assessment can be found, for a more detailed breakdown, go to 9.1.

The following table contains the identified risks.

ID	Risk	Category	Description
1	Lack of experience	Organizational	The author has little experience in the field of emulation, which can led to time underestimation since he has no time frame in which an emulator can be finished
2	Over scoping	Organizational	As the author does not know what is needed for the emulator to be considered finished, some unnecessary features can be added, which will led to more time
3	Internship	Managerial	At the time of developing this project, the author is currently looking for an internship, which can led to work stoppage.
4	Nintendo	External	Even though emulators are legal, Nintendo has been known to go after emulators with shady legal practices to make them stop development
5	Incorrect estimation	Organizational	Some sections of the project can be wrongly estimated since the author is not the best at project planning and management
6	External libraries	Technical	This project makes use of external third party libraries, of which the author has no control, these libraries can have bugs, or have incorrect documentation.
7	Hard concepts	Technical	Low Level Emulators are made emulating the hardware of consoles, in order to do that, it is crucial to have a correct and accurate breakdown of the hardware, if some part of the hardware is not correctly documented, the author will have to look for that information elsewhere.

Table 7. Risk identification

4.1.5. Initial Budget

The following section consists in a breakdown of the budget allocated for this project. This budget is calculated using the previous planning defined in 4.1.3. and using the team organization previously defined.

This developer will have a salary consisting of 18.75 Euros, since a C++ developer earns 39k a year in a 40 hrs week (according to Glassdoor (20)).

The Budget will be broken down in different budget items, this budget items are made up of the main sections of the WBS.

To get the total budget of this project refer to **¡Error! No se encuentra el origen de la referencia.** for the direct costs and 4.1.5.7 for the indirect costs, for the client budget, refer to 4.1.5.8

4.1.5.1. Budget Item 1. System Analysis

System Analysis					
I1	**Description**	**Amount**	**Units**	**Price**	**Total**
1	Problem Statement	4	hrs	18.75	75
2	Identifying Stakeholders	3	hrs	18.75	56.25
3	Identifying Users	2	hrs	18.75	37.5
4	Identifying Requirements	8	hrs	18.75	150
5	Scope Definition	2	hrs	18.75	37.5
6	WBS Definition	8	hrs	18.75	150

TOTAL SUM	**506.25**

Table 8. System Analysis Budget Item

4.1.5.2. Budget Item 2. System Design

System Design					
I1	**Description**	**Amount**	**Units**	**Price**	**Total**
1	Use case Design	16	hrs	18.75	300
2	Architecture Design	16	hrs	18.75	300
3	Class Design	8	hrs	18.75	150
4	User Interface Design	16	hrs	18.75	300

TOTAL SUM	**1050**

Table 9. System Design Budget Item

4.1.5.3. Budget Item 3. System Documentation

System Documentation							
I1	**I2**	**Description**	**Amount**	**Units**	**Price**	**Subtotal(2)**	**Total**
1		Study	100	hrs	18.75		1875
2		Information System Plan	32	hrs	18.75		600
3		System Feasibility Study	32	hrs	18.75		600
4		Analysis of the Information System	52	hrs	18.75		975
5		Information System Design	80	hrs	18.75		1500
6		Manuals	40	hrs	18.75		750
	1	User Manuals	40	hrs	18.75	750	
7		Conclusions	40	hrs	18.75		750
8		Annexes	40	hrs	18.75		750

TOTAL SUM	**7800**

Table 10. System Documentation Budget Item

4.1.5.4. Budget Line 4. System Development

System Development											
I1	I2	I3	I4	Description	Amount	Units	Price	Subtotal (4)	Subtotal (3)	Subtotal (2)	Total
1				Emulator	108	hrs	18.75				2025
	1			CPU	40	hrs	18.75			750	
	2			PPU	60	hrs	18.75			1125	
		1		Base	40	hrs	18.75		750		
		2		Scroll	20	hrs	18.75		375		
	3			Cartridge Loader	8	hrs	18.75			150	
		1		Mappers	8	hrs	18.75		150		
			1	NROM	8	hrs	18.75	150			
2				Renderer	40	hrs	18.75				750
3				Input Handler	4	hrs	18.75				75
4				Window	4	hrs	18.75				75
5				User Interface	60	hrs	18.75				1125
	1			PPU Status	20	hrs	18.75			375	
	2			PPU Status	20	hrs	18.75			375	
	3			Memory View	20	hrs	18.75			375	

TOTAL SUM	**4050**

Table 11. System Development Budget Item

4.1.5.5. Budget Line 5. System Testing

I1	Description	Amount	Units	Price	Total
1	CPU Tests	8	hrs	18.75	150
2	PPU Tests	4	hrs	18.75	75
3	ROM Tests	4	hrs	18.75	75

TOTAL SUM	**300**

Table 12. System Testing Budget Line

4.1.5.6. Direct Costs

Direct Costs					
I1	Description	Amount	Units	Price	Total
1	System Analysis	25	hrs	18.75	468.75
2	System Design	56	hrs	18.75	1050
3	System Documentation	416	hrs	18.75	7800
4	System Development	216	hrs	18.75	4050
5	System Testing	16	hrs	18.75	300

TOTAL SUM	**13668.75**

Table 13. Initial Direct Costs

4.1.5.7. *Indirect Costs*

Electricity cost has been calculated taking the kWh cost from 9 am to 17 pm (21) and averaging them, then multiplying it by the average consumption of a personal computer (22).

Internet cost has been calculated by taking Movistar's monthly subscription (23) and multiplying it by the project length (around 5 months)

Indirect Costs						
I1	**Description**	**Amount**	**Units**	**Price**	**Total**	**Type**
1	Microsoft 365	1	1 yr license	69.99	69.99	Rent
2	Computer	1	computer	1500	1500.00	Amortization
3	Electricity	320	kWh	0.141	45.12	Rent
4	Internet	1	Subscription	35	186.67	Rent

TOTAL SUM	**1801.78**

Table 14. Initial Indirect Costs

4.1.5.8. *Client Budget*

To calculate the client budget, indirect costs have been added to the direct costs and multiplied by a 25% benefit increase.

Client Budget		
I1	**Description**	**Price**
1	System analysis	716.23
2	System Design	1485.51
3	System Documentation	11035.22
4	System development	5729.83
5	System Testing	424.43

TOTAL SUM	**19391.22**

Table 15. Initial Client Budget

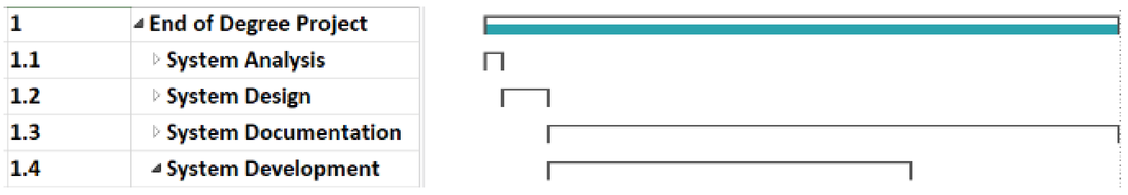
4.2. Closing the Project

Aside from some small changes to some sections, there are two main differences between the final planning and the initial planning, the first change is a work stoppage from 2024/02/08 to 2024/03/15 since the author was in an internship; the second main change is in system development, since the author had a flawed understating of NES's PPU, which lead to a restructuring of that section.

4.2.1. Final Planning

WBS	*Task Name*	*Work*	*Duration*	*Start*	*Finish*
1	***End of Degree Project***	***857 hrs***	***139.38 days***	***Wed 20/09/23***	***Tue 02/04/24***
1.1	***System Analysis***	***25 hrs***	***3.38 days***	***Wed 20/09/23***	***Mon 25/09/23***
1.2	***System Design***	***80 hrs***	***10 days***	***Mon 25/09/23***	***Mon 09/10/23***
1.3	***System Documentation***	***416 hrs***	***126 days***	***Mon 09/10/23***	***Tue 02/04/24***
1.4	***System Development***	***320 hrs***	***80 days***	***Mon 09/10/23***	***Mon 29/01/24***
1.5	***System Testing***	***16 hrs***	***4 days***	***Mon 29/01/24***	***Fri 02/02/24***

Table 16. Final Planning



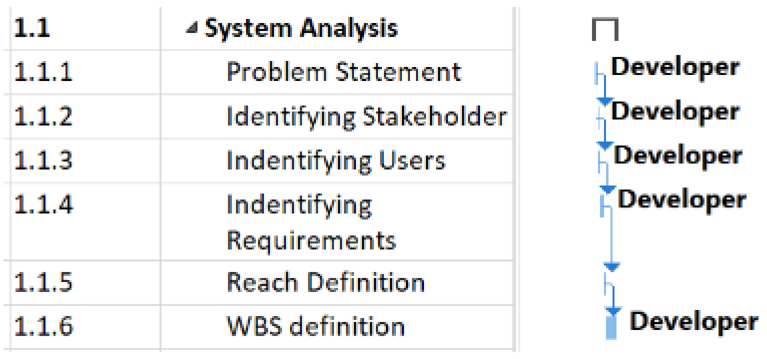
1	◢ End of Degree Project
1.1	▷ System Analysis
1.2	▷ System Design
1.3	▷ System Documentation
1.4	◢ System Development

Figure 7. Final Planning Gantt Diagram

4.2.1.1. System Analysis

WBS	*Task Name*	*Work*	*Duration*	*Start*	*Finish*
1.1	***System Analysis***	***25 hrs***	***3.38 days***	***Wed 20/09/23***	***Mon 25/09/23***
1.1.1	*Problem Statement*	*4 hrs*	*4 hrs*	*Wed 20/09/23*	*Wed 20/09/23*
1.1.2	*Identifying Stakeholder*	*3 hrs*	*3 hrs*	*Wed 20/09/23*	*Wed 20/09/23*
1.1.3	*Indentifying Users*	*2 hrs*	*2 hrs*	*Wed 20/09/23*	*Thu 21/09/23*
1.1.4	*Indentifying Requirements*	*8 hrs*	*1 day*	*Thu 21/09/23*	*Fri 22/09/23*
1.1.5	*Scope Definition*	*0 hrs*	*2 hrs*	*Fri 22/09/23*	*Fri 22/09/23*
1.1.6	*WBS definition*	*8 hrs*	*1 day*	*Fri 22/09/23*	*Mon 25/09/23*

Table 17. Final System Analysis



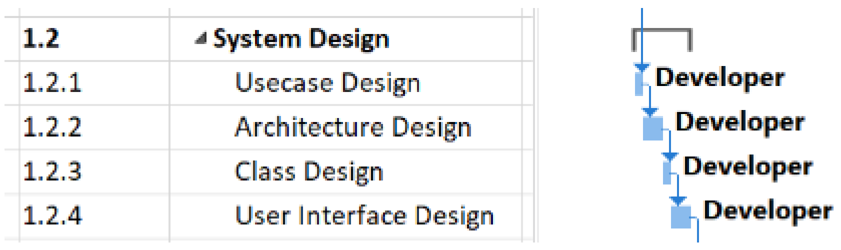
1.1	◢ System Analysis	
1.1.1	Problem Statement	Developer
1.1.2	Identifying Stakeholder	Developer
1.1.3	Indentifying Users	Developer
1.1.4	Indentifying Requirements	Developer
1.1.5	Reach Definition	
1.1.6	WBS definition	Developer

Figure 8. Final System Analysis Gantt Diagram

4.2.1.2. System Design

WBS	*Task Name*	*Work*	*Duration*	*Start*	*Finish*
1.2	***System Design***	***80 hrs***	***10 days***	***Mon 25/09/23***	***Mon 09/10/23***
1.2.1	*Usecase Design*	*16 hrs*	*16 hrs*	*Mon 25/09/23*	*Wed 27/09/23*
1.2.2	*Architecture Design*	*24 hrs*	*3 days*	*Wed 27/09/23*	*Mon 02/10/23*
1.2.3	*Class Design*	*16 hrs*	*2 days*	*Mon 02/10/23*	*Wed 04/10/23*
1.2.4	*User Interface Design*	*24 hrs*	*3 days*	*Wed 04/10/23*	*Mon 09/10/23*

Table 18. Final System Design



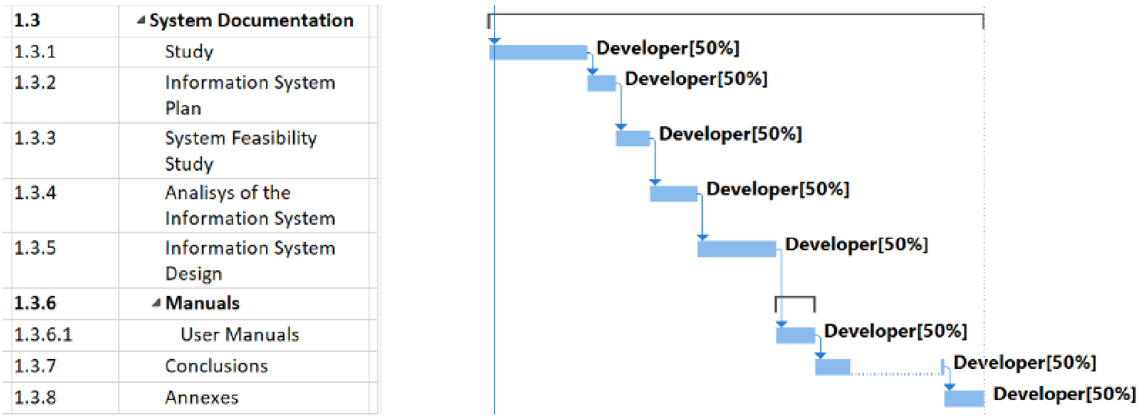
1.2	**System Design**
1.2.1	Usecase Design
1.2.2	Architecture Design
1.2.3	Class Design
1.2.4	User Interface Design

Figure 9. Final System Design Gantt Diagram

4.2.1.3. System Documentation

WBS	*Task Name*	*Work*	*Duration*	*Start*	*Finish*
1.3	***System Documentation***	***416 hrs***	***126 days***	***Mon 09/10/23***	***Tue 02/04/24***
1.3.1	*Study*	*100 hrs*	*25 days*	*Mon 09/10/23*	*Mon 13/11/23*
1.3.2	*Information System Plan*	*32 hrs*	*8 days*	*Mon 13/11/23*	*Thu 23/11/23*
1.3.3	*System Feasibility Study*	*32 hrs*	*8 days*	*Thu 23/11/23*	*Tue 05/12/23*
1.3.4	*Analisys of the Information System*	*52 hrs*	*13 days*	*Tue 05/12/23*	*Fri 22/12/23*
1.3.5	*Information System Design*	*80 hrs*	*20 days*	*Fri 22/12/23*	*Fri 19/01/24*
1.3.6	***Manuals***	***40 hrs***	***10 days***	***Fri 19/01/24***	***Fri 02/02/24***
1.3.6.1	*User Manuals*	*40 hrs*	*10 days*	*Fri 19/01/24*	*Fri 02/02/24*
1.3.7	*Conclusions*	*40 hrs*	*10 days*	*Fri 02/02/24*	*Tue 19/03/24*
1.3.8	*Annexes*	*40 hrs*	*10 days*	*Tue 19/03/24*	*Tue 02/04/24*

Table 19. Final System Documentation



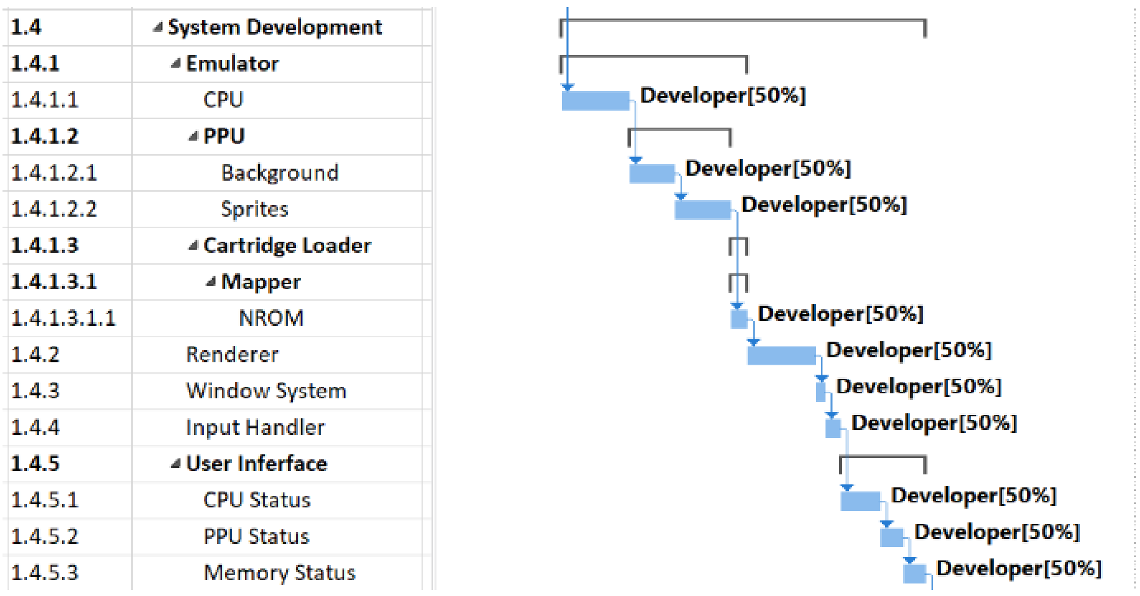
1.3	System Documentation
1.3.1	Study
1.3.2	Information System Plan
1.3.3	System Feasibility Study
1.3.4	Analisys of the Information System
1.3.5	Information System Design
1.3.6	Manuals
1.3.6.1	User Manuals
1.3.7	Conclusions
1.3.8	Annexes

Figure 10. Final System Documentation Gantt Diagram

4.2.1.4. System Development

WBS	*Task Name*	*Work*	*Duration*	*Start*	*Finish*
1.4	***System Development***	***320 hrs***	***80 days***	***Mon 09/10/23***	***Mon 29/01/24***
1.4.1	***Emulator***	***164 hrs***	***41 days***	***Mon 09/10/23***	***Tue 05/12/23***
1.4.1.1	*CPU*	*60 hrs*	*15 days*	*Mon 09/10/23*	*Mon 30/10/23*
1.4.1.2	***PPU***	***92 hrs***	***23 days***	***Mon 30/10/23***	***Thu 30/11/23***
1.4.1.2.1	*Background*	*40 hrs*	*10 days*	*Mon 30/10/23*	*Mon 13/11/23*
1.4.1.2.2	*Sprites*	*52 hrs*	*13 days*	*Mon 13/11/23*	*Thu 30/11/23*
1.4.1.3	***Cartridge Loader***	***12 hrs***	***3 days***	***Thu 30/11/23***	***Tue 05/12/23***
1.4.1.3.1	***Mapper***	***12 hrs***	***3 days***	***Thu 30/11/23***	***Tue 05/12/23***
1.4.1.3.1.1	*NROM*	*12 hrs*	*3 days*	*Thu 30/11/23*	*Tue 05/12/23*
1.4.2	*Renderer*	*60 hrs*	*15 days*	*Tue 05/12/23*	*Tue 26/12/23*
1.4.3	*Window System*	*12 hrs*	*3 days*	*Tue 26/12/23*	*Fri 29/12/23*
1.4.4	*Input Handler*	*12 hrs*	*3 days*	*Fri 29/12/23*	*Wed 03/01/24*
1.4.5	***User Inferface***	***72 hrs***	***18 days***	***Wed 03/01/24***	***Mon 29/01/24***
1.4.5.1	*CPU Status*	*32 hrs*	*8 days*	*Wed 03/01/24*	*Mon 15/01/24*
1.4.5.2	*PPU Status*	*20 hrs*	*5 days*	*Mon 15/01/24*	*Mon 22/01/24*
1.4.5.3	*Memory Status*	*20 hrs*	*5 days*	*Mon 22/01/24*	*Mon 29/01/24*

Table 20. Final System Development



1.4	**System Development**
1.4.1	**Emulator**
1.4.1.1	CPU
1.4.1.2	**PPU**
1.4.1.2.1	Background
1.4.1.2.2	Sprites
1.4.1.3	**Cartridge Loader**
1.4.1.3.1	**Mapper**
1.4.1.3.1.1	NROM
1.4.2	Renderer
1.4.3	Window System
1.4.4	Input Handler
1.4.5	**User Inferface**
1.4.5.1	CPU Status
1.4.5.2	PPU Status
1.4.5.3	Memory Status

Figure 11. Final System Development Gantt Diagram

4.2.1.5. System Testing

WBS	*Task Name*	*Work*	*Duration*	*Start*	*Finish*
1.5	***System Testing***	***16 hrs***	***4 days***	***Mon 29/01/24***	***Fri 02/02/24***
1.5.1	*CPU Tests*	*8 hrs*	*2 days*	*Mon 29/01/24*	*Wed 31/01/24*
1.5.2	*PPU Tests*	*4 hrs*	*1 day*	*Wed 31/01/24*	*Thu 01/02/24*
1.5.3	*ROM Tests*	*4 hrs*	*1 day*	*Thu 01/02/24*	*Fri 02/02/24*

Table 21. Final System Testing

1.5	**System Testing**
1.5.1	CPU Tests
1.5.2	PPU Tests
1.5.3	ROM Tests

Developer[50%]
Developer[50%]
Developer[50%]

Figure 12. Final System Testing Gantt Diagram

4.2.2. Final Risk Report

No changes.

4.2.3. Final Cost Budget

This section won't be as detailed as 4.1.5, since it will be mostly the same breakdown but changing some values, instead, only direct costs, indirect costs and client budget will be shown.

4.2.3.1. Direct Costs

Direct Costs					
I1	Description	Amount	Units	Price	Total
1	System Analysis	25	hrs	18.75	468.75
2	System Design	80	hrs	18.75	1500
3	System Documentation	416	hrs	18.75	7800
4	System Development	320	hrs	18.75	6000
5	System Testing	16	hrs	18.75	300

TOTAL SUM	**16068.75**

Table 22. Final Direct Costs

4.2.3.2. Indirect Costs

Indirect Costs						
I1	Description	Amount	Units	Price	Total	Type
1	Microsoft 365	1	1 yr license	69.99	69.99	Rent
2	Computer	1	computer	1500	1500.00	Amortization
3	Electricity	390	kWh	0.141	54.99	Rent
4	Internet	1	Subscription	35	227.50	Rent

TOTAL SUM	**1852.48**

Table 23. Final Indirect Costs

4.2.3.3. Client Budget

To see how this table is defined, refer to 4.1.5.8.

Client Budget		
I1	Description	Price
1	System analysis	653.49
2	System Design	2091.16
3	System Documentation	10874.03
4	System development	8364.64
5	System Testing	418.23

TOTAL SUM	**22401.54**

Table 24. Final Client Budget

4.2.4. Lessons Learned Report

Estimating work in a field in which I have little to no previous experience is hard, this led to an underestimation of development that was almost half of the result.

5. Analysis of the Information System (ASI)

5.1. ASI 1: System Definition

5.1.1. Determination of the System Scope

The main objective of this work is, as stated before, to provide a system capable of basic NES emulation, and to teach what it takes to build an emulator, how low-level emulators (LLE from now on) work, and to help users understand some intricacies of the NES.

To determine the reach of the system, first, some decision will be explained, this decision include the limitations of the system, these are:

- Only support Mapper 0 ROMs.
- Only support NTSC ROMs.
- No audio.
- Inaccurate colour display.
- Can only control emulation flow forward without save states.

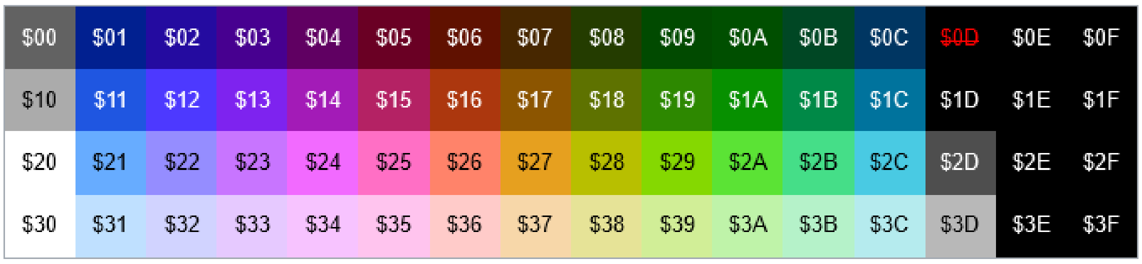
The first question I'll answer is. Why the NES? It's because it's a very simple system that is not in the realm of very old arcades or Atari consoles. Its processor, the 6502, has been heavily documented, so I felt like taking this challenge instead of a more complicated machine, like the PS1, would be possible to handle.

Why only mapper 0 ROMs, first I'll explain what a mapper is. The NES only has 2kb of memory, which is extremely limited, to counter that, the original engineers that designed the system, added a circuit to the game cartridges, these systems could extend the memory providing bank switching capabilities, adding new audio channels, storing the sprites and graphical data of the games, and storing the executable code of the games. These circuits are called mappers, since they map memory in the cartridge to memory in NES RAM. The problem was, since these capabilities were in the cartridge and not in the system, each game had their own, obviously, Nintendo had their own proprietary mappers, and those are the most common, but there are other mappers, be it designed by other companies at the time, or by the homebrew scene these days. I decided to only go with mapper 0, since it's the simplest, only providing graphical and code data, and I feel that adding more complicated mappers could extend the development time of this project considerably (24).

The decision to only support NTSC ROMs can be controversial, since we are in a PAL region. First, why not support both? The PPU is different across these systems, which would hinder development, then, Why NTSC? Simple, almost all guides I found were NTSC exclusive, including a brilliant frame timing the greatly simplified PPU development for the NTSC system, other reasons include, frame rate and resolution, since I prefer 60 FPS over 50.

No audio, there are two main reasons for audio, first, even though I believe emulating audio is a very interesting topic I would love to dig in, I believe time constraints would skyrocket adding those, coupled that with my inexperience with audio programming, and that adding audio would require adding multithreading to the system, I think that the technical difficulty of that subsystem greatly tops any other system included in this project excluding the renderer. The other reason, albeit more selfish, it's that I hate dealing with Windows APIs, and I did not find an audio library that could provide audio playback from memory, not that I looked around that much.

Inaccurate colour display, what does this mean, first, some background. The NES is a console designed to be used with CRT Televisions, so its PPU does not display an RGB signal, it displays a composite signal to be interpreted by the television, this makes it so no two TVs have the same colours. Some emulators do emulate that composite signal, but I believe that the intricacies of a CRT are not part of the emulator, so instead of delving into composite signal encoding/decoding, I would just use a colour palette and completely negate that problem. The following Colour palette is courtesy of NES Dev Wiki (25).



$00	$01	$02	$03	$04	$05	$06	$07	$08	$09	$0A	$0B	$0C	~~$0D~~	$0E	$0F
$10	$11	$12	$13	$14	$15	$16	$17	$18	$19	$1A	$1B	$1C	$1D	$1E	$1F
$20	$21	$22	$23	$24	$25	$26	$27	$28	$29	$2A	$2B	$2C	$2D	$2E	$2F
$30	$31	$32	$33	$34	$35	$36	$37	$38	$39	$3A	$3B	$3C	$3D	$3E	$3F

Figure 13. Colour Palettes Used

Why only control forwards. Adding ways to fine control the emulation backwards would be extremely taxing to the system, so save states will be able to emulate the functionality of backwards control.

Now that the limitations and the reasons these limitations exist have been defined, the next few paragraphs will entail what can be done with this system.

The user can load ROMs if they adhere to the previously stated limitations, and that they follow the iNES format. The user can stop the emulation and resume it at any time, and they can also advance the emulation by frame, scanline, pixel, PPU clock cycle, CPU instruction, and CPU clock cycle, must be noted that advancing by PPU or CPU will also advance the other, since they are coupled together. The user can use the interface to see the current PPU status, that means, see the values in MMIO registers, the palettes being used, and the sprites being loaded. The user can see the memory status, with image representations of the memory for both the full memory range and the RAM, and a full HxD inspired memory inspector (can't modify data). The user can see the CPU status, that means, the current values on the stack, the values on registers, and the disassembly of the program, the disassembler used is a custom-made tool for this project.

5.2. ASI 2: Requirements Specification

5.2.1. Obtaining System Requirements

5.2.1.1. Functional Requirements

RF.1. The system must let the user load a valid ROM

- RF.1.1. The User must provide the ROM
    - RF.1.1.1. iNES format
    - RF.1.1.2. Mapper 0 ROM
    - RF.1.1.3. NTSC ROM
- RF.1.2. The ROM is valid

        - RF.1.2.1. The ROM is immediately executed
    - RF.1.3. The ROM is not valid
        - RF.1.3.1. Invalid ROM is not executed
        - RF.1.3.2. Error message is shown
    - RF.1.4. The user loads a new ROM while the system is running
        - RF.1.4.1. The ROM is valid
            - RF.1.4.1.1. The emulator is reset
            - RF.1.4.1.2. The new ROM is executed
        - RF.1.4.2. The ROM is invalid
            - RF.1.4.2.1. The emulator is not reset
            - RF.1.4.2.2. The ROM is not executed
            - RF.1.4.2.3. The current ROM keeps being executed
- RF.2. The user resets the emulator
    - RF.2.1. ROM is loaded
        - RF.2.1.1. Reset emulator parameters to reset values
        - RF.2.1.2. Restart ROM
    - RF.2.2. ROM is not loaded
        - RF.2.2.1. Reset emulator parameters to reset values
    - RF.2.3. Emulator is stopped
        - RF.2.3.1. Reset emulator parameters to reset values
        - RF.2.3.2. Emulator keeps being stopped
    - RF.2.4. Emulator is running
        - RF.2.4.1. Reset emulator parameters to reset values
        - RF.2.4.2. Emulator keeps running
- RF.3. The system must let the user control the advance of emulation
    - RF.3.1. The user stops the emulation
        - RF.3.1.1. ROM is being executed
            - RF.3.1.1.1. Emulation is stopped
        - RF.3.1.2. ROM is not being executed
            - RF.3.1.2.1. When a ROM is loaded the Emulation will stop on load
    - RF.3.2. The user continues emulation
        - RF.3.2.1. ROM is being executed
            - RF.3.2.1.1. Emulation is resumed
        - RF.3.2.2. ROM is not being executed
            - RF.3.2.2.1. When a ROM is loaded the emulation will start immediately
    - RF.3.3. The user runs a frame
        - RF.3.3.1. Rom is loaded
            - RF.3.3.1.1. Emulation is running
                - RF.3.3.1.1.1. Emulation is stopped after running reminder of current frame
            - RF.3.3.1.2. Emulation is stopped
            - RF.3.3.1.3. Emulation is resumed
            - RF.3.3.1.4. Emulation is stopped after running reminder of current frame
        - RF.3.3.2. Rom is not loaded
            - RF.3.3.2.1. Emulation is running
                - RF.3.3.2.1.1. Emulation is stopped
            - RF.3.3.2.2. Emulation is stopped
                - RF.3.3.2.2.1. Nothing happens
    - RF.3.4. The user runs a scanline

- RF.3.4.1. Same structure as 2.3, swapping reminder of current frame with reminder of current scanline
- RF.3.5. The user runs a pixel
  - RF.3.5.1. Same structure as 2.3, swapping reminder of current frame with reminder of current pixel
- RF.3.6. The user runs a PPU cycle
  - RF.3.6.1. Same structure as 2.3, swapping reminder of current frame with PPU cycle
- RF.3.7. The user runs an instruction
  - RF.3.7.1. Same structure as 2.3, swapping reminder of current frame with reminder of current instruction
- RF.3.8. The user runs a CPU cycle
  - RF.3.8.1. Same structure as 2.3, swapping reminder of current frame with reminder of current CPU cycle

RF.4. The user views PPU status

- RF.4.1. ROM is being executed
  - RF.4.1.1. Show pattern tables
    - RF.4.1.1.1. Backgrounds
    - RF.4.1.1.2. Sprites
  - RF.4.1.2. Show palettes
    - RF.4.1.2.1. Backgrounds
    - RF.4.1.2.2. Sprites
  - RF.4.1.3. Show timing information
    - RF.4.1.3.1. Current scanline
    - RF.4.1.3.2. Current cycle
    - RF.4.1.3.3. Current frame number
    - RF.4.1.3.4. Frame time
    - RF.4.1.3.5. Time since last frame
    - RF.4.1.3.6. FPS counter
  - RF.4.1.4. Show internal registers
    - RF.4.1.4.1. V (VRAM address)
    - RF.4.1.4.2. X (fine x scroll)
    - RF.4.1.4.3. T (Temp address)
    - RF.4.1.4.4. W (address latch)
  - RF.4.1.5. Show MMIO registers
    - RF.4.1.5.1. PPU Control
    - RF.4.1.5.2. PPU Status
    - RF.4.1.5.3. PPU Mask
  - RF.4.1.6. Show current OAM
    - RF.4.1.6.1. Number
    - RF.4.1.6.2. X position
    - RF.4.1.6.3. Y position
    - RF.4.1.6.4. ID
    - RF.4.1.6.5. Attribute information
      - RF.4.1.6.5.1. Palette
      - RF.4.1.6.5.2. Behind background
      - RF.4.1.6.5.3. Filp vertically
      - RF.4.1.6.5.4. Flip horizontally

- RF.4.2. ROM is not being executed
  - RF.4.2.1. Show 3.1 fields with console reset information.
- RF.5. The user views CPU status
  - RF.5.1. ROM is being executed
    - RF.5.1.1. Show registers
      - RF.5.1.1.1. Processor status
        - RF.5.1.1.1.1. Negative flag
        - RF.5.1.1.1.2. Overflow flag
        - RF.5.1.1.1.3. Unused flag
        - RF.5.1.1.1.4. Break flag
        - RF.5.1.1.1.5. Decimal flag
        - RF.5.1.1.1.6. Interrupt disable flag
        - RF.5.1.1.1.7. Zero flag
        - RF.5.1.1.1.8. Carry flag
      - RF.5.1.1.2. Index register X
        - RF.5.1.1.2.1. Decimal
        - RF.5.1.1.2.2. Hexadecimal
      - RF.5.1.1.3. Index register Y
        - RF.5.1.1.3.1. Decimal
        - RF.5.1.1.3.2. Hexadecimal
      - RF.5.1.1.4. Accumulator
      - RF.5.1.1.5. Stack pointer
        - RF.5.1.1.5.1. Decimal
        - RF.5.1.1.5.2. Hexadecimal
    - RF.5.1.2. Show disassembly
      - RF.5.1.2.1. Show PC
      - RF.5.1.2.2. Show instruction
      - RF.5.1.2.3. Show branch labels
      - RF.5.1.2.4. Show tooltip
      - RF.5.1.2.5. Highlight Current instruction
    - RF.5.1.3. Show stack
      - RF.5.1.3.1. Memory position
      - RF.5.1.3.2. Value
  - RF.5.2. ROM is not loaded
    - RF.5.2.1. The same as 5.1.1 with reset values
    - RF.5.2.2. Show empty disassembly
    - RF.5.2.3. The same as 5.1.3 with reset values
- RF.6. The user view memory status
  - RF.6.1. ROM is being executed
    - RF.6.1.1. Show full memory range as image
    - RF.6.1.2. Show CPU RAM as image
      - RF.6.1.2.1. 1 bit per pixel
      - RF.6.1.2.2. 4 bits per pixel
    - RF.6.1.3. Show memory explorer
      - RF.6.1.3.1. Memory position
      - RF.6.1.3.2. Memory representation
        - RF.6.1.3.2.1. Hexadecimal
        - RF.6.1.3.2.2. String

RF.6.1.3.2.3. The user can search for memory position
RF.6.1.3.2.3.1. The memory position is highlighted
RF.6.2. ROM is not loaded
RF.6.2.1. The same as 6.1 with reset values
RF.7. The user uses a modern controller to run the game
RF.7.1. User can control the game with controller
RF.7.2. The user can't control the emulator with controller
RF.8. The user uses keyboard and mouse to control the emulator
RF.8.1. The user can control the emulator with keyboard and mouse
RF.8.2. The user can't control the game with keyboard and mouse
RF.9. The user can Save and load states
RF.9.1. The save states are ROM based
RF.9.2. Only up to MAX_STATE + 1 states per ROM
RF.9.3. The user must be able to change the current save state number
RF.9.3.1. The user can increment save state
RF.9.3.1.1. If the save state is greater than the max, it will wrap to MIN_STATE
RF.9.3.2. The user can decrement the save state
RF.9.3.2.1. If the save state is lower than zero, it will wrap to MAX_STATE

5.2.1.2. Non-Functional Requirements

RNF. 1. The system must run at least at FPS_TARGET FPS
RNF. 2. The system must run on current systems
RNF. 3. The emulated games must be controlled with a modern controller
RNF. 4. The system will not provide ROMS
RNF.4.1. The system will not infringe in the intellectual property of Nintendo
RNF.4.2. The system will not teach how to obtain ROMs
RNF. 5. The system will not teach how to play emulated games
RNF. 6. The system will not teach basic computer concepts
RNF.6.1. The system will expect the user to have a baseline knowledge
RNF. 7. The save states will be stored in STATE_PATH

5.2.1.3. Requirements Dictionary

Name	Value
STATE_PATH	%appdata%/EMOO/states/<ROM_NAME>/
MAX_STATE	5
MIN_STATE	0
FPS_TARGET	60

5.2.2. System Actors Identification

5.2.2.1. System

System represents the application. Every action the user takes, it taken through the system, so the system has been omitted for clarity in diagrams unless necessary.

5.2.2.2. User

User represents the person or group of persons that are using the system.

5.2.3. Use Case Specification

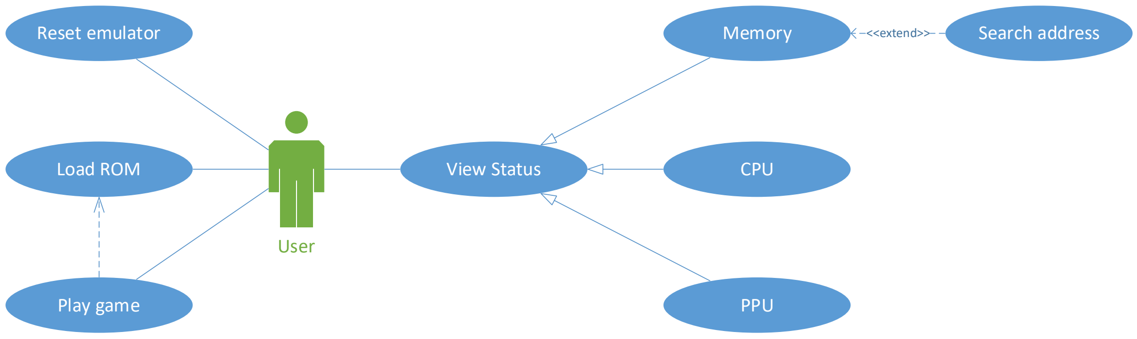


Figure 14. User use cases

Name
Reset emulator
Description
A user can reset the emulator to be the in a similar state as when it was started, the loaded rom is not removed.

Table 25. Use case specification: Reset emulator

Name
Load ROM
Description
A user can load a ROM to be executed by the emulator

Table 26. Use Case specification: Load ROM

Name
Play game
Description
Is the user has loaded a ROM; the ROM can be played

Table 27. Use Case specification: Play game

Name
View Status: Memory
Description
A user can see the contents in the memory of the emulator

Table 28. Use Case specification: View Status: Memory

Name
View Status: CPU
Description
A use can see the status of the emulators CPU

Table 29. Use Case specification: View Status: CPU

Name
View Status: PPU
Description
A user can see the status of the emulators PPU

Table 30. Use Case Specification: View Status: PPU

Name
Search address
Description
A user can inspect any memory address in the emulator’s memory space

Table 31. Use Case specification: Search address

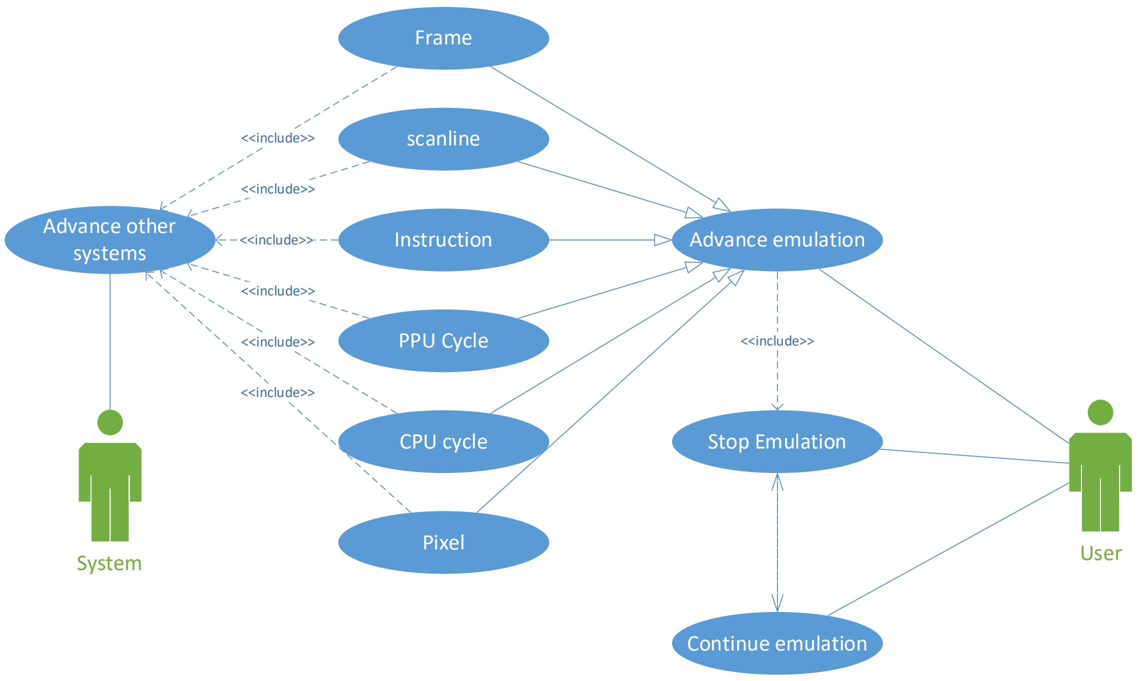


Figure 15. User use cases, continued

Name
Stop Emulation
Description
If the emulation is running, a user can stop it

Table 32. Use Case specification: Stop emulation

Name
Continue Emulation
Description
If the emulation is stopped, a user can resume it

Table 33. Use Case specification: Continue emulation

Name
Advance emulation: Frame
Description
A user can advance the emulation by the reminder of the current frame

Table 34. Use Case specification: Advance emulation: Frame

Name
Advance emulation: Scanline
Description
A user can advance the emulation by the reminder of the current scanline

Table 35. Use Case specification: Advance emulation: Scanline

Name
Advance emulation: Pixel
Description
A user can advance the emulation by the reminder of the current pixel

Table 36. Use Case specification: Advance emulation: Run Pixel

Name
Advance emulation: PPU cycle
Description
A user can advance the emulation by the reminder of the current PPU cycle

Table 37. Use Case specification: Advance emulation: PPU Cycle

Name
Advance emulation: CPU cycle
Description
A user can advance the emulation by the reminder of the current PPU cycle

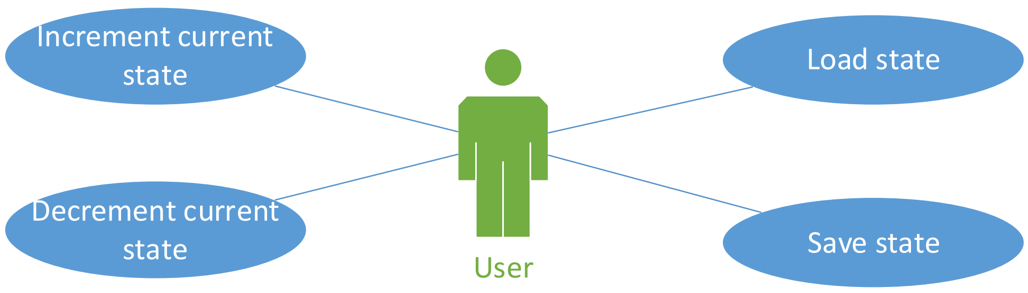
Table 38. Use Case specification: Advance emulation: CPU cycle

Name
Advance emulation: Instruction
Description
A user can advance the emulation by the reminder of the current instruction.

Table 39. Use Case specification: Advance emulation: Instruction

Name
Advance other systems
Description
The system will advance other systems to keep the emulator synchronized when a user advances emulation.

Table 40: Use Case specification: Advance other systems



Name
Load state
Description
The previously saved state will be loaded from disk from STATE_PATH, and the emulator will be set to it.

Table 41. Use Case specification: Load state

Name	
Save state	
Description	
The current state of the emulator will be saved to disk at STATE_PATH.	

Table 42. Use Case specification: Save state

Name	
Increment state	
Description	
The current state number will be incremented or set to MIN_STATE as needed	

Table 43. Use Case specification: Increment state

Name	
Decrement state	
Description	
The current state number will be incremented or set to MAX_STATE as needed	

Table 44. Use Case specification: Decrement state

5.3. ASI 3: Identification of Analysis Subsystems

The next sections will explain the different subsystems this work is made of in order to ease the understanding of its functions.

5.3.1. Subsystems Description

5.3.1.1. Emulator

This subsystem is the core of this work, it's the part tasked with emulating the original hardware, providing image data to the renderer, data to be displayed by the interface, letting the user control the flow of emulation, providing disassembly when needed and loading ROMs.

This system is made to be completely agnostic of other systems, this means that the emulator does not know of the existence of other external systems, only provides data when asked about it, and requests input data when necessary. The emulator could be completely run without an interface, as it's made to be controlled by other systems that do know about the emulator.

5.3.1.2. Renderer

The renderer is tasked with displaying the aforementioned image data provided by the emulator on the screen, other functions this subsystem has are providing a rendering backend for the interface system.

5.3.1.3. Window system

This is an extremely simple system that is only tasked with providing an screen to the renderer to display onto and a collection of events to allow moving, closing and resizing the screen.

5.3.1.4. Input Handler

Another very simple system that is tasked with getting a controller and polling that controller to send its data to the emulator, its also tasked with providing shortcuts to the interface.

5.3.1.5. Application

This subsystem is the system that is run when the system is executed, is tasked with bridging the different components, and with sending the user controls to the systems, it also provides an interface for the user to see the emulators data.

5.3.1.6. File Manager and Serialization

This subsystem is the one that handles serialization, file path, and file access. It provides a set of functions to change current file path, to open files in the current folder, and to serialize and deserialize data.

5.3.2. Description of Interfaces between Subsystems

All communication is performed through code, most of it it's directly through the application, with the notable exception of the communication between application and renderer, since the renderer has been heavily abstracted, all communication is made through API calls to not muddy the application code with rendering structures.

The Application holds references to the Input Handler, Emulator and Window System, holds the emulator to instruct it and get its data, the window system to poll events regarding the window, the input handler to send data to the emulator and to poll input events. The application does not hold a reference to the renderer and only calls its API when necessary.

The renderer does need a reference to the window subsystem to create a surface for it, and to initialise some window data.

Both the application and the emulator make use of the FileManager, the application uses it to send a path for the imgui ini file, and the emulator uses it for serialization and deserialization.

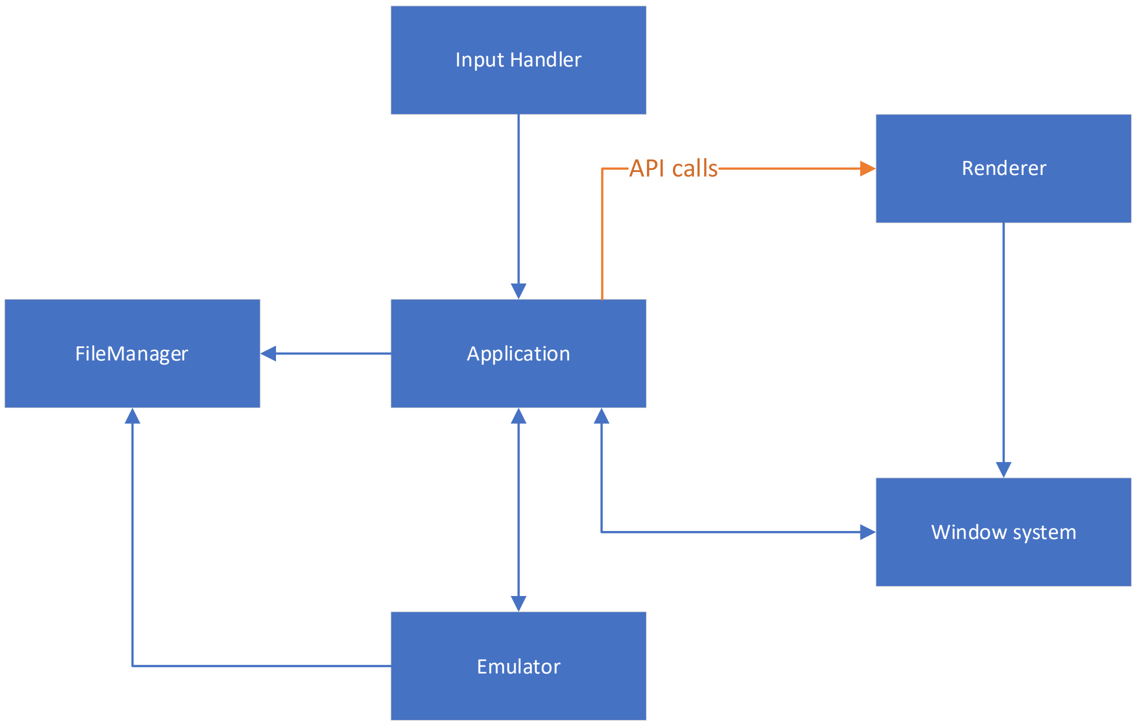


Figure 16. Interfaces between subsystems

5.4. ASI 4: Use Case Analysis

Restart Emulator	
Preconditions	None
Postconditions	Emulator is set to initial state
Actors	User
Description	1. User press restart button or shift + F8 2. Emulator is restarted
Variations	1. If the emulator has a ROM loaded, the ROM is not removed, so the emulator is set to initial state with a ROM
Exceptions	None
Notes	None

Table 45. Use Case analysis: Restart emulator

Load ROM	
Preconditions	None
Postconditions	Rom is loaded by the emulator
Actors	User
Description	1. User presses load ROM button or ctrl + O 2. File dialog is open 3. User selects ROM with valid extension 4. ROM is loaded by the system 5. ROM is executed if able
Variations	1. If the emulator is stopped, the ROM is not executed until emulation is resumed
Exceptions	If the ROM is invalid the ROM will not be loaded.
Notes	If a ROM is already loaded when the invalid ROM is being opened, the old ROM will continue to be the loaded ROM

Table 46. Use Case analysis: Load ROM

Play game	
Preconditions	1. Loaded ROM 2. Emulation running
Postconditions	None
Actors	User
Description	1. User plays the emulated ROM
Variations	None
Exceptions	None
Notes	None

Table 47. Use Case analysis: Play game

View status: Memory	
Preconditions	None
Postconditions	View memory interface is shown
Actors	User
Description	1. User presses view -> Memory status or ctrl + M 2. View memory status interface is shown
Variations	None
Exceptions	None
Notes	If other status is shown, older status is hidden but can be accessed by pressing a tab button.

Table 48. Use Case analysis: View Status: Memory

View status: CPU	
Preconditions	None
Postconditions	CPU status interface is shown
Actors	User
Description	1. User presses view -> CPU status or ctrl + C 2. CPU status interface is shown
Variations	None
Exceptions	None
Notes	If there is no ROM loaded, the disassembly is empty, and the stack is full. If other status is shown, older status is hidden but can be accessed by pressing a tab button.

Table 49. Use Case analysis: View status: CPU

View status: PPU	
Preconditions	None
Postconditions	View PPU status interface is shown
Actors	User
Description	1. User presses view -> PPU status or ctrl + P 2. PPU status interface is shown
Variations	None
Exceptions	None
Notes	If other status is shown, older status is hidden but can be accessed by pressing a tab button.

Table 50. Use Case analysis: View Status: PPU

Search address	
Preconditions	View memory status interface is shown
Postconditions	Memory inspector is scrolled to around address position, and address is highlighted
Actors	User
Description	1. User right clicks memory inspector 2. User inputs memory address in text input 3. User presses enter or OK button 4. Address is searched
Variations	If user presses cancel button, address is not searched
Exceptions	None
Notes	Text input is in hexadecimal, and must be in unsigned 16-bit value [0, 0xFFFF]

Table 51. Use Case analysis: Search address

Stop emulation	
Preconditions	Emulation is running
Postconditions	Emulation is stopped
Actors	User
Description	1. User presses stop button or F9 2. Emulation is stopped
Variations	None
Exceptions	None
Notes	None

Table 52. Use Case analysis: Stop emulation

Continue emulation	
Preconditions	Emulation is stopped
Postconditions	Emulation is resumed
Actors	User
Description	1. User presses run button or F9 2. Emulation is resumed
Variations	None
Exceptions	None
Notes	None

Table 53. Use Case analysis: Continue emulation

Advance emulation: Frame	
Preconditions	None
Postconditions	1. Emulation is advanced to reminder of current frame. 2. Emulation is stopped
Actors	User
Description	1. User presses run frame button or shift + F9 2. Emulation is stopped
Variations	If emulation is already stopped, emulation is resumed, reminder of current frame is advanced, and then stopped again
Exceptions	If ROM is not loaded, nothing happens when you press the button
Notes	None

Table 54. Use Case analysis: Advance emulation: Frame

Advance emulation: Scanline	
Preconditions	None
Postconditions	1. Emulation is advanced to reminder of current scanline. 2. Emulation is stopped
Actors	User
Description	1. User presses run scanline button or F10 2. Emulation is stopped
Variations	If emulation is already stopped, emulation is resumed, reminder of current scanline is advanced, and then stopped again
Exceptions	If ROM is not loaded, nothing happens when you press the button
Notes	None

Table 55. Use Case analysis: Advance emulation: Scanline

Advance emulation: PPU cycle	
Preconditions	None
Postconditions	1. Emulation is advanced to reminder of current PPU cycle. 2. Emulation is stopped
Actors	User
Description	1. User presses run PPU cycle button or ctrl + F10 2. Emulation is stopped
Variations	If emulation is already stopped, emulation is resumed, reminder of current ppu cycle is advanced, and then stopped again
Exceptions	If ROM is not loaded, nothing happens when you press the button
Notes	None

Table 56. Use Case analysis: Advance emulation: PPU Cycle

Advance emulation: pixel	
Preconditions	None
Postconditions	1. Emulation is advanced to reminder of current pixel. 2. Emulation is stopped
Actors	User
Description	1. User presses pixel button or shift + F10 2. Emulation is stopped
Variations	If emulation is already stopped, emulation is resumed, reminder of current pixel is advanced, and then stopped again
Exceptions	If ROM is not loaded, nothing happens when you press the button
Notes	The same as running PPU cycle, but skipping nonvisible cycles and scanlines

Table 57. Use Case analysis: Advance emulation: Pixel

Advance emulation: Instruction	
Preconditions	None
Postconditions	1. Emulation is advanced to reminder of current Instruction. 2. Emulation is stopped
Actors	User
Description	1. User presses advance instruction button or F11 2. Emulation is stopped
Variations	If emulation is already stopped, emulation is resumed, reminder of current instruction is advanced, and then stopped again
Exceptions	If ROM is not loaded, nothing happens when you press the button
Notes	None

Table 58. Use Case analysis: Advance emulation: Instruction

Advance emulation: CPU cycle	
Preconditions	None
Postconditions	1. Emulation is advanced to reminder of current Instruction. 2. Emulation is stopped
Actors	User
Description	1. User presses advance CPU cycle button or shift + F11 2. Emulation is stopped
Variations	If emulation is already stopped, emulation is resumed, reminder of current CPU cycle is advanced, and then stopped again
Exceptions	If ROM is not loaded, nothing happens when you press the button
Notes	None

Table 59. Use Case analysis: Advance emulation: CPU cycle

Advance emulation: Advance other systems	
Preconditions	User has advanced emulation
Postconditions	System has advanced all other systems.
Actors	User
Description	1. User advances emulation 2. System advances rest of emulator systems to keep synchronization
Variations	If emulation is already stopped, emulation is resumed, reminder of all other systems in equivalent time to selected option is advanced, and then stopped again
Exceptions	If ROM is not loaded, nothing happens when you press the button
Notes	This is done automatically when the user advances anything, since they are in the same Step function call

Table 60. Use Case analysis: Advance other systems

Save state	
Preconditions	ROM is loaded
Postconditions	Save state is loaded to file
Actors	User
Description	1. The user saves state 2. Emulator state is saved to file
Variations	If no previous state has been created, a folder with name equal to the ROM without the extension will be created in the states path. If a previous state has been created with the same number, the old one will be replaced.
Exceptions	If ROM is not loaded, nothing happens when you press the button
Notes	None

Table 61. Use Case analysis: Save state

Load state	
Preconditions	ROM is loaded
Postconditions	Emulator state is loaded from file
Actors	User
Description	1. The user loads state 2. Emulator state is loaded from file
Variations	None
Exceptions	If ROM is not loaded, nothing happens when you press the button. If no state exists for that ROM and that number, nothing happens.
Notes	None

Table 62. Use Case analysis: Load state

Increment state	
Preconditions	None
Postconditions	None
Actors	User
Description	1. The current state number is incremented
Variations	If the current state number is greater than MAX_STATE, it will be set to MIN_STATE
Exceptions	None
Notes	None

Table 63. Use Case analysis: Increment state

Decrement state	
Preconditions	None
Postconditions	None
Actors	User
Description	1. The current state number is decremented
Variations	If the current state number is lower than MIN_STATE, it will be set to MAX_STATE
Exceptions	None
Notes	None

Table 64. Use Case analysis: Decrement state

5.5. Class Analysis

5.5.1. Class Diagram

5.5.1.1. Emulator

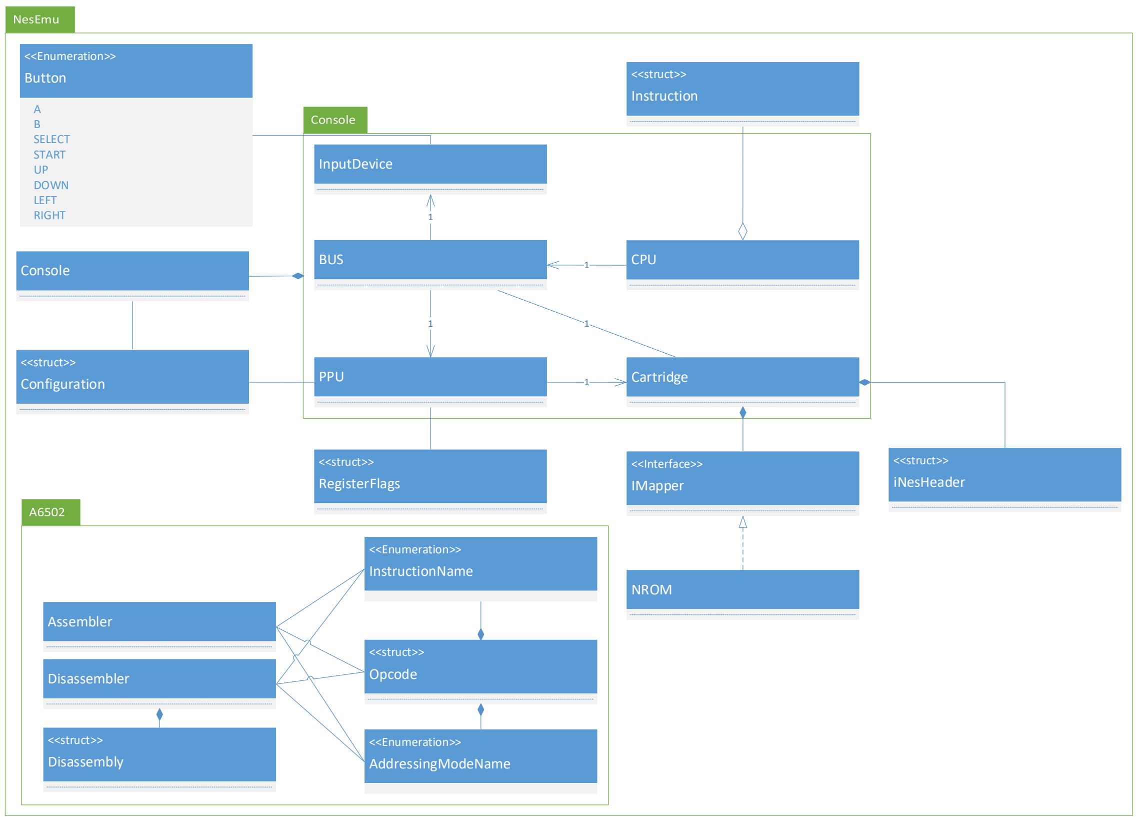


Figure 17. Emulator class diagram

InstructionName and AdressingModeName are only the names of instructions and addressing modes.

5.5.1.2. Renderer

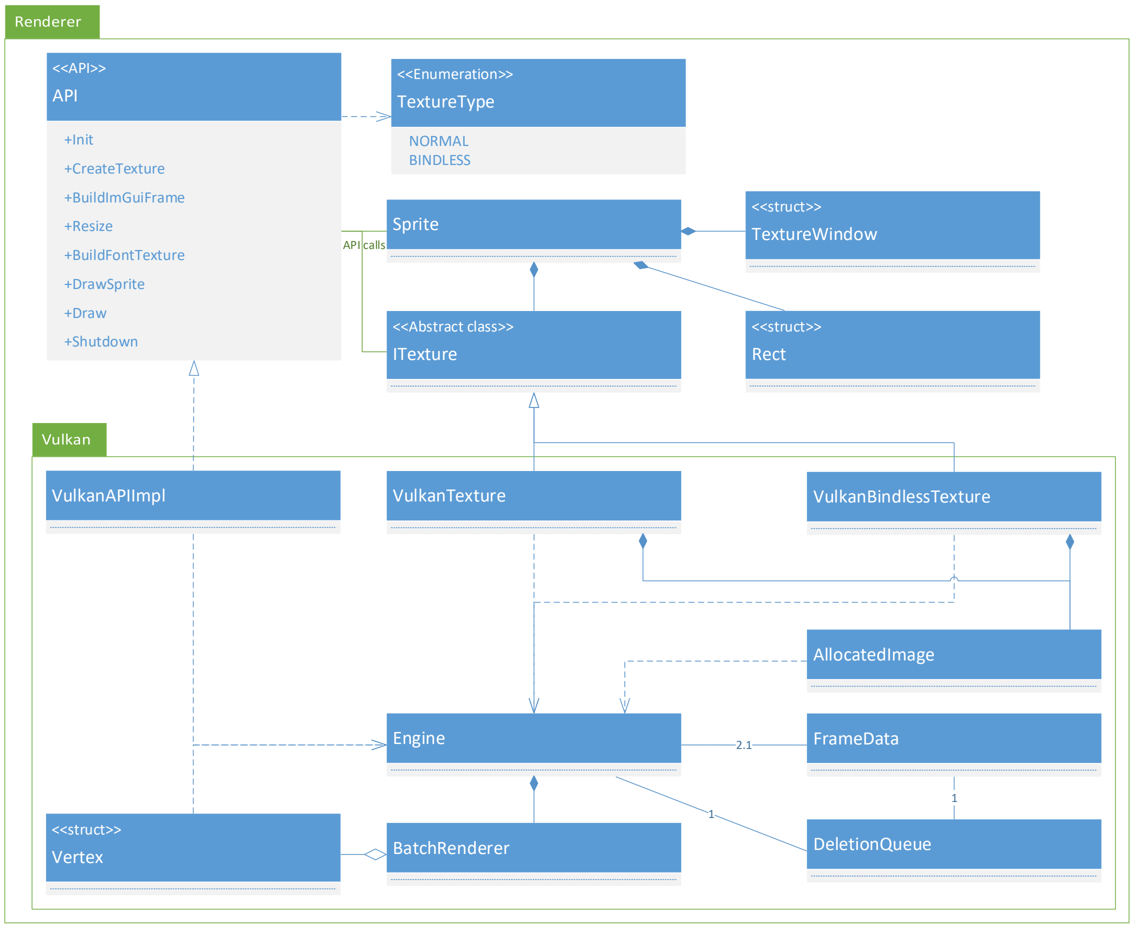


Figure 18. Renderer class diagram (simplified)

This is an extremely oversimplified diagram of the renderer, as the information omitted is there to provide structures for Vulkan and explaining it would be pointless since it’s not the point of this project and would only contribute to mudding the waters.

5.5.1.3. Input Handler

Why SDL3Input is included even though is not used will be explained in 5.5.2.3

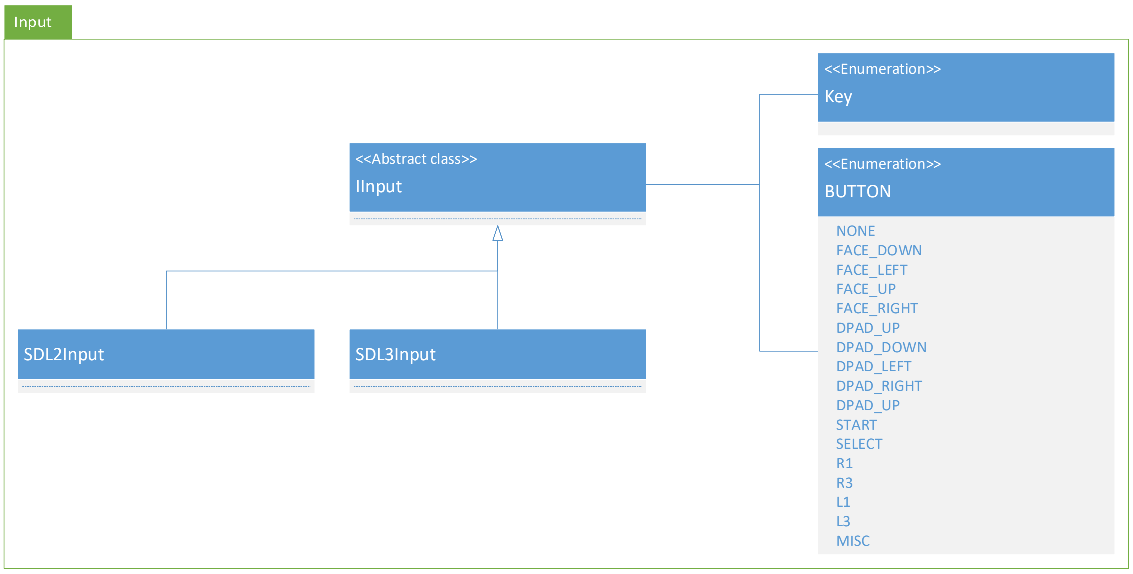


Figure 19. Input Handler class diagram

The members of Key are not show due to image constraints, as it contains 114 members, but Key contains, the Keys of a keyboard.

5.5.1.4. Window System

Why SDL3Window is included even though is not used will be explained in 5.5.2.4.

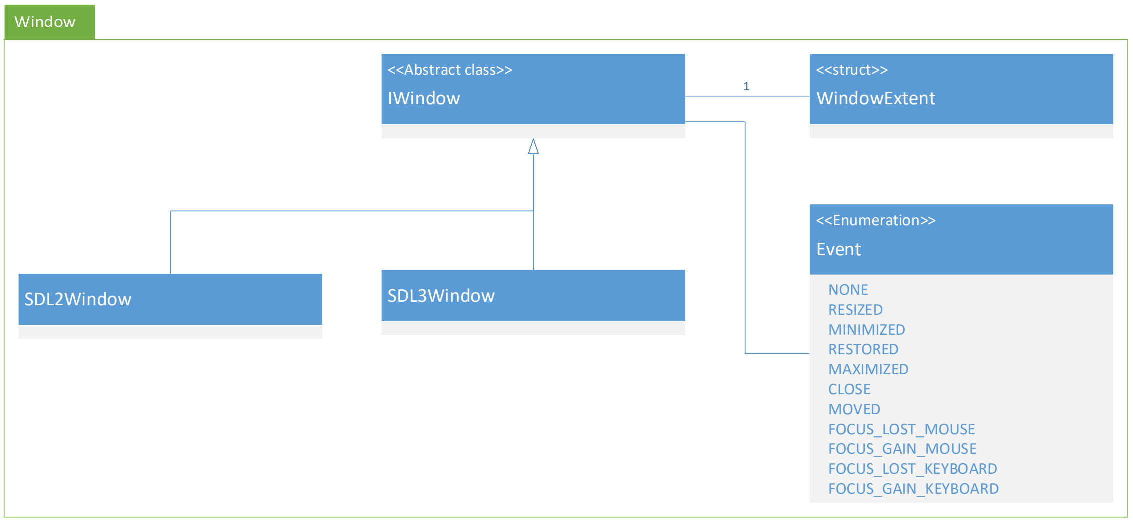


Figure 20. Window Class Diagram

5.5.1.5. FileManager

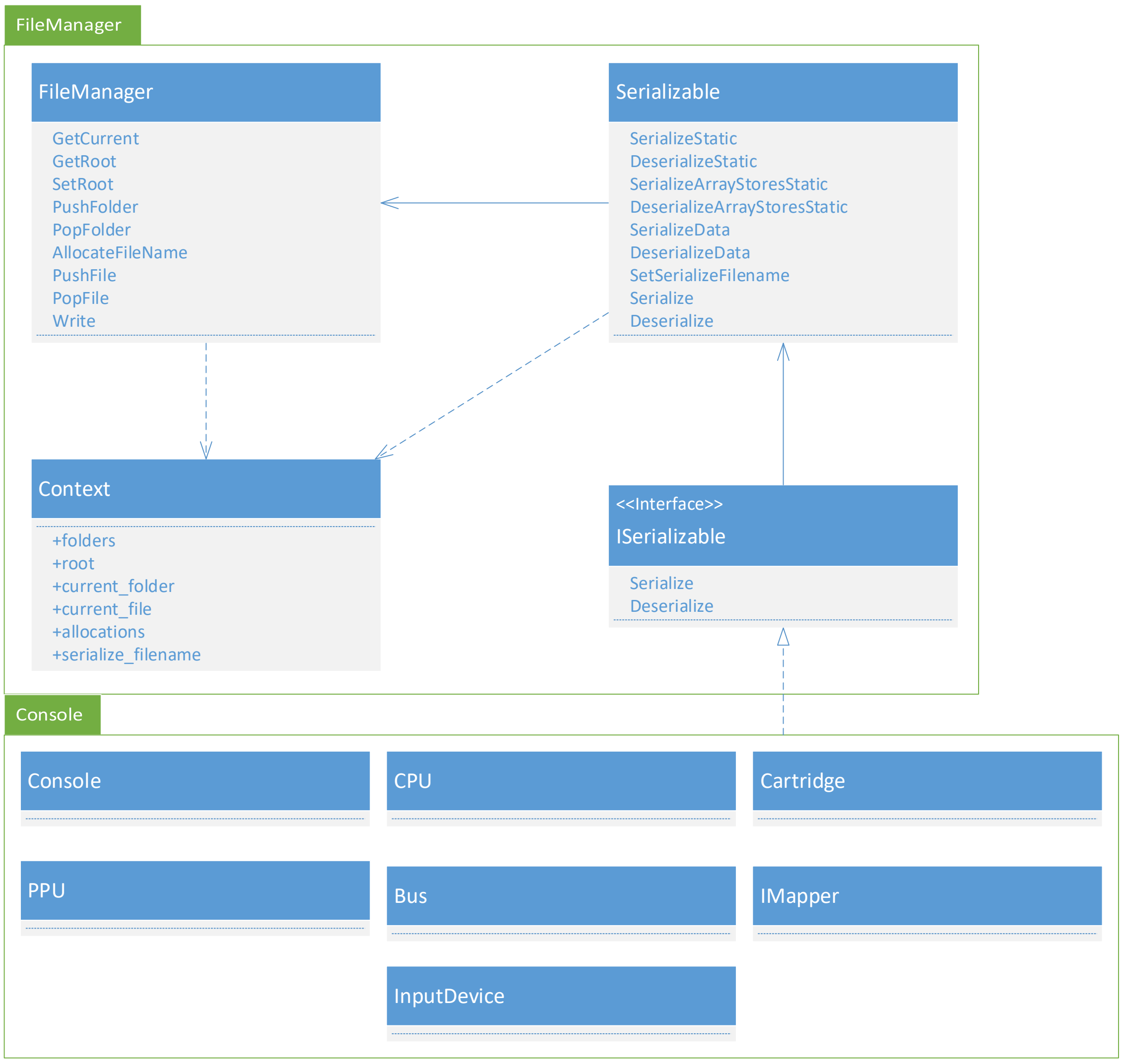


Figure 21. FileManager Class Diagram

5.5.1.6. Application

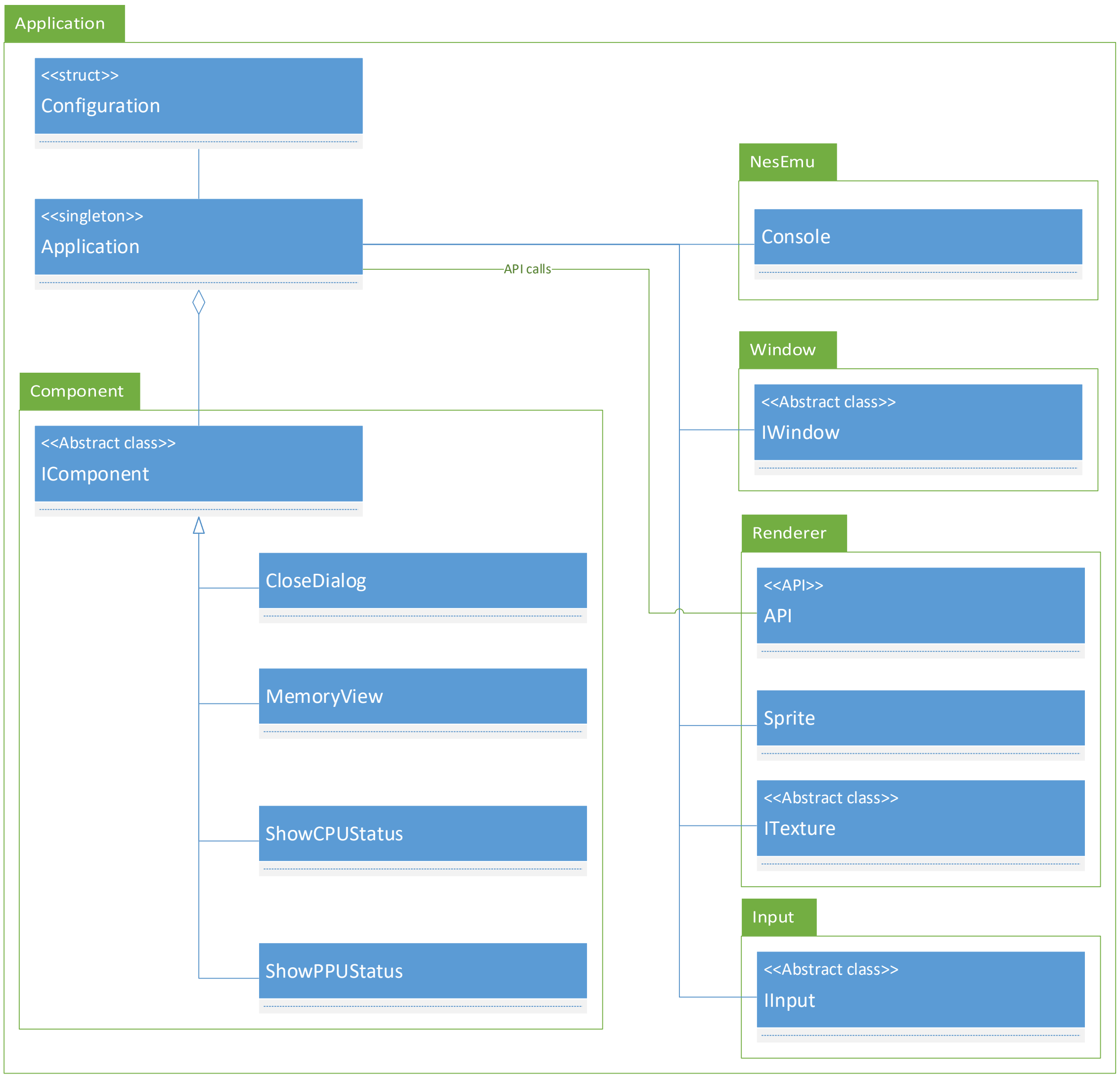


Figure 22. Application Class Diagram

5.5.2. Class Description

5.5.2.1. Emulator

Name
Console
Description
Holds all the different components of the emulator, interfaces with the outside, also runs the emulator.
Proposed attributes
None.
Proposed methods
• Step: runs the emulation for 1 master clock cycle. • Reset: resets the emulator (RF.2). • LoadCartridge(string): loads a cartridge from disk (RF.1). • LoadCartridgeFromMemory: loads a cartridge from disk (testing purposes). • UnloadCartridge: unloads current cartridge. • RunFrame: runs reminder of current frame (RF.3.3). • RunCpuInstruction: runs reminder of current instruction (RF.3.7). • RunPpuPixel: runs reminder of current pixel (RF.3.5). • RunPpuScanline: runs reminder of current scanline (RF.3.4). • RunPpuCycle: runs reminder of current PPU cycle (RF.3.6). • RunCpuCycle: runs reminder of current CPU cycle (RF.3.8). • CanRun: true if the emulator can run. • GetFrameTime: return time this frame took (RF.4.1.3.4). • GetTimeSinceLastFrame: returns time since last frame ended (RF.4.1.3.5). • GetCpu: getter for CPU. • GetPpu: getter for PPU. • GetBus: getter for bus. • GetConfig: getter for Configuration. • GetController: returns requested controller port.

Table 65. Console Class Description

Name
Configuration
Description
Contains information regarding the type of emulation to be done, like target framerate and resolution.
Proposed attributes
• cpu_clock_divisor: required amount of master clock ticks to run a CPU cycle. • ppu_clock_divisor: required amount of master clock ticks to run a PPU cycle. • frame_rate: desired frame rate. • frame_time: ms that a single frame takes. • width: width of the emulator screen (in pixels). • height: height of the emulator screen (in pixels).

Table 66. Configuration Class Description

Name
InputDevice
Description
Writes to the specific memory location reserved for the first input device of the NES.
Proposed attributes
• data: data to be set by input handler.
Proposed methods
• Write: updates inner data with data. • Read: sends a bit of inner data to bus. • Peek: return inner data. • SetPressed: ors data with button mask. • SetPressed: ors data with button mask array.

Table 67. Input Device Class Description

Name
Button
Description
Enum that contains the buttons of a NES controller
Proposed attributes
• A • B • SELECT • START • UP • DOWN • LEFT • RIGHT

Table 68. Button (emulator) Class Description

Name
Bus
Description
An abstraction of the contents of the NES memory range, components use this to read or write to memory.
Proposed attributes
None.
Proposed methods
• Read: returns data in memory address, may modify emulator state. • Peek: returns data in memory address, does not modify state (RF.5.1.3, RF.6.1.1, RF.6.1.3). • Write: writes data in memory address. • ConnectCartridge: connects to cartridge. • ConnectPPU: connects to PPU. • DMA: does DMA operation in PPU. • ConnectController: connects to controller. • Reset: resets bus state.

Table 69. Bus Class Description

Name
PPU
Description
Component that emulates the Ricoh 2C02, the “graphical unit” of the NES.
Proposed attributes
None.
Proposed members
• Step: steps a ppu cycle. • GetCycles: returns current ppu cycle number (RF.4.1.3.2). • GetFrames: returns number of frames since last reset (RF.4.1.3.3). • GetScanlines: returns number of current scanlinse (RF.4.1.3.1) • IsScanlineDone: true if scanline is finished. • IsFrameDone: true if frame is done. • IsNMI: true if DMA is in progress. • SetNMI: to be set by bus. • IsDMATransfer: if DMA transfer is in progress. • IsDMADummy: false if DMA can begin this frame. • SetDMADummy: to be set by bus. • SetDMAData: current dma byte. • GetOAMEntry: returns OAM entry at position (RF.4.1.6). • HasUpdatedPatternTables: true if the pattern tables have changed. • HasUpdatedPalettes: true if the palettes have changed. • ConnectCartridge: connects to the cartridge. • Reset: resets PPU state. • CpuRead: to be called by the bus in MMIO register range, returns MMIO register data, may modify state. • CpuPeek: to be called by the bus in MMIO register range, returns MMIO register data, does not modify state. • CpuWrite: to be called by the bus in MMIO register range, sets MMIO register data. • DMA: starts DMA. • X: returns X register (RF.4.1.4.2). • W: returns W register (RF.4.1.4.4). • V: returns V register (RF.4.1.4.1). • T: returns T register (RF.4.1.4.3). • GetScreen: returns screen colour data. • GetPatternTable: returns colour data for requested pattern table with requested palette colours (RF.4.1.1). • GetPalette: returns palettes color data (RF.4.1.2). • GetColorFromPalette: returns color for palette and index.

Table 70. PPU Class Description

Name
RegisterFlags
Description
A struct to made flags operations on MMIO registers easier
Proposed attributes
• reg u8: attribute holding the register data
Proposed methods
• Constructor: Constructs and sets the value of the register • operator(): returns reg. • is_flag_set: ands reg with value provided. • set_flags: sets or unsets flags according to control parameter

Table 71. RegisterFlags Class Description

Name
CPU
Description
Component that emulates the Ricoh 2A03, the CPU of the NES .
Proposed attributes
Constants: • STACK_VECTOR: position of the stack in the memory. • IRQ_VECTOR_LO: low byte of the irq vector. • IRQ_VECTOR_HI: high byte of the irq vector. • NMI_VECTOR_LO: low byte of the nmi vector. • NMI_VECTOR_HI: hight byte of the nmi vector. • RESET_VECTOR_LO: low byte of the reset vector. • RESET_VECTOR_HI: high byte of the reset vector.
Proposed methods
• ConnectBus: connects to bus. • IsDone: true if the current instruction is done. • Step: runs a cpu cycle. • Reset: resets state. • IRQ: raises hardware interrupt. • NMI: raises non maskable interrupt. • A: getter for accumulator (RF.5.1.1.4). • X: getter for index register x (RF.5.1.1.2). • Y: getter for index register y (RF.5.1.1.3). • S: getter for stack pointer (RF.5.1.1.5). • P: getter for processor status (RF.5.1.1.1). • PC: getter for program counter. • SetA: setter for accumulator. • SetX: setter for index register x. • SetY: setter for index register y. • SetS: setter for stack pointer. • SetP: setter for processor status. • SetPC: setter for program counter. • GetCycles: returns current cycles until CPU is done. • GetTotalCycles: returns total cycles executed by CPU.

Table 72. CPU Class Description

Name
Instruction
Description
Struct that bundles function pointers for both instruction and addressing mode
Proposed attributes
• name: the name of the instruction. • addressing_mode_fn: function that represents the addressing mode, does the addressing mode operation so the operand can be sent to the instruction. • instruction_fn: function that represents the instruction, does the instruction with the operand obtained from addressing_mode_fn. • cycles: number of cycles this instruction will take
Proposed methods
None

Table 73. Instruction Class Description

Name
Cartridge
Description
Emulates the cartridge that holds the "game", provides access to PRG and CHR rom.
Proposed attributes
None
Proposed methods
• Constructor: Loads cartridge from disk. • Constructor: Loads cartridge from memory. • ConnectBus: connects to bus. • CpuRead: for CPU reads in cartridge memory space, from PRG ROM. • CpuWrite: for CPU reads in cartridge memory space, from PRG ROM. • PPURead: for PPU reads in cartridge memory space, from CHR ROM. • PPUWrite: for PPU writes in cartridge memory space, from CHR ROM. • GetMirroring: returns mirroring type.

Table 74. Cartridge Class Description

Name
IMapper
Description
Interface for mapper implementations
Proposed attributes
None
Proposed methods
• Constructor: constructs setting amount of PRG and CHR banks. • CpuMapRead: maps CPU read to new address in cartridge space, in PRG ROM. • CpuMapWrite: maps CPU write to new address in cartridge space, in PRG ROM. • PpuMapRead: maps PPU read to new address in cartridge space, in CHR ROM. • PpuMapWrite: maps PPU write to new address in cartridge space, in CHR ROM. • GetName: returns name assigned to mapper, for NROM would be "NROM".

Table 75. IMapper Class Description

Name
iNesHeader
Description
Structure containing the data of an iNES header file format.
Proposed attributes
• name: contains the characters for NES and following by MS-DOS end-of-file "\x4E\x45\x53\x1A". • prg_rom_chunks: amount of PRG ROM chunks. • chr_rom_chunks: amount of CHR ROM chunks. • flags_6: mapper, mirroring, battery, trainer. • flags_7: maper, vs/Playchoice, NES 2.0. • prg_ram_size: PRG RAM size • flags_9: tv system. • flags_10: tv system, PRG RAM presence. • unused: padding.
Proposed methods
None

Table 76. iNesHeader Class Description

Name
NROM
Description
Emulates mapper 0 "NROM", implements IMapper.
Proposed attributes
Same as IMapper.
Proposed methods
Same as IMapper.

Table 77. NROM Class Description

Name
Assembler
Description
A custom tool made to make tests easier.
Proposed attributes
None
Proposed methods
• ConnectBus: connects to bus in order to insert assembled code. • Assemble: assembles provided string into code. • GetInstructionName: gets InstructionName from string. • ParseAddressingMode: parses addressing mode from string.

Table 78. Assembler Class Description

Name
Disassembler
Description
A custom tool made to show disassembly in the interface
Proposed attributes
None
Proposed methods
• ConnectBus: connects to bus in order to insert assembled code. • Get: returns disassembly of provided address in the PRG ROM, if address is not disassembled, it will call DisassembleFromAddress from it. • Cotains: returns true if has disassembled address provided. • Init: initialises data and calls DisassembleFromAddress into the reset, IRQ and NMI vectors, if the control parameter is set to true, it would replace some values with known constants. • GetCache: returns all disassembly (RF.5.1.2). • DisassembleFromAddress: disassembles addresses starting at provided value and stops when it founds existing disassembly or a return instruction, if the control parameter is set to true, it would replace some values with known constants.

Table 79. Disassembler Class Diagram

Name
Disassembly
Description
Struct representing disassembly.
Proposed attributes
• repr: string representation of the disassembly. • label: if exists, label for disassembly. • size: size in bytes of instruction. • instruction: string name of the instruction. • addressing: string addressing mode of the instruction. • has_register: true if the disassembly is accessing a known constant. • register_name: name of the known constant. • register_value: value of the known constant.
Proposed methods
None

Table 80. Disassembly Class Description

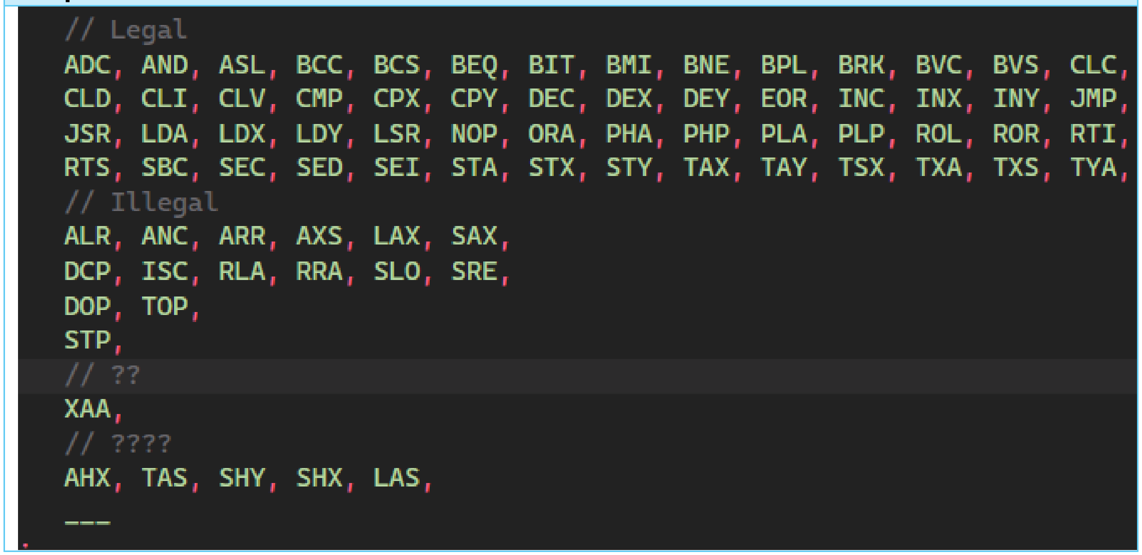
Name

InstructionName

Description

Enumeration of instruction names.

Proposed attributes



```
// Legal
ADC, AND, ASL, BCC, BCS, BEQ, BIT, BMI, BNE, BPL, BRK, BVC, BVS, CLC,
CLD, CLI, CLV, CMP, CPX, CPY, DEC, DEX, DEY, EOR, INC, INX, INY, JMP,
JSR, LDA, LDX, LDY, LSR, NOP, ORA, PHA, PHP, PLA, PLP, ROL, ROR, RTI,
RTS, SBC, SEC, SED, SEI, STA, STX, STY, TAX, TAY, TSX, TXA, TXS, TYA,
// Illegal
ALR, ANC, ARR, AXS, LAX, SAX,
DCP, ISC, RLA, RRA, SLO, SRE,
DOP, TOP,
STP,
// ??
XAA,
// ????
AHX, TAS, SHY, SHX, LAS,
___
```

Table 81. InstructionName Class Description

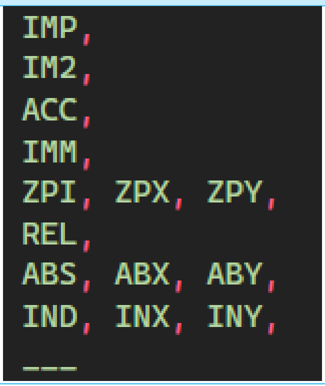
Name

AddressingModeName

Description

Enumaration of addressing mode names.

Proposed attributes



```
IMP,
IM2,
ACC,
IMM,
ZPI, ZPX, ZPY,
REL,
ABS, ABX, ABY,
IND, INX, INY,
___
```

Table 82. AddressingModeName Class Description

Name

Opcode

Description

Struct representing instruction addressing mode pair.

Proposed attributes

- instruction InstructionName: instruction.
- mode AddressingModeName: addressing mode.

Proposed methods

- operator<=>: defaulted to provide ordering.

5.5.2.2. Renderer

Name
API
Description
Not a class, a collection of unimplemented functions grouped in a namespace, it abstracts the rendering backend in a collection of functions to be used by the other systems without the complexity and structures required for the renderer to work
Proposed attributes
None
Proposed functions
• Init: Initialises the underlying renderer • Shutdown: shuts down the underlying renderer • Draw: draws all sprites sent to renderer • Resize: resize renderer surface • DrawSprite: sents the sprite to be drawn by the batcher, is not presented to the screen until Draw is called • CreateTexture: Creates a texture of the desired type with desired dimensions • CreateTexture: Create a texture of the desired type with desired dimensions, and fills it with provided data. • BeginImGuiFrame: Starts ImGui frame. • BuildFontTexture: Builds ImGui Font texture.

Table 83. Renderer API Class Description

Name
TextureType
Description
Enumeration holding the supported types of textures
Proposed attributes
• BINDLESS: texture is not bound to a texture slot • NORMAL: texture is bound to texture slot

Table 84. TextureType Class Description

Name
Sprite
Description
A rect that holds a texture a position and a size
Proposed attributes
• rect: position and size of the sprite • z_index: “depth” of the sprite • texture: texture of the sprite • texture_window: texture window of the sprite, more on table 78
Proposed methods
• Constructor: Constructs a sprite with a rect. • ConstructorConstructs an sprite with a texture, a rect and a texture window • SetTexturesets texture and texture window • Draw: sends sprite to the batcher

Table 85. Sprite Class Description

Name
TextureWindow
Description
Class used to be able to load different textures in the shape of a spritesheet, not a useful feature in this work, but a good thing to have If I reuse this renderer. Provides a "window" into a texture rendering only the parts of the texture between (x0, y0) and (x1, y1)
Proposed attributes
• x0: point 0 x • y0: point 0 y • x1: point 1 x • y1: point 1 y
Proposed methods
None

Table 86. TextureWindow Class Description

Name
Rect
Description
A struct that holds a point and a size
Proposed attributes
• x: x position of the rect • y: y position of the rect • w: width of the rect • h: height of the rect
Proposed methods
None

Name
ITexture
Description
Abstract class like interface, definition of a texture without the renderer implementation
Proposed attributes
None
Proposed methods
• Constructor: constructs texture with width and height. • Constructor: constructs texture with width, height, and texel data. • SetData: sets texel data of the texture. • ToImgui: returns imgui equivalent to be able to be used with the interface.

Table 87. ITexture Class Description

Name
VulkanAPIImpl
Description
Not a class, provides implementation of renderer API functions
Proposed attributes
None
Proposed functions
Implementation of API functions

Table 88. VulkanAPIImpl Class Description

Name
Engine
Description
Backbone of the renderer, initialise and closes vulkan and imgui, provides rendering support.
Proposed attributes
None
Proposed methods
• Get: class is a singleton, returns instance. • Init: inits renderer and sets it to render to window, if control parameter is set, initialises imgui. • Cleanup: cleans up renderer. • Draw: draws background and draws batched data. • Resize: resizes swapchain. • RequestResize: requests a swapchain resize. • AddTextureToBatcheradds texture to batcher, only works with bindless textures in order to not create a texture atlas. • RemoveTextureFromBatcher: removes given texture from batcher. • CreateImage: creates empty vulkan, if control parameter is set, image will be mipmapped. • CreateImage: creates vulkan image with texel data, if control parameter is set, image will be mipmapped. • DestroyImage: destroys vulkan image. • SetImageData: sets texel data in image. • CreateBuffer: creates vulkan buffer. • DestroyBuffer: destroys vulkan buffer. • SubmitDrawRect: adds rect to batcher. • GetDevice: returns vulkan device (GPU abstraction from Vulkan). • GetAllocator: returns vma allocator. • GetSampler: returns default image sampler.

Table 89. Engine Class Description

Designed using both vkguide and vulkan-tutorial until 3D concepts start appearing, modified it with bindless support in order to save texture slots in the gpu, since I did not find enough information on how to build a texture atlas from textures of different sizes.

Name
BatchRenderer
Description
Class that stores provided rects and textures, and renders them at once in order to save draw calls.
Proposed attributes
Constants • MAX_SPRITE_AMOUNT: max number of rects that can be submitted to the batcher.
Proposed methods
• StartBatch: copies vertex and index data to GPU memory • Add: adds a rect to batcher. • PrepareDescriptor: prepares descriptor set used by the rendererer, binds textures to it • Draw: draws data stored in GPU memory. • AddTexture(VulkanBindlessTexture*): adds texture to batcher. • RemoveTexture(VulkanBindlessTexture*): removes texture from batcher. • HasTexture(VulkanBindlessTexture*): true if batcher contains texture • Flush: cleans vertices, indices and textures. • GetVertexCount: returns number of vertices in the batcher currently. • GetIndexCount: removes current number of indices. • GetAddress: returns address of the mesh buffer used by the batcher.

Table 90. BatchRenderer Class Description

Engine has a list of BatchRenderers, when one is full, it adds another. Completely overkill for the end implementation of this project, but at the time of analysis, it was not known whether to draw the emulator screen it would be better to use a quad for each internal screen pixel, or a single texture that uses internal screen pixels as texels.

Name
VulkanTexture
Description
Vulkan specific implementation of ITexture, uses a texture slot.
Proposed attributes
None.
Proposed methods
Implementation of ITexture.

Table 91. VulkanTexture Class Description

Name
VulkanBindlessTexture
Description
Vulkan specific implementation of ITexture, does not use a texture slot.
Proposed attributes
None.
Proposed methods
Implementation of ITexture.

Table 92. VulkanBindlessTexture Class Description

The existence of this class is complex, its existence is tied to the BatchRenderer, since I did not find much literature on 2D rendering, like, I found a lot, but not on more technical details like building texture atlas of misshaped textures and batch rendering, so I decided to investigate this “new” technology named Bindless resources.

Name
AllocatedImage
Description
Vulkan image and memory allocation bundled together.
Proposed attributes
• Image: handle to vulkan image • view: handle to vulkan image view • format: the image format • extent: the size of the image • allocation: memory allocation of the image
Proposed methods
None

Table 93. AllocatedImage Class Description

This is one of the internals of the renderer that I am cautious on adding, but I since this is basically what textures are, an AllocatedImage and maybe a descriptor set depending on the image, I decided to show it here since it provides context.

Name
Vertex
Description
Vertex data that will be sent to the GPU when drawing.
Proposed attributes
• position: position of this vertex in space. • tex_coords: texture coordinates used for this vertex, also named UV in other programs. • texture_id: id of the texture, if not defined, will use a default texture
Proposed methods
None

Table 94. Vertex Class Description

Other internal I am not keen to add, but vertex definitions are extremely important, and I think adding them is important.

Name
FrameData
Description
A class used to delete structures when are not used.
Proposed attributes
• command_pool: pool from which Vulkan commands will be allocated. • command_buffer: Command buffer that will record Vulkan commands. • swapchain_semaphore: GPU to GPU synchronization, used to make render commands wait on the swapchain request • render_semaphore: GPU to GPU synchronization, used to control presenting mage • render_fence: GPU to CPU synchronization, blocks CPU until GPU finishes work • deletion_queue: deletion queue of this frame • frame_descriptor: descriptor of this frame, a descriptor set hold information on how to send data to the GPU.
Proposed methods
None

Table 95. FrameData Class Description

The last of the set of implementation details that will be shown here, added since this is the most important structure of the renderer. The engines will have n frames in-flight, that means, that it renders one while it cleans or prepares the next.

Name
DeletionQueue
Description
A class used to delete structures when are not used.
Proposed attributes
None.
Proposed methods
• PushFunction: adds a cleanup function (deletor) to be added to the deletion queue • Flush: calls every function added and then cleans the queue.

Table 96. DeletionQueue Class Description

There is a main deletion queue for the engine class, and each FrameData has its own.

5.5.2.3. *Input Handler*

Name

IInput

Description

Abstract class for Input handlers, provides actions functionality, but not event or keyboard/mouse functionality, that is to be provided by implementers (RF.7, RF.8).

Proposed attributes

Implementation details:

- current key is the key to which current action being run is linked to
- current button is the button to which current action being run is linked to

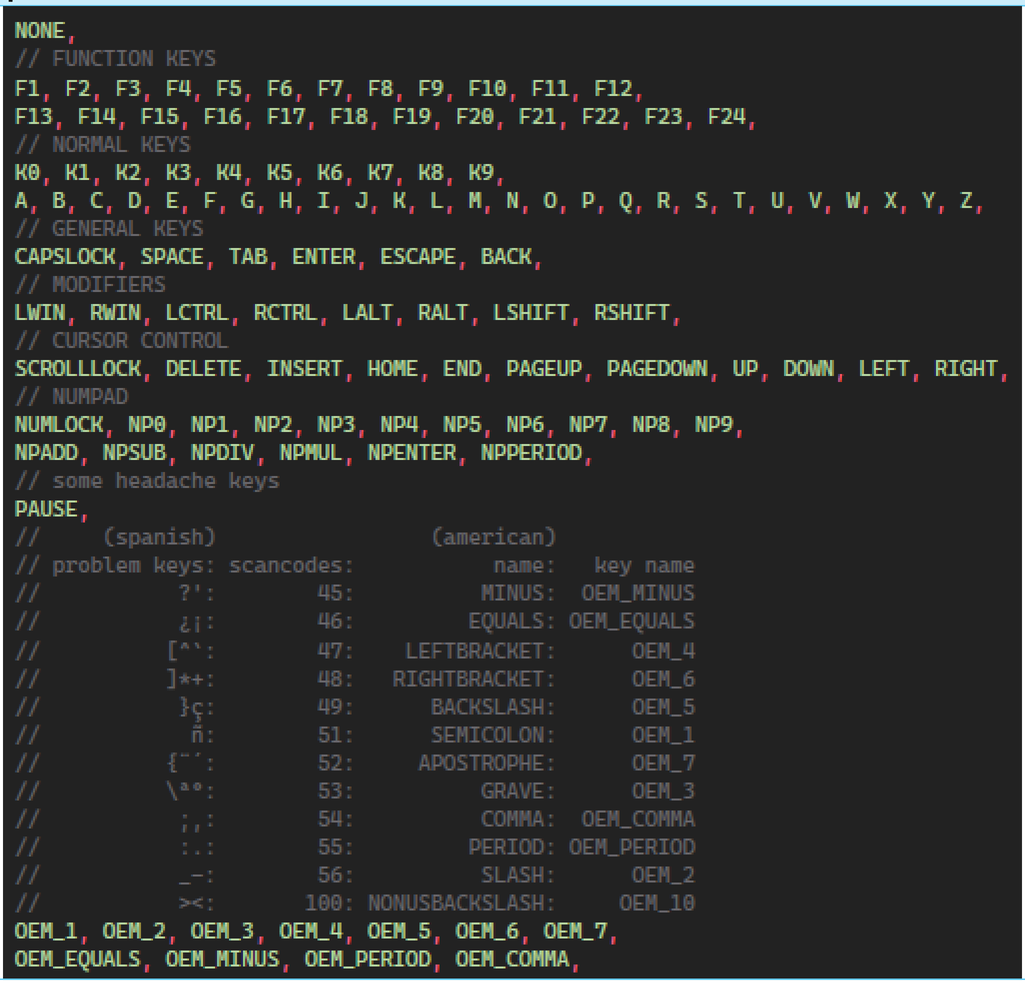
if current key is defined, current button is not defined, and vice versa.

Proposed methods

- GetButton: returns true whether passed button is pressed.
- GetKey: returns true whether passed key is pressed.
- IsKeyModified: returns true if key is being modified by a mod key (ctrl, shift, alt, win)
- IsRepeating: returns true if current key or button is being hold down after being pressed.
- IsKeyRepeating: returns true if given key is being hold down after being pressed.
- IsButtonRepeating: returns true if given button is being hold down after being pressed.
- CanRepeatAfter: returns true if time amount has passed since last current key or button was pressed
- CanRepeatKeyAfter: returns true if time amount has passed since last given key press.
- CanRepeatButtonAfter: returns true if time amont has passed since last given button press.
- CanRepeatEvery: returns true every time some time amount has passed since current key or button was pressed.
- CanRepeatKeyEvery: returns true every time amount has passed since given key was pressed.
- CanRepeatButtonEvery: returns true every time amount has passed since given button was pressed.
- ClearActions: deletes all actions
- AddGamepadAction: ads an action to be run when a given button is pressed,
- AddKeyboardAction: ads an action to be run when a given button is pressed.
- RunActions: runs all actions that can be run, (key or button is pressed)
- RunGamePadActions:
- RunKeyboardActions: runs actions for key if key is pressed.
- RunGamePadActions: runs actions for button if button is pressed.
- Update: calls platform specific update implementation, updates state and runs actions.
- ProcessEvents: runs input related events of underlying implementation.

Table 97. IInput Class Description

An implantation of IInput has to be an adapter to easily slot in new technologies in case the old one is faulty, or a new one is better, without modifying existing code; this is the reason SDL2Input exists, as SDL3Input was the original, and at some point in development, a rollback from SDL3 to SDL2 was needed. This is made more complex than it needs to be in order to make easier future reuses n other projects or ampliations in this work if any.

Name	
Key	
Description	
An enumeration of keys that can be used in this input handler	
Proposed attributes	
	

```
NONE,
// FUNCTION KEYS
F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F12,
F13, F14, F15, F16, F17, F18, F19, F20, F21, F22, F23, F24,
// NORMAL KEYS
K0, K1, K2, K3, K4, K5, K6, K7, K8, K9,
A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z,
// GENERAL KEYS
CAPSLOCK, SPACE, TAB, ENTER, ESCAPE, BACK,
// MODIFIERS
LWIN, RWIN, LCTRL, RCTRL, LALT, RALT, LSHIFT, RSHIFT,
// CURSOR CONTROL
SCROLLLOCK, DELETE, INSERT, HOME, END, PAGEUP, PAGEDOWN, UP, DOWN, LEFT, RIGHT,
// NUMPAD
NUMLOCK, NP0, NP1, NP2, NP3, NP4, NP5, NP6, NP7, NP8, NP9,
NPADD, NPSUB, NPDIV, NPMUL, NPENTER, NPPERIOD,
// some headache keys
PAUSE,
//     (spanish)                (american)
// problem keys: scancodes:          name:   key name
//          ?':        45:          MINUS:  OEM_MINUS
//          ¿¡:        46:         EQUALS: OEM_EQUALS
//         [^`:        47:    LEFTBRACKET:      OEM_4
//         ]*+:        48:   RIGHTBRACKET:      OEM_6
//          }ç:        49:      BACKSLASH:      OEM_5
//           ñ:        51:      SEMICOLON:      OEM_1
//         {¨´:        52:     APOSTROPHE:      OEM_7
//         \ªº:        53:          GRAVE:      OEM_3
//          ;,:        54:          COMMA:  OEM_COMMA
//          :.:        55:         PERIOD: OEM_PERIOD
//          _-:        56:          SLASH:      OEM_2
//          ><:       100: NONUSBACKSLASH:     OEM_10
OEM_1, OEM_2, OEM_3, OEM_4, OEM_5, OEM_6, OEM_7,
OEM_EQUALS, OEM_MINUS, OEM_PERIOD, OEM_COMMA,
```

Table 98. Key Class Description

Name
Button
Description
An enumeration of all valid buttons for this input handler
Proposed attributes
• NONE • FACE_DOWN • FACE_LEFT • FACE_UP • FACE_RIGHT • DPAD_DOWN • DPAD_LEFT • DPAD_UP • DPAD_RIGHT • START • SELECT • R1 • R3 • L1 • L3

Table 99. Button (Input Handler) Class Description

Name
SDL2Input
Description
IInput implementation using SDL2 as backed.
Proposed attributes
None.
Proposed methods
Same as IInput.

Table 100. SDL2Input Class Description

Name
SDL3Input
Description
IInput implementation using SDL3 as backed.
Proposed attributes
None.
Proposed methods
Same as IInput.

Table 101. SDL3Input Class Description

5.5.2.4. Window System

Name
IWindow
Description
The same idea as IInput, provide an adapter with which implementors interface with underlying system to make swapping libraries easier, almost pure virtual only provides adapter interface.
Proposed attributes
None.
Proposed methods
• GetDimensions: returns a WindowExtend with window dimensions. • GetDimensions: modifies passed parameters to window dimensions, made this way since its common for window systems to have a similar function. • CreateRendererSurface: used to provide a rendering surface to renderer, to be called exclusively by renderer. • GetWindowID: to know what window is the main one when using multiple windows. • InitImguiForRenderer: initialises Imgui to work with vulkan, since imgui window requires it, to be called by renderer. • ShutdownImGuiWindow: shuts down imgui window subsystem. • BeginImGuiFrame: Begins window imgui frame. • AddEventFunction: sets function to be run when given event is being polled. • ProcessEvents: runs event related code for the underlying system.

Table 102. IWindow Class Description

As stated before, IWindow only provides a adapter interface for other implementations to use so in case of a library change, the main code does not need to be changed; SDL3 was the original implementation, but a bug in the SDL3 imgui implementation that was way above what I could fix forced me to roll back to SDL2, luckily, this system made swapping from SDL3 to SDL2 incredibly easy.

This system and the input handler have been used in other personal projects, and thanks to their flexibility, I could implement a new version with the Windows API in no time.

Name
WindowExtent
Description
Struct that represents the size of a window.
Proposed attributes
• w: the width of the window. • h: the height of the window.
Proposed methods
None.

Table 103. WindowExtent Class Description

Contrary to what GetWindowID might led you to believe, this system is not designed to have multiple windows, and the renderer is definitely not designed to render to multiple windows, that's why WindowExtend does not support position, and why IWindow does not have a position getter of any sort, GetWindowID to not close the main window when closing an imgui viewport.

Name
SDL2Window
Description
SDL2 implementation of IWindow.
Proposed attributes
None.
Proposed methods
Implementation of IWindow.

Table 104. SDL2Window Class Description

Name
SDL3Window
Description
SDL3 implementation of IWindow.
Proposed attributes
None.
Proposed methods
Implementation of IWindow.

Table 105. SDL3Window Class Description

Name
Event
Description
Enumeration of supported events by window system
Proposed attributes
• NONE • RESIZED • MINIMIZED • RESTORE • MAXIMIZED • CLOSE • MOVED • FOCUS_LOST_MOUSE • FOCUS_GAIN_MOUSE • FOCUS_LOST_KEYBOARD • FOCUS_GAIN_KEYBOARD

Table 106. Event Class Description

5.5.2.5. FileManager

Name
FileManager
Description
Not a class but a collection of functions that provide file managing support
Proposed attributes
None.
Proposed functions
• GetCurrent: returns path of current folder. • GetRoot: returns path of current root folder. • SetRoot: sets new root folder. • PushFolder: appends folder structure to current folder. • AllocateFileName: returns new file name put in current folder. • PopFolder: goes back last folder structure added. • PushFile: opens file in current folder in provided openmode, openmode is a thin wrapper on C++ std::ios::openmode. • PopFile: closes current file. • Write: writes to current file.

Table 107. FileManager Class Description

Name
ISerializable
Description
Interface that any class that wants to be serialized has to implement, the class only provides two methods to be called by Serializable functions
Proposed attributes
None
Proposed methods.
• Serialize: Serializes data of class • Deserialize: Deserialize data of class

Table 108. ISerializable Class Description

Name
Serializable
Description
Collection of functions that provide serialization support
Proposed attributes
None
Proposed methods.
• SerializeData: provides flexible way to serialize data • DeserializeData: provides flexible way to deserialize data • SerializeStatic: serializes static data, uses SerializeData internally • DeserializeStatic: Deserializes static data, uses DeserializeData internally • SerializeArrayStoresStatic: Serializes static arrays, (no vectors, deques or lists) that contain static data (no strings, or custom classes that can vary in size) • DeserializeArrayStoresStatic: Deserializes static arrays, (no vectors, deques or lists) that contain static data (no strings, or custom classes that can vary in size) • Serialize: To be called in application, serializes provided ISerializable • Deserialize: To be called in application, deserializes provided ISerializable • SetSerializeFilename: Sets the filename to serialize to

Table 109. Serializable Class Description

Name
Context
Description
Internal struct used in FileManager
Proposed attributes
• folders: list of folders pushed to current path • root: current root • current_folder: current folder • current_file: current opened file • allocations: list of allocated file names • serialize_filename: name of serialization filename
Proposed methods.
None

Table 110. Context Class Description

5.5.2.6. Application

Name
Application
Description
This is the "main" class of this project, is the one that is run when starting the project and is the one that ties all other systems together.
Proposed attributes
None.
Proposed methods.
• Constructor: constructs class with given configuration. • Get: class is a singleton, returns instance. • GetDelta: returns time since last frame. • Error: puts error message on screen. • GetConsole: returns emulator. • SetUpdate: if set to false then nor emulator nor components will be updated each frame. • AddComponent: adds a component to the interface. • RemoveComponent: removes component from the interface. • Run: starts main loop of the application. • Close: closes application • RestartEmulator: restarts emulator. • GetScreenSize: returns screen size

Table 111. Application Class Description

Name
Application
Description
A struct representing the configuration of the application window.
Proposed attributes
• name: name of the application, will be passed down to the window. • w: width of the window. • h: height of the window.
Proposed methods.
None

Table 112. Configuration (Application) Class Description

Name
IComponent
Description
Abstract class like interface, classes that inherit this will be able to be put into the interface and interact with the application, will be used to show different “windows” or to display error messages.
Proposed attributes
• removed: if set to true, the component will be removed from the application on the end of this frame. • name: name of the component, must be unique
Proposed methods.
• Constructor: constructs the component with a name • OnCreate: called when a component is added to the application. • OnRender: called when the component is renderer. • OnUpdate: called when the component is updated. • operator==: overload of == with another component • operator==: overload of == with the name of another component

Table 113. IComponent Class Description

Name
CloseDialog
Description
Implementation of IComponent, used as a popup to display error messages, can be unrecoverable or recoverable.
Proposed attributes
Same as IComponent.
Proposed methods.
Same as IComponent. New: • Constructor: constructs with a name, an error message, and if the error is unrecoverable.

Table 114. CloseDialog Class Description

Name
MemoryView
Description
Implementation of IComponent, used to view the memory of the emulator (RF.6).
Proposed attributes
Same as IComponent.
Proposed methods.
Same as IComponent. New: • Constructor: constructs with a name, and a monospaced font to display the memory inspector.

Table 115. MemoryView Class Description

Name
ShowCPUStatus
Description
Implementation of IComponent, shows the status of the CPU (RF.5)
Proposed attributes
Same as IComponent.
Proposed methods.
Same as IComponent. New: • Constructor: constructs with a name, and a monospaced font to display information.

Table 116. ShowCPUStatus Class Description

Name
ShowPPUStatus
Description
Implementation of IComponent, shows the status of the PPU (RF.4).
Proposed attributes
Same as IComponent.
Proposed methods.
Same as IComponent. New: • Constructor: constructs with a name, and a monospaced font to display information.

Table 117. ShowPPUStatus Class Description

5.6. Asi 8: Defining User Interfaces

All images shown that contain a loaded ROM are from GPL-3.0 licensed game thwaite by Damian Yerrick (26), this emulator will not shown materials under copyright of any company.

5.6.1. Interface Description

5.6.1.1. Main screen

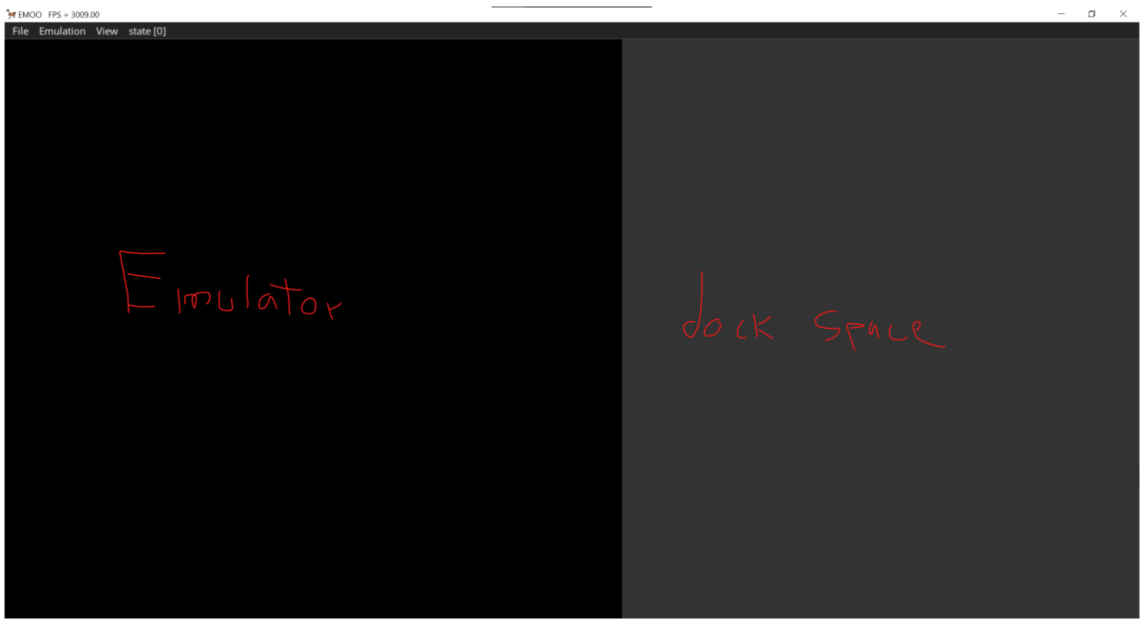


Figure 23. Interface: Main Screen

The main screen has three parts, two of which are always visible, the first visible part is the menu on top, it has three buttons, File, Emulation and View, their functions will be shown in a moment; the other always visible part is the emulator screen, the black square, the emulator will display to that; the part that is not always visible is the dock space, the dock space is the part of the screen where the components go by default.

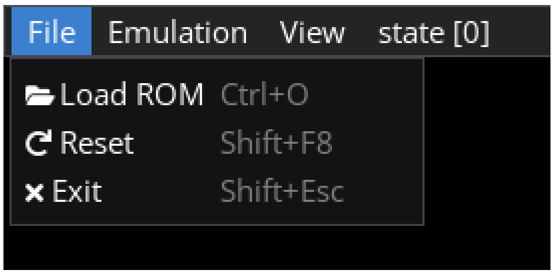


Figure 24. Interface: File Menu

The file menu has the load ROM, reset, and exit capabilities, exit has not been described until now, but it closes the application, it acts as another way to close the application.

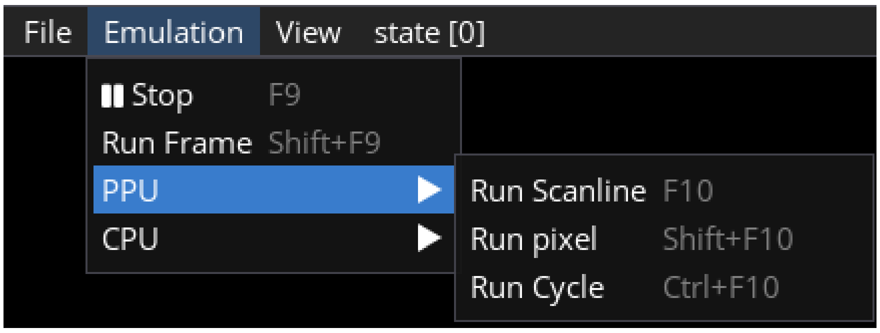


Figure 25. Interface: Emulation Menu 1

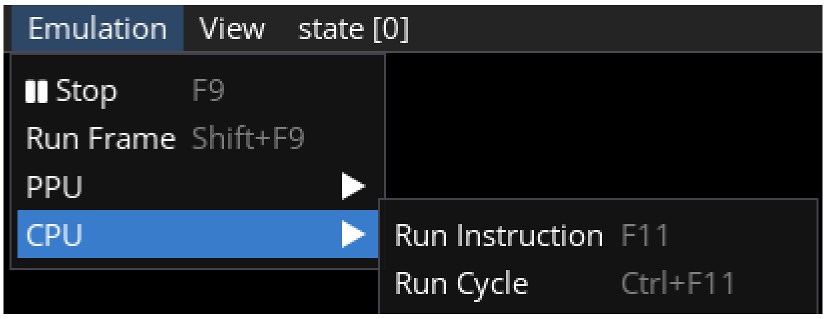


Figure 26. Interface: Emulation Menu 2

The emulation button has the advance emulation controls, if the emulator is stopped, the Stop button is converted to a Run button.

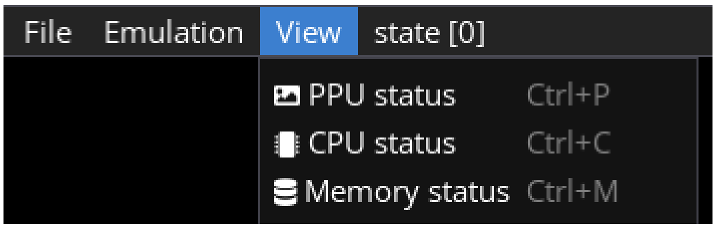


Figure 27. Interface: View Menu

The buttons in the view lets the user show or hide the different elements of the UI.

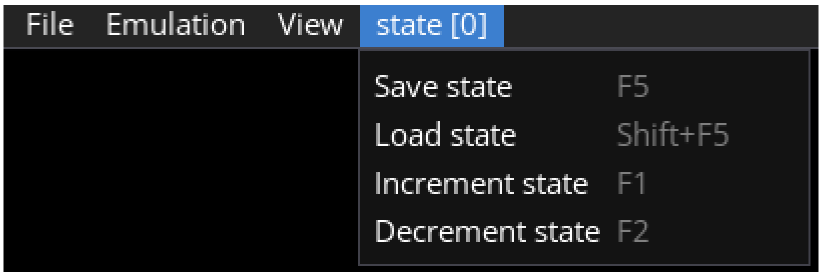
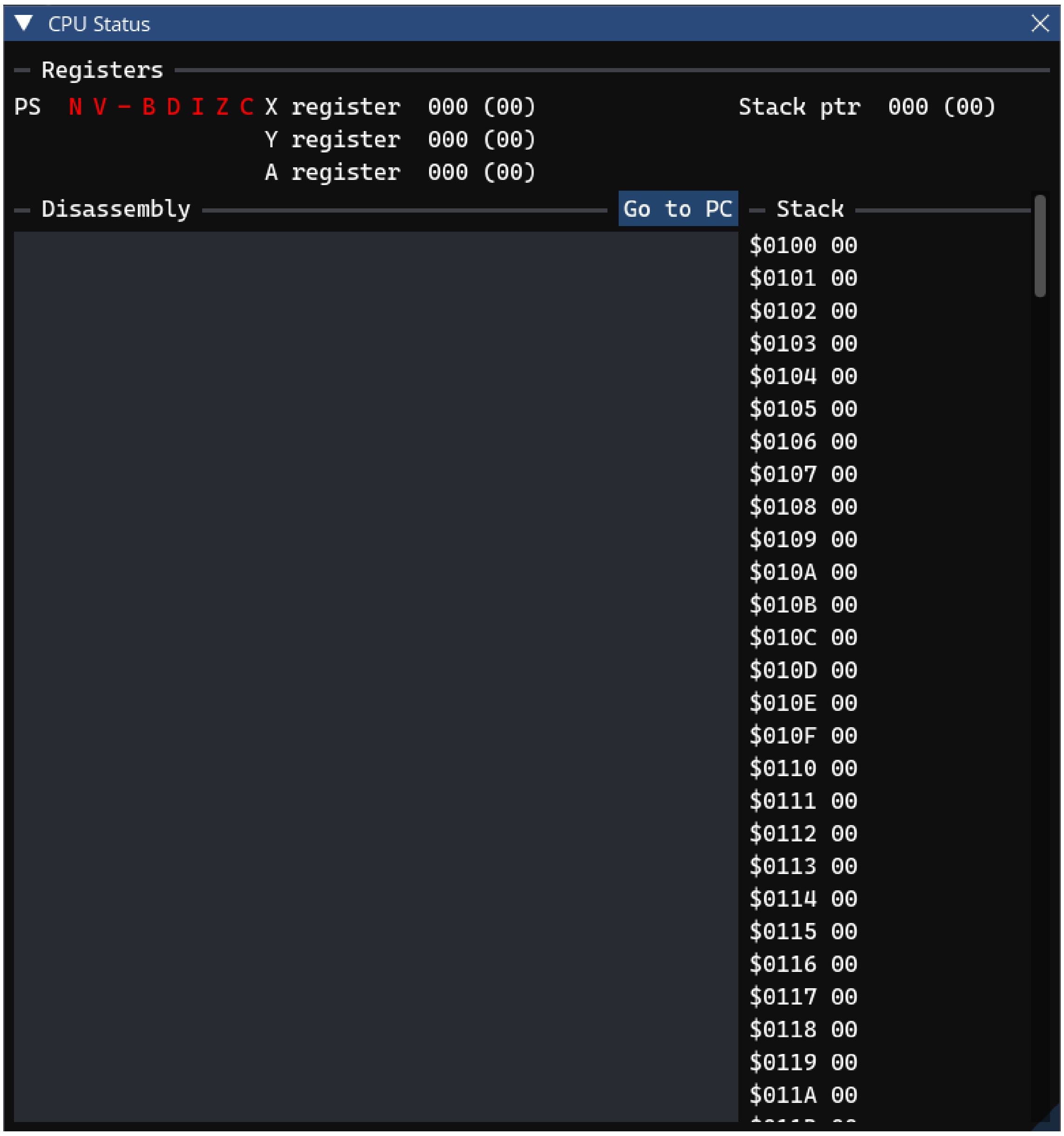


Figure 28. Interface: State Menu

The buttons in the state menu lets the user save, load, decrement and increment state.

5.6.1.2. CPU status

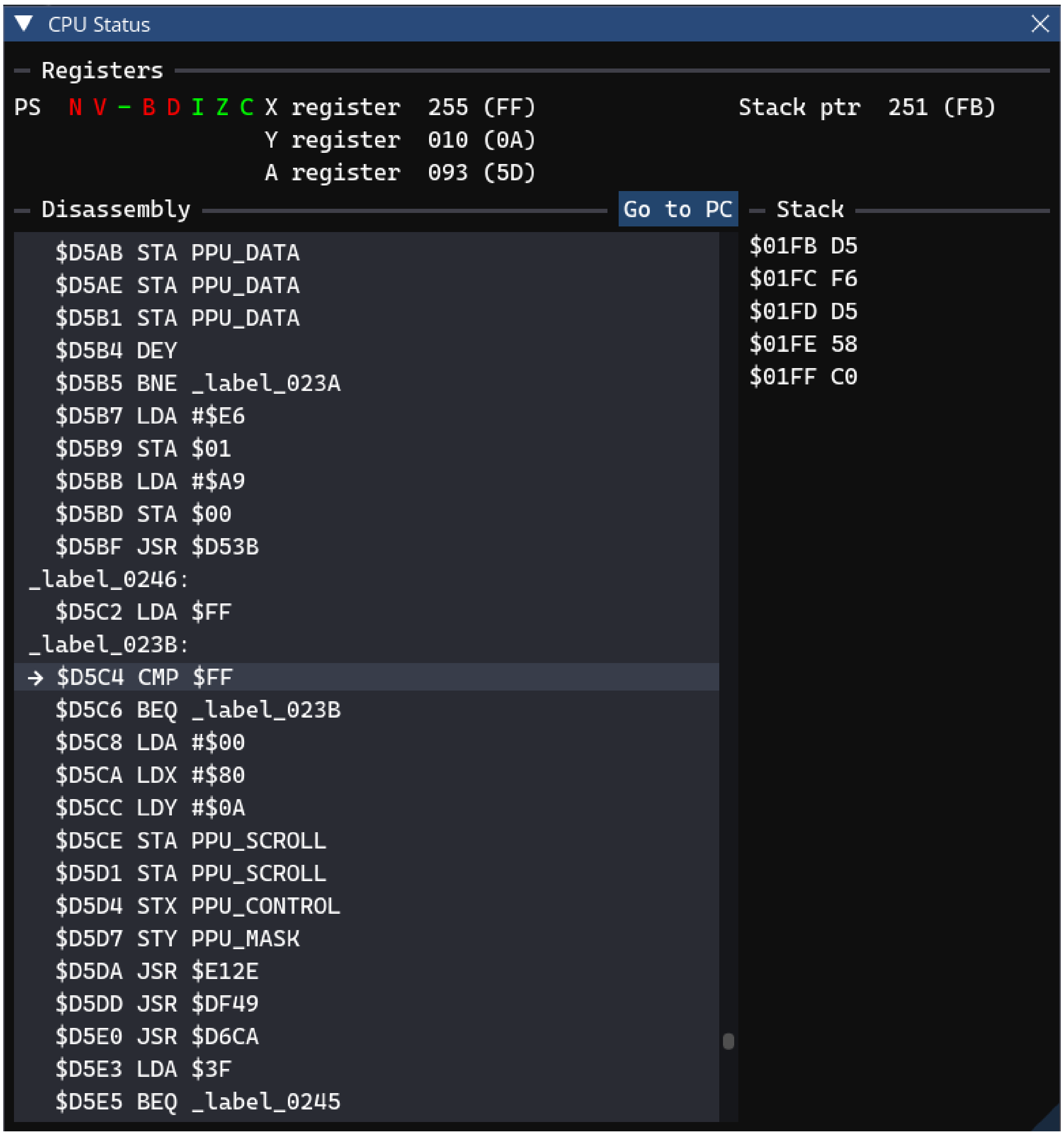


```
PS  N V - B D I Z C X register  000 (00)          Stack ptr  000 (00)
                    Y register  000 (00)
                    A register  000 (00)
```

```
$0100 00
$0101 00
$0102 00
$0103 00
$0104 00
$0105 00
$0106 00
$0107 00
$0108 00
$0109 00
$010A 00
$010B 00
$010C 00
$010D 00
$010E 00
$010F 00
$0110 00
$0111 00
$0112 00
$0113 00
$0114 00
$0115 00
$0116 00
$0117 00
$0118 00
$0119 00
$011A 00
```

Figure 29. Interface: CPU status 1

The CPU status component, show all data regarding the CPU. The image shown above is without a ROM loaded.



```
PS  N V - B D I Z C X register  255 (FF)        Stack ptr  251 (FB)
                    Y register  010 (0A)
                    A register  093 (5D)
```

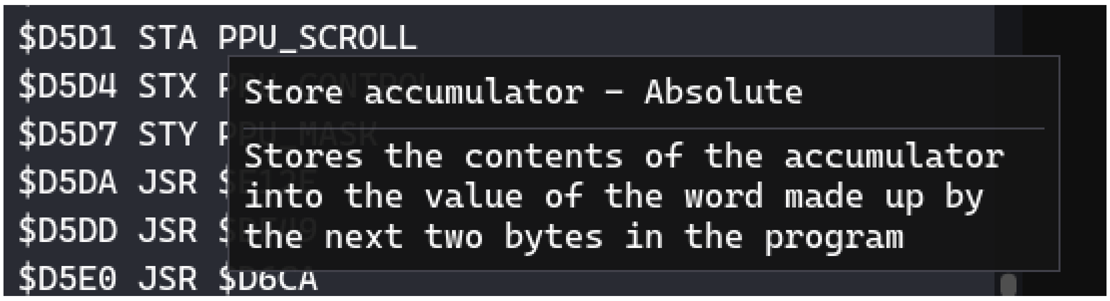
```
   $D5AB STA PPU_DATA
   $D5AE STA PPU_DATA
   $D5B1 STA PPU_DATA
   $D5B4 DEY
   $D5B5 BNE _label_023A
   $D5B7 LDA #$E6
   $D5B9 STA $01
   $D5BB LDA #$A9
   $D5BD STA $00
   $D5BF JSR $D53B
 _label_0246:
   $D5C2 LDA $FF
 _label_023B:
 → $D5C4 CMP $FF
   $D5C6 BEQ _label_023B
   $D5C8 LDA #$00
   $D5CA LDX #$80
   $D5CC LDY #$0A
   $D5CE STA PPU_SCROLL
   $D5D1 STA PPU_SCROLL
   $D5D4 STX PPU_CONTROL
   $D5D7 STY PPU_MASK
   $D5DA JSR $E12E
   $D5DD JSR $DF49
   $D5E0 JSR $D6CA
   $D5E3 LDA $3F
   $D5E5 BEQ _label_0245
```

```
$01FB D5
$01FC F6
$01FD D5
$01FE 58
$01FF C0
```

Figure 30. Interface: CPU status 2

The image presented above is with a ROM loaded, the Go to PC button allows the user to go back to the current value of the program counter when scrolling, the disassembly window can only be scrolled when the emulator is stopped.

If the user hovers a line of the disassembly that is not a label while the emulator is stopped, a tooltip will be shown explaining the name of the instruction, and what it does.

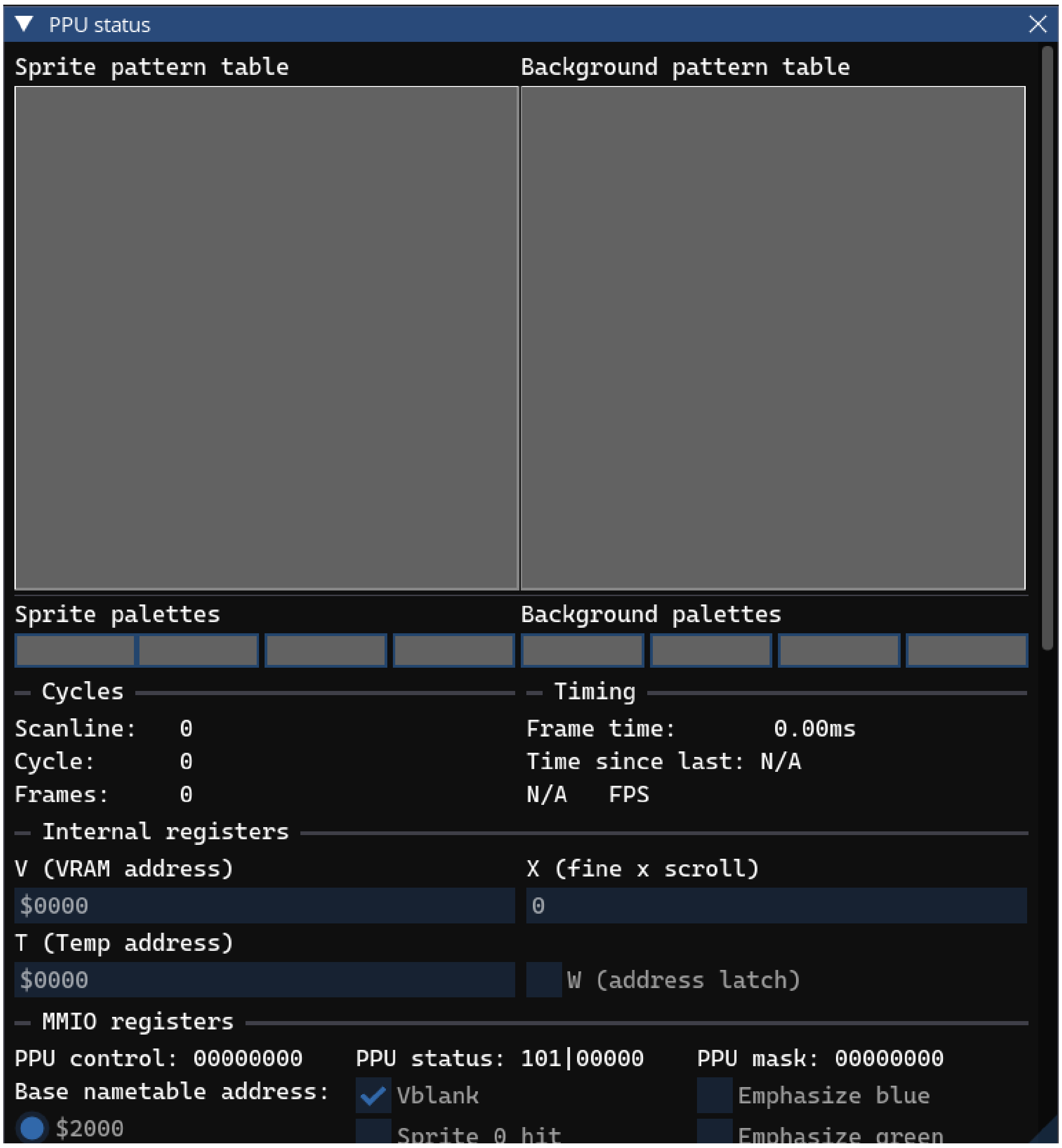


```
$D5D1 STA PPU_SCROLL
$D5D4 STX P
$D5D7 STY P
$D5DA JSR $
$D5DD JSR $
$D5E0 JSR $D6CA
```

```
Store accumulator - Absolute

Stores the contents of the accumulator
into the value of the word made up by
the next two bytes in the program
```

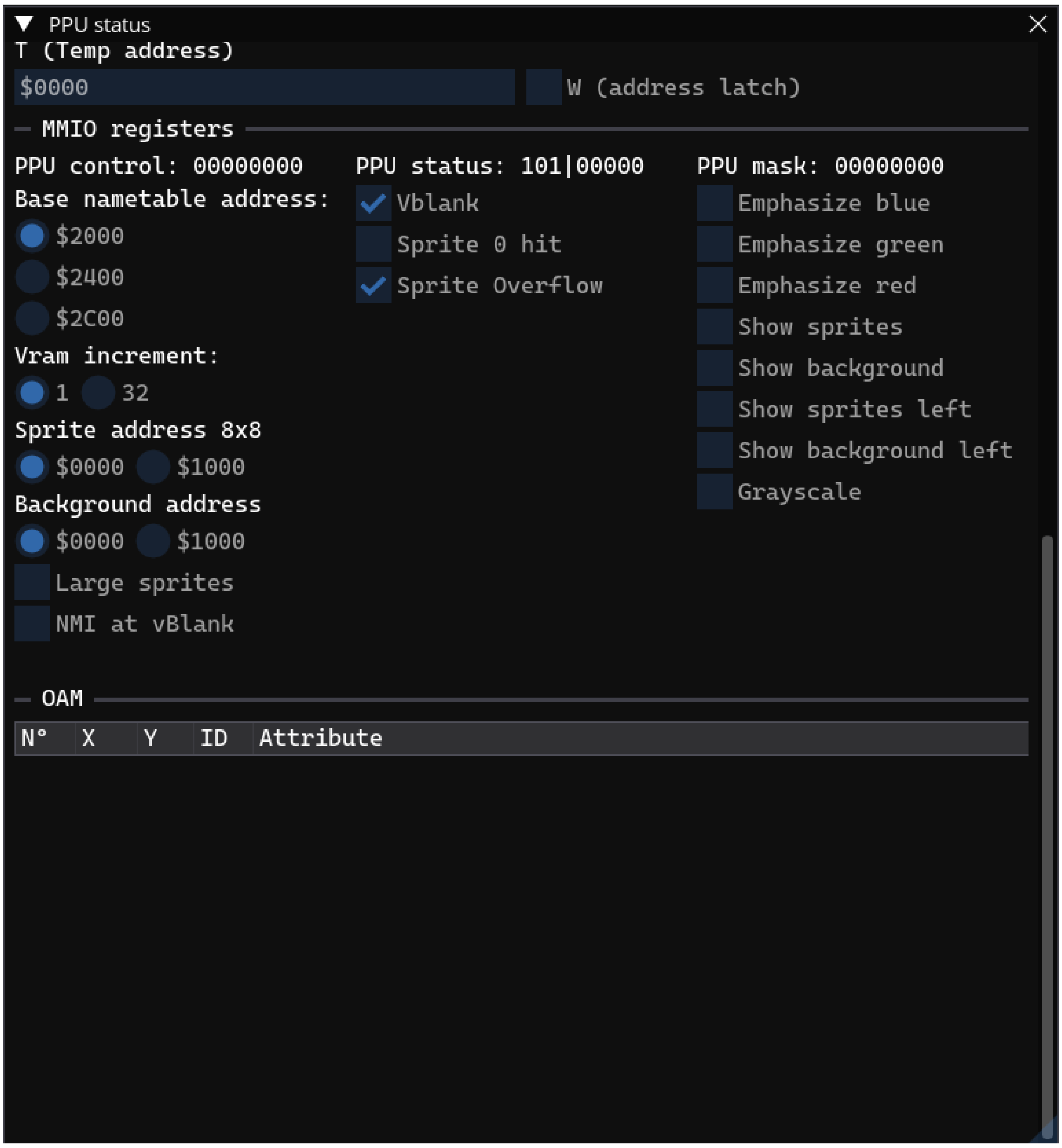
Figure 31. Interface: Disassembly tooltip

5.6.1.3. PPU Status



```
PPU status

Sprite pattern table              Background pattern table

Sprite palettes                   Background palettes

— Cycles                          — Timing
Scanline:    0                    Frame time:        0.00ms
Cycle:       0                    Time since last: N/A
Frames:      0                    N/A    FPS
— Internal registers
V (VRAM address)                  X (fine x scroll)
$0000                             0
T (Temp address)
$0000                             W (address latch)
— MMIO registers
PPU control: 00000000    PPU status: 101|00000    PPU mask: 00000000
Base nametable address:  ✓ Vblank                 Emphasize blue
● $2000                    Sprite 0 hit           Emphasize green
```

Figure 32. Interface: PPU status 1



PPU status

T (Temp address)

$0000 [ ] W (address latch)

MMIO registers

PPU control: 00000000
Base nametable address:
[● $2000] $2400 $2C00
Vram increment:
[● 1] 32
Sprite address 8x8
[● $0000] $1000
Background address
[● $0000] $1000
[ ] Large sprites
[ ] NMI at vBlank

PPU status: 101|00000
[✓] Vblank
[ ] Sprite 0 hit
[✓] Sprite Overflow

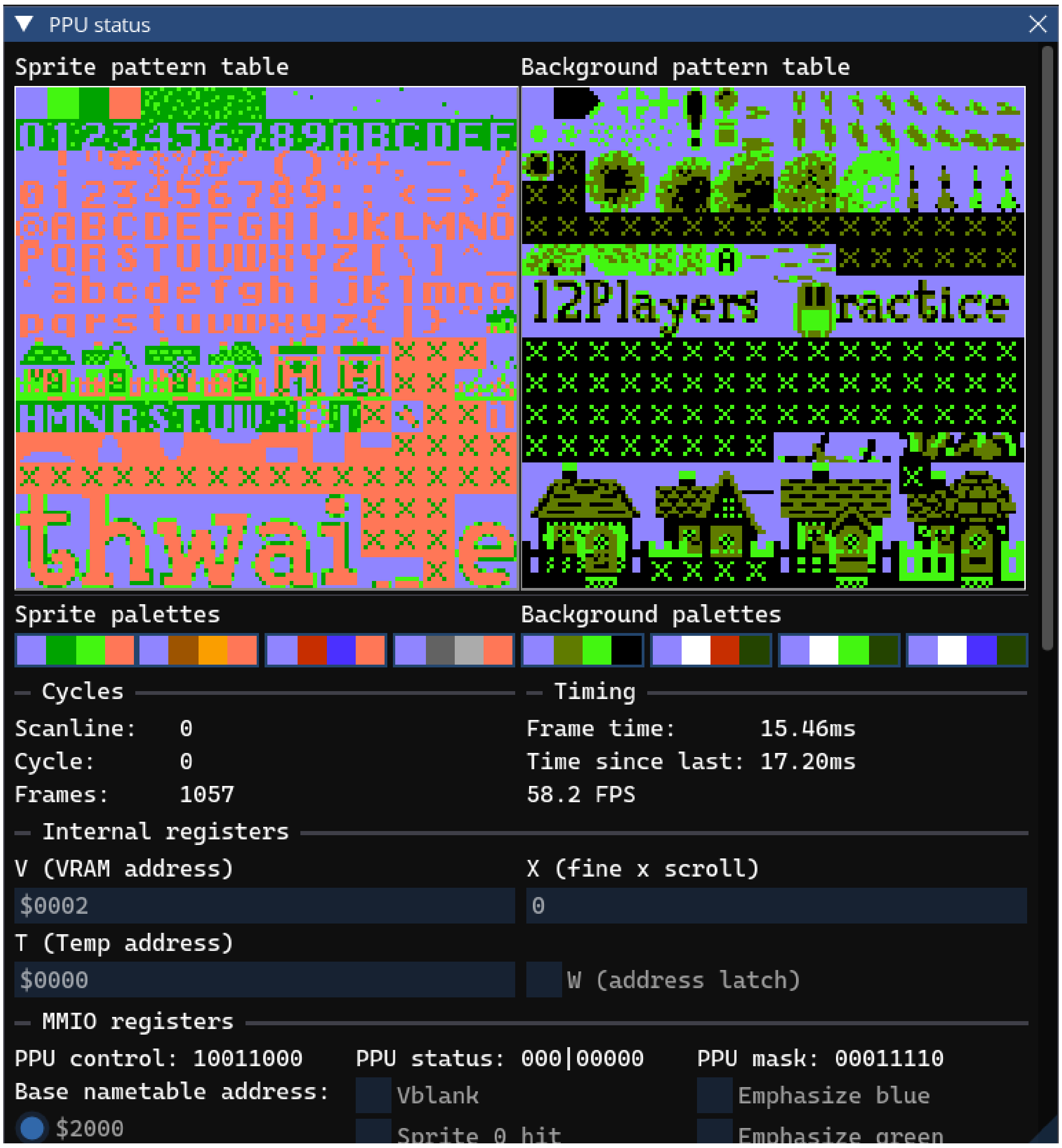
PPU mask: 00000000
[ ] Emphasize blue
[ ] Emphasize green
[ ] Emphasize red
[ ] Show sprites
[ ] Show background
[ ] Show sprites left
[ ] Show background left
[ ] Grayscale

OAM

N°	X	Y	ID	Attribute

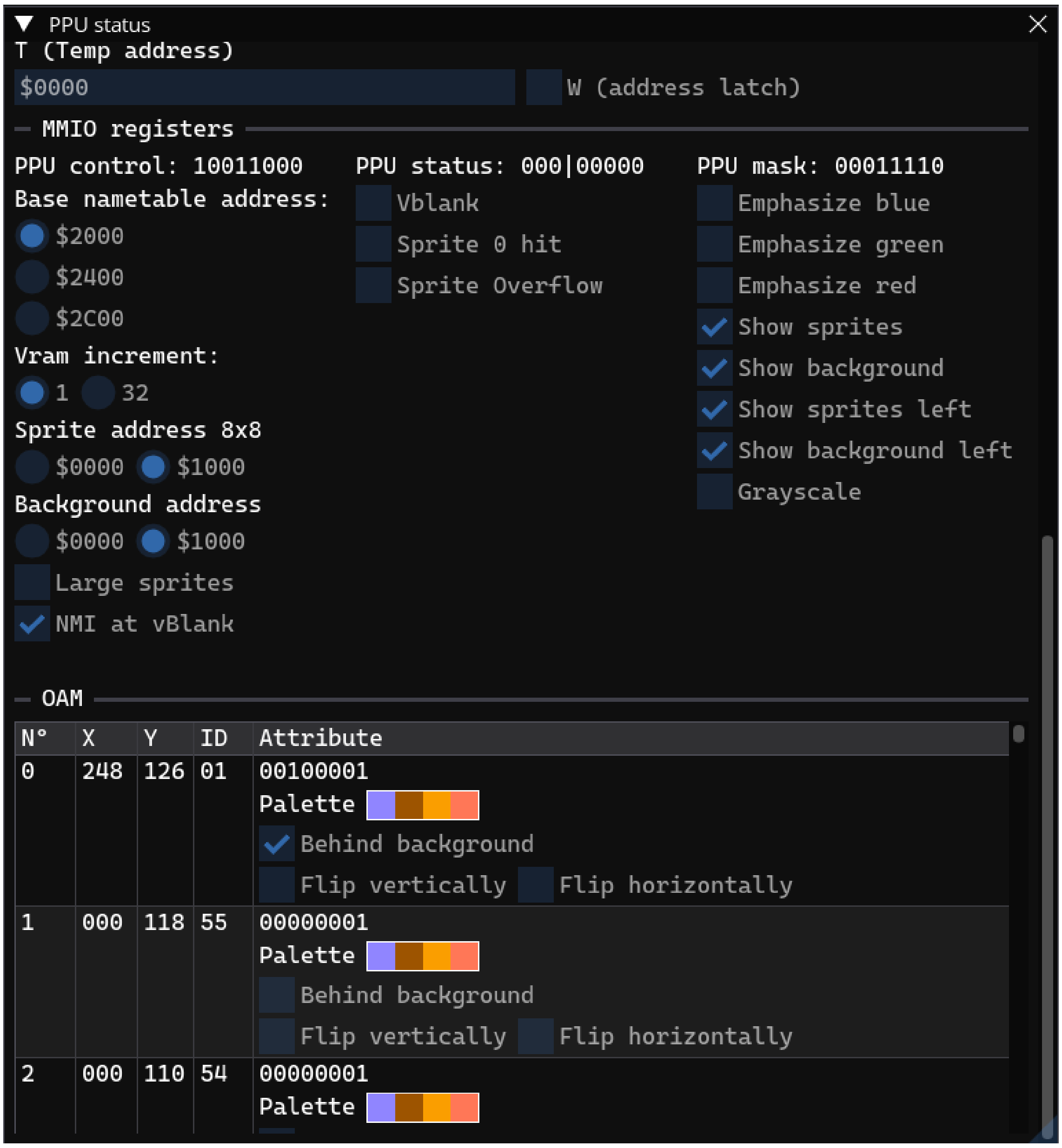
Figure 33. Interface: PPU status 1 cont.

The images show above are the UI of PPU status when no ROM is laoded



```
Cycles
Scanline:    0
Cycle:       0
Frames:      1057

Timing
Frame time:       15.46ms
Time since last:  17.20ms
58.2 FPS

Internal registers
V (VRAM address)            X (fine x scroll)
$0002                       0
T (Temp address)
$0000                       W (address latch)

MMIO registers
PPU control: 10011000    PPU status: 000|00000    PPU mask: 00011110
Base nametable address:  Vblank                   Emphasize blue
$2000                    Sprite 0 hit             Emphasize green
```

Figure 34. PPU status 2



PPU status

T (Temp address)

$0000 ☐ W (address latch)

MMIO registers

PPU control: 10011000

Base nametable address:
- [✓ $2000]
- $2400
- $2C00

Vram increment:
[✓ 1] 32

Sprite address 8x8
$0000 [✓ $1000]

Background address
$0000 [✓ $1000]

☐ Large sprites
✓ NMI at vBlank

PPU status: 000|00000
- ☐ Vblank
- ☐ Sprite 0 hit
- ☐ Sprite Overflow

PPU mask: 00011110
- ☐ Emphasize blue
- ☐ Emphasize green
- ☐ Emphasize red
- ✓ Show sprites
- ✓ Show background
- ✓ Show sprites left
- ✓ Show background left
- ☐ Grayscale

OAM

N°	X	Y	ID	Attribute
0	248	126	01	00100001 Palette 🟪🟫🟧🟥 ✓ Behind background ☐ Flip vertically ☐ Flip horizontally
1	000	118	55	00000001 Palette 🟪🟫🟧🟥 ☐ Behind background ☐ Flip vertically ☐ Flip horizontally
2	000	110	54	00000001 Palette 🟪🟫🟧🟥

Figure 35. Interface: PPU Status 2 cont.

The other two images are when a ROM is loaded. There are three distinct parts on this interface, the first part is the pattern tables and the palettes, these show what sprites and palettes are currently loaded by the ROM, a user can click in any part of a pattern table to see its pattern index

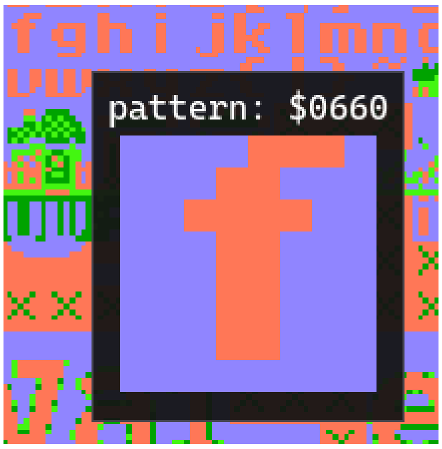


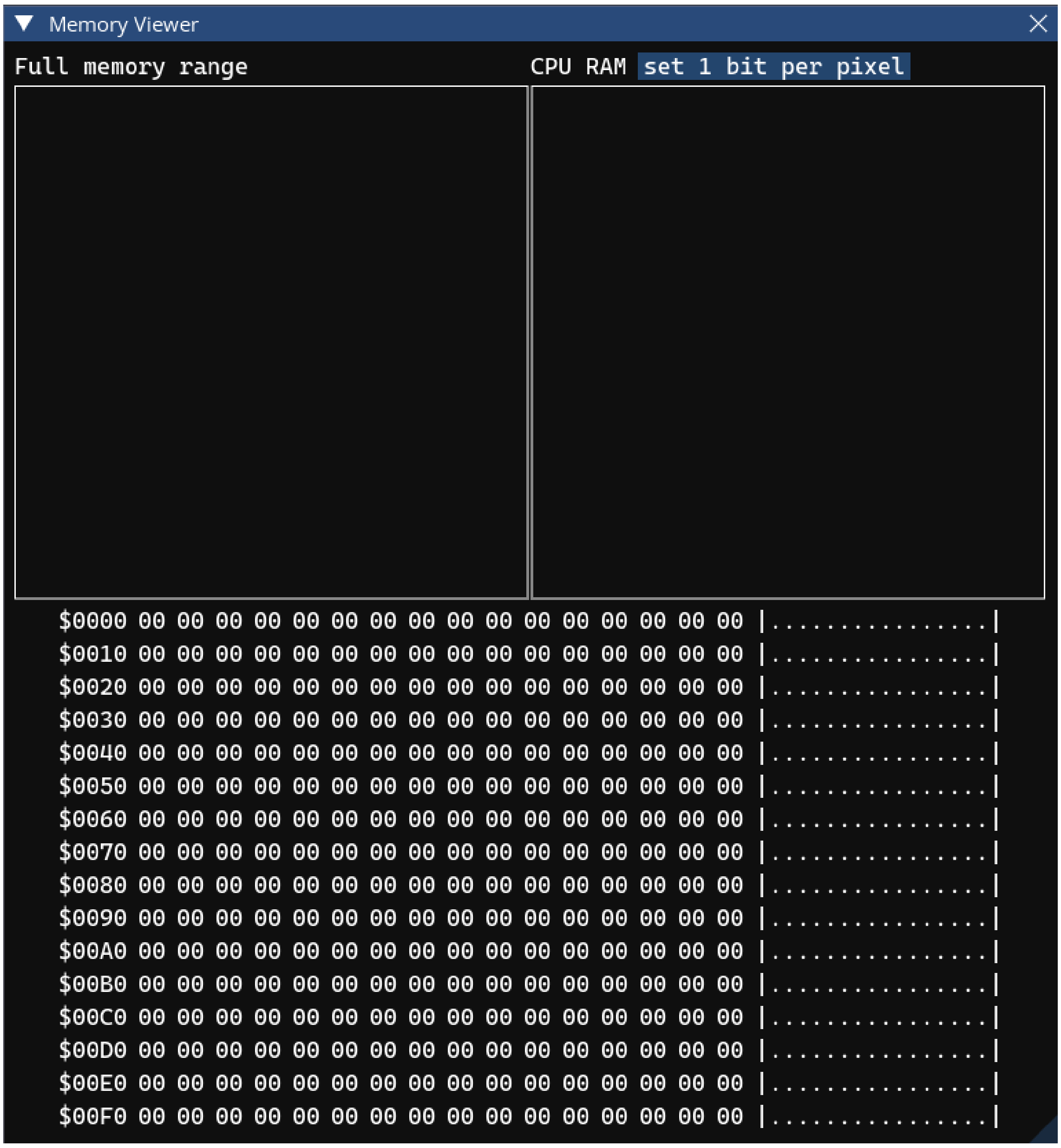
Figure 36. Interface: User holding click on pattern

The user can also click on the palettes to swap the palette of the pattern table above, can only use sprite palettes with sprite patterns and background palettes with background patterns.

The second part of this interface is where the bulk of the information is, like timing information, internal registers, and MMIO registers. None of this is interactable in any way, and its only there to show information.

The last part is the OAM, OAM means Object Attribute Memory, but we call them sprites, a OAM, this only shows the loaded sprites, be it 1 or 64, the user can scroll this part to see more.

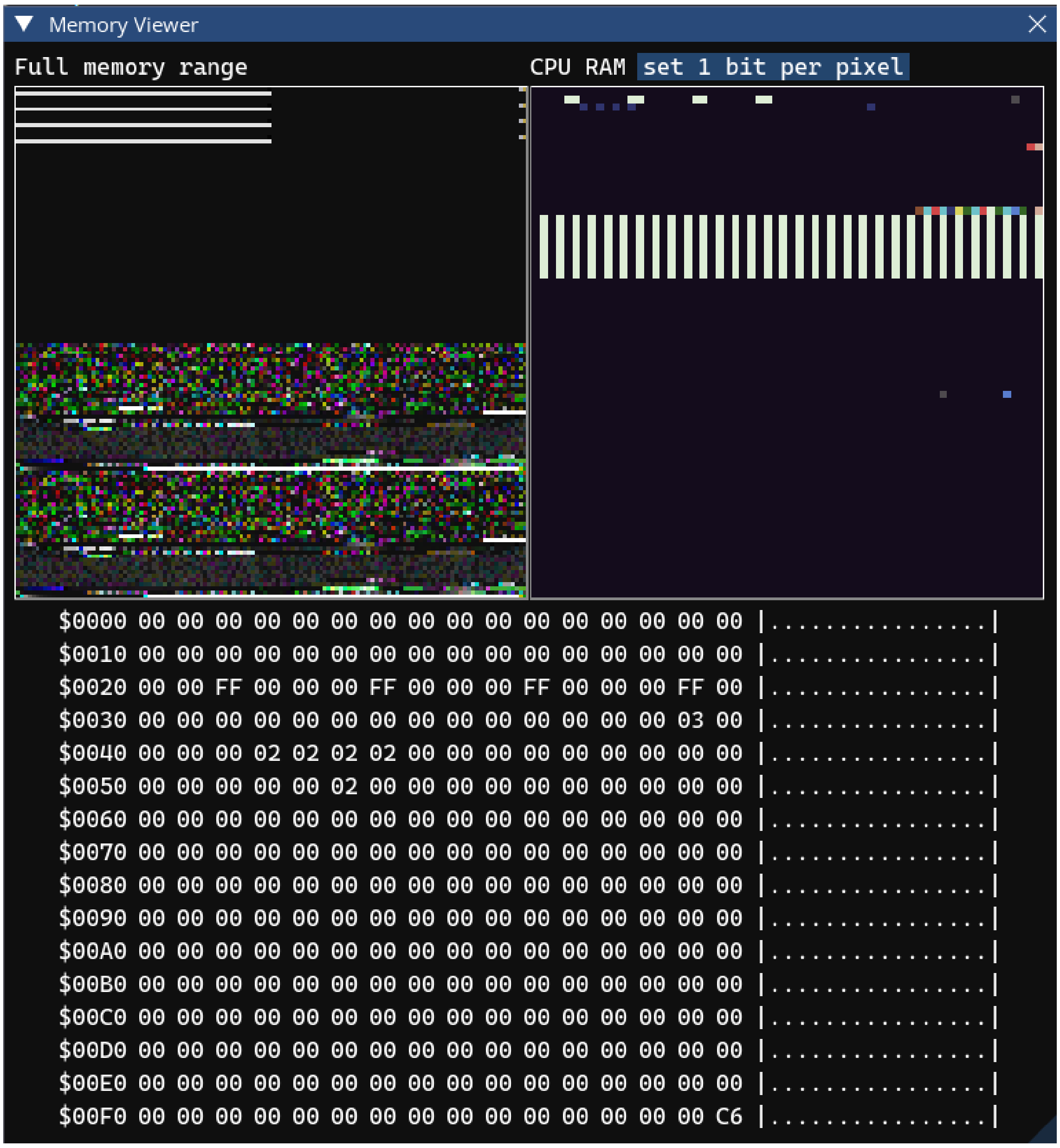
5.6.1.4. Memory Status



```
$0000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$00A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$00B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$00C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$00D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$00E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$00F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
```

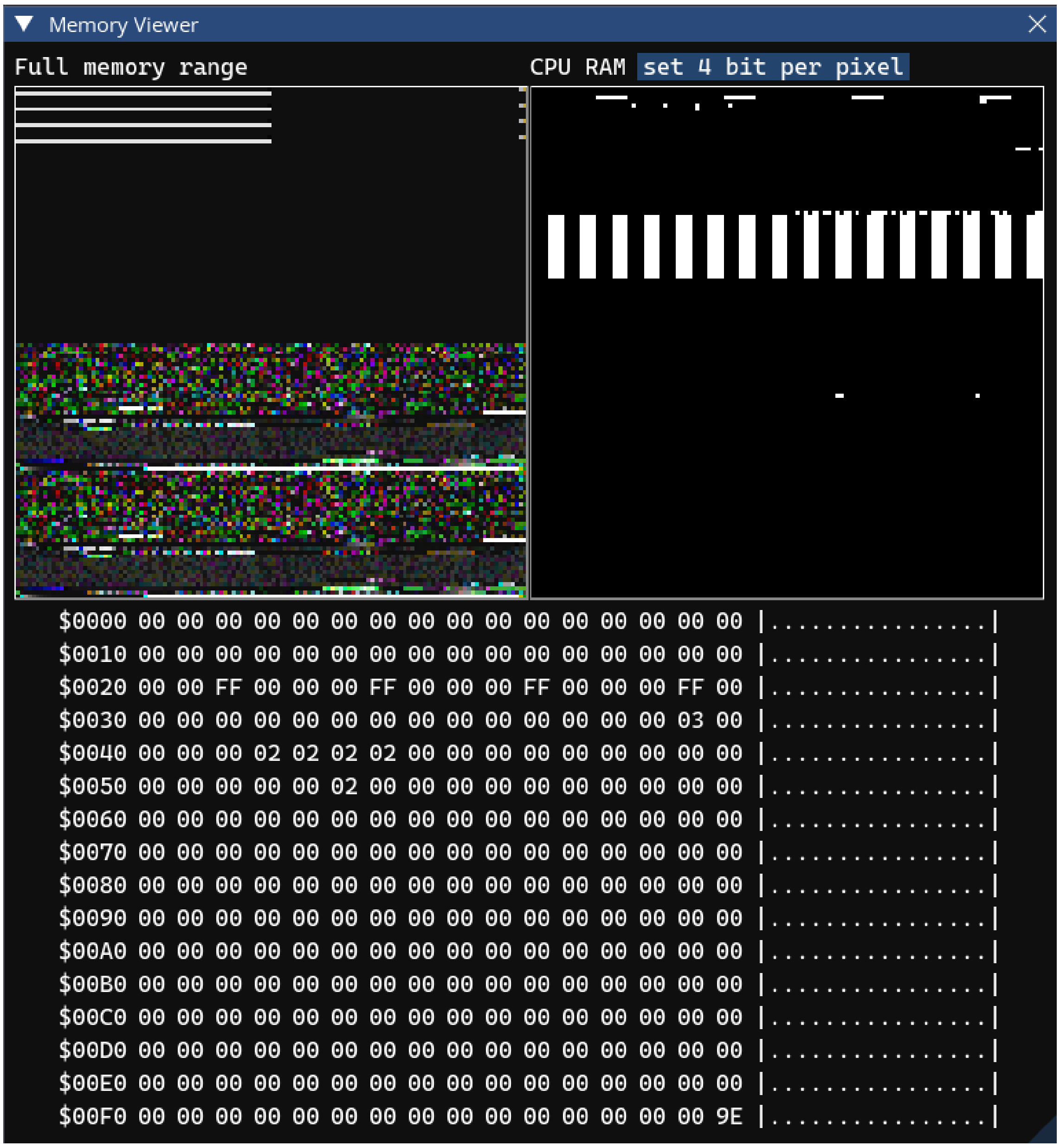
Figure 37. Interface: Memory Status 1

The image presented above is how the memory status looks when no ROM is loaded.



```
$0000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0020 00 00 FF 00 00 00 FF 00 00 00 FF 00 00 00 FF 00 |................|
$0030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03 00 |................|
$0040 00 00 00 02 02 02 02 00 00 00 00 00 00 00 00 00 |................|
$0050 00 00 00 00 00 02 00 00 00 00 00 00 00 00 00 00 |................|
$0060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$00A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$00B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$00C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$00D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$00E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$00F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 C6 |................|
```

Figure 38. Interface: Memory Status 2 4 bits per pixel



```
$0000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0020 00 00 FF 00 00 00 FF 00 00 00 FF 00 00 00 FF 00 |................|
$0030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03 00 |................|
$0040 00 00 00 02 02 02 02 00 00 00 00 00 00 00 00 00 |................|
$0050 00 00 00 00 00 02 00 00 00 00 00 00 00 00 00 00 |................|
$0060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$00A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$00B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$00C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$00D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$00E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$00F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 9E |................|
```

Figure 39. Interface: Memory Status 2 1 bit per pixel

The two images shown above this paragraph are shown when a ROM is loaded and the respective bits per pixel are set for the CPU RAM, 4 bits per pixel uses 16 colours, and uses the dawnbringer 16 colour palette (27).

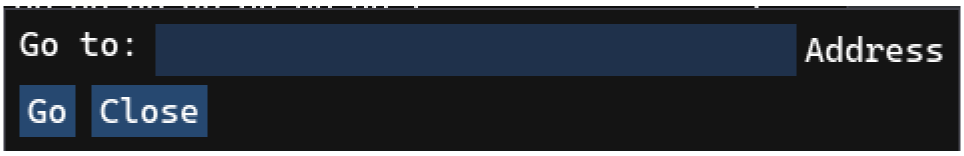


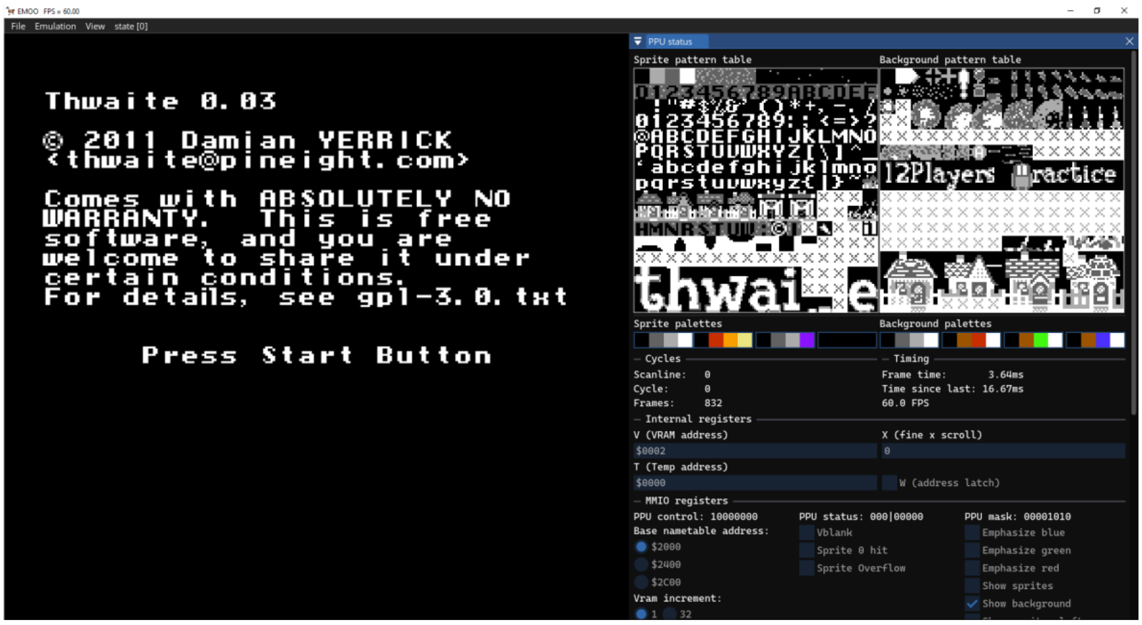
Figure 40. Interface: Search Address

This dialog appears when the user right clicks the memory inspector, the dialog in question allows the user to go to any memory address they wish if it's a valid address; when the address is selected, it will be highlighted for a few moments. The user can also scroll the inspector with their mouse wheel.

5.6.2. Description of the Interface Behaviour

5.6.2.1. Docks and Viewports

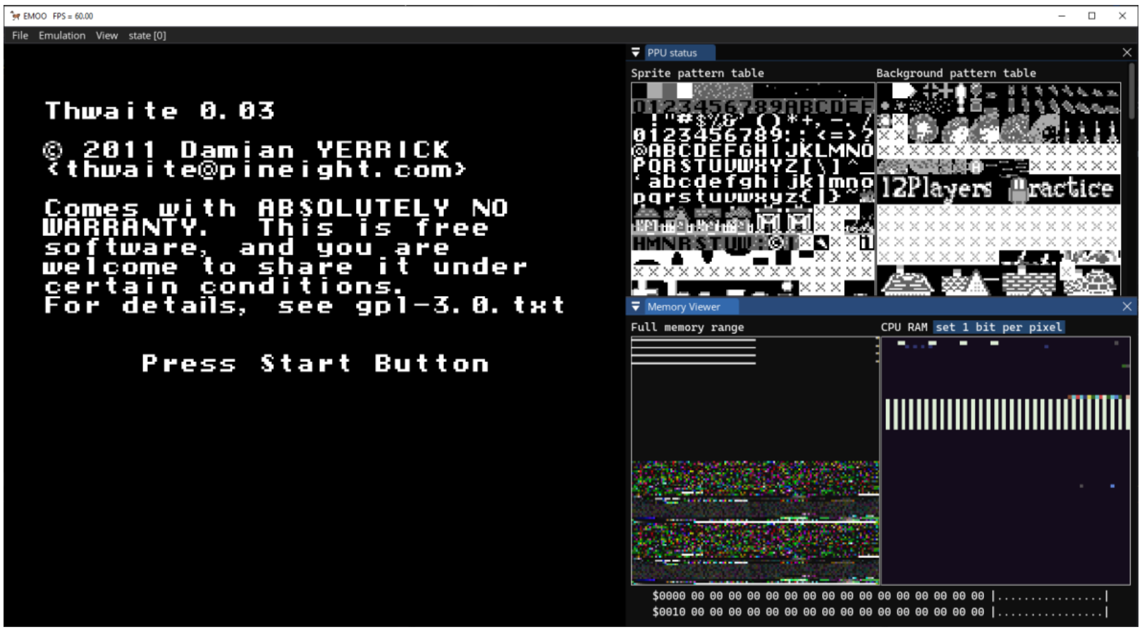
The interface is made with the concept of docking and viewports in mine, that means, that the application has a dock space, and components are put in the dock space by default, but components can be undocked and put on other screens or docked into each other to enable user customization.



```
Thwaite 0.03

© 2011 Damian YERRICK
<thwaite@pineight.com>

Comes with ABSOLUTELY NO
WARRANTY.  This is free
software, and you are
welcome to share it under
certain conditions.
For details, see gpl-3.0.txt

Press Start Button
```

```
— Cycles                          — Timing
Scanline:   0                     Frame time:       3.64ms
Cycle:      0                     Time since last: 16.67ms
Frames:     832                   60.0 FPS
— Internal registers
V (VRAM address)                  X (fine x scroll)
$0002                             0
T (Temp address)
$0000                             W (address latch)
— MMIO registers
PPU control: 10000000   PPU status: 000|00000   PPU mask: 00001010
Base nametable address: Vblank                  Emphasize blue
● $2000                 Sprite 0 hit            Emphasize green
$2400                   Sprite Overflow         Emphasize red
$2C00                                           Show sprites
Vram increment:                                 ✓ Show background
● 1  32
```

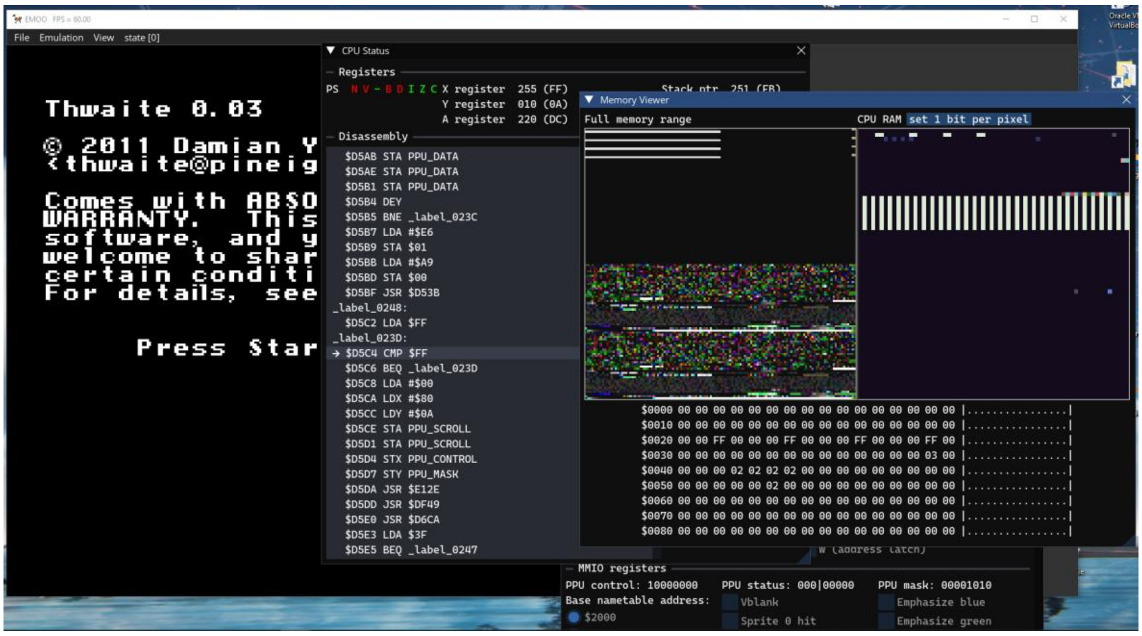
Figure 41. Components tabbed in dock space



```
Thwaite 0.03

© 2011 Damian YERRICK
<thwaite@pineight.com>

Comes with ABSOLUTELY NO
WARRANTY.  This is free
software, and you are
welcome to share it under
certain conditions.
For details, see gpl-3.0.txt

Press Start Button
```

```
$0000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
```

Figure 42. Components Docked onto each other



```
Registers
PS  N V - B D I Z C X register  255 (FF)
                    Y register  010 (0A)
                    A register  220 (DC)
Disassembly
  $D5AB STA PPU_DATA
  $D5AE STA PPU_DATA
  $D5B1 STA PPU_DATA
  $D5B4 DEY
  $D5B5 BNE _label_023C
  $D5B7 LDA #$E6
  $D5B9 STA $01
  $D5BB LDA #$A9
  $D5BD STA $00
  $D5BF JSR $D53B
_label_0248:
  $D5C2 LDA $FF
_label_023D:
→ $D5C4 CMP $FF
  $D5C6 BEQ _label_023D
  $D5C8 LDA #$00
  $D5CA LDX #$80
  $D5CC LDY #$0A
  $D5CE STA PPU_SCROLL
  $D5D1 STA PPU_SCROLL
  $D5D4 STX PPU_CONTROL
  $D5D7 STY PPU_MASK
  $D5DA JSR $E12E
  $D5DD JSR $DF49
  $D5E0 JSR $D6CA
  $D5E3 LDA $3F
  $D5E5 BEQ _label_0247
```

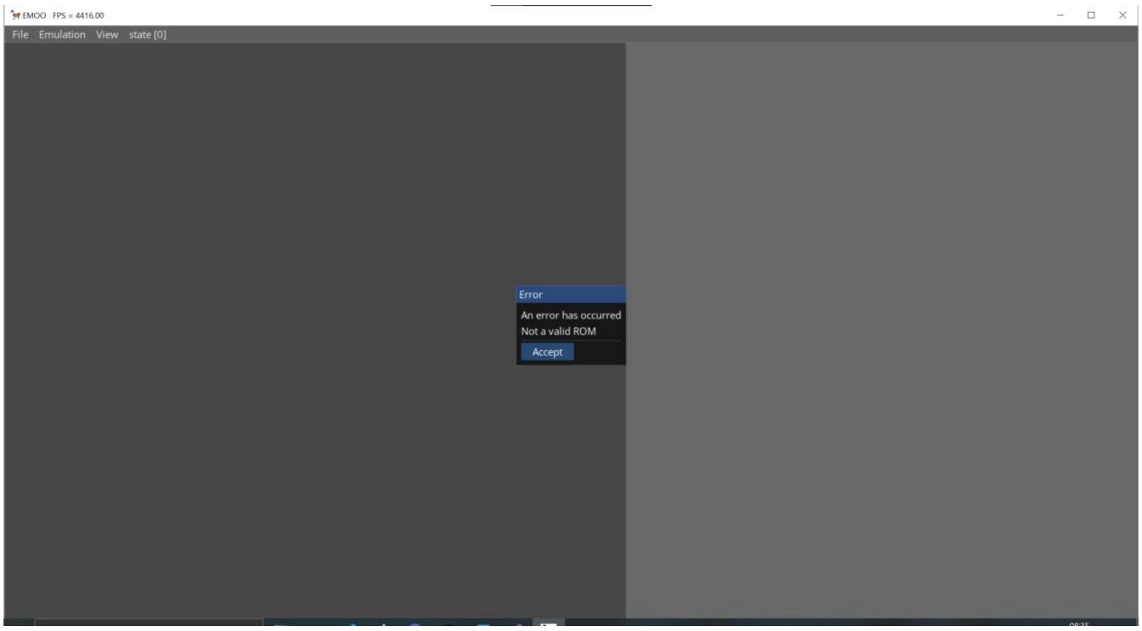
```
$0000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0020 00 00 FF 00 00 00 FF 00 00 00 FF 00 00 00 FF 00 |................|
$0030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03 00 |................|
$0040 00 00 00 02 02 02 02 00 00 00 00 00 00 00 00 00 |................|
$0050 00 00 00 00 00 02 00 00 00 00 00 00 00 00 00 00 |................|
$0060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
```

```
MMIO registers
PPU control: 10000000    PPU status: 000|00000    PPU mask: 00001010
Base nametable address:  Vblank                   Emphasize blue
● $2000                  Sprite 0 hit             Emphasize green
```

Figure 43. Components free on the screen

5.6.2.2. Error messages.

If an error message occurs, be it invalid ROM or STP opcode reached, an error message will be shown, if the error is unrecoverable, the application will be closed, if the error is recoverable, the application will be stop while the error message is on screen.



An error has occurred
Not a valid ROM

Figure 44. Error Message

5.6.3. Navigability Diagram

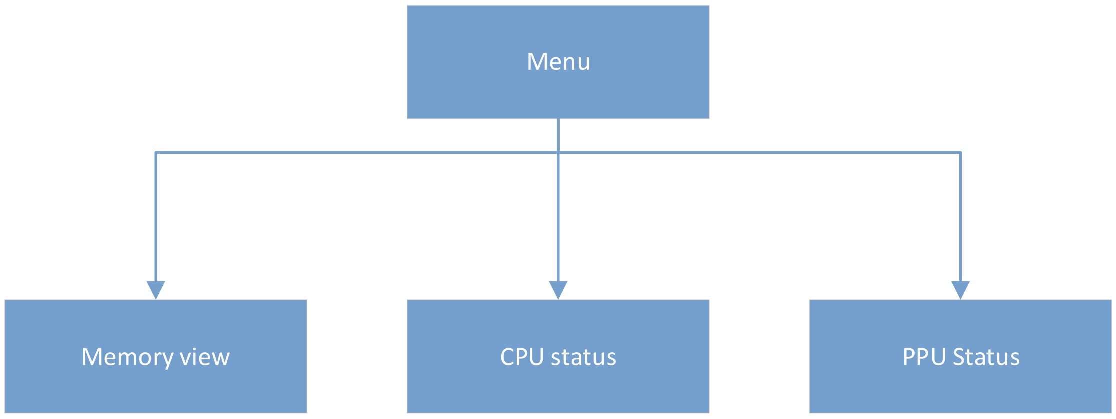


Figure 45. Navigability Diagram

All parts of the interface are accessible at any time using the menu on top of the application.

5.7. ASI 10: Testing Plan Identification

The tests in this project are quite unique, since we will be using ROMs to do system and integration testing, and normal Unit Testing to test the CPU.

Only the CPU contains unit tests, since it's the most important part of the project, the 'brain' so to speak; every single instruction has been tested, and every single addressing mode has been tested, but not every single combination of both has been tested, this is because every single type of addressing mode is equal in every opcode that makes use of it, since they use the same function call and testing them each time would be redundant.

To run the tests, the startup project must be changed from Application to NesEmu-Test, and MSVC integration must not be used, since it would break project generation as it uses some external scripting to patch it. Once the startup project is set, just press F5 or run it.

The tests are made as follow, CPU tests are unit tests, PPU tests are integration tests, and ROM tests are integration tests, the ROM tests are made with the NesTest ROM (28). There are more unit tests, like the tests made to the disassembler or the save-states, but those are less thorough than the CPU ones.

Other testing was made 'on the field' that means, testing notorious ROMs with known bugs, like green sky in Super Mario Bros, or any other hard mapper 0 ROM; as this is an emulator, if there are some code in bug, it would be extremely easy to spot since the game will look extremely off, this includes (bugs found in development) mangled sprites, a continuously scrolling screen, black screen, colour palette out of place...

Other Systems have not been tested, this includes parts of the project that are not the focus of this work, like the Renderer, Window System, Input Handler and interface.

6. Information System Design (ISD)

6.1. ISD 4: Class Design

This section contains a more detailed description of the class design, and how that design came to be.

I'll preface this section with general ideas on the design of this application.

First, this application is completely single threaded, since the performance goal I wanted was easily achievable with one thread.

Second, I really dislike DLLs, so I had to recompile a lot of libraries to use them as static libraries. This was to achieve a single executable file.

As I want to have a single executable file, I don't want to have other files around, like files containing fonts, texts, shaders or colour palettes; to achieve this, I used different embedding techniques, from compiling shaders and fonts to a file and then feeding that file to the compiler, from putting texts in a unordered map in a header file to be used by the application, to putting palettes in an array to be accessed when needed. Each strategy has their reason to exist, the palettes where put in arrays since they were constant and already defined, fonts where put in an ".embed" file, that was a header, since I used a tool called font-to-cpp, texts where compiled into a map with a python script, and that file was a header since it could be accessed by different cpp files and I did not want to deal with redefinitions, and the shaders where compiled to a cpp during their compilation step and they were loaded by engine using extern since they were only used there.

The only files this application produces are from imgui, a file that stores user preferences, and I did not want to remove it, its stored in %appdata%/EMOO; and the files containing the serialization data for the save states.

6.1.1. Emulator

Before starting this section, it must be noted that this is an LLE or Low-Level Emulator, as this emulator is done by individually implementing the necessary components and CPU instructions. An HLE or High-Level Emulator would intercept and execute system calls sent by the game.

The architecture of the emulator is based on a fetch decode execute loop that uses a jump table. The different parts of the system can speak through internal memory provided by the bus class.

One decision that had to be taken extremely early, was how to synchronize the components, to that, I had two answers, one was using a master clock and running the components when their cycles were at 0, and the other possibility was allocating a thread per component, and then sleeping the thread and notifying them accordingly. The chosen answer was to not parallelise the system at all, running the whole application in a single thread, so the components are run in a sequential way, controlled by a master clock, since synchronization is easier this way. This design is fundamentally flawed, as there are games that require dummy reads to function, but this approach will generally work and was close enough. Another decision was to fake parallelism using coroutines, but at the time of development, I was not confidence on making that system work. Other solution to this problem is how Mesen does it, it makes the CPU execute the PPU, when the CPU reads or writes memory, it advances the PPU, so in this case, the CPU would command synchronization.

The NES works in the following way: all components are being run in their own compartments, without caring much about each other, except when the PPU triggers an NMI, then, the CPU can communicate with the PPU by modifying the PPUs MMIO registers, in truth, this can be done with the programmers discerption, but, accessing PPU at any other time that NMI would lead to corruption or unexpected behaviour, so it's recommended to only access MMIO registers in the NMI subroutine. So, in less precise terms, first, the CPU is running game logic, while the PPU renders the frame, when the PPU ends the frame, it triggers an NMI and the work being done by the CPU is completely stopped, which is normally an endless loop waiting for the NMI, and the CPU goes to the NMI subroutine, which in this case, its setting the PPU data to run the next frame, and this repeats until the game is closed or it crashes. If you ever wondered how lag can happen in the NES, it would be if the time the CPU takes to complete a frame of game code was longer than the time the PPU takes to render that frame.

All motherboard images have been extracted from this video, credits to NesHacker.

6.1.1.1. Console

The console class is the 'emulator' in itself, it holds and links all the components together, and it's the piece of software that runs these components in a synchronized way, there are a bunch of ways to run this class, as described in 5.5.2.1 in the Console Class, but all those methods internally call the Step method, the Step method is the way this emulator is run. The Step method works as following, first, it checks if any of the components can be run, and then it increments the master clock, basically the master clock is modelled after the real system, which is different from the PPU or CPU clock, and the clock of the individual components is derived from it, assigning It a divisor, its done this way since PAL and NTSC have different CPU and PPU clocks. So every time the Step function is called, the master clock is checked against the different divisions, like 12 for the CPU and 4 for the PPU in NTSC of 16 for the CPU and 5 for the PPU in the PAL version.

Other methods provided for this class are ways to interface with its components without needing to access them, like the LoadCartridge method, that takes a path to a file, and opens a ROM.

The final responsibility of this class is to link the components together, since they are not linked by default.

6.1.1.1.1. Configuration

The original idea with this structure, was so it could be passed to the console, and you would be able to load PAL or NTSC ROMs, this was thought with a very basic understanding of the NES, since I thought that those versions were only differentiated by the televisions they were using, but reality is stranger, since they do have different components, so this class ended up being kind of useless in the end.

It did provide some useful features, like resolution and framerate, but it could not be used for its original purpose in the end.

6.1.1.2. CPU

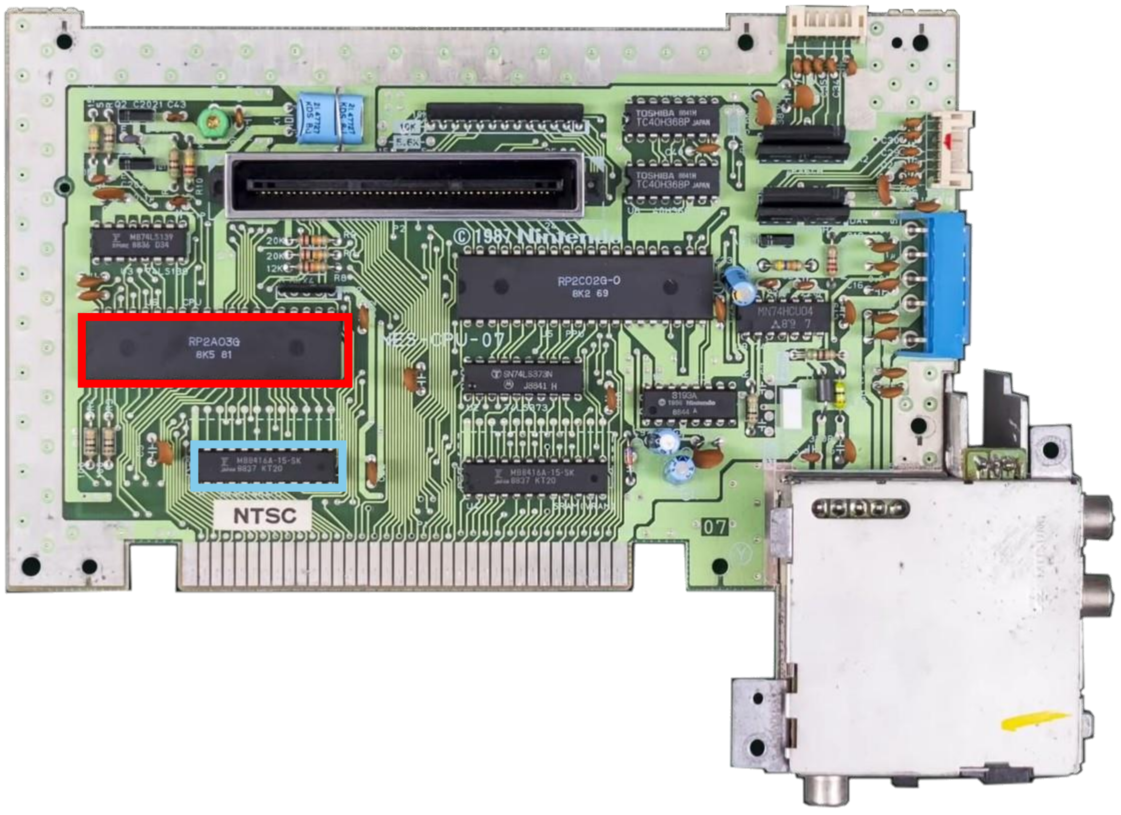


Figure 46. CPU Chip in NES motherboard (red), WRAM (blue)

The CPU class is modelled after the Ricoh A2A07 chip, which is a modified 6502 that does not have decimal mode and contains the APU of the NES (APU is the Audio Processing Unit, is not implemented in this work).

This class is extremely simple, when its reset, it gets the address at the reset vector, which is provided by the ROM, and sets the PC to that address, then, every time the Step method is called, it decrements the internal clock, and if the clock is 0, it fetches, decodes and executes the next instruction in PGR ROM.

The fetch decode execute cycle is made with a jump table, an array that contains functions, so, it fetches the next instruction by means of getting the value at PC, then with the value obtained, it gets the next addressing mode and instruction function, that are stored in the jump table, this is the decode part, then, the execute part, first, the function of the addressing mode is called, to prepare the data that is going to be sent to the instruction, and then, the instruction is executed, setting the internal clock to the number of clock cycles required for that instruction to be executed.

As implied, this CPU does not have sub-cycle precision which makes it not cycle accurate, so if a game requires it, it won't run.

All official opcodes have been implemented, and the unofficial opcodes that are required to pass NesTest are also implemented, if an unimplemented opcode is reached, the system will throw, and the application will notify of an unrecoverable error.

Funnily enough, to accurately implement this CPU, some bugs were implemented knowingly, since the real system had them, like in indirect addressing, where it does not advance page if the low byte of the address is $FF, a list of hardware bugs can be found here.

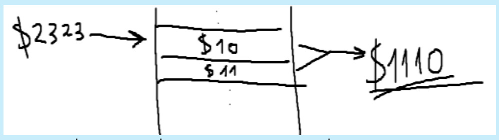
An alternative design would be to use a state machine, as that implementation would provide feasible cycle accuracy, but it would completely break the current implementation of opcodes. Either way a reimplementation would be needed in order to implement an APU.

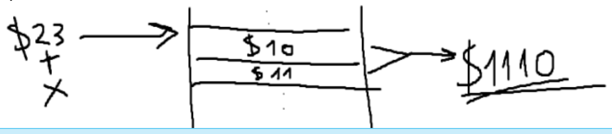
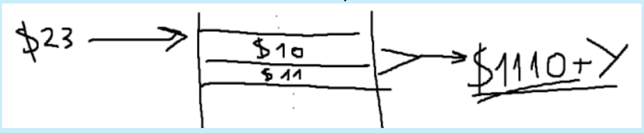
A solution for the cycle accuracy problem, that does not require on redoing the whole CPU or adding coroutines would be to run it the same way its being run now, but in a dummy memory, and recording memory accesses, and then, doing the memory accesses in real memory when needed.

6.1.1.2.1. Instruction

This is what is stored in the jump table, is a collection of function pointers, and aggregated data, like the clock cycles required and the name of the instruction.

The following tables contain a simple explanation of what each addressing mode and instruction does, the operand is whatever the addressing mode returns.

Name	Description	Syntax
IMP	Implicit. What the instruction does is implied by the instruction, i.e. INX.	INX
ACC	Accumulator. The operand is the accumulator.	LSR A
IMM	Immediate. The operand is the byte next to the instruction (next in the program code after the one that signals the instruction)	LDA #23
ZPI	Zero Page. The operand is the contents of the memory location in the zero page [$0000-00FF] provided by the next byte in the program	LDA $23
ZPX	Zero Page, X. The operand in a the contents of the memory location in the zero page provided by the next byte in the program plus X	STY 34,x
ZPY	Zero Page, Y The same as ZPX but uses Y instead of X	LDX 23,y
REL	Relative The operand is the next signed byte in the program	BCC -15 BCC label BCC *-3
ABS	Absolute The operand is the contents of the word located in the address provided by the next two bytes in the program, low first, high second.	JMP $2323
ABX	Absolute, X The operand is the contents of the word located in the address provided by the next two bytes in the program, then adding X. If there is a page cross, that means, that the contents of the high byte have changed after adding X, this takes an extra cycle (oops cycle).	STA $2323,x
ABY	Absolute, Y. The same as ABX but changing X with Y.	ORA $2323,y
IND	Indirect. Only used by JMP. The next word in the program contains 16 bits that identifies the location of the least significant byte of another word memory address which is the operand.  (i.e. $2323 contains $10, and $2323+1 contains $11, then the operand is $1110) This mode is bugged in real hardware, if the first byte in the program equals $FF, then the high byte of the operand is taken again from the word in the program instead of the word + 1	JMP ($2323)
INX	Indirect, X (Indexed indirect).	LDA ($23,x)

	The next word in the program added to X is used as a pointer to the address of the low byte of the operand. 	
INY	Indirect, Y (Indirect indexed). The next word in the program is used as a pointer to the address of the low significant byte of the word that will be added to Y to form the operand.  The difference between INX and INY is that in INX you add before reading, and in INY you add after reading.	LDA ($23),y

Name	Description	Flags
ADC	Add with carry. Adds the value of the operand with the accumulator and the carry bit.	• Negative • Carry • Overflow • Zero
SBC	Subtract with carry. Subtract the value of the operand to the accumulator together with the not of the carry bit.	• Negative • Carry • Overflow • Zero
CMP	Compare accumulator. Compares the operand and the accumulator, result is discarded.	• Negative • Zero • Carry
CPX	Compare X register. Compares the operand and the X register, result is discarded	• Negative • Zero • Carry
CPY	Compare Y register. Compares the operand and the Y register, result is discarded.	• Negative • Zero • Carry

Table 118. Instructions: Arithmetic

A detail, the SBC instruction was one of the most difficult things to implement correctly, since in the beginning I tried to implement it like a normal subtraction, but that did not pass the ROM tests, so in the end I used twos complement and the ADC implementation since that was the implementation used in other emulators.

Name	Description	Flags
LDA	Load accumulator. Loads a byte of the operand into the accumulator.	• Negative • Zero
LDX	Load X register. Loads a byte of the operand into the X register.	• Negative • Zero
LDY	Load Y register. Loads a byte of the operand into the Y register.	• Negative • Zero
STA	Store accumulator. Stores the accumulator into the operand.	
STX	Store X register. Stores the X register into the operand.	
STY	Store Y register. Stores the Y register into the operand.	

Table 119. Instructions: Store/load

Name	Description	Flags
TAX	Transfer accumulator to X. Copies the contents of the accumulator to X.	• Negative • Zero
TXA	Transfer X to accumulator. Copies the contents of X to the accumulator.	• Negative • Zero
TAY	Transfer accumulator Y. Copies the contents of the accumulator to Y.	• Negative • Zero

TYA	Transfer Y to accumulator. Copies the contents of Y to the accumulator	• Negative • Zero

Table 120. Instructions: Register transfers

Name	Description	Flags
TSX	Transfer stack pointer to X. Copies the contents of the stack pointer to X.	• Negative • Zero
TXS	Transfer X to stack pointer. Copies the contents of X to the stack pointer.	
PHA	Push accumulator on stack. Pushes the contents of the accumulator to the stack and increments stack pointer.	
PHP	Push processor status to stack. Pushes the contents of the processor status to the stack and increments stack pointer.	
PLA	Pull accumulator from stack. Pulls a byte from the stack and copies it to the accumulator, decrements the stack pointer.	• Negative • Zero
PLP	Pull processor status from stack. Pulls a byte from the stack and copies it to the processor status, decrements the stack pointer.	All of them.

Table 121. Instructions: Stack operations

Name	Description	Flags
AND	Logical AND. Performs logical and, bit by bit, on the accumulator with the operand.	• Negative • Zero
EOR	Exclusive OR. Performs an exclusive or, bit by bit, on the accumulator with the operand.	• Negative • Zero
ORA	Logical inclusive OR. Performs an inclusive or, bit by bit, on the accumulator with the operand.	• Negative • Zero
BIT	Bit test. Tests if bits in operand are set with a bitmask in the accumulator.	• Negative • Overflow • Zero

Table 122. Instructions: Logical

Name	Description	Flags
INC	Increment the operand.	• Negative • Zero
INX	Increment the X register.	• Negative • Zero
INY	Increment the Y register.	• Negative • Zero
DEC	Decrement the operand.	• Negative • Zero
DEX	Decrement the X register.	• Negative • Zero
DEY	Decrement the Y register.	• Negative • Zero

Table 123. Instructions: Increments/decrements

Name	Description	Flags
ASL	Arithmetic Shift Left. Shifts all bits of the operand to the left and sets the carry to bit old bit 7.	• Negative • Zero • Cary
LSR	Logical Shift Right. Shifts all bits of the operand to the right and sets the carry bit to old bit 0.	• Negative • Zero • Carry
ROL	Rotate Left. Rotates all bits of the operand with the carry bit to the left, the carry is set to old bit 7 and bit 0 is set to old carry.	• Negative • Zero • Carry
ROR	Rotate Right. Rotates all bits of the operand with the carry bit to the right, the carry is set to old bit 0 and bit 7 is set to old carry.	• Negative • Zero • Carry

Table 124. Instructions: Shifts

Name	Description	Flags

JMP	Jump to another location. Sets the program counter to the operand	
JSR	Jump to a subroutine. Sets the program counter the operand and pushes the old program counter minus one to the stack	
RTS	Return from subroutine. Pulls twice from the stack and sets the program counter to the word made from the pulls.	

Table 125. Instructions: Jumps/Calls

Name	**Description**	**Flags**
BCC	Branch if carry clear. Add the operand to the PC if the carry is clear.	
BCS	Branch if carry set. Add the operand to the PC if the carry is set.	
BEQ	Branch if zero set. Add the operand to the PC if the zero is set.	
BNE	Branch if zero clear. Add the operand to the PC if the zero is clear.	
BMI	Branch if negative set. Add the operand to the PC if the negative is set.	
BPL	Branch if negative clear. Add the operand to the PC if the negative is clear.	
BVC	Branch if overflow clear. Add the operand to the PC if the overflow is clear.	
BVS	Branch if overflow set. Add the operand to the PC if the overflow is set.	

Table 126. Instructions: Branches

Name	**Description**	**Flags**
CLC	Clear carry flag.	• Carry
SEC	Set carry flag.	• Carry
CLD	Clear decimal mode flag.	• Decimal
SED	Set decimal mode flag.	• Decimal
CLI	Clear interrupt disable flag.	• Interrupt disable
SEI	Set interrupt disable flag.	• Interrupt disable
CLV	Clear overflow flag.	• Overflow

Table 127. Instructions: Status Flag Changes

Name	**Description**	**Flags**
BRK	Force and interrupt. The program counter and the processor status are pushed to the stack and then the interrupt vector is loaded onto the PC.	• Break
NOP	No operation.	
RTI	Return from interrupt. Pulls the processor status from the stack followed by the program counter	All of them

Table 128. Instructions: System functions

These tables do not contain unofficial instructions, to learn more about them you can check these websites.

- https://www.oxyron.de/html/opcodes02.html
- https://www.pagetable.com/?p=39
- https://www.nesdev.org/wiki/Programming_with_unofficial_opcodes

6.1.1.3. PPU

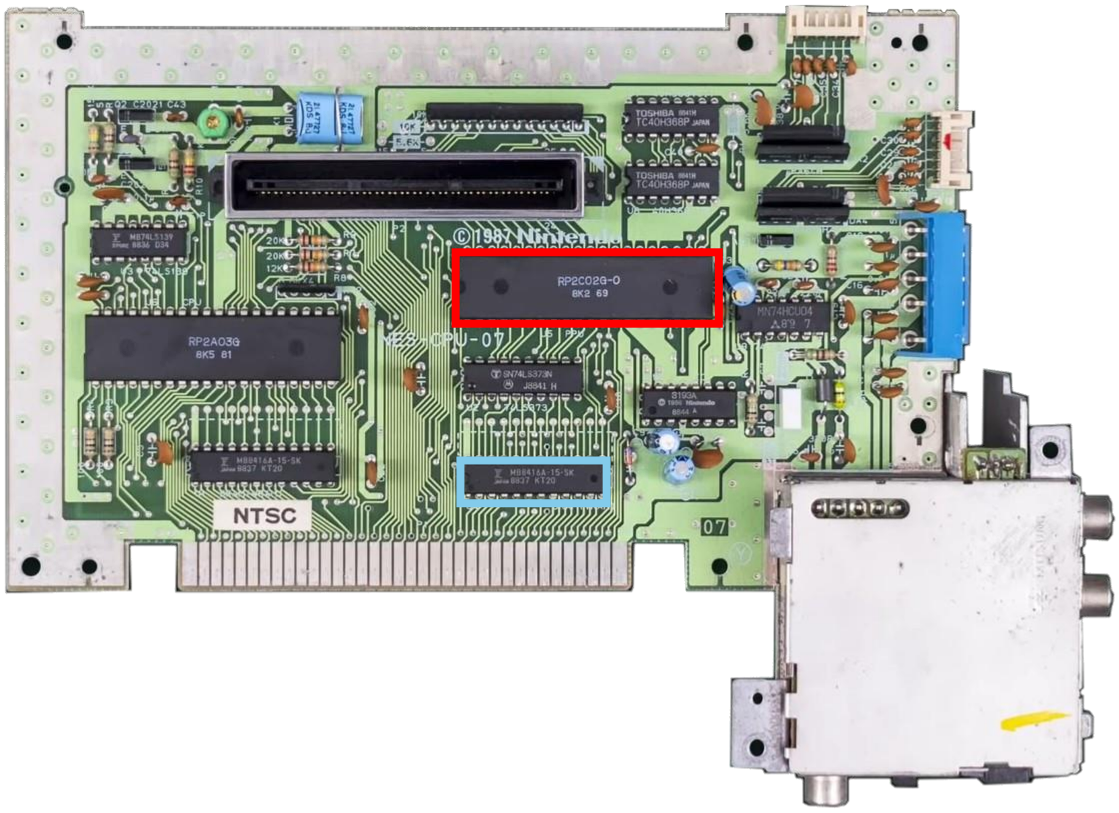


Figure 47. PPU Chip in NES motherboard (red) VRAM (blue

This class is modelled after the chip used in the NTSC NES the Ricoh RP20C2. This was by far the hardest part of this project to develop, especially the sprite rendering part since I couldn't find information on how the sprites are stored after the sprite evaluation process is completed.

To preface the implementation details, and general thoughts on the design, I'd like to explain how the chip works on a surface level.

MMIO registers have been mentioned already, they are eight memory-mapped registers exposed by the PPU to achieve communication with the CPU. They are stored in the $2000-$2007 range, and they are mirrored every 8 bytes until $3FFF, there is a ninth register outside this range that is used to start the DMA process, more on that later.

Name	Description	Location	Access
PPU Control	Miscellaneous settings	$2000	Write
PPU Mask	Rendering settings	$2001	Write
PPU Status	Rendering events	$2002	Read
OAM Address	Sprite RAM address	$2003	Write
OAM Data	Sprite RAM data	$2004	Read/Write
PPU Scroll	X and Y Scroll	$2005	Write twice
PPU Address	VRAM address	$2006	Write twice
PPU Data	VRAM data	$2007	Read/Write
OAM DMA	Sprite DMA	$4014	Write

Table 129. MMIO Registers

Most registers are very easy to understand, like the address and data ones, the address ones set the address that will be used to read/write data, and the data one is used to read what is on that address or write data on that address. When a register has write twice, it means that a sequential write will do different things, in the case of PPU Scroll, it writes x scroll on first write and y on second, and in the case of PPU Address is to make a 16 bit address since the address space is large enough to require it, OAM Address does not require 16 bit addressing. The most special registers are those of PPU Control, PPU Mask and PPU Status.

PPU Control is the register that controls how the rendering is done, this register can only be written to and is used by the CPU to instruct the PPU on how to render things, providing where the sprites are located, and how to render them.

Bit	Controls
0-1	Base nametable address • 0 = $2000 • 1 = $2400 • 2 = $2800 • 3 = $2C00
2	VRAM increment per CPU read or write of PPUDATA. • 0 = 1, going across • 1 = 32, going down
3	Sprite pattern table address for 8x8 sprites, large sprites, or 8x16 ignore this. • 0 = $0000 • 1 = $1000
4	Background pattern table address • 0 = $0000 • 1 = $1000
5	Sprite size • 0 for 8x8 sprites • 1 for 8x16 sprites
6	PPU mater/slave select, unused in this work as its never used on stock consoles (29).
7	Can trigger NMI on Vblank

Table 130. PPU Control Register

PPU Status is mainly used for timing purposes with the CPU and reflects the current state of the CPU. This register can only be read and when read it clears the W register.

Bit	Controls
0-4	PPU open bus, this has stale bus contents.
5	Sprite overflow flag, as stated before, a scanline can only have 8 sprites, if there are found more than 8, this bit is set.
6	Sprite 0 hit flag, this flag is set when the sprite 0 (the first in Object Attribute Memory) and a non-transparent background pixel are on top of each other. used to synchronize with the CPU.
7	VBlank flag, set when entering VBlank, will be cleared after being read.

Table 131. PPU Status Register

PPU Mask is the register that contains rendering settings, it can only be written by the CPU and modifies the behaviour of the PPU by modifying colour output or disabling parts of the rendering process.

Bit	Controls
0	Grayscale mode.
1	Show background on the leftmost 8 pixels of the screen.
2	Show sprites on the leftmost 8 pixels of the screen.
3	Enable background rendering.
4	Enable sprite rendering.
5	Emphasize red on NTSC, green on PAL.
6	Emphasize green on NTSC, red on PAL.
7	Emphasize blue.

Table 132. PPU Mask Register

This register is a bit tricky to understand, especially bits 1 and 2, this is mainly done in games with horizontal scroll with one page or vertical mirroring, since that does not provide smooth scrolling. The grayscale mode is done by binary ANDing the palette index with $30, making it draw the only the first column on the palette, if the palette had different colours there, it would not draw grayscale.

The PPU also contains some internal registers.

Name	Description
X	Fine X scroll
V	VRAM address
T	Temporal VRAM address
W	Address latch

Table 133. PPU Internal registers

These registers are used internally by the PPU, they are used in tandem with the MMIO registers, i.e. T is used on the first write to PPU ADDRESS and is used mostly to store scroll position and to send 16-bit address to V after the second read to PPU ADDRESS. W is used on the MMIO registers that require two reads, after the first one, its set to 1, and after the second one is set to 0. X is used to store the remaining scroll, most of the scroll data is stored in T and V in the following way yyyNNYYYYYXXXXX, where y is the fine y scroll, N is the nametable selection, Y is the coarse y scroll and X is the coarse x scroll, since there is not enough bits to store the needed information, the PPU contains a 3-bit register to store fine x scroll.

Now that some concepts like pattern table, palettes, or attributes are starting to appear I'd like to explain them.

The pattern tables are the areas of memory that make the backgrounds and the sprites, they are in CHR ROM, there are 2 pattern tables, usually one for sprites and one for backgrounds and each contains 256 tiles, every tile is comprised of 2 8-byte planes. If you do the math, you will realize that 16 bytes per tile is not enough to store colour information in a RGB setting, this is because the NES is limited to 4 colour per tile, this is achieved by adding the two planes since each plane provide two colours. For example, in each plane, a pixel would be 0 or 1, and when added, they can be 00, 01, 10 or 11, this is the index of the palette that will be used in this tile (30).

Even though I already mentioned that the NES outputs composite signal, it uses an internal palette RAM that groups colours together for the sprites to use, there are 8 palettes, 4 for background and 4 for sprites, and each palette contains 4 colours (31).

Nametables are the area of memory in which the PPU would lay backgrounds, the NES contains 4 nametables, and each nametable contains 30 rows of 32 tiles, the remaining 64 tiles needed to create a perfect square do exists, but they are called attribute memory. As mentioned, the NES technically contains 4 nametables, but in reality, it contains two, while the other two may be provided by the mapper, when these two nametables are not provided, a mirroring, also provided by the mapper is used, there are mainly two types of mirroring, horizontal and vertical, but there are more uncommon ones like four-screen and single-screen mirroring (32).

Name	Original address	Mapped address
Horizontal	$2000	$2800
	$2400	$2C00
Vertical	$2000	$2400
	$2800	$2C00
Single Page	All nametables refer to a single nametable.	
Four-screen	The cartridge provides additional nametables.	

Table 134. Nametable mirroring

This is a neat implementation detail, but since the only thing changing in different mirroring is where the address is mapped to, I used template metaprogramming to reduce code amount.

As mentioned above, attribute memory is contained in the last two rows of each nametable, there are 4 attribute tables, one for each nametable, and they contain information regarding how the tiles in the nametables are rendered, it's a 64-byte section of memory arranged in and 8 by 8 byte array, and each byte contains information on a 4 by 4 tile sections, you may notice that since a byte controls 8 palettes it may not be enough to cover palette needs for each tile since 1 bit is not enough to discern between the 4 palettes used for backgrounds, that's because each byte is divided into 4 nibbles, and each nibble control a 16 by 16 pixel area, or 2 by 2 tiles, this translates that each 16 by 16 pixel area is limited to 3 unique colours plus the universal background colour (33).

The last important concept of the PPU is the Object Attribute Memory or OAM, not to be confused with attribute memory, different things, this is an internal memory of the PPU that contains Sprite information, that is, the palette the sprite uses, if the sprite is flipped horizontally or vertically, priority of the sprite and the ID of the sprite (which sprite is it on its pattern table). In order to fill this memory, MMIO registers can be used, but this is incredibly slow, taking 4 write to OAM Address and 4 writes to OAM Data per sprite, in order to combat that, the DMA or Direct Memory Access is used, when this process is started, the CPU is suspended, and then it copies 256 bytes from CPU memory to the OAM memory in the PPU, once this process ends, the CPU is resumed (34) (35).

Now I'd like to explain the rendering process.

To render a frame, the PPU must complete 261 scanlines with 340 cycles each, scanlines can be roughly divided into three categories, pre-render scanlines, visible scanlines and

VBlank scanlines, cycles can be divided into visible cycles, and HBlank cycles, each visible cycle draws a pixel to the screen.

The first scanline, is the pre-render scanline, normally identified as scanline -1 or scanline 261, this scanline exists to fill the data for the first two tiles in the first scanline.

The visible scanlines are the ones that render both the backgrounds and the sprites, while the PPU is busy rendering, the CPU must not access any MMIO register, or that would lead to corruption or errors in the rendering process. In the HBlank cycles, data for the sprites and first two tiles in the next scanline are fetched.

This process is repeated until scanline 2 40 is reached, this is the post-render scanline and is an idle scanline. In scanline 241 cycle 1, the VBlank flag is set, and the NMI process is started, the remaining VBlank scanlines are completely idle, and the PPU is safe to be accessed to, for a more detailed information refer to the wiki, this process is repeated every frame (36).

This Sprite evaluation process is more complicated and is the one that gave me the most trouble.

The sprite evaluation process takes place in two stages, first, a memory section called secondary OAM is filled with $FF in cycles 1-64 of visible scanlines, and then, that secondary OAM is filled with the first eight sprites that are contained in the next scanline, when the process ends, the data from the secondary OAM is put in internal memory, this is the part that I'm not sure about; for a sprite, to be in a scanline, its y position has to intersect the next scanline, if more than 8 sprites are found in the next scanline, a bug name sprite evaluation bug appears, in which the memory is increased diagonally instead of linearly (think of ach entry in OAM as a 4 byte box, normally, you would go from beginning of one box to the beginning of the next one, but in this case it would go from beginning of that box to the second byte in the next box, and then the third, and etc).

This class works in a similar way to the CPU class, it has an step function that is called every time the master clock is perfectly divided by the PPU Clock divisor, in NTSC that would be each 4 master clock ticks, and in PAL it would be every 5, this class is also fitted with helpers used to represent data in the application, like a helper to get a palette or a helper to get a pattern table.

6.1.1.3.1. RegisterFlags

At the beginning, I thought of using an union containing an u8 and a bit field to model the registers that were fields of flags (37), but since using unions this way (accessing non active member) is undefined behaviour, I decided against it as even if it worked on MSVC, it might not work in other system or compiler.

So the solution was between using a single u8 and modifying it with bit manipulation on the spot, or with using a thin wrapper of that u8 that contained some quality of life members to set flags, remove them, or check if they were set.

This is a class whose only purpose is to reduce code duplication, since the PPU has three registers that are a set of flags, these registers are: PPU control, PPU mask and PPU status.

This class is only used in the PPU, even though the CPU has a register composed of flags (Processor Status), since the CPU was already finished when I made it, and I didn't want to change the CPU.

6.1.1.4. Bus

This class is, as stated before, an abstraction of the memory and real bus that exists in the NES, it only does one thing, and is communicating memory accesses to their respective components, for example, if the CPU reads CPU range, it would access CPU memory, but if it access MMIO registers, it would access PPU memory.

Address Range		Size	Device
$0000	$07FF	$0800	2 KB internal RAM
$0000	$00FF	$0100	Zero page
$0200	$02FF	$0100	Stack
$0800	$0FFF	$0800	Mirrors of $0000-$07FF
$1000	$17FF	$0800	
$1800	$1FFF	$0800	
$2000	$2007	$0008	PPU MMIO Register
$2008	$3FFF	$1FF9	Mirrors of $2000-$2007 repeats every 8 bytes
$4000	$4017	$0018	APU and I/O registers
$4018	$401F	$0008	APU and I/O registers, test mode
$4020	$FFFF	$BFE0	Cartridge space
$6000	$7FFF	$2000	Usually, cartridge RAM
$8000	$FFFF	$8000	Usually, cartridge ROM

Table 135. NES Memory Map

The table above contains the contents of the NES memory range, the bus class is tasked with properly communicating devices.

Address	Name
$2000	PPU Control
$2001	PPU Mask
$2002	PPU Status
$2003	OAM Address
$2004	OAM Data
$2005	PPU Scroll
$2006	PPU Address
$2007	PPU Data
$4014	OAM DMA (when written to it, will trigger DMA)

Table 136. PPU MMIO Registers

When the CPU accesses memory in the range of the PPU Registers, the bus will call the CpuRead or CpuWrite method in PPU.

6.1.1.5. *Input Device*

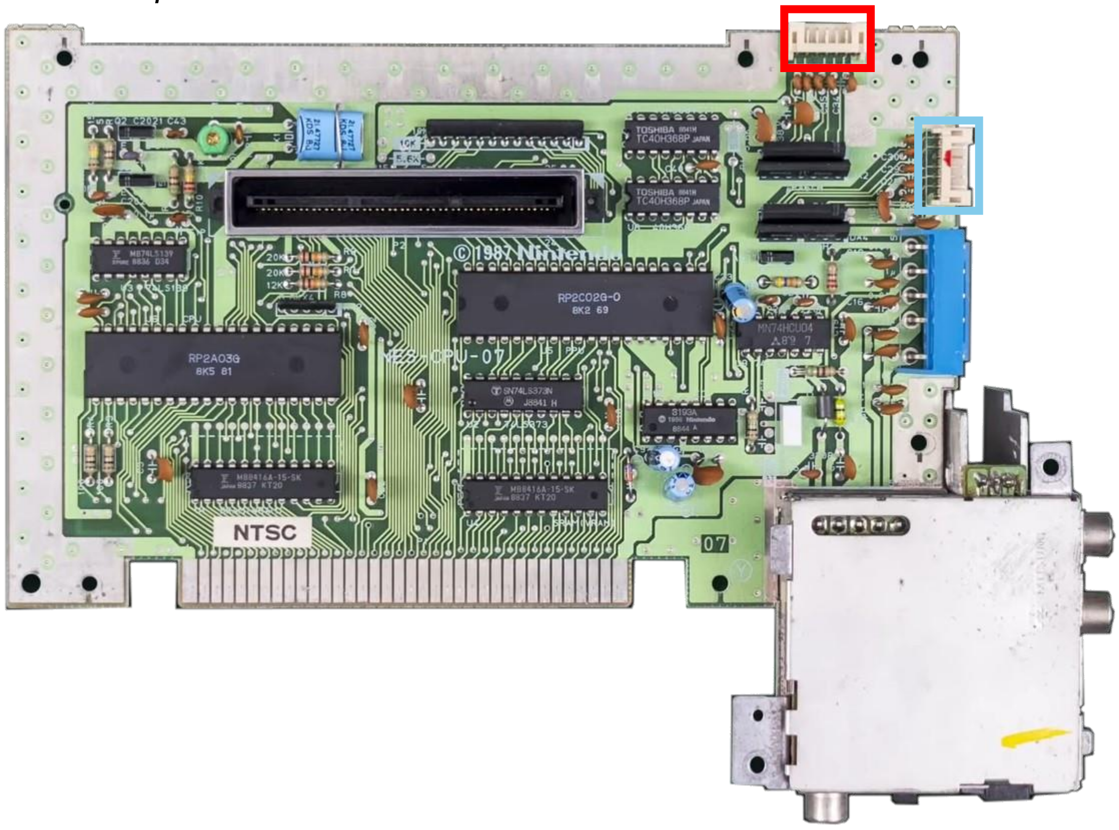


Figure 48. Controller Port 1 (red) and controller port 2 (blue) in NES motherboard

There's not a lot to explain here, but the functionality of this class is as follows: the application sets which buttons have been pressed this frame, and then, the CPU writes to $4016 or $4017, this sets the internal register to the buttons pressed, after that, the CPU reads the same address 8 times, to check which buttons have been pressed, since the input device shifts the internal register 1 bit to the left every time it has been read.

This is done by having two internal registers, one that is being modified all the time by the application, and one that is only modified when the address is written to, so, the application sets the data every time the user uses their controller, and the emulator overwrites the internal register with that data when needed.

6.1.1.6. Cartridge

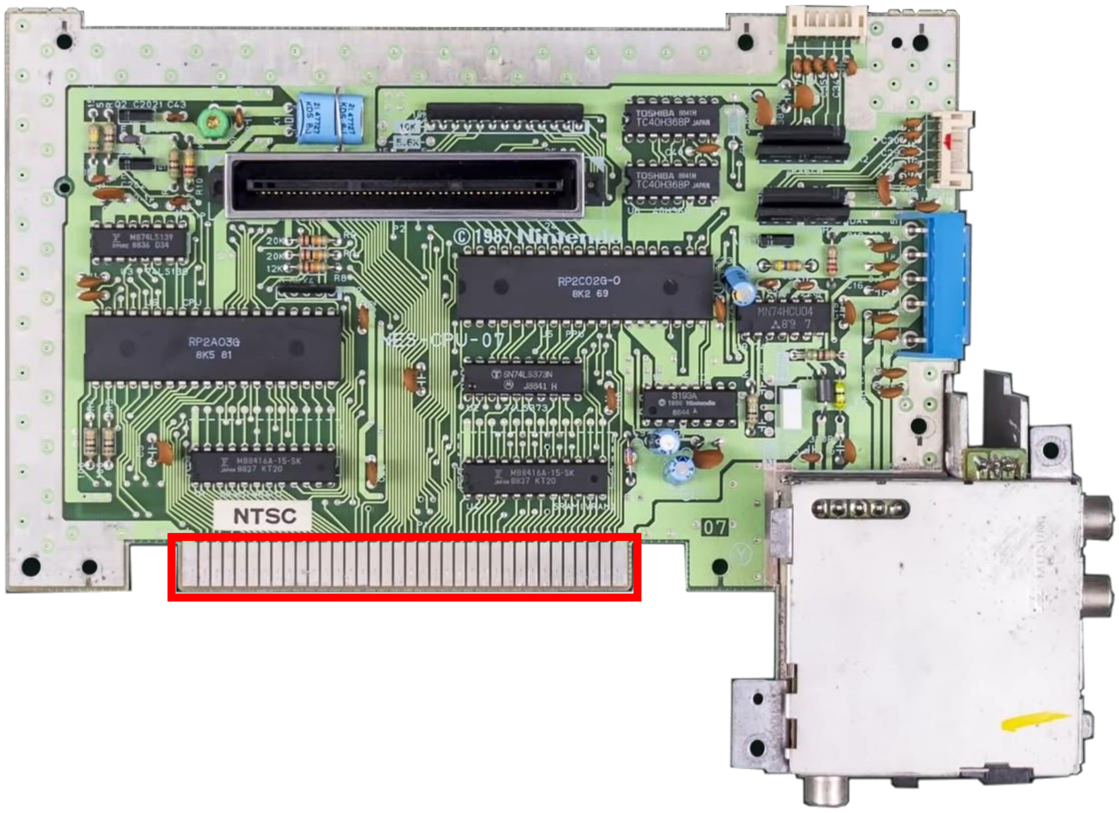


Figure 49. Cartridge Slot in NES motherboard (red)

This class is made to model the physical cartridge that would be used with the real hardware. It's tasked with reading the ROM from file or memory, with making sure it's valid and with returning the correct data from the PRG or CHR ROM when accessed.

When a ROM is loaded, this class will load the whole contents of the ROM, they are kind of small, and it will validate it, the validation process is simple, it checks the header in the iNES format, I decided to use iNES 1.0 and not iNES 2.0 since the ROMS that will be supported are simple enough to not need the extensions that iNES 2.0 provides, and iNES 2.0 ROMS are retro compatible with iNES 1.0.

If the header is valid, the loader will set its internal memories, PRG and CHR, so they can be accesses from the bus when needed.

There is a functionality to load ROMs from memory that is only used in testing.

6.1.1.6.1. INesHeader

This is a struct that contains the information that can be found in a mapper, it only exists to provide a way to access those fields without encumbering the code more than necessary.

6.1.1.7. IMapper

Image extracted from nescartdb.

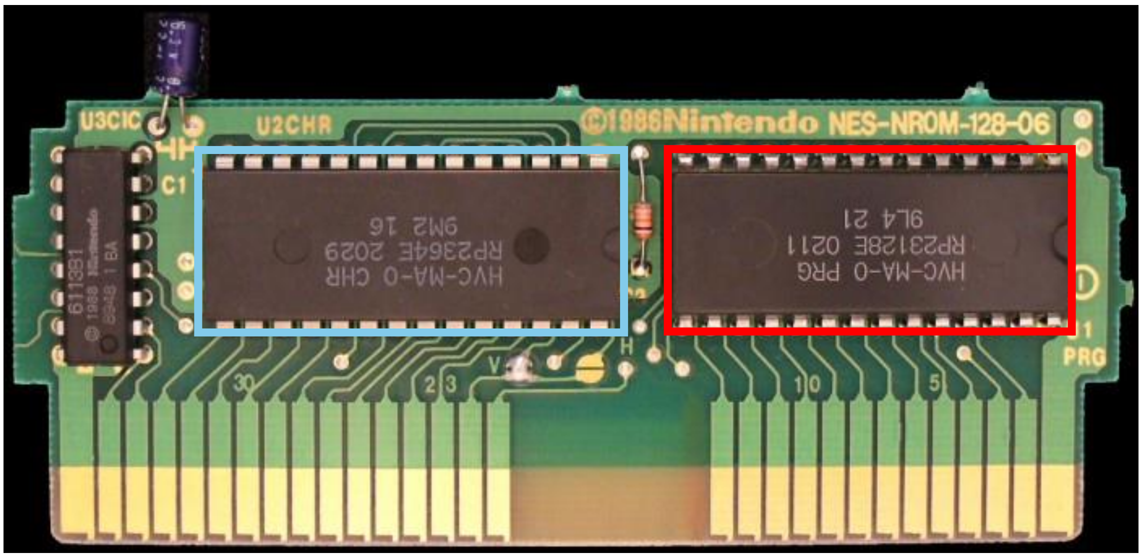


Figure 50. NROM cartridge PCB, CHR ROM (blue) and PRG ROM (red)

The other visible chip is the CIC lockout chip used to validate the cartridge with the CIC lockout chip present in the NES motherboard.

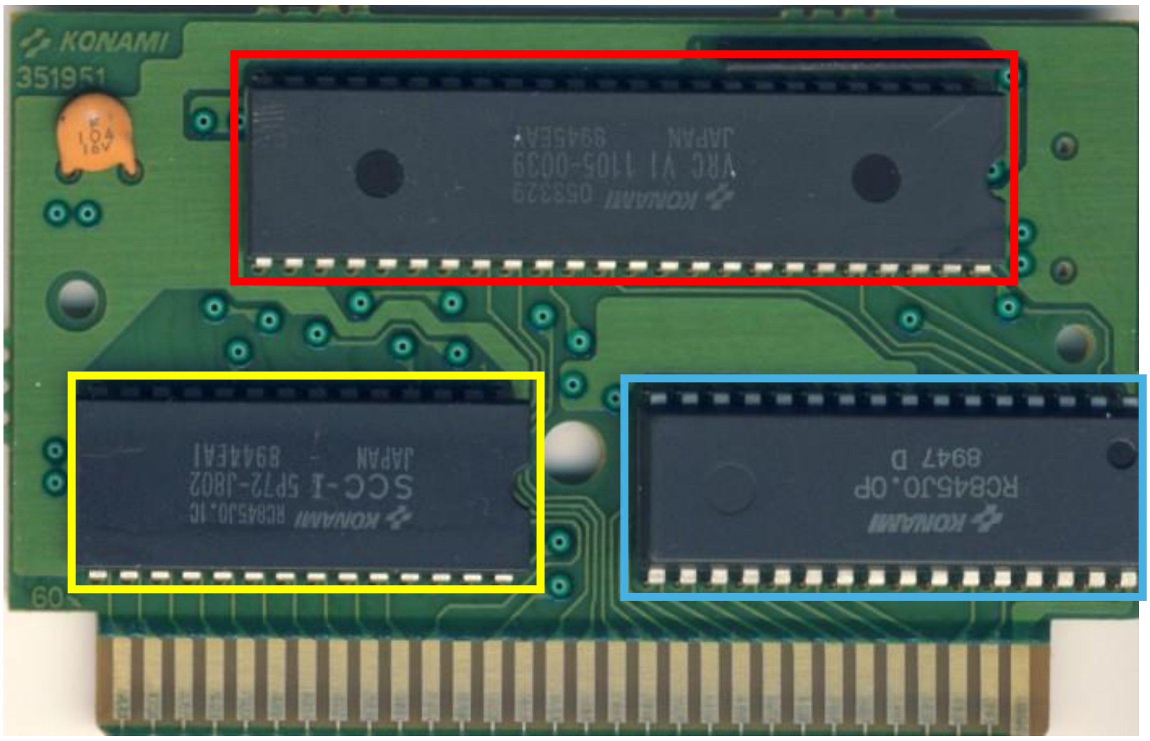


Figure 51. Cartridge using mapper 024 (VRC6a) (red), CHR ROM (yellow) and PRG ROM (blue)

This class is made to be equivalent to a real mapper, a mapper is a piece of hardware made to extend the capabilities of the system, it can be done by adding more memory and having the system access it with bank switching capabilities, adding persistent memory with the help of batteries, or adding RAM to the system.

The current interface only has methods to map addresses, so in the future will need to be expanded to provide the extra functionalities.

6.1.1.7.1. NROM

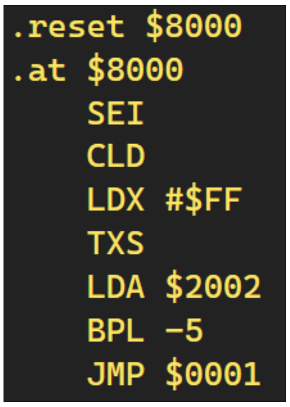
This is mapper 0, a mapper that basically holds maps the ROMs that use it to themselves, it can only map PRG ROM.

The real mapper has more things, in case of an arcade version of the Nes, the Famliy Basic, as that version contains PRG RAM, but this version is only intended to work with the NES, so it does not have that functionality.

6.1.1.8. Assembler

The assembler class is an extremely simple 6502 assembler made as a custom tool to be able to test the CPU without needing to check a machine code table all the time.

Its regex based and has almost no features outside of assembling the provided code to machine code, the only features it has is, set reset, nmi and irq vector position and change current memory address with the .at directive.



```
.reset $8000
.at $8000
    SEI
    CLD
    LDX #$FF
    TXS
    LDA $2002
    BPL -5
    JMP $0001
```

Figure 52. Example of valid assembly

Other limitations are that it only supports hexadecimal numbers and does not support labels.

6.1.1.9. Disassembler

A more complex sister to the assembler, its tasked with sending a disassembly to the application.

It’s not perfect disassembly, since that would require executing the code beforehand thanks to the indirect jump instruction. But its correct enough, since accessing an address that is not currently mapped would add it to the disassembly.

It works by following paths to reach other sectors of code, since a sector has to be traversable in order to be executed. When the disassembler is created, it will traverse three paths of code, one would be the reset vector, this includes power up section, and most of the game code, other path is the NMI vector, that is the section when the CPU speaks with the PPU, and the last version is the IRQ vector, that is almost empty. The disassembler will disassemble until it either founds an instruction that has already been disassembled, and instruction that would return from a subroutine, like BRK, RTS or RTI, and it would exit when encountered the STP instruction. Every time the disassembler finds a branch, be it conditional or unconditional, it would recursively call another DisassembleFromAddress from that branch so all paths are reached, the disassembler does not check if that branch can be taken, it will take it nonetheless.

The current number one limitation of this approach is that it can not take indirect jumps, since those jumps require executing the code. Other pressing limitations are that it does not support bank switching since the emulator neither does.

6.1.1.9.1. Disassembly

This is what is returned from the disassembly, it contains a string version of the machine code, the "disassembly", a string version of the label this address may have, if it has it, and a register name and value in case this disassembly contains a known constant.

Name	Value
PPU_CONTROL	$2000
PPU_MASK	$2001
PPU_STATUS	$2002
OAM_ADDRESS	$2003
OAM_DATA	$2004
PPU_SCROLL	$2005
PPU_ADDRESS	$2006
PPU_DATA	$2007
PULSE_1_VOLUME	$4000
PULSE_1_SWEEP	$4001
PULSE_1_LO	$4002
PULSE_1_HI	$4003
PULSE_2_VOLUME	$4004
PULSE_2_SWEEP	$4005
PULSE_2_LO	$4006
PULSE_2_HI	$4007
TRIANGLE_VOLUME	$4008
TRIANGLE_SWEEP	$4009
TRIANGLE_LO	$400A
TRIANGLE_HI	$400B
NOISE_VOLUME	$400C
NOISE_LO	$400E
NOISE_HI	$400F
DMC_FREQUENCY	$4010
DMC_LOAD_COUNTER	$4011
DMC_START	$4012
DMC_LENGHT	$4013
OAM_DMA	$4014
APU_STATUS	$4015
JOYPAD_1	$4016
JOYPAD_2	$4017
APU_TEST_1	$4018
APU_TEST_2	$4019
APU_TEST_3	$401A
CPU_TIMER_1	$401C
CPU_TIMER_2	$401D
CPU_TIMER_3	$401E
CPU_TIMER_4	$401F

Table 137. Known Disassembly Constants

6.1.1.10. Opcode

This is an internal structure used in both assembler and disassembler to bundle instruction and addressing mode names.

6.1.2. Renderer

I will not go into much detail, since its not the focus on this work. I will only detail the functions of the Engine, Batch Renderer, Sprite, ITexture and API classes.

Creating a renderer was not necessary, since I could use the SDL Renderer one, and I had experience using that renderer, but I wanted to learn how to make one, and decided to use this work as training and study in this realm, this is way this renderer has features that are not needed for this work.

6.1.2.1. Engine

Engine is the so called “renderer” it was made following mainly vkguide and with some vulkan-tutorial and was modified to have bindless resources since I intended to use a batch renderer.

Another difference between those two guides, is that these guides are meant mainly for 3D rendering, and I do not want 3D rendering, so I had to modify it a bit to achieve that. One massive challenge with that was a deceivingly complicated concept called projection, 3D generally uses perspective projection, but that does not really work in 2D rendering, since 2D rendering generally uses orthographic projection, at the time I found it incredibly hard to access information related to orthographic projection, since I did not know how it was called, or much about the concept, so this part was a huge stump in development. Other problem with projection was understanding orthographic projection, especially what zNear and zFar parameters in glm meant, since I misunderstood those parameters and thought they worked inversely as they really work.

Other small roadblock was understanding the different types of coordinates, specially NDC coordinates (38).

This class is the one that holds all Vulkan structures and is the one that sends data to the GPU, it’s kind of messy, that’s why I used an API to interface with it.

6.1.2.1.1. Batch Renderer

Batch rendering is a rendering strategy used to save draw calls, thus improving the performance of the rendering process, this strategy is achieved by putting all the vertex information in a single vertex buffer and drawing that vertex buffer.

This batch renderer exists since at the time I did not know if rendering a single texture and modifying its texel data or drawing multiple quads of different colour would be better, so I decided to implement a batch renderer to prepare me for future refactoring, and to learn how a batch renderer works.

In the end, I decided to use a single texture, but the batch renderer was functional, and it made no sense to remove it.

The batch renderer is the reason I decided to use bindless resources, the reason was that I did not found much literature on 2D rendering that explained more complex processes like creating a texture atlas from different sized textures, so I decided to circumvent that problem using bindless textures and adding a texture ID to the vertex structure.

On small detail, is that in the current implementation, the descriptors are being recreated every frame, and I don’t know if I should reuse them or not since I was unable to find information on it.

6.1.2.2. ITexture

This class is a generic interface for a texture, it’s meant to have an implementation that uses the rendering API being used in the background, in this case, Vulkan.

This is a texture meant to be modifiable and to be created from memory, so it does not provide a way to load a texture from file, only from memory data.

In this framework, textures are not bound to sprites, since they could be used by multiple sprites, so they must be created and managed outside a sprite.

To create a texture, I decided to create them through the API, and the API would return the correct implementation as requestsd.

6.1.2.3. Sprite

A sprite is a bundle containing a texture, a position and a size, to be seen, it requires a rect and a texture, the texture can be reused, and the texture can be windowed in case of using a sprite sheet.

To be drawn, the sprite will call functionality in the API.

6.1.2.4. API

This is not a class, is a collection of functions in the Renderer namespace, it could be put into a fully static class, but I saw no benefit to that so instead of using an interface, all these functions are unimplemented by default, and the renderer must create these implementations.

This class exists to abstract the complexity of the renderer, for example, creating a texture, requires a size, data, and a type instead of a image format, a size, an usage or if the image is mipmapped, as those are the parameters required by engine.

This could be done by changing the interface of engine, but I liked the idea of having an API for the renderer.

6.1.2.4.1. RendererAPIVulkanImpl

This is the Vulkan implementation of the API, these functions interface with the Engine class and are the ones that are meant to be called by the outside.

6.1.3. Input Handler

The input handler is made up to be an action-based input handler, that means that you assign actions to inputs, and when the input is pressed, the actions are run.

I am not completely happy with how this ended up working, since it currently does not have a way to rebind inputs, but that could be expanded in the future by means of adding alias to keys and swapping the aliased keys.

6.1.3.1. IInput

This class has been changed from the design step, as it has been flattened, originally the enums where enum classes, the C++ type enforced enum, but I found that working with that was extremely annoying to this use case, so the enums have been changed to pure int backed enums, so a function like CanRepeatKeyAfter now is CanRepeatAfter, and can be used with both Key and Button.

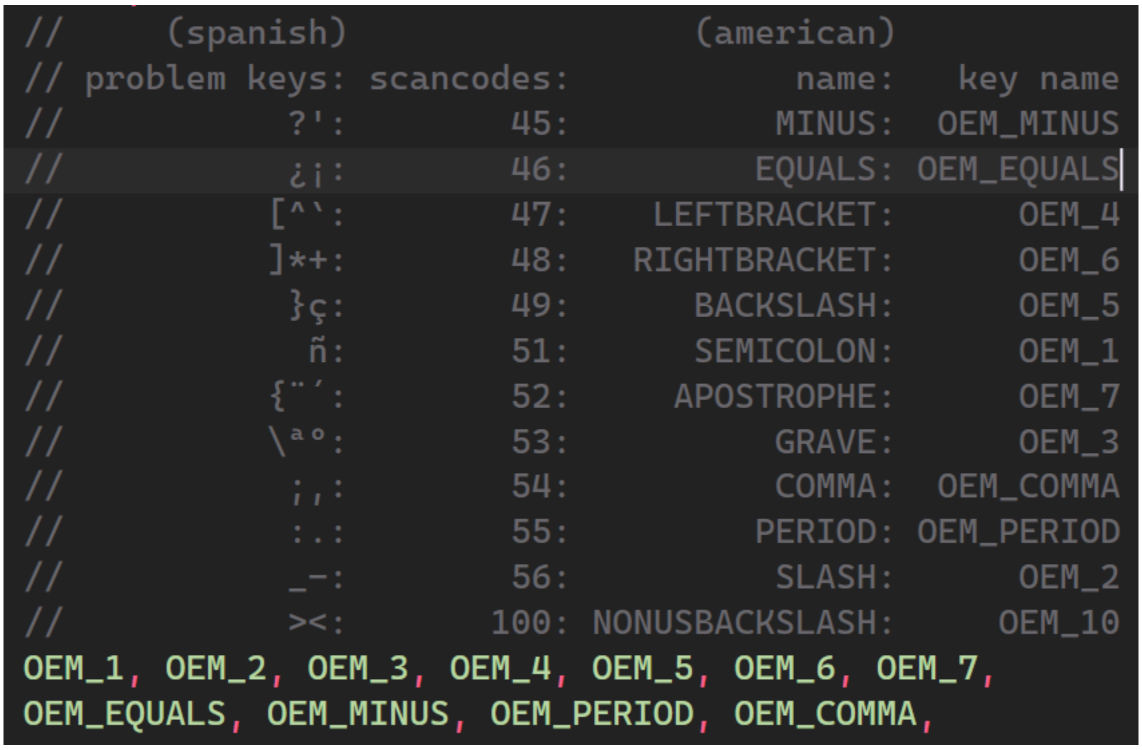
Name
IInput
Description
Abstract class for Input handlers, provides actions functionality, but not event or keyboard/mouse functionality, that is to be provided by implementers (RF.7, RF.8).
Proposed attributes
Implementation details: • current is the trigger whose action is being run • a trigger is a key or a button if current key is defined, current button is not defined, and vice versa.
Proposed methods
• GetButton: returns true whether passed button is pressed. • GetKey: returns true whether passed key is pressed. • IsKeyModified: returns true if key is being modified by a mod key (ctrl, shift, alt, win) • IsRepeating: returns true if current is being hold down after being pressed. • CanRepeatAfter: returns true if time amount has passed since last current was pressed • CanRepeatEvery: returns true every time some time amount has passed since current was pressed. • ClearActions: deletes all actions for all triggers or a trigger. • RunActions: runs all actions that can be run for all triggers and a trigger. • Update: calls platform specific update implementation, updates state and runs actions. • ProcessEvents: runs input related events of underlying implementation.

Table 138. New IInput class description

This is the class that provides the action system, so that makes it not an interface, in C++, more like an abstract class, but I think its fine.

This class is also made to be an adapter interface, that means, that an implementation of IInput must interface with another library, and IInput provides a common interface for different libraries to be used in application code. This is achieved thanks to the enums provided by the system, there are two, one for keys and one for buttons, the implementee would need to convert that enum member to a proprietary version of that enum provided by the library being used.

One huge problem I had developing this library, was naming the keys enum, since I use an Spanish keyboard, but keyboards are different depending to layout, so I had to discover how to name the members, I used a combination of how SDL names its keys, and kbdlayout.



```
//     (spanish)                   (american)
// problem keys: scancodes:           name:   key name
//           ?':        45:          MINUS:  OEM_MINUS
//           ¿¡:        46:         EQUALS: OEM_EQUALS
//           [^`:       47:    LEFTBRACKET:      OEM_4
//           ]*+:       48:   RIGHTBRACKET:      OEM_6
//           }ç:        49:      BACKSLASH:      OEM_5
//            ñ:        51:      SEMICOLON:      OEM_1
//           {¨´:       52:     APOSTROPHE:      OEM_7
//           \ªº:       53:          GRAVE:      OEM_3
//           ;,:        54:          COMMA:  OEM_COMMA
//           :.:        55:         PERIOD: OEM_PERIOD
//           _-:        56:          SLASH:      OEM_2
//           ><:       100: NONUSBACKSLASH:     OEM_10
OEM_1, OEM_2, OEM_3, OEM_4, OEM_5, OEM_6, OEM_7,
OEM_EQUALS, OEM_MINUS, OEM_PERIOD, OEM_COMMA,
```

Figure 53. List of problematic keys

6.1.3.1.1. SDL2Input

This is the implementation of IInput that uses SDL2, the original used SDL3, but thanks to a bug in ImGuis SDL3 implementation, I had to rollback the library to SDL2, luckily, it was an easy process.

As stated before, this class maps the enums defined on IInput to sdl constants, i.e. Button::FACE_DOWN would be mapped to SDL_GameControllerButton::SDL_CONTROLLER_BUTTON_DPAD_DOWN.

This class is also tasked with connecting to an active controller, it connects automatically when a controller is connected, and if a controller is disconnected, it would disconnect from the controller.

6.1.3.1.2. SDL3Input

The same as SDL2Input but with the more modern version of SDL3.

6.1.4. Window System

There is not much insight to be provided hare as this library follows the same principles as the Input handler.

This library works very similarly to the Input, it provides an adapter interface to be used to interface with other window libraries like win32 API, GLFW or SDL.

A difference between this and the Input Handler, is that this system is made to work with the Renderer, and the Input Handler does not care about other systems.

6.1.4.1. IWindow

Class that provides the adapter interface to be implemented, mayor difference with IInput, is that it requires the user to define an event loop with callbacks, in IInput, the same event loop is static.

The events work in a similar way to the enums in IInput, they are a generic definition of events that can be used by this application, and they are mapped to the real events in the library.

6.1.4.1.1. SDL2Window

SDL2 inplementation of IWindow, as stated before, a rollback from SDL3.

6.1.4.1.2. SDL3Window

The same as SDL2Window, but with SDL3.

6.1.5. File Manager

6.1.5.1. FileManager

As stated before, this is a collection of functions inspired by the imgui architecture, that means that the library uses push and pop to add or subtract to the current file structure.

When I say that this library was inspired by imgui, I mean that the library holds a "context" that represents its current state, in this case it would be the current file path or open file being used, so if you want to modify the context, you would "push" or "pop" a folder or file to move the context to it.

6.1.5.2. Context

This is an internal struct that holds all data required for the FileManager functions to work.

6.1.5.3. ISerializable

An interface that provides a way for other classes to be serialized or deserialized.

6.1.5.4. Serializable

Another collection of functions that serialize or deserialize data, all of them make use of SerializeData or DeserializeData internally. They also provide a way to set the current serialization file.

6.1.6. Application

This is the system meant to be run by the user, it's the one that sets up all other systems and is the one that link them together, like creating a window, sending it to the renderer, and setting up the key binds.

6.1.6.1. Application

The application class is a singleton and is the main class, it has a collection of IComponents to be renderer as the interface and is the one that does all the things described above.

A problem I have with it right now is that the top menu bar is not a component, so is coupled to the application instead of a component.

6.1.6.2. IComponent

This is the “interface” to be used in the application, this class provides an interface that has a way to update the component, to create the component and to render the component.

A detail in the implementation of this class, is that I struggled to identify components, at first I thought of using UUIDs, but in the end I went with unique names, that way I could access a component without retrieving its UUID and only knowing their given name, this is mainly used to delete them and to check if they are already added to the application.

Components also have a way to be deleted by setting a member to true.

If someone inspects the code, they will realize that I gradually change my approach to working with ImGui, as I was testing different ways to do things with it.

6.1.6.2.1. CloseDialog

A component used to show error messages, they can be recoverable, like loading an invalid ROM, or unrecoverable, like reaching an STP instruction.

If the error is unrecoverable, the application will be closed.

6.1.6.2.2. ShowCPUStatus

This class is the one that shows the information that is in the CPU, like the stack, its registers and a disassembly, the user can scroll the disassembly and go back to the PC when scrolled.

A problem with the representation of the disassembly, is that the icon font I’m using is not monospaced, so the instruction that holds the PC is slightly to the right compared to the rest.

6.1.6.2.3. ShowPPUStatus

This is the component that holds the information of the PPU, this class creates a lot of images to show data, like the pattern tables or the palettes, this is also the class that caused the rollback from SDL3, this was because I was adding a tooltip to the pattern tables when holding click, it shows the pattern you are clicking, but zoomed in, and shows its number, the problem, when moving the mouse outside the window the application would crash, and that affected normal tooltips if you moved the mouse fast enough, I thought that the tooltips provided important information, and I would rather rollback to SDL2 than losing them.

6.1.6.2.4. MemoryView

This class is the one I dislike the most, it’s the one that show the memory, but it’s also the most bugged class, if you made the component window small, the memory inspector disappears.

This class has two images on top that provide representation of the memory, one of the full memories, and one of the CPU RAM, they are not useful, but I think they show cool patterns in memory, like PPU MMIO registers and mirroring in the different sections.

The memory inspector part is the problematic one, is made in a way that does not render the full memory to avoid lag, but thanks to that implementation, a lot of problems appear, since its height is based in available scroll, and does not provide feedback when scrolling, this is part that will need to be redesigned in the future. Other things the user can do with the inspector, is search for an address, by right clicking the inspector, a dialog that asks for an address will appear, and if the user inputs something, the inspector will be scrolled to it and highlight it for a while.

6.2. ISD 5: Designing the System Module Architecture

6.2.1. ISD 5.1: System Module Design

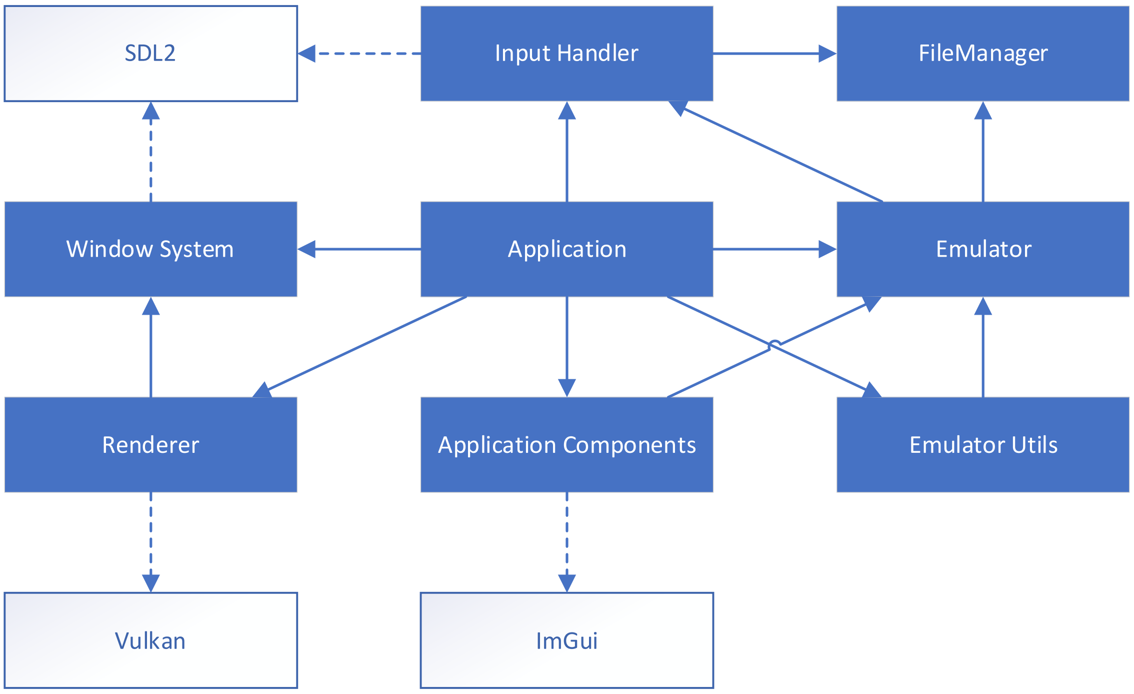


Figure 54. Package Diagram

This package diagram includes the main libraries used in the background.

6.2.2. ISD 5.2: Inter-Module Communications Design

6.2.2.1. Component Diagram

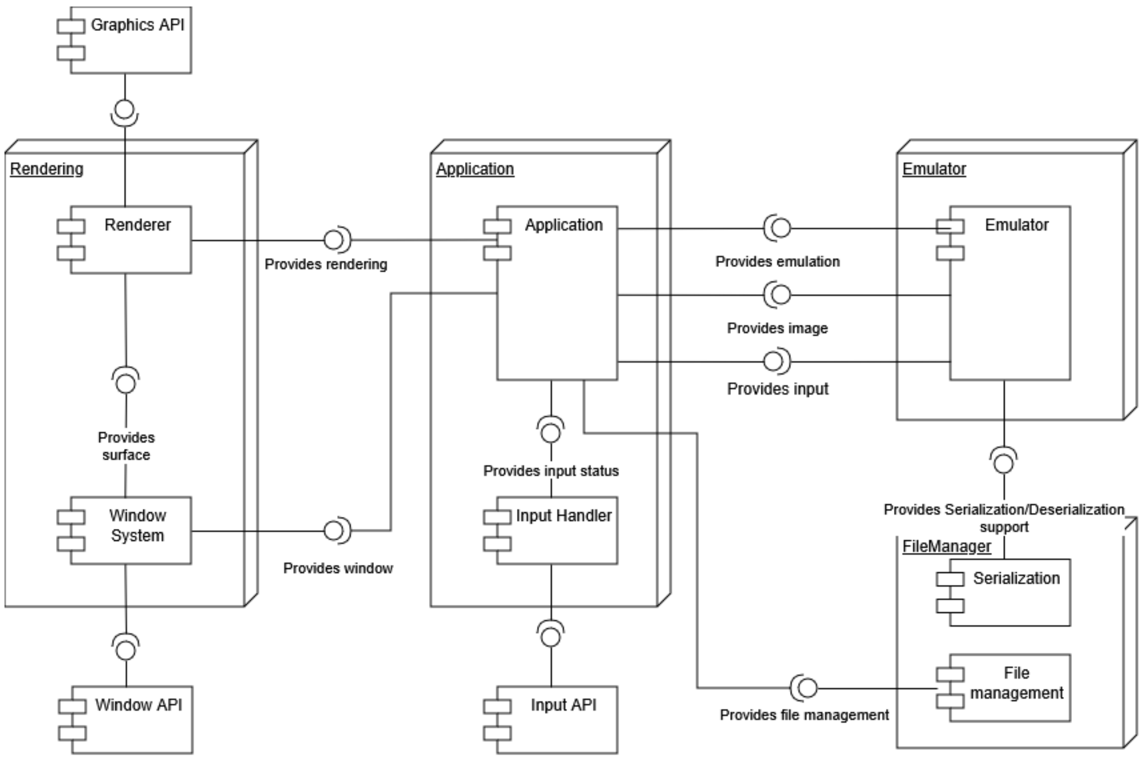


Figure 55. Components Diagram

6.3. ISD 10: Technical Specification of the Test Plan

6.3.1. Unit Testing

Unit testing is done in the CPU part of the emulator, since other components cannot work alone, the tests are divided by instruction type. These tests were made during CPU development and are made completely obsolete by test ROMs.

As stated before, there are more non-CPU related unit tests for the disassembler and the save states functionality.

These tests are not really “Unit” tests since they come with the precondition that the bus is functioning correctly and are more a way to test that an input program executes as intended while isolating parts of the emulator, i.e. the CPU tests do not take into account the PPU.

6.3.1.1. Arithmetic

The addressing modes are only tested once since they share the same function call for all instructions. ADC are used as a baseline test for the cpu and after them all tests use heavy assembly to run tests

Name	Description	Expected result
ADC_IMM_N	Test ADC instruction in immediate mode while setting negative flag. • Set the accumulator to 10. • Execute adc #$80 (assembly).	• The result is stored in the accumulator. • The result is 138. • The negative flag is set. • Correct cycle amount passed

ADC_ZPI_C	Test ADC instruction in zero-page index mode while setting carry flag. • Write $80 to memory position $0002. • Set the accumulator to $80. • Execute adc $02 (assembly).	• The result is stored in the accumulator • The operand is taken from $0002. • The operand is $80. • The result is 0. • The carry flag is set. • Correct cycle amount passed.
ADC_ZPX_Z	Test ADC instruction in Zero Page X mode while setting zero flag. • Write $80 to $0003. • Set the accumulator to $80. • Set X to 1. • Execute adc $02,x (assembly).	• The result is stored in the accumulator. • The result is 0. • The operand is taken from $02 + X. • The operand is $80. • The Zero flag is set. • Correct cycle amount passed.
ADC_ABS_C	Test ADC instruction in Absolute mode while setting carry flag before the operation. • Write 13 to $0003. • Set the carry flag. • Set the accumulator to 12. • Execute adc $0003 (assembly).	• The result is stored in the accumulator. • The operand is taken from $0002. • The operand is 13. • Correct cycle amount passed. • The result is 26. • The carry flag is cleared.
ADC_ABX_OOPS	Test ADC instruction is Absolute X mode while making it oops cycle (taking an extra cycle) • Write 13 to $0100. • Set carry flag. • Set the accumulator to 12. • Set X to 1. • Execute adc $00ff,x.	• The result is stored in the accumulator. • Correct cycle amount passed. • The result is 26. • The operand is extracted from $00FF + X. • The operand is 13. • The instruction takes extra cycle thanks to page wrap.
ADC_ABY_OOPS	Test ADC instruction in Absolute Y mode while making it oops. • Write 13 to $0100. • Set the carry flag. • Set the accumulator to 12. • Set Y to 1. • Execute adc $00ff,y.	• The result is stored in the accumulator. • Correct cycle amount passed • The result is 26. • The operand is extracted from $00FF + Y. • The operand is 13. • The instruction takes extra cycle thanks to page wrap.
ADC_INX	Test ADC instruction in indirect X mode. • Write 0 to $000D. • Write 15 to $000C. • Write 23 to $000F. • Set the accumulator to 12. • Set X to 2. • Execute adc ($000A,x).	• The result is stored in the accumulator. • Correct cycle amount passed. • The result is 25. • The operand is taken from $000F. ○ First read at $000A + X to get $0F. ○ Second read at $000A + X + 1. ○ To get $00. ○ Make word to get $000F. • The operand is 23
ADC_INY_NO_OOPS	Test ADC instruction in indirect Y mode and make it no oops. • Write 13 to $000A. • Write 0 to $000B. • Write 23 to $000F. • Set the accumulator to 12. • Set Y to 2. • Execute adc ($000A),Y.	• The result is stored in the accumulator. • Correct cycle amount passed. • The result is 25 • The operand is taken from $000D + Y ○ First read at $000A to get $D. ○ Second read at $000A + 1 to get $00.

		◦ Make word to get $000D. • The operand is 23.
ADC_INY_OOPS	Test ADC instruction in indirect Y mode and make it oops • Write $FF to $000A. • Write 0 to $000B. • Write 23 to $0101. • Set the accumulator to 12. • Set Y to 2. • Execute adc ($000A),y.	• The result is stored in the accumulator. • Correct cycle amount passed. • The result is 25 • The operand is taken from $0101 ◦ First read at $000A to get $FF. ◦ Second read at $000A + 1 to get $00. ◦ Make word to get $00FF. ◦ Add Y to get $0101. • The operand is 23.
ADC_V	Test ADC instruction and make it overflow. First make it overflow. Then make it not overflow.	• After first program the overflow is set. • After second program the overflow is cleared.
SBC	Test SBC instruction. Test it to set carry flag. Test it to set negative flag. Test it to set zero flag. Test it to set overflow flag.	• The result is correct. • The flags are correct.
CMP	Test CMP instruction. Test it to set carry flag. Test it to set zero flag. Test it to set negative flag.	• The flags are correct.
CPX	Test CPX instruction. Test it to set carry flag. Test it to set zero flag. Test it to set negative flag.	• The flags are correct.
CPY	Test CPY instruction. Test it to set carry flag. Test it to set zero flag. Test it to set negative flag.	• The flags are correct.
BIT	Test BIT instruction. Test it to set Zero and overflow flags. Test it to set negative and clear zero flag.	• The flags are correct.

Table 139. Unit Tests: Arithmetic instructions

6.3.1.2. Branch

Name	Description	Expected result
GENERAL_BRANCH_PAGECROSS	General branch test to check if the branch adds extra cycle on page cross.	• The extra cycle is registered.
BCC_BRANCH	Tests if the BCC instruction branches when the carry is clear.	• Branches.
BCC_NOBRANCH	Tests if the BCC instruction branches when the carry is set.	• Does not branch.
BCS_BRANCH	Tests if the BCS instruction branches when the carry is set.	• Branches.
BCS_NOBRANCH	Tests if the BCS instruction branches when the carry is clear.	• Does not branch.
BEQ_BRANCH	Tests if the BEQ instruction branches when the zero flag is set.	• Branches.
BEQ_NOBRANCH	Tests if the BEQ instruction branches when the zero flag is clear.	• Does not branch.
BMI_BRANCH	Tests if the BMI instruction branches when the zero flag is clear.	• Branches
BMI_NOBRANCH	Tests if the BMI instruction branches when the zero flag is set.	• Does not branch.
BNE_BRANCH	Tests if the BNE instruction branches when the negative flag is set.	• Branches.

BNE_NOBRANCH	Tests if the BNE instruction branches when the negative flag is clear.	• Does not branch.
BPL_BRANCH	Tests if the BPL instruction branches when the negative flag is clear.	• Branches.
BPL_NOBRANCH	Tests if the BPL instruction branches when the negative flag is set.	• Does not branch.
BVC_BRANCH	Tests if the BVC instruction branches when the overflow flag is clear.	• Branches.
BVC_NOBRANCH	Tests if the BVC instruction branches when the overflow flag is set.	• Does not branch.
BVS_BRANCH	Tests if the BVS instruction branches when the overflow flag is set.	• Branches.
BVS_NOBRANCH	Tests if the BVS instruction branches when the overflow flag is clear.	• Does not branch.

Table 140. Unit tests: Branch instructions

6.3.1.3. *Increment & Decrement*

Name	Description	Expected result
INC	Test the INC instruction. Set negative flag with INC. Set zero flag with INC.	• The value at the operand is incremented by one. • The flags are set correctly.
INX	Test the INX instructions. Set the negative flag with INX. Set the zero flag with INX.	• The value of X is incremented by one. • The flags are set correctly.
INY	Test the INY instruction. Set the negative flag with INY. Set the zero flag with INY.	• The value of Y is incremented by one. • The flags are set correctly.
DEC	Test the DEC instruction. Set the negative flag with DEC. Set the zero flag with DEC.	• The value of the operand is decreased by one. • The flags are set correctly.
DEX	Test the DEX instruction. Set the negative flag with DEX. Set the zero flag with DEX.	• The value of X is decreased by one. • The flags are set correctly.
DEY	Test the DEY instruction. Set the negative flag with DEY. Set the zero flag with DEY.	• The value of Y is decreased by one • The flags are set correctly

Table 141. Unit tests: Increment/Decrement instructions

6.3.1.4. *Jump & Call*

Name	Description	Expected result
JMP	Tests the JMP instruction in both absolute and indirect addressing modes. Tests the hardware bug in indirect addressing.	• The program counter is set correctly in both modes. • The bug is reproduced correctly.
JSR	Tests the JSR instruction.	• The old program counter is pushed to the stack • The program counter is set to the operand.
RTS	Tests the RTS instruction.	• The PC is retrieved from the stack. • The PC is set to the retrieved value. • The program continues normal execution after RTS.

Table 142. Unit Tests: Jump/Call instructions

6.3.1.5. *Load & Store*

Name	Description	Expected result
LDA	Tests the LDA instruction.	• The accumulator is set to the correct value. • The negative and zero flags are set correctly.
LDX	Tests the LDX instruction.	• The X register is set to the correct value. • The negative and zero flags are set correctly.
LDY	Tests the LDY instruction.	• The Y register is set to the correct value. • The negative and zero flags are set correctly.
STA	Tests the STA instruction.	• The value stored in the accumulator is written to the correct memory value.
STX	Tests the STX instruction.	• The value stored in the X register is written to the correct memory value.
STY	Tests the STY instruction.	• The value stored in the Y register is written to the correct memory value.

Table 143. Unit Tests: Load/Store instructions

6.3.1.6. *Register Transfer*

Name	Description	Expected result
TAX	Tests the TAX instruction.	• The value in the accumulator is copied to the X register. • The negative and zero flags are set correctly.
TAY	Tests the TAY instruction.	• The value in the accumulator is copied to the Y register. • The negative and zero flags are set correctly.
TXA	Tests the TXA instruction.	• The value in the X register is copied to the accumulator. • The negative and zero flags are set correctly.
TYA	Tests the TYA instruction.	• The value in the Y register is copied to the accumulator. • The negative and zero flags are set correctly.

Table 144. Unit Tests: Register transfer instructions

6.3.1.7. *Shift*

Name	Description	Expected result
ASL	Tests the ASL instruction and the accumulator addressing mode.	• The value in the accumulator is properly shifted to the left. • The carry and negative flags are set correctly.
ASL_WRITE_MEMORY	Tests the ASL instruction without an immediate type addressing mode.	• The value in the operand is properly shifted to the left. • The carry and negative flags are set correctly.
LSR	Tests the LSR instruction.	• The value in the accumulator is properly shifted to the right. • The carry and negative flags are set correctly.
ROL	Tests the ROL instruction.	• The value in the accumulator is properly rotated with the carry to the left. • The carry and negative flags are set correctly.
ROR	Tests the ROR instruction.	• The value in the accumulator is properly rotated with the carry to the right. • The carry and negative flags are set correctly.

Table 145. Unit Tests: Shift instructions

6.3.1.8. *Stack Operations*

Name	Description	Expected result
TSX	Tests the TSX instruction.	• The value in the stack pointer is copied to the X register. • The zero and negative flags are set correctly.
TXS	Tests the TXS instruction.	• The value in the X register is copied to the stack pointer. • The zero and negative flags are set correctly.
STACK_PUSH	Tests push operations to the stack.	• The stack pointer is incremented correctly. • The pushed values are in the correct memory locations. • The pushed values are correct. ○ For PHA the value of the accumulator is pushed. ○ For PHP the value of the processor status is pushed.
STACK_POP	Tests the pop operations to the stack.	• The stack pointer is decremented correctly. • The values are popped from the correct memory locations. • The popped values are correct. ○ For PLA the accumulator is set to the popped value ○ For PLP the processor status is set to the popped value and the unused flag is set

Table 146. Unit Tests: Stack operations instructions

6.3.1.9. *Status Flag Changes*

Name	Description	Expected result
SEC	Tests the SEC instruction.	The carry flag is set.
CLC	Tests the CLC instruction.	The carry flag is clear.
SED	Tests the SED instruction.	The decimal flag is set.
CLD	Tests the CLD instruction.	The decimal flag is clear.
SEI	Tests the SEI instruction.	The IRQ disable flag is set.
CLI	Tests the CLI instruction.	The IRQ disable flag is clear.
CLV	Tests the CLV instruction.	The overflow flag is clear.

Table 147. Unit Tests: Status Flag Changes instructions

6.3.1.10. *System Functions*

Name	Description	Expected result
NOP	Tests the NOP instruction.	Nothing happens
BRK	Tests the BRK instruction.	• The PC and processor status are pushed to the stack. • The PC is set to the value in the NMI vector. • The break flag is set.
RTI	Tests the RTI instruction	• The operations in the BRK instruction are reverted. • The PC and processor status are set to the popped values from the stack. • The unused flag is set.

Table 148. Unit Tests: System Functions instructions.

6.3.1.11. *Disassembler*

These tests are made to tests the disassembler used in the CPU view interface.

Name	Description	Expected result
ASSEMBLE_DISASSEMBLE	Tests that a previously assembled set of instructions are disassembled correctly.	The output program is the same as the assembled one.
ASSEMBLE_NESTEST	Tests that the nestest ROM is disassembled correctly.	The output program is the same as the nestest one.

Table 149. Unit Tests: Disassembler

6.3.1.12. Save States

Name	Description	Expected result
SAVE_STATE_DEFAULT	Tests that the state of the emulator is saved correctly to disk.	The state is saved properly and can be loaded later.
SAVE_STATE_MULTIPLE_SAME_ROM	Tests that multiple save states can exists at the same time for a singular ROM.	Multiple stats are saved and can be loaded individually while preserving their correct values.
SAVE_TEST_LOAD_EMPTY	If a save state that does not exist is loaded nothing happens.	Nothing happens.
SAVE_STATE_MULTIPLE_ROM	Tests that multiple ROMs can have different save states.	The different states are saved properly and they can be loaded when their ROM is active.

Table 150. Unit Tests: Save states

6.3.2. Integration Testing

The PPU tests are used as integration tests since the PPU requires the CPU to work.

There are not a lot of PPU tests since it being a "visual" component, any glaring issue would make the emulated image completely broken, i.e, mangled sprites or the screen always scrolling.

6.3.2.1. MMIO

These are the most delicate MMIO registers, the rest can be tested by running ROMs.

Name	Description	Expected result
CONTROL	Tests the PPU Control MMIO register.	The register is properly modified when writing to $2000.
ADDRESS_DATA	Tests the PPU Data MMIO register.	• The register is properly read or wrote when accessing the $2007 address. • The Dummy read functionality works.
SCROLL	Tests the PPU Scroll MMIO register.	• The Scroll Values are set correctly.

Table 151. Integration Tests: MMIO

6.3.3. System Testing

System testing is made by running ROMs, an initial test was run by tracing the execution of the emulator (both PPU and CPU) and comparing it to a known correct trace, that test no longer exists since it was made purely to bugfix the PPU and have it in a working state before testing it with nestest.

6.3.3.1. Nestest

Nestest is a test ROM made to test emulators, it can be run both manually and automatically, the tests are made with the automatic mode, but the manual mode ahs been run multiple times to tests the correctness of the CPU.

Sadly, a lot of information on this ROM has been lost to time, like the meaning of the error codes, so a lot of blind steps were run when first testing the emulator with it, that's why the trace was needed, since I couldn't figure why something was failing and the error code meant nothing. In the future, I'd like to test the emulator with multiple test ROMs like the ones provided by the NES development wiki.

Name	Description	Expected result
RUN_NESTEST	The nestest ROM is run until it crashes, each time the console is stepped, the values in the error memory locations ($0002 and $0003) are checked, if they are non-zero the test fails.	The tests runs until it crashes.

Table 152. System Tests: Nestest

7. Building the Information System

7.1. ISC 1: Preparation of the Generation and Construction Environment

7.1.1. Standards and Regulations Followed

I did not follow any particular standard in the development, of this project, I did loosely follow the Google C++ Style guide (39), but I did not followed 100% of the time since this project was a learning experience in my C++ development and I wanted to experiment with different naming conventions, like swapping to snake case in private methods; and in architecture design.

7.1.2. Programming Languages

7.1.2.1. C++

I mainly used C++20, but I wanted to use features from C++23, like std print, or std unreachable, so in the end I used the C++latest configuration in MSVC.

7.1.2.2. 6502 Assembly

6502 Assembly was used for programming the emulated CPU in tests, I did not use any official 6502 Assembly definition since I built my own assembler to my own needs, but it still follows the main 6502 assembly instruction set.

7.1.2.3. Lua

Lua is the native language used for PreMake, the build tool used by this project.

7.1.2.4. Python

Python is a scripting language used in this project to help with the compilation and building process.

7.1.3. Tools and Programs Used for Development

7.1.3.1. Microsoft Visual Studio Community Edition 2022

Visual Studio is a very popular IDE for windows C++ development, it can be extremely bulky and demanding, but it provides excellent debugging and refactoring tools.

It also provides a way to easily compile C++.

7.1.3.2. Visual Studio Code

Even though MSVC is my main development platform, sometimes I want to quickly modify something without loading MSVC, so I normally use Visual Studio since I prefer the coding experience in it, I would love to be able to 100% code in Visual Studio, but the ctre hpp header was crashing the C++ extension for Visual Studio, so I couldn't do that

7.1.3.3. PreMake

PreMake is a Lua-based build tool for C++, it's not as popular as CMake, but, as stated before, is the one I know how to use.

7.1.3.4. GitHub

I used GitHub for Source Control, and code portability to have the same code base in multiple machines.

7.1.3.5. Compiler Explorer

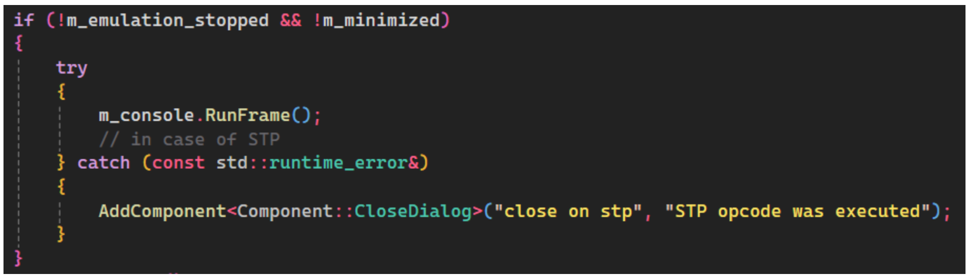
Compiler explorer is a way to test C++ code generation in different compiler setups without having to set up the compiler pipeline in a local machine. I use compiler explorer mainly to check if the code I'm writing works as I expect it to work since I'm not the best C++ developer out there.

7.2. ISC 2: Code Generation of Components and Procedures

The following sections contains an overview that highlights some of the most important parts of this project.

7.2.1. Stepping a Frame

The first relevant part of this functionality happens on Application.cpp on the update method.

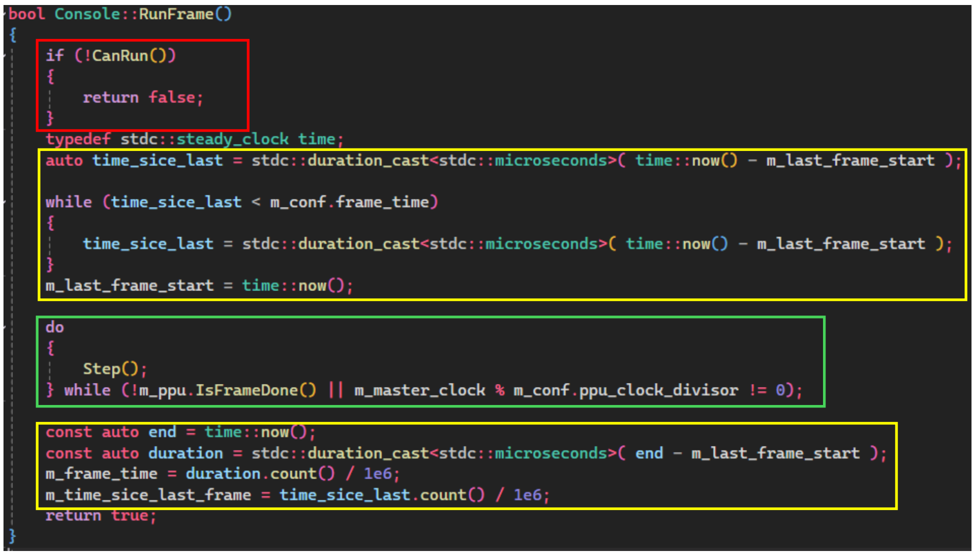


```cpp
if (!m_emulation_stopped && !m_minimized)
{
    try
    {
        m_console.RunFrame();
        // in case of STP
    } catch (const std::runtime_error&)
    {
        AddComponent<Component::CloseDialog>("close on stp", "STP opcode was executed");
    }
}
```

Figure 56. Running a frame 1

First, it is check if the application is not minimized and if the emulation is currently running, if both checks pass, the console will run a frame, the try catch is only for STP opcode, since that opcode crashes the console in real hardware.

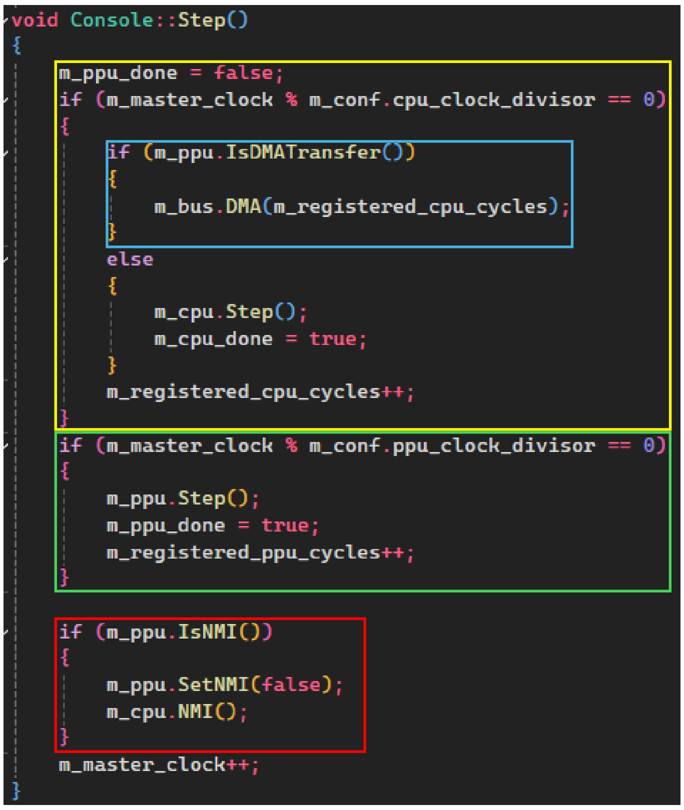
The second relevant part happens on Console.cpp, this is the method that runs the frame.



```cpp
bool Console::RunFrame()
{
    if (!CanRun())
    {
        return false;
    }
    typedef stdc::steady_clock time;
    auto time_sice_last = stdc::duration_cast<stdc::microseconds>( time::now() - m_last_frame_start );

    while (time_sice_last < m_conf.frame_time)
    {
        time_sice_last = stdc::duration_cast<stdc::microseconds>( time::now() - m_last_frame_start );
    }
    m_last_frame_start = time::now();

    do
    {
        Step();
    } while (!m_ppu.IsFrameDone() || m_master_clock % m_conf.ppu_clock_divisor != 0);

    const auto end = time::now();
    const auto duration = stdc::duration_cast<stdc::microseconds>( end - m_last_frame_start );
    m_frame_time = duration.count() / 1e6;
    m_time_sice_last_frame = time_sice_last.count() / 1e6;
    return true;
}
```

Figure 57. Running a frame 2

There are three relevant parts to RunFrame, as highlighted in the image above, the red part checks if there is a ROM loaded so the emulation can be run, the yellow part is the one that handles frame timing to ensure stable FPS so the emulation does not run faster than required and the green part is the one that calls the underlying Step method.

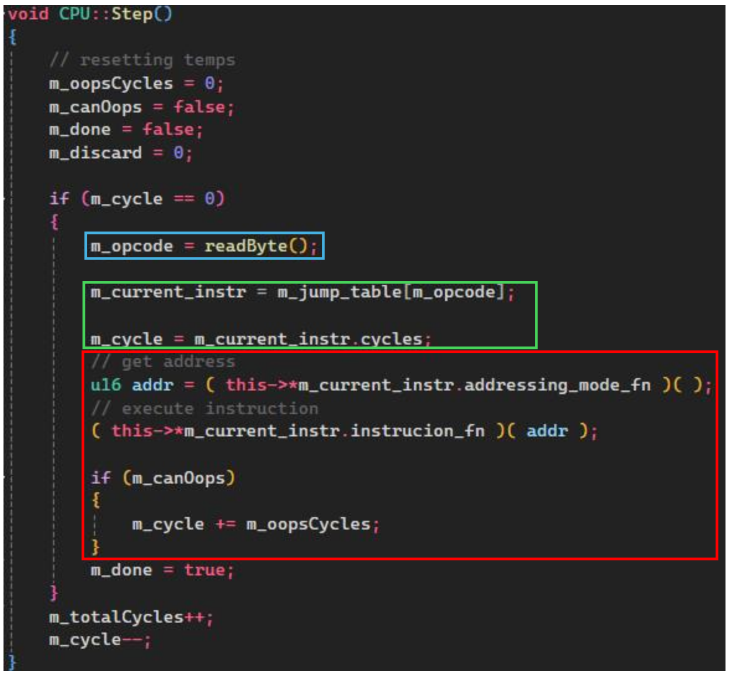


```
void Console::Step()
{
    m_ppu_done = false;
    if (m_master_clock % m_conf.cpu_clock_divisor == 0)
    {
        if (m_ppu.IsDMATransfer())
        {
            m_bus.DMA(m_registered_cpu_cycles);
        }
        else
        {
            m_cpu.Step();
            m_cpu_done = true;
        }
        m_registered_cpu_cycles++;
    }
    if (m_master_clock % m_conf.ppu_clock_divisor == 0)
    {
        m_ppu.Step();
        m_ppu_done = true;
        m_registered_ppu_cycles++;
    }

    if (m_ppu.IsNMI())
    {
        m_ppu.SetNMI(false);
        m_cpu.NMI();
    }
    m_master_clock++;
}
```

Figure 58. Running a frame 3

Again, relevant parts highlighted in the image. The yellow part steps the CPU when needed, the green part handles the PPU, the blue part is the one that runs the DMA process when the PPU asks for it and the red part is the one that makes a hardware interrupt happen.

7.2.1.1. CPU Step



```
void CPU::Step()
{
    // resetting temps
    m_oopsCycles = 0;
    m_canOops = false;
    m_done = false;
    m_discard = 0;

    if (m_cycle == 0)
    {
        m_opcode = readByte();

        m_current_instr = m_jump_table[m_opcode];

        m_cycle = m_current_instr.cycles;
        // get address
        u16 addr = ( this->*m_current_instr.addressing_mode_fn )( );
        // execute instruction
        ( this->*m_current_instr.instrucion_fn )( addr );

        if (m_canOops)
        {
            m_cycle += m_oopsCycles;
        }
        m_done = true;
    }
    m_totalCycles++;
    m_cycle--;
}
```

Figure 59. CPU step

As stated before, the CPU step follows a fetch decode execute, the fetch part is in blue, the decode part in green and the execute part in red. This process works by subtracting to the cycles the last instruction took until it reaches 0 to run the next fetched instruction. The fetch part works in conjunction with a jump table that holds the relevant function pointers for addressing mode and instruction and holds the base cycles of the opcode, the oops part is in case the opcode took more cycles than normal.

7.2.1.2. PPU Step

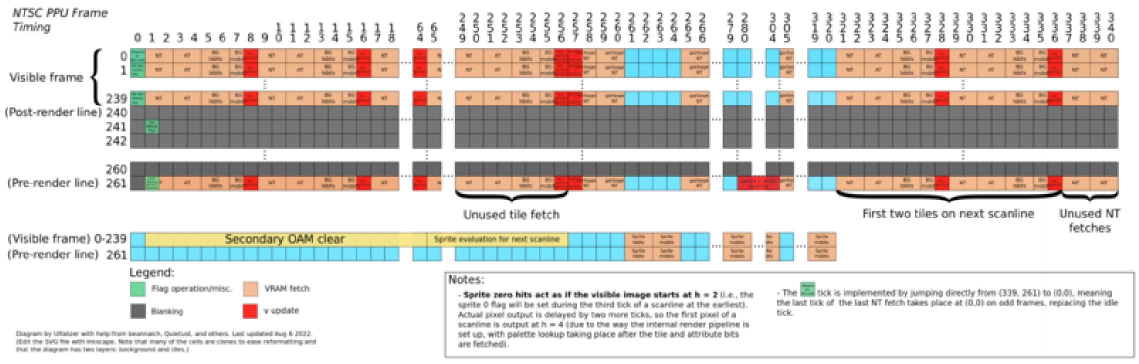


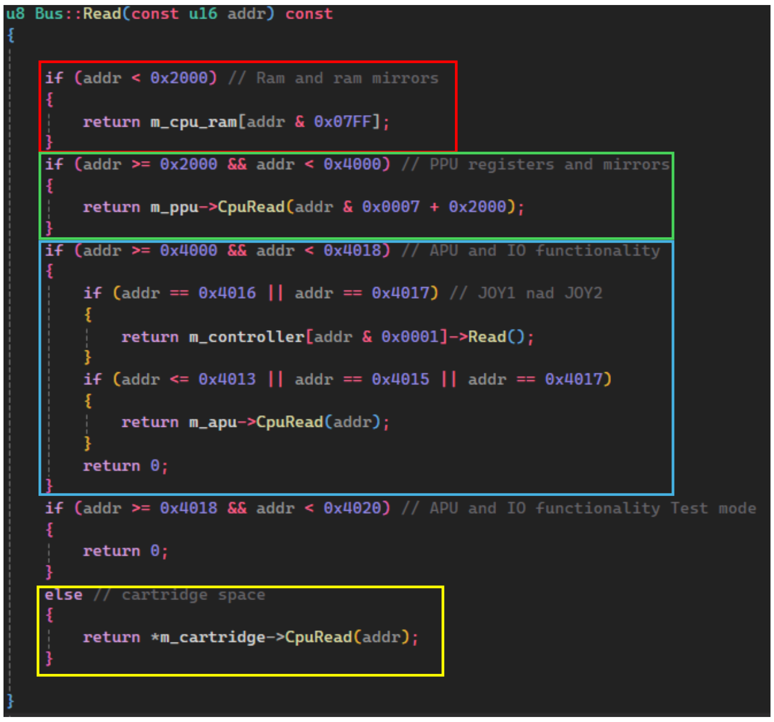
Figure 60.PPU NTSC Frame timing

I won't show the code of this part since it is a very long process, but in happens in the Step method on PPU.cpp and it follows the image shown above.

The Frame image is created by filling an image one pixel at a time until is full, that image is then sent to the renderer and loaded into a texture to be rendered in the screen.

7.2.2. Sending an Address Through the Bus

This happens in Bus.cpp in the Read or Write methods.



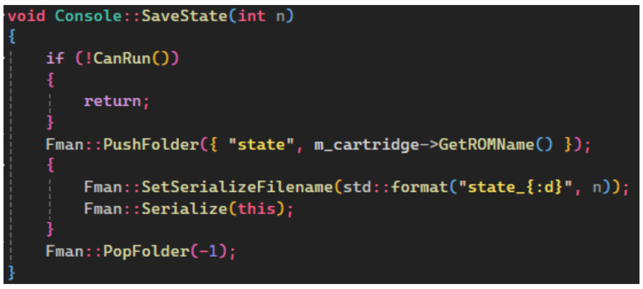
```
u8 Bus::Read(const u16 addr) const
{
    if (addr < 0x2000) // Ram and ram mirrors
    {
        return m_cpu_ram[addr & 0x07FF];
    }
    if (addr >= 0x2000 && addr < 0x4000) // PPU registers and mirrors
    {
        return m_ppu->CpuRead(addr & 0x0007 + 0x2000);
    }
    if (addr >= 0x4000 && addr < 0x4018) // APU and IO functionality
    {
        if (addr == 0x4016 || addr == 0x4017) // JOY1 nad JOY2
        {
            return m_controller[addr & 0x0001]->Read();
        }
        if (addr <= 0x4013 || addr == 0x4015 || addr == 0x4017)
        {
            return m_apu->CpuRead(addr);
        }
        return 0;
    }
    if (addr >= 0x4018 && addr < 0x4020) // APU and IO functionality Test mode
    {
        return 0;
    }
    else // cartridge space
    {
        return *m_cartridge->CpuRead(addr);
    }
}
```

Figure 61. Bus Address mapping

The code excerpt shown above, shows the memory range of the NES and how each section of it is sent to a different component (the APU part is a placeholder and does nothing). The rede part is the NES RAM, which is completely handled by the bus, the green part represents the PPU MMIO registers, the blue one is relevant to input and audio, but also holds the PPU DMA MMIO register in the write version of this method and the yellow part is the cartridge space; the cartridge space will be mapped further in accordance to the current mapping circuit, in this case, only mapper 0 is allowed, so the map would be 1 to 1 with some restrictions in range.

7.2.3. Saving State

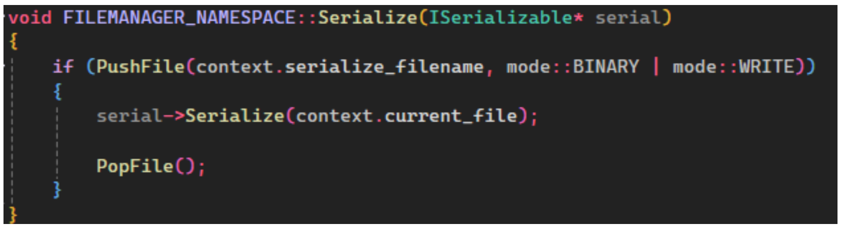
This call happens in Console.cpp.



```
void Console::SaveState(int n)
{
    if (!CanRun())
    {
        return;
    }
    Fman::PushFolder({ "state", m_cartridge->GetROMName() });
    {
        Fman::SetSerializeFilename(std::format("state_{:d}", n));
        Fman::Serialize(this);
    }
    Fman::PopFolder(-1);
}
```

Figure 62. Saving state

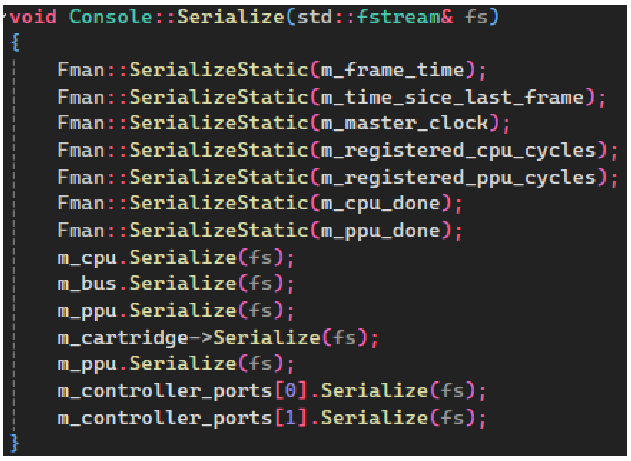
The save/load state is very straight forward; first, you move to the relevant file path for states, then you call the Serialize method in the ISerializable to be serialized.



```cpp
void FILEMANAGER_NAMESPACE::Serialize(ISerializable* serial)
{
    if (PushFile(context.serialize_filename, mode::BINARY | mode::WRITE))
    {
        serial->Serialize(context.current_file);

        PopFile();
    }
}
```

Figure 63. Serialization in the File Manager

This code is part of FileManagerImpl.cpp, it opens the file in binary write mode and then calls the serialization method of the object to be serialized, it passes the current fstream in case the provided API for serialization is not enough.



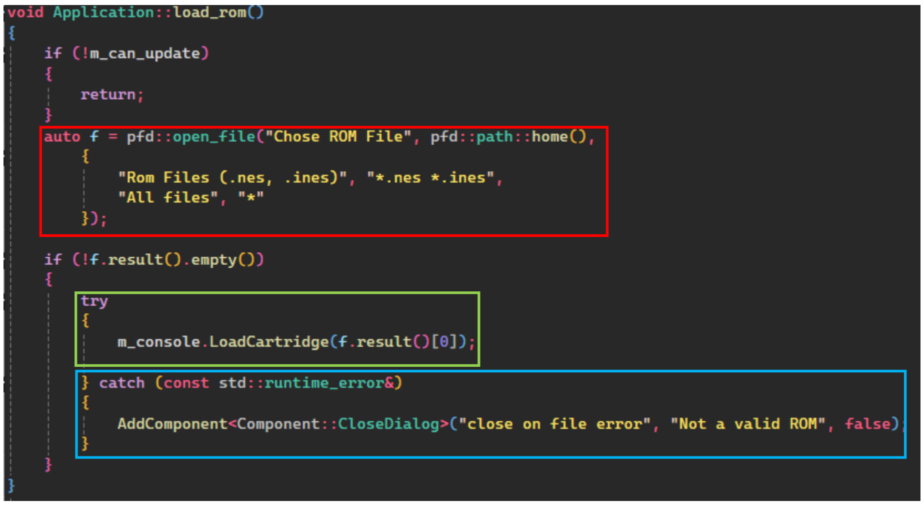
```cpp
void Console::Serialize(std::fstream& fs)
{
    Fman::SerializeStatic(m_frame_time);
    Fman::SerializeStatic(m_time_sice_last_frame);
    Fman::SerializeStatic(m_master_clock);
    Fman::SerializeStatic(m_registered_cpu_cycles);
    Fman::SerializeStatic(m_registered_ppu_cycles);
    Fman::SerializeStatic(m_cpu_done);
    Fman::SerializeStatic(m_ppu_done);
    m_cpu.Serialize(fs);
    m_bus.Serialize(fs);
    m_ppu.Serialize(fs);
    m_cartridge->Serialize(fs);
    m_ppu.Serialize(fs);
    m_controller_ports[0].Serialize(fs);
    m_controller_ports[1].Serialize(fs);
}
```

Figure 64. Console Serialization

This is what the serialization method in the console class looks like, I won't show how the different components serialization works as it is almost the same, you take all the relevant data needed to maintain the current state of the emulation, and you save it to a file; for a similar reason I won't show the deserialization part since it would be the same but changing the Serialize prefix with Deserialize.

7.2.4. Loading a ROM

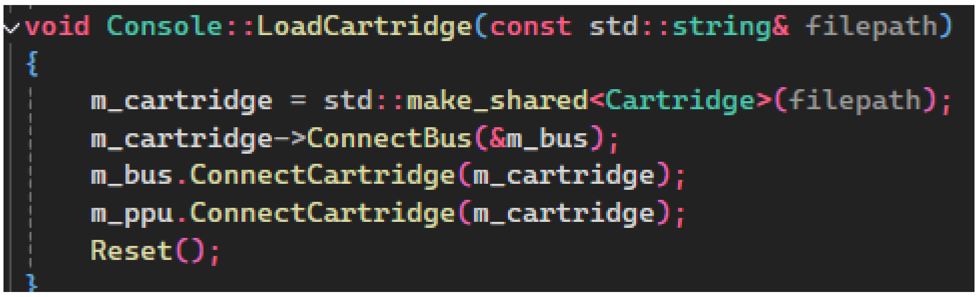
The ROM loading process starts in application.cpp.



```cpp
void Application::load_rom()
{
    if (!m_can_update)
    {
        return;
    }
    auto f = pfd::open_file("Chose ROM File", pfd::path::home(),
        {
            "Rom Files (.nes, .ines)", "*.nes *.ines",
            "All files", "*"
        });

    if (!f.result().empty())
    {
        try
        {
            m_console.LoadCartridge(f.result()[0]);
        } catch (const std::runtime_error&)
        {
            AddComponent<Component::CloseDialog>("close on file error", "Not a valid ROM", false)
        }
    }
}
```

Figure 65. Loading a ROM 1

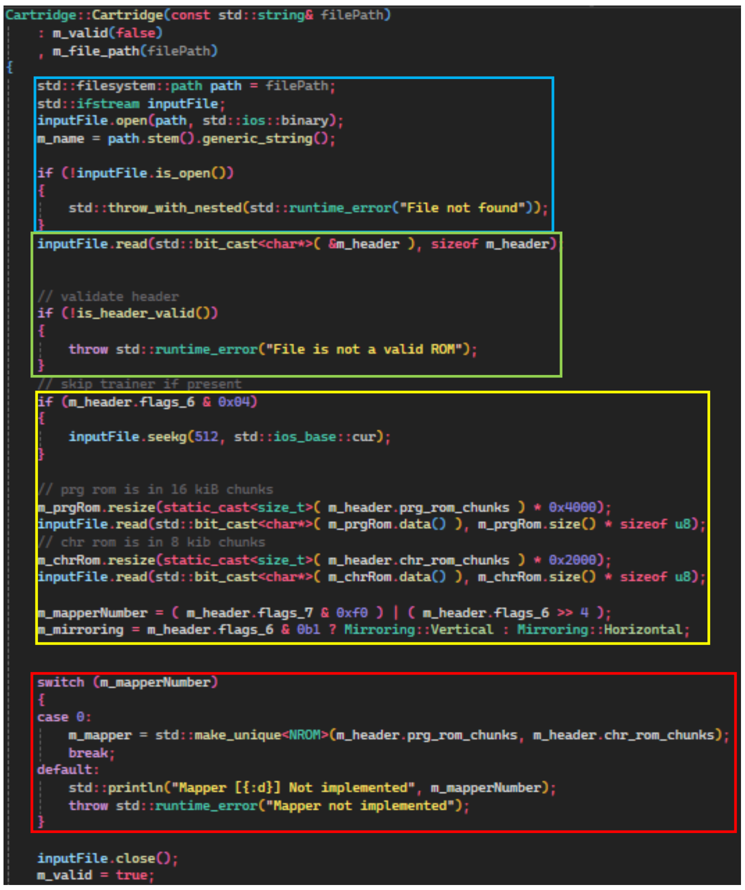
The red part is the one that handles opening a native file dialog to load the ROM, the green part is the one that tries to load a ROM after the file has been selected and the blue part is error handling in case the ROM is not valid.



```cpp
void Console::LoadCartridge(const std::string& filepath)
{
    m_cartridge = std::make_shared<Cartridge>(filepath);
    m_cartridge->ConnectBus(&m_bus);
    m_bus.ConnectCartridge(m_cartridge);
    m_ppu.ConnectCartridge(m_cartridge);
    Reset();
}
```

Figure 66. Loading a ROM 2

This is the function called in application, this fragment of code creates the cartridge, and links it to the different components, after linking it, it resets the console.



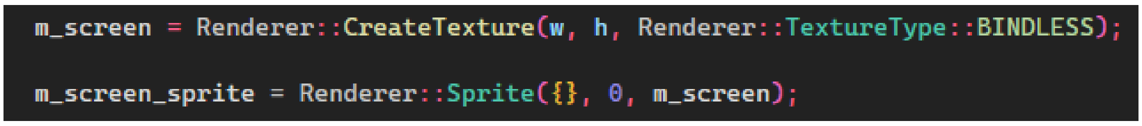
```cpp
Cartridge::Cartridge(const std::string& filePath)
    : m_valid(false)
    , m_file_path(filePath)
{
    std::filesystem::path path = filePath;
    std::ifstream inputFile;
    inputFile.open(path, std::ios::binary);
    m_name = path.stem().generic_string();

    if (!inputFile.is_open())
    {
        std::throw_with_nested(std::runtime_error("File not found"));
    }
    inputFile.read(std::bit_cast<char*>( &m_header ), sizeof m_header);


    // validate header
    if (!is_header_valid())
    {
        throw std::runtime_error("File is not a valid ROM");
    }
    // skip trainer if present
    if (m_header.flags_6 & 0x04)
    {
        inputFile.seekg(512, std::ios_base::cur);
    }

    // prg rom is in 16 kiB chunks
    m_prgRom.resize(static_cast<size_t>( m_header.prg_rom_chunks ) * 0x4000);
    inputFile.read(std::bit_cast<char*>( m_prgRom.data() ), m_prgRom.size() * sizeof u8);
    // chr rom is in 8 kib chunks
    m_chrRom.resize(static_cast<size_t>( m_header.chr_rom_chunks ) * 0x2000);
    inputFile.read(std::bit_cast<char*>( m_chrRom.data() ), m_chrRom.size() * sizeof u8);

    m_mapperNumber = ( m_header.flags_7 & 0xf0 ) | ( m_header.flags_6 >> 4 );
    m_mirroring = m_header.flags_6 & 0b1 ? Mirroring::Vertical : Mirroring::Horizontal;

    switch (m_mapperNumber)
    {
    case 0:
        m_mapper = std::make_unique<NROM>(m_header.prg_rom_chunks, m_header.chr_rom_chunks);
        break;
    default:
        std::println("Mapper [{:d}] Not implemented", m_mapperNumber);
        throw std::runtime_error("Mapper not implemented");
    }

    inputFile.close();
    m_valid = true;
```

Figure 67. Loading a ROM 3

This is the main constructor of the Cartridge class; quite a lot of things are happening here, as highlighted they are: the blue part is file opening, the green part is header validation, the yellow part is header parsing and the red part is mapper selection.

7.2.5. Creating a Texture

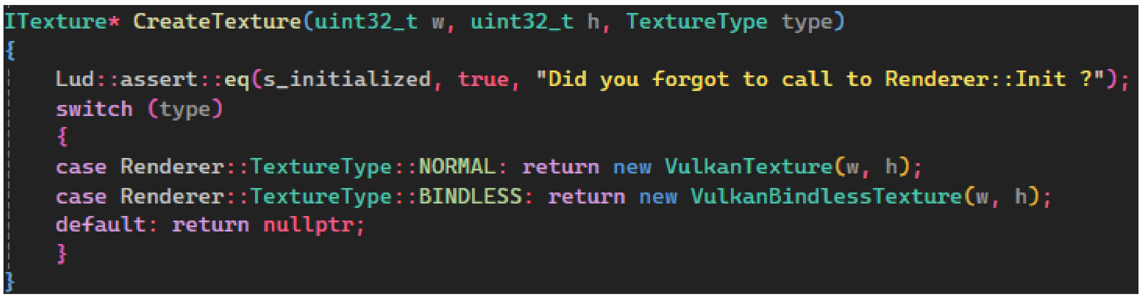
Creating a texture is a vital part of the visual part of the application, since a texture is used to render the emulation screen.



```cpp
m_screen = Renderer::CreateTexture(w, h, Renderer::TextureType::BINDLESS);

m_screen_sprite = Renderer::Sprite({}, 0, m_screen);
```

Figure 68. Create a Texture Sprite pair

In this fragment of code, we are creating a bindless texture, and then tying it to a sprite that will hold it since the sprite is the one that holds all the positional data.



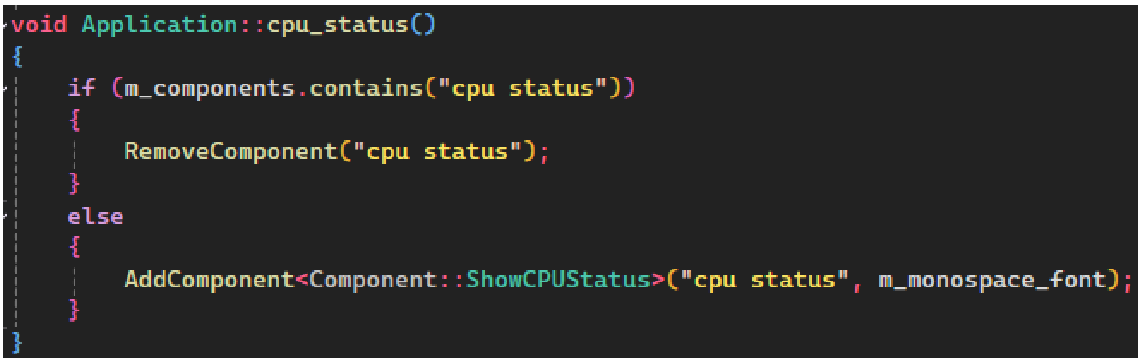
```cpp
ITexture* CreateTexture(uint32_t w, uint32_t h, TextureType type)
{
    Lud::assert::eq(s_initialized, true, "Did you forgot to call to Renderer::Init ?");
    switch (type)
    {
    case Renderer::TextureType::NORMAL: return new VulkanTexture(w, h);
    case Renderer::TextureType::BINDLESS: return new VulkanBindlessTexture(w, h);
    default: return nullptr;
    }
}
```

Figure 69. Creating the texture in the API

Then, in the Renderer API, the texture is created. Two types of textures can be created, the normal ones are meant for ImGui, and the bindless ones are meant for the application. Then, the texture is created in Engine.cpp, the finer detail will not be show here, but it must be noted that it is verbatim the same as the one in vkguide chapter 4.

7.2.6. Creating an Interface

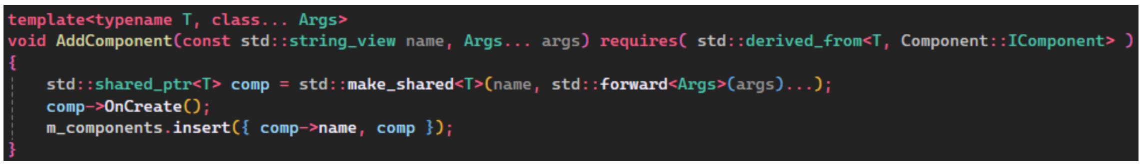
To create the different components, as stated before, I went with a component system, in that way I could create different components without needing to modify the main code to accommodate them. This is an example of how the CPU status component is created.



```cpp
void Application::cpu_status()
{
    if (m_components.contains("cpu status"))
    {
        RemoveComponent("cpu status");
    }
    else
    {
        AddComponent<Component::ShowCPUStatus>("cpu status", m_monospace_font);
    }
}
```

Figure 70. Creating an interface 1

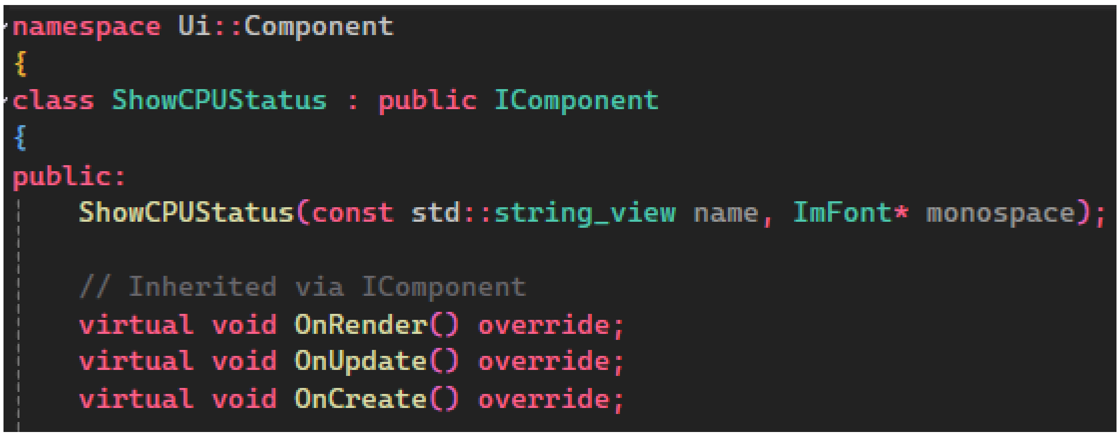
First, we check if the component is already added, and if it is, we remove it, if it isn't, the AddComponent teamplate function is called.



```cpp
template<typename T, class... Args>
void AddComponent(const std::string_view name, Args... args) requires( std::derived_from<T, Component::IComponent> )
{
    std::shared_ptr<T> comp = std::make_shared<T>(name, std::forward<Args>(args)...);
    comp->OnCreate();
    m_components.insert({ comp->name, comp });
}
```

Figure 71. Creating an interface 2

The AddComponent template function will create a smart ptr to that component, call the OnCreate of that component, and insert it into an unordered map to be stored.



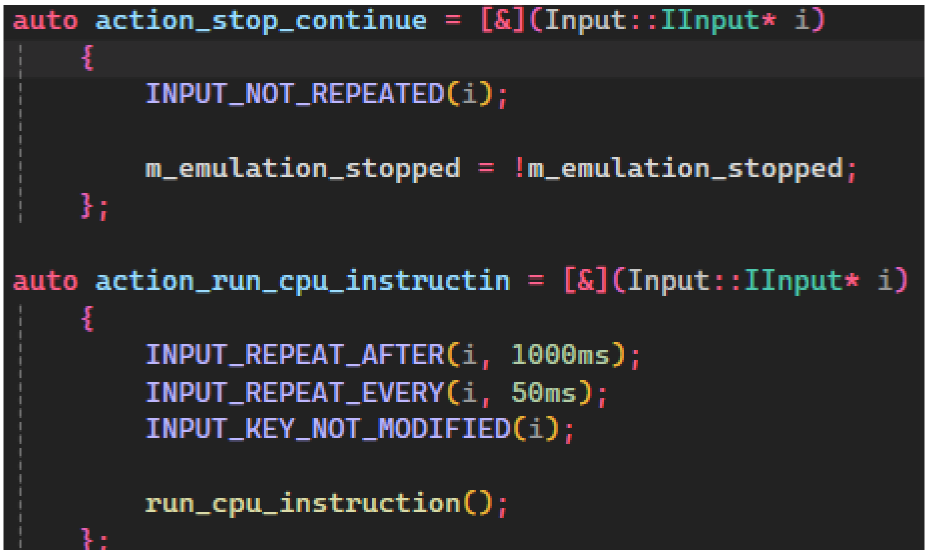
```
namespace Ui::Component
{
class ShowCPUStatus : public IComponent
{
public:
    ShowCPUStatus(const std::string_view name, ImFont* monospace);

    // Inherited via IComponent
    virtual void OnRender() override;
    virtual void OnUpdate() override;
    virtual void OnCreate() override;
```

Figure 72. Creating an interface 3

Then all components have an OnRender function, that is the one that creates the user interface; an onUpdate function, that is called on the update part of the application main loop; and an OnCreate function that is called when the component is created.

7.2.7. Input Handling

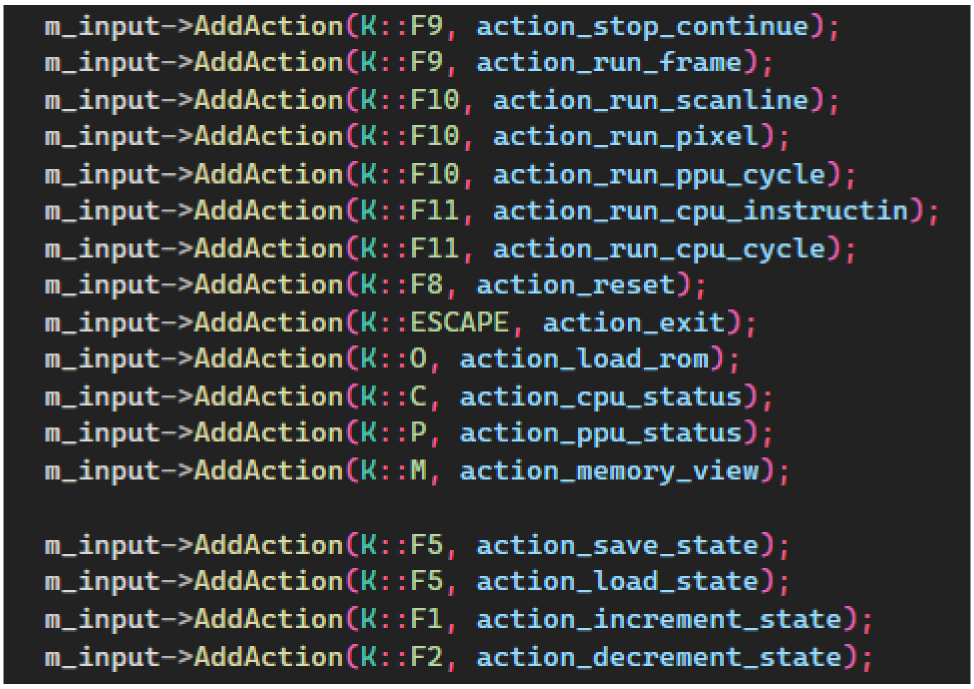
The Input Handler library that I built for this project works with actions, that means, “prerecorded” functions that are executed when some conditions are met.



```
auto action_stop_continue = [&](Input::IInput* i)
    {
        INPUT_NOT_REPEATED(i);

        m_emulation_stopped = !m_emulation_stopped;
    };

auto action_run_cpu_instructin = [&](Input::IInput* i)
    {
        INPUT_REPEAT_AFTER(i, 1000ms);
        INPUT_REPEAT_EVERY(i, 50ms);
        INPUT_KEY_NOT_MODIFIED(i);

        run_cpu_instruction();
    };
```

Figure 73. Input Handling 1

In this case, some macros are used to restrict how this actions are called, for example, the INPUT_NOT_REPEATED one makes the action to be called only once until the button is released and pressed again, the INPUT_KEY_NOT_MODIFIED forbids the action to be run if any modifiers (ctrl, shit, alt...) are pressed.

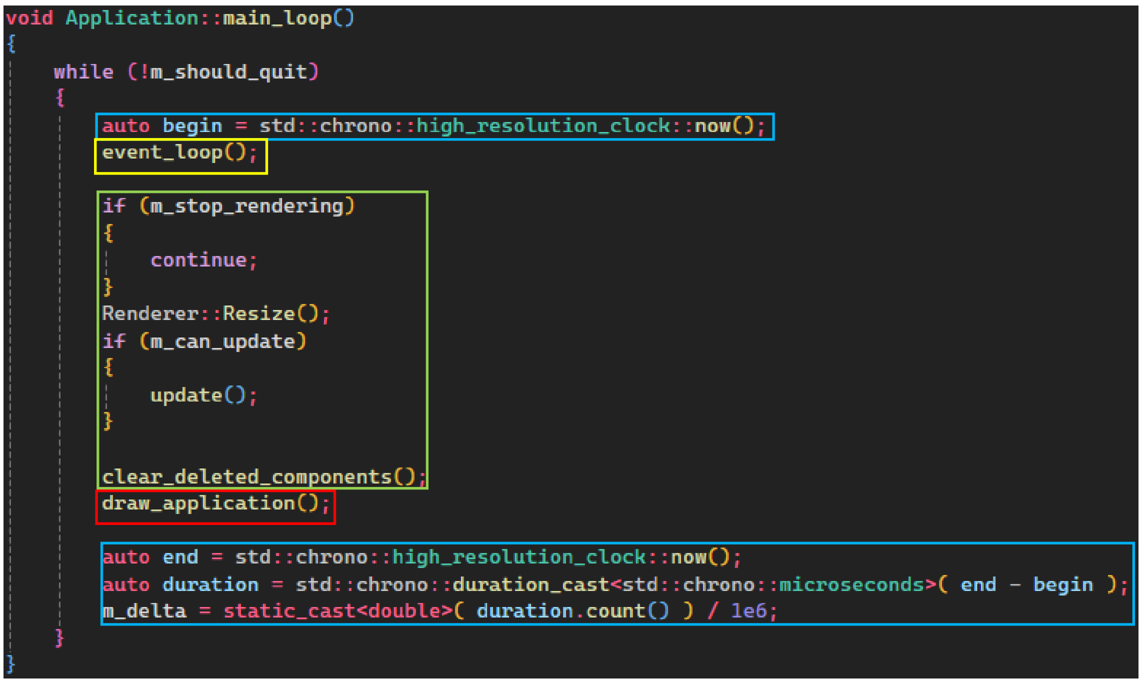


```cpp
m_input->AddAction(K::F9, action_stop_continue);
m_input->AddAction(K::F9, action_run_frame);
m_input->AddAction(K::F10, action_run_scanline);
m_input->AddAction(K::F10, action_run_pixel);
m_input->AddAction(K::F10, action_run_ppu_cycle);
m_input->AddAction(K::F11, action_run_cpu_instructin);
m_input->AddAction(K::F11, action_run_cpu_cycle);
m_input->AddAction(K::F8, action_reset);
m_input->AddAction(K::ESCAPE, action_exit);
m_input->AddAction(K::O, action_load_rom);
m_input->AddAction(K::C, action_cpu_status);
m_input->AddAction(K::P, action_ppu_status);
m_input->AddAction(K::M, action_memory_view);

m_input->AddAction(K::F5, action_save_state);
m_input->AddAction(K::F5, action_load_state);
m_input->AddAction(K::F1, action_increment_state);
m_input->AddAction(K::F2, action_decrement_state);
```

Figure 74. Input Handling 2

Once the action is recorded, then it is assigned to a trigger, that can be a key or a gamepad button. I'm not happy with the current design, and I'd like to change it to an alias-based action system, that would mean, that you would create an alias, then you would assign the triggers to the alias, and finally you would bind the actions to the alias, that would make rebinding actions pretty easy.

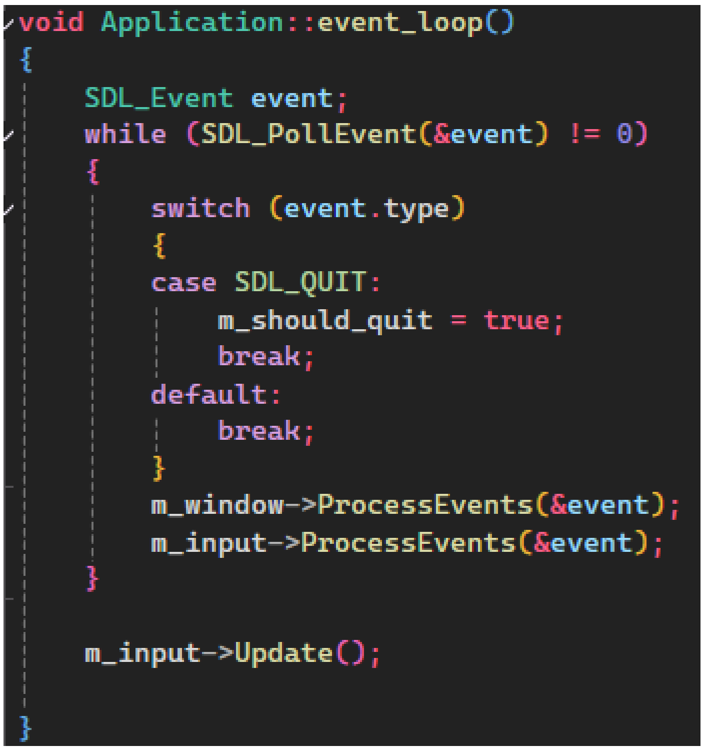
7.2.8. Main Loop



```cpp
void Application::main_loop()
{
    while (!m_should_quit)
    {
        auto begin = std::chrono::high_resolution_clock::now();
        event_loop();

        if (m_stop_rendering)
        {
            continue;
        }
        Renderer::Resize();
        if (m_can_update)
        {
            update();
        }

        clear_deleted_components();
        draw_application();

        auto end = std::chrono::high_resolution_clock::now();
        auto duration = std::chrono::duration_cast<std::chrono::microseconds>( end - begin );
        m_delta = static_cast<double>( duration.count() ) / 1e6;
    }
}
```

Figure 75. Main Loop

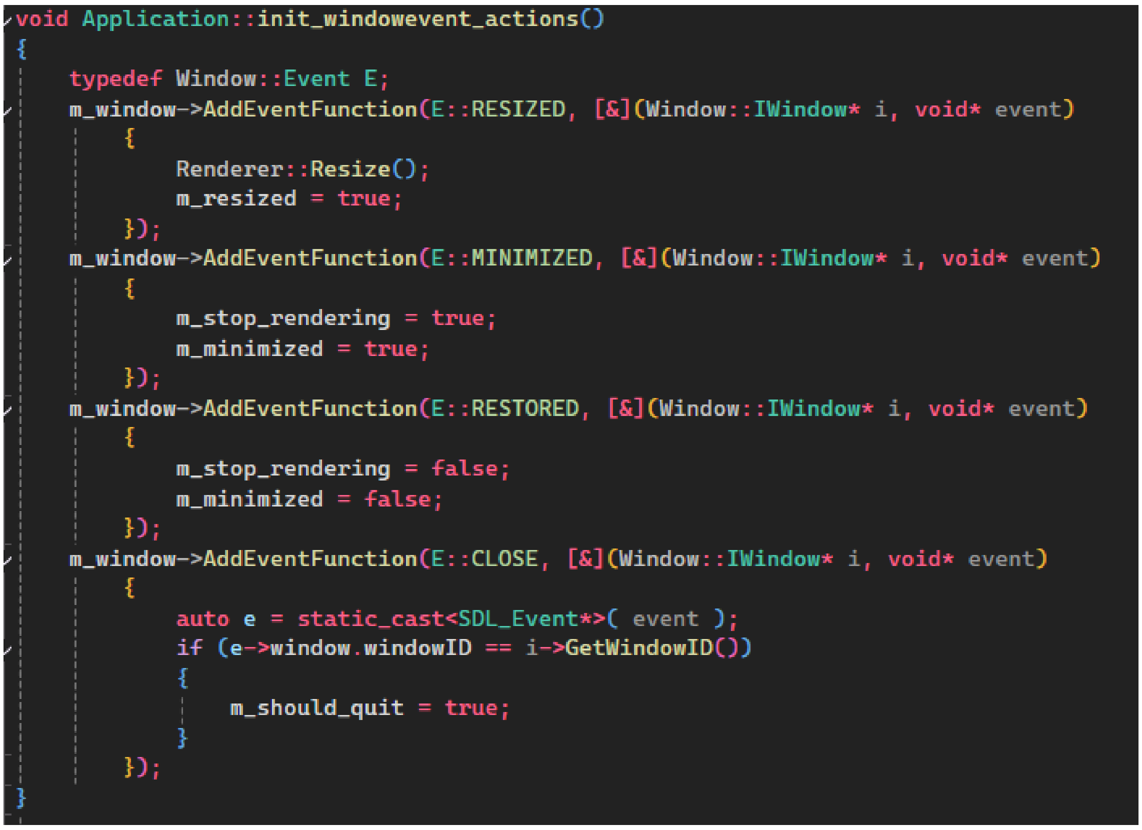
This is the main loop of the application, this is the beating heart of the application, is the code that makes sure that events are propagated to their components (yellow), is the code that keeps track of the application delta (blue), is the code that tells the renderer to draw to screen and is the code (red) that updates the application (green).



```cpp
void Application::event_loop()
{
    SDL_Event event;
    while (SDL_PollEvent(&event) != 0)
    {
        switch (event.type)
        {
        case SDL_QUIT:
            m_should_quit = true;
            break;
        default:
            break;
        }
        m_window->ProcessEvents(&event);
        m_input->ProcessEvents(&event);
    }

    m_input->Update();

}
```

Figure 76. Event loop

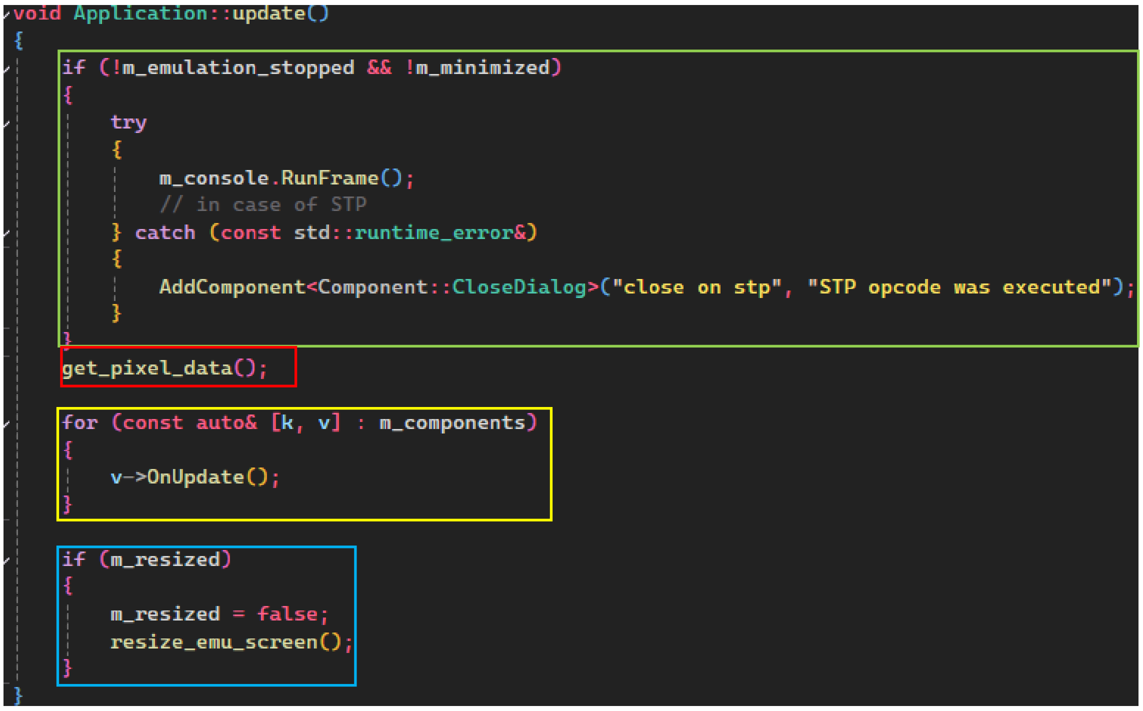
This is the event loop, since both the window and the input handler need to know when their respective events are fired, this code will extract all events from the SDL event queue until its exhausted (that's why the application freezes when its being dragged or resized, I don't like it) and propagates the events to the input handler and the window.



```cpp
void Application::init_windowevent_actions()
{
    typedef Window::Event E;
    m_window->AddEventFunction(E::RESIZED, [&](Window::IWindow* i, void* event)
        {
            Renderer::Resize();
            m_resized = true;
        });
    m_window->AddEventFunction(E::MINIMIZED, [&](Window::IWindow* i, void* event)
        {
            m_stop_rendering = true;
            m_minimized = true;
        });
    m_window->AddEventFunction(E::RESTORED, [&](Window::IWindow* i, void* event)
        {
            m_stop_rendering = false;
            m_minimized = false;
        });
    m_window->AddEventFunction(E::CLOSE, [&](Window::IWindow* i, void* event)
        {
            auto e = static_cast<SDL_Event*>( event );
            if (e->window.windowID == i->GetWindowID())
            {
                m_should_quit = true;
            }
        });
}
```

Figure 77. Programming window events

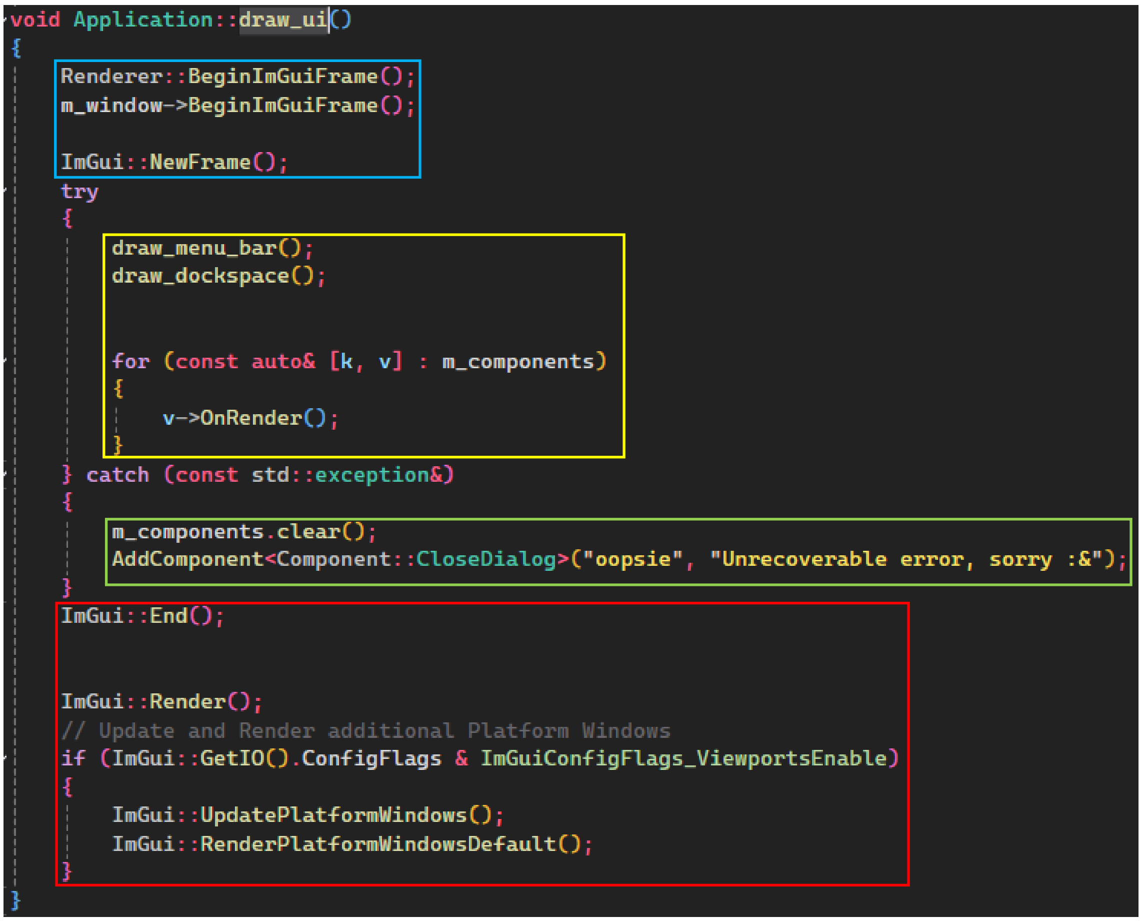
Since I intend for the window library to be standalone, I needed a way to program the events since I could use this library in other application with different needs in the window events. These events are then mapped in the SDL implementation to SDL window events, and when they are fired, the recorded function is executed. The last event, the close one, is needed so when a window that is not the main one is closed, for example, the memory view, the whole application is not closed.



```
void Application::update()
{
    if (!m_emulation_stopped && !m_minimized)
    {
        try
        {
            m_console.RunFrame();
            // in case of STP
        } catch (const std::runtime_error&)
        {
            AddComponent<Component::CloseDialog>("close on stp", "STP opcode was executed");
        }
    }
    get_pixel_data();

    for (const auto& [k, v] : m_components)
    {
        v->OnUpdate();
    }

    if (m_resized)
    {
        m_resized = false;
        resize_emu_screen();
    }
}
```

Figure 78. Update method

This is the update function, it runs the emulator (green), sets the texel data for the emulator screen (red), updates the components (yellow) and when the app is resized, it changes the dimensions of the sprite that holds the screen texture (blue).



```cpp
void Application::draw_ui()
{
    Renderer::BeginImGuiFrame();
    m_window->BeginImGuiFrame();

    ImGui::NewFrame();
    try
    {
        draw_menu_bar();
        draw_dockspace();


        for (const auto& [k, v] : m_components)
        {
            v->OnRender();
        }
    } catch (const std::exception&)
    {
        m_components.clear();
        AddComponent<Component::CloseDialog>("oopsie", "Unrecoverable error, sorry :&");
    }
    ImGui::End();


    ImGui::Render();
    // Update and Render additional Platform Windows
    if (ImGui::GetIO().ConfigFlags & ImGuiConfigFlags_ViewportsEnable)
    {
        ImGui::UpdatePlatformWindows();
        ImGui::RenderPlatformWindowsDefault();
    }
}
```

Figure 79. Drawing the User Interface

This is the code that orders the renderer to draw things, first, it needs to begin the frame for ImGui (blue), then it draws the application (yellow), if any error happens in drawing the application, an error component is created, and the application is closed when the user OKs the error message (green) and finally it ends the ImGui frame (red).

In future iterations of this project, I'd like to rework the menu bar as a component to decouple it from the application.

7.3. ISC 3: Unit Tests Execution

This section contains the results found while executing the tests defined in 6.3.1.

The first run was very successful with 79 successful tests of 80 total. The singular test that failed is the following.

Description	Expected result
6.3.1.11.ASSEMBLE_NESTEST The NesTest ROM will be run through the disassembler and compared to a known excerpt of the original assembly.	The retrieved assembly should match the known assembly.
	Obtained result
	The retrieved assembly contained a lot of labels pointing to empty addresses before the known assembly.
Problem	**Solution**
The disassembler is creating paths through the JMP instruction in indirect addressing.	The disassembler cannot trace through Indirect addressing with the JMP instruction.

Table 153. 6.3.1.11.ASSEMBLE_NESTEST results

7.4. ISC 4: Integration Tests Execution

No errors were found in the integrations tests.

7.5. ISC 5: System Tests Execution

This section contains the results found while executing the tests defined in 6.3.3. It also contains the reason these results were found to begin with.

7.5.1. NesTest

The NesTest test was unsuccessful at first, it kept showing errors that traced back to the SBC instruction and to the JMP instruction, this errors were not detected by the unit tests because I made the tests with a flawed understanding of how the overflow flag should work in the SBC instruction since I applied the same process to the ADC overflow, and in the case of the JMP, the error was in the indirect addressing mode, since the original hardware has aa bug that I was not replicating properly; so even if 79 of the 80 tests were successful, some of them were faulty since they were not properly made.

In order to fix these errors, the execution of the emulator had to be traced and compared to a known valid trace created by a highly accurate emulator since the meaning of the error codes provided by NesTest were lost to the internet long ago.

Description	Expected result
6.3.3.1.RUN_NESTEST The NesTest ROM is run.	No error codes are generated by NesTest.
	Obtained result
	Errors are generated by NesTest.
Problem	**Solution**
SBC overflow flag is faulty. Indirect addressing hardware bug is not properly implemented.	Fix to SBC overflow. Fix to the indirect addressing hardware bug.

Table 154.6.3.3.1.RUN_NESTEST results

7.6. ISC 6: Elaboration of User Manuals

7.6.1. Installation Manual

This project is a desktop app compiled for windows, in order to instal it properly, the user needs the following:

- A Vulkan SDK installation.
- Up to date drivers that support Vulkan 1.3

If those two are installed, just download the executable file.

7.6.2. Compilation Manual

Some previous requirements are needed to compile this system.

- C++ compiler that is compliant with C++20/23 (MSVC).
- Vulkan SDK.
- Python 3.

PreMake is not needed since it’s included in the project files. For the compiler I recommend MSVC, and I don’t promise that it works with Cygwin or MinGW since I have not tested, I know for a fact that the GNU make result generated with PreMake does not work.

For the Vulkan SDK I used the 1.3.280.0 version, I don’t know if it complies with newer versions.

For Python I use the 3.10.11 version.

If the project was to be clone from its repository when I decide to make it public, it’s imperative to clone with the recursive flag since It makes heavy use of submodules.

Once all requirements are fulfilled, run the MakeProkect.bat batch file, this program will generate the project files needed for the compilation to work, after running, open MSVC with the generated solution file, and make sure that the starting project is Application, once the start project is properly defined, just build as normal in either release or debug, this will create an exe file in the bin/<configuration-platform-architecture>/Application folder.

7.6.3. User Manual

This section contains a detailed guide on the systems contained in the system.

7.6.3.1. *The File Menu*

The Rom Loading process, reset process and exit process can be found in the File menu.

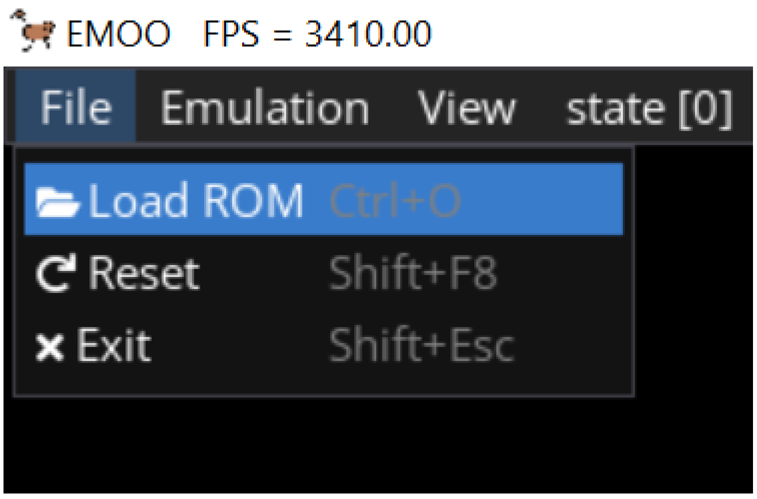


Figure 80. User Manual: File Menu

These processes can also be accessed with the help of the following key binds:

- Load ROM: Ctrl + O.
- Reset: Shift + F8.
- Exit: Shift + Esc.

7.6.3.2. *The Emulation Menu*

The emulation menu contains the tools to stop and advance the emulation in a controlled way.

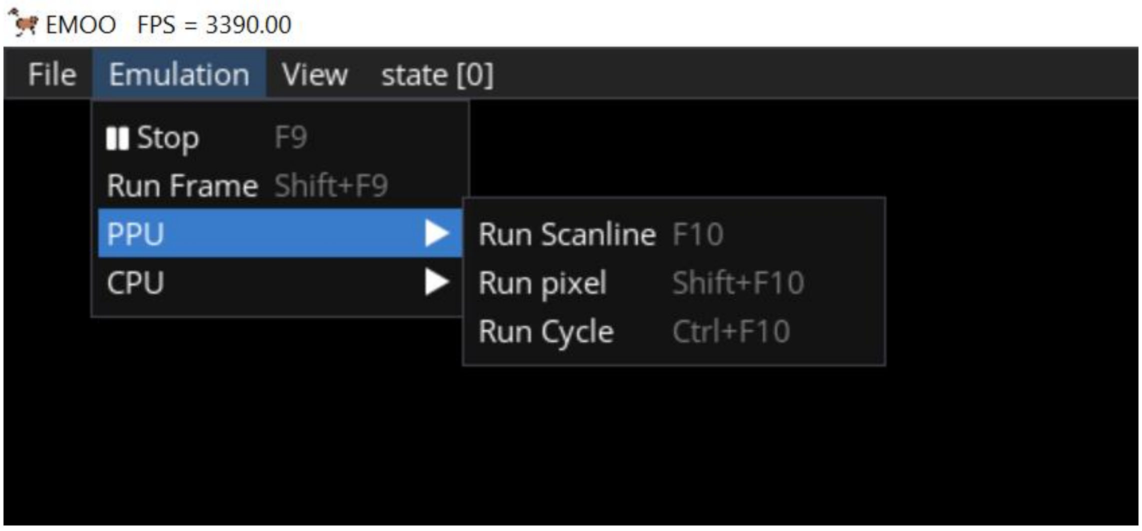


Figure 81. User Manual: Emulation Menu 1

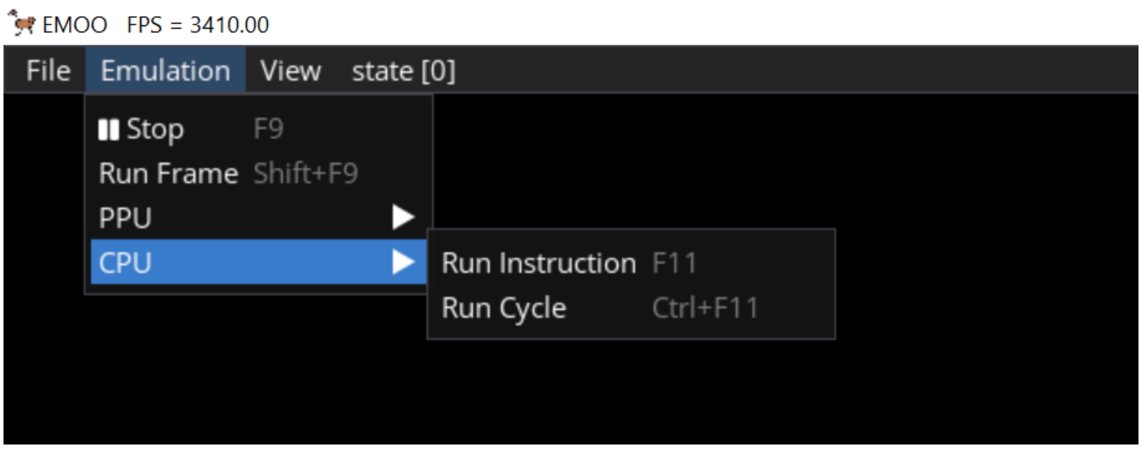


Figure 82. User Manual: Emulation Menu 2

The functionality of each button has been explained already, but these actions can be accessed with the following key binds:

- Stop or continue: F9.
- Run frame: Shift + F9.
- Run scanline: F10.
- Run pixel: Shift + F10.
- Run PPU cycle: Ctrl + F10.
- Run Instruction: F11.
- Run CPU Cycle: Ctrl + F11.

7.6.3.3. The View Menu

The view menu contains the tools to show and hide the CPU, PPU and memory views.

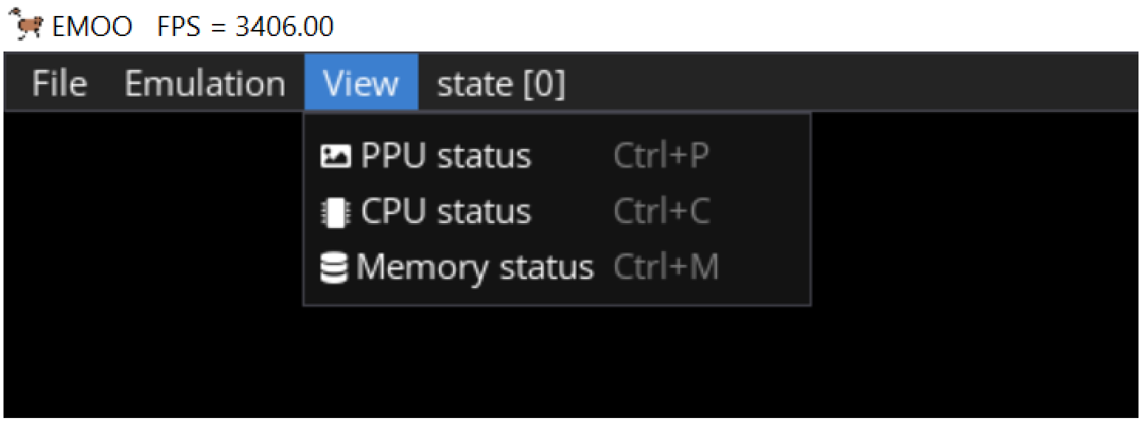


Figure 83. User Manual: View Menu

These views can also be accessed with the following key binds:

- PPU status: Ctrl + P.
- CPU status: Ctrl + C.
- Memory status: Ctrl + M.

7.6.3.3.1. PPU Status

The PPU status view is mostly used to check on the PPU status with little to do besides that, that does not mean the user can't do anything.

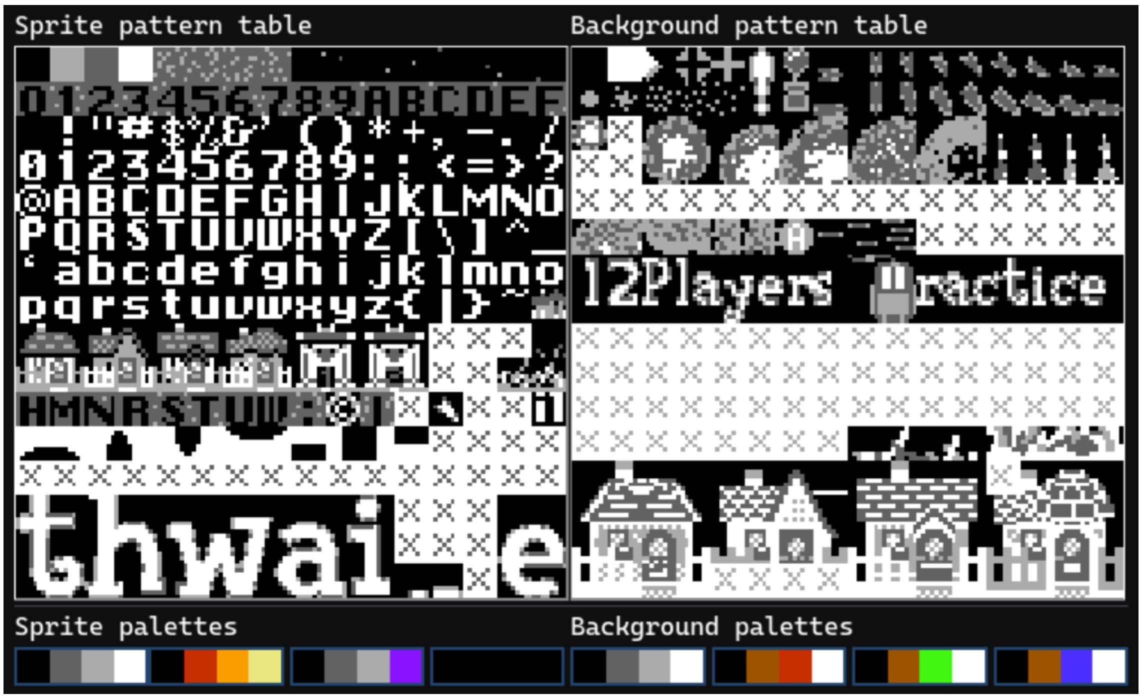


Figure 84. User Manual: PPU Status 1

In the section that contains the pattern tables and palettes the user can swap the palettes by pressing the desired palette and check the patterns by holding click on the pattern.

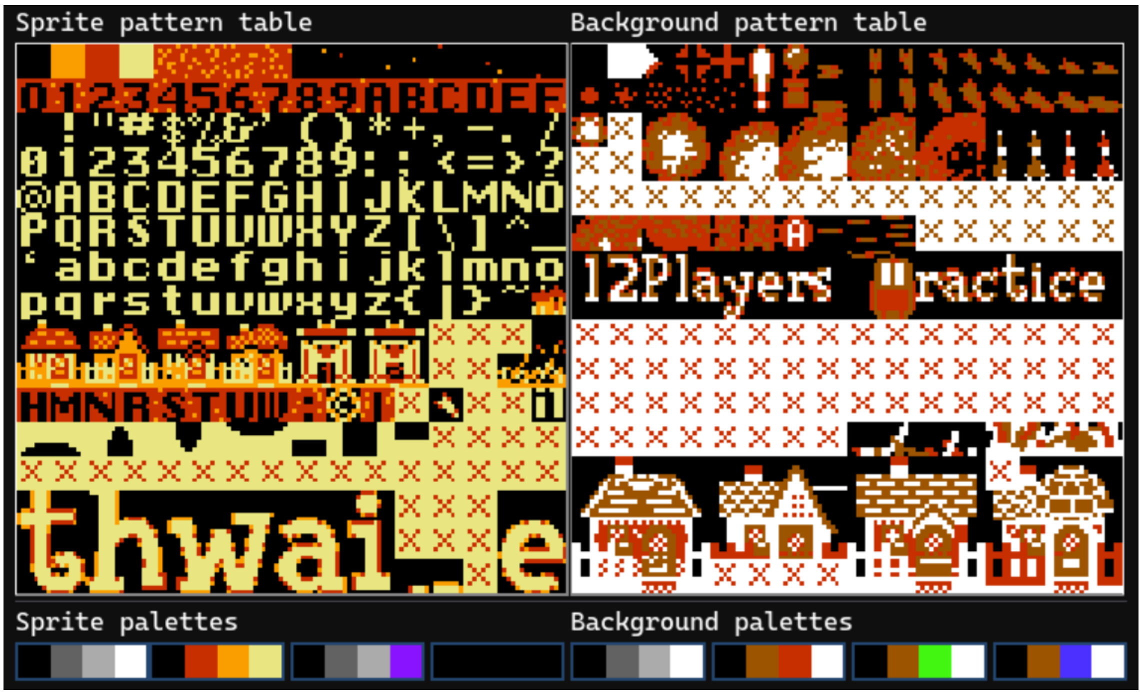


Figure 85. User Manual: PPU status 2

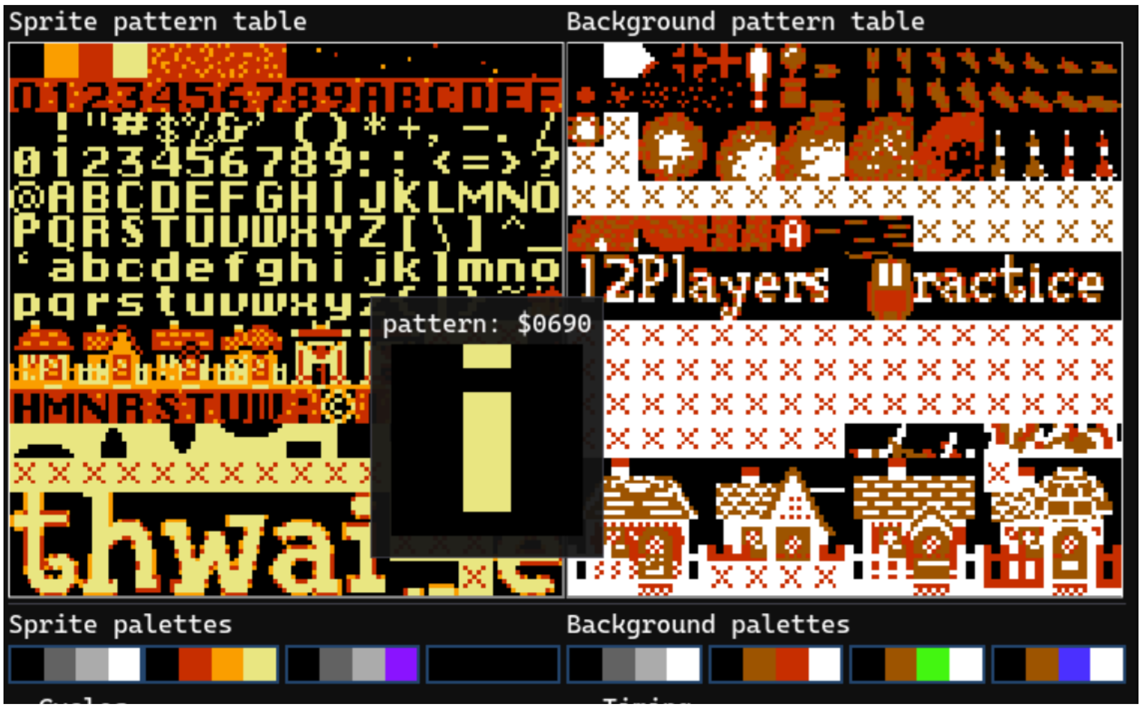
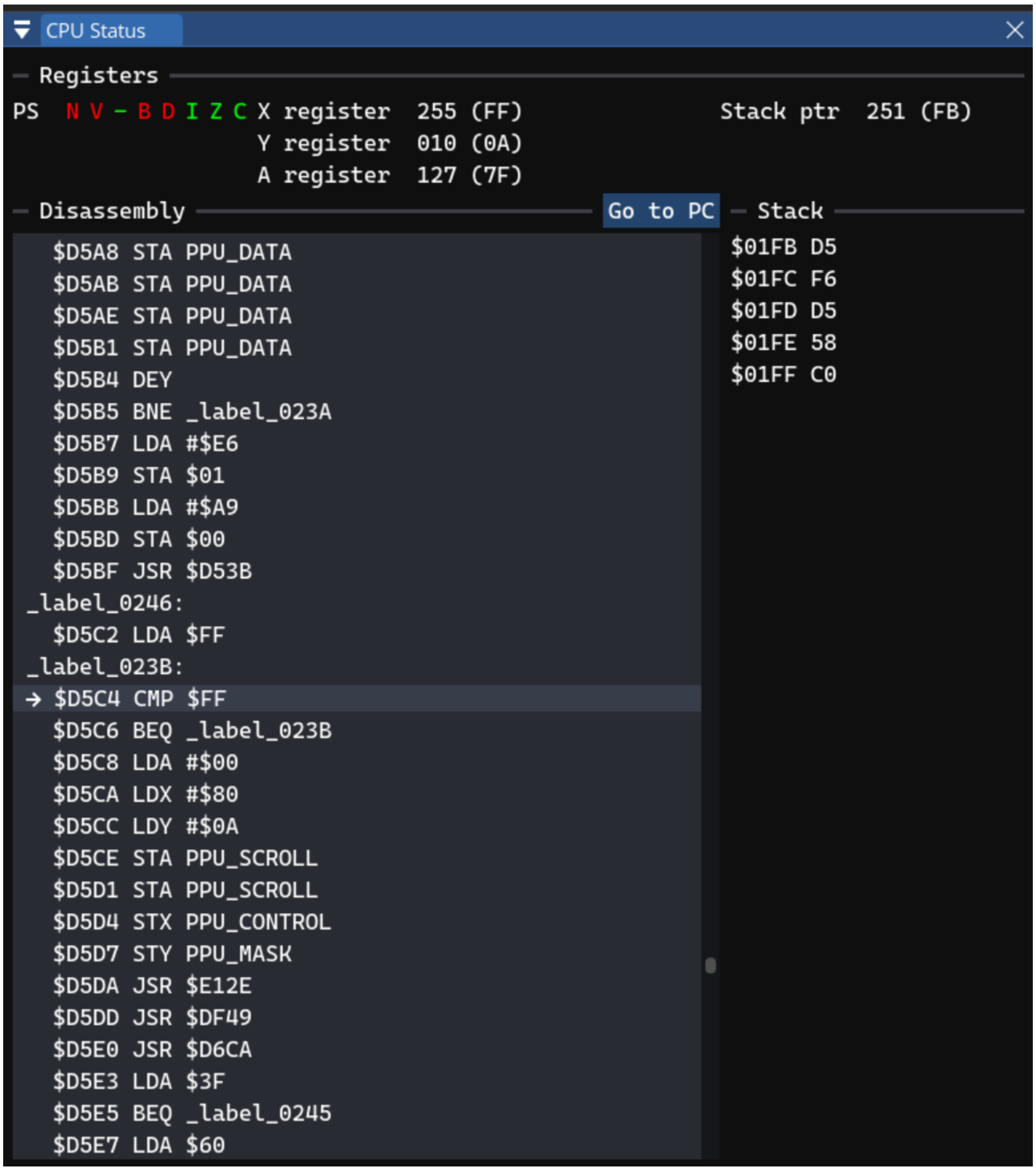


Figure 86. User Manual: PPU status 3

7.6.3.3.2. CPU status

The CPU status is like the PPU status, the only way to interact with it is to check the disassembly when the emulation is stopped.



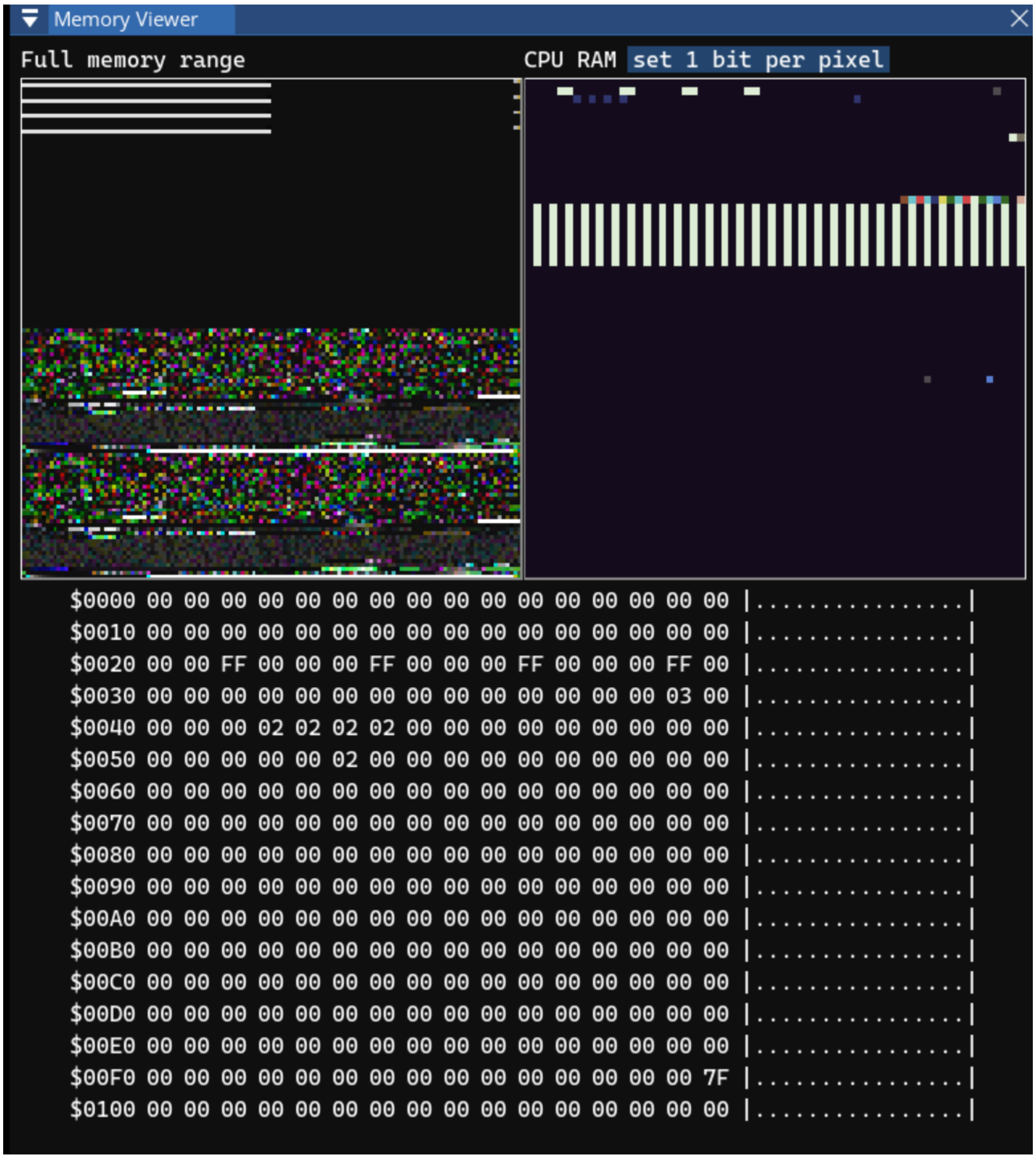
```
Registers
PS  N V - B D I Z C X register  255 (FF)          Stack ptr  251 (FB)
                    Y register  010 (0A)
                    A register  127 (7F)
Disassembly                                  Stack
   $D5A8 STA PPU_DATA                         $01FB D5
   $D5AB STA PPU_DATA                         $01FC F6
   $D5AE STA PPU_DATA                         $01FD D5
   $D5B1 STA PPU_DATA                         $01FE 58
   $D5B4 DEY                                  $01FF C0
   $D5B5 BNE _label_023A
   $D5B7 LDA #$E6
   $D5B9 STA $01
   $D5BB LDA #$A9
   $D5BD STA $00
   $D5BF JSR $D53B
 _label_0246:
   $D5C2 LDA $FF
 _label_023B:
 → $D5C4 CMP $FF
   $D5C6 BEQ _label_023B
   $D5C8 LDA #$00
   $D5CA LDX #$80
   $D5CC LDY #$0A
   $D5CE STA PPU_SCROLL
   $D5D1 STA PPU_SCROLL
   $D5D4 STX PPU_CONTROL
   $D5D7 STY PPU_MASK
   $D5DA JSR $E12E
   $D5DD JSR $DF49
   $D5E0 JSR $D6CA
   $D5E3 LDA $3F
   $D5E5 BEQ _label_0245
   $D5E7 LDA $60
```

Figure 87. User Manual: CPU status

The user can also press the “Go to PC” button to scroll to the current instruction.

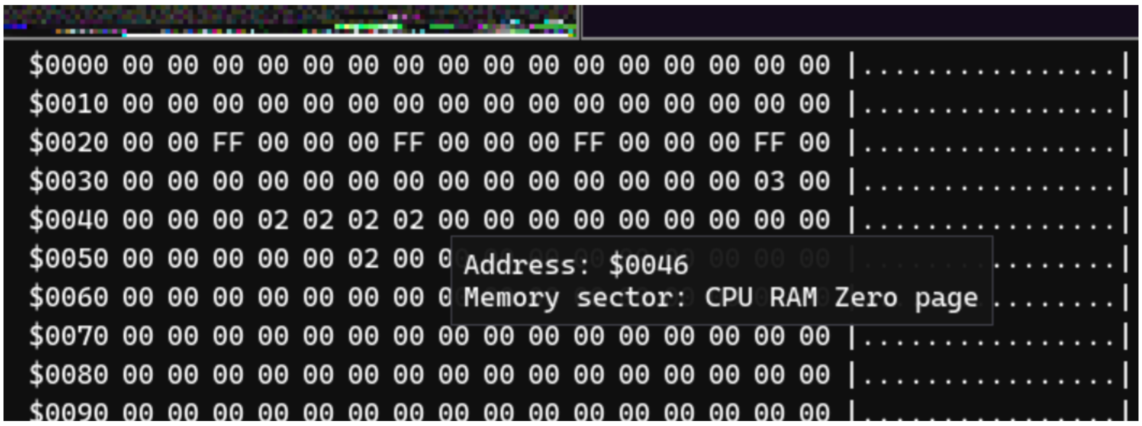
7.6.3.3.3. Memory status

The memory status view is the most flexible way, the user can change the representation of the CPU RAM (the second image) by pressing the button above it, and can inspect any memory address.



```
$0000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0020 00 00 FF 00 00 00 FF 00 00 00 FF 00 00 00 FF 00 |................|
$0030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03 00 |................|
$0040 00 00 00 02 02 02 02 00 00 00 00 00 00 00 00 00 |................|
$0050 00 00 00 00 00 02 00 00 00 00 00 00 00 00 00 00 |................|
$0060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$00A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$00B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$00C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$00D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$00E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$00F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 7F |................|
$0100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
```

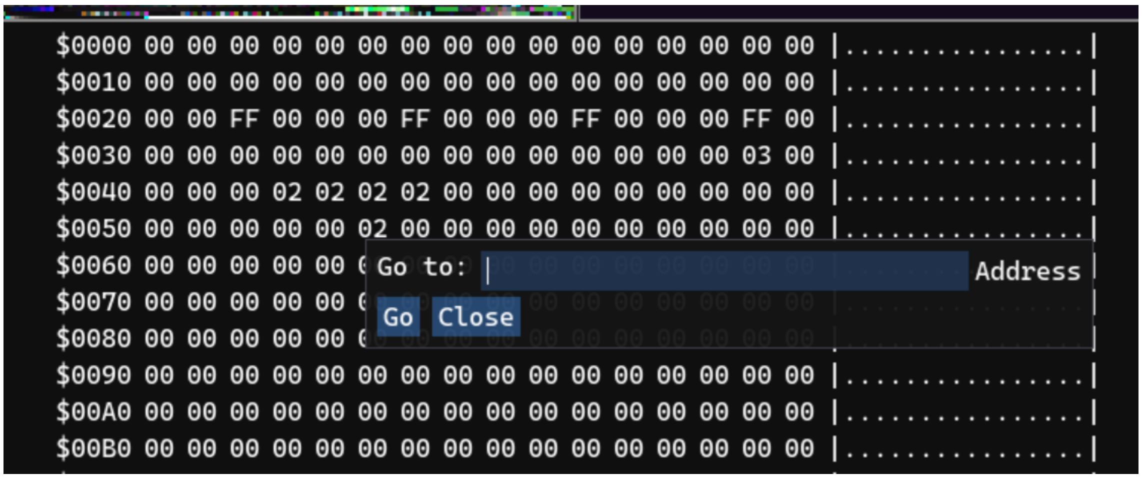
Figure 88. User Manual: Memory Status 1



```
$0000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0020 00 00 FF 00 00 00 FF 00 00 00 FF 00 00 00 FF 00 |................|
$0030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03 00 |................|
$0040 00 00 00 02 02 02 02 00 00 00 00 00 00 00 00 00 |................|
$0050 00 00 00 00 00 02 00 0 Address: $0046                  ........|
$0060 00 00 00 00 00 00 00 0 Memory sector: CPU RAM Zero page ........|
$0070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
```

Figure 89. User Manual: Memory status 2

The user can hover any location in the inspector, and it will show, what section of the memory it belongs to, and its address.



```
$0000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$0020 00 00 FF 00 00 00 FF 00 00 00 FF 00 00 00 FF 00 |................|
$0030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03 00 |................|
$0040 00 00 00 02 02 02 02 00 00 00 00 00 00 00 00 00 |................|
$0050 00 00 00 00 00 02 00 00 00 00 00 00 00 00 00 00 |................|
$0060 00 00 00 00 00 0 Go to: |                          Address |
$0070 00 00 00 00 00 0 Go  Close                                  |
$0080 00 00 00 00 00 0                                            |
$0090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$00A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
$00B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
```

Figure 90. User Manual: Memory status 3

The user can also right-click in the inspector to look up any desired address.

7.6.3.4. The State Menu

The state menu holds the functionality to save state, load state, increment state and decrement state, the current state is shown in the menu bar between brackets.

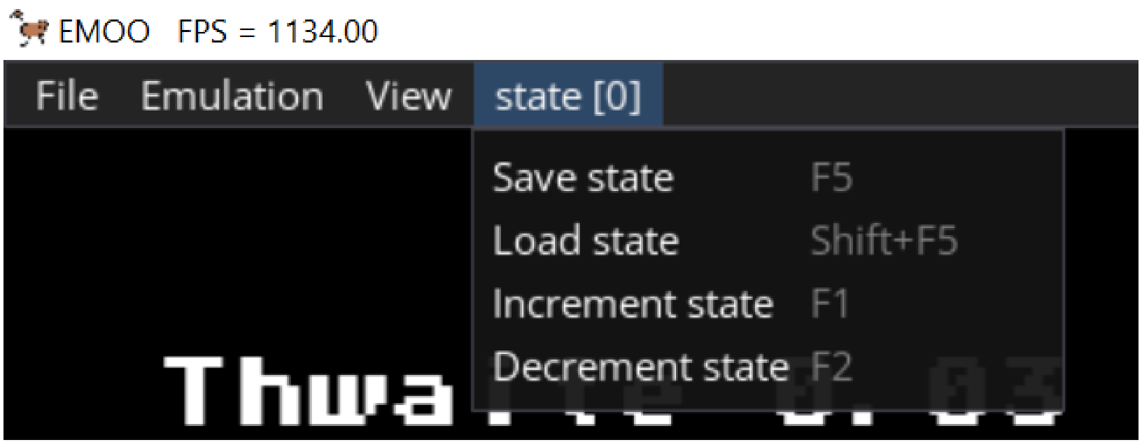


Figure 91. User Manual: State menu

These functionalities can also be accessed via the following key binds:

- Save state: F5
- Load state: Shift + F5
- Increment state: F1
- Decrement state: F2

7.6.3.5. Playing a game

To play a game, a controller is needed, I use an Xbox controller.

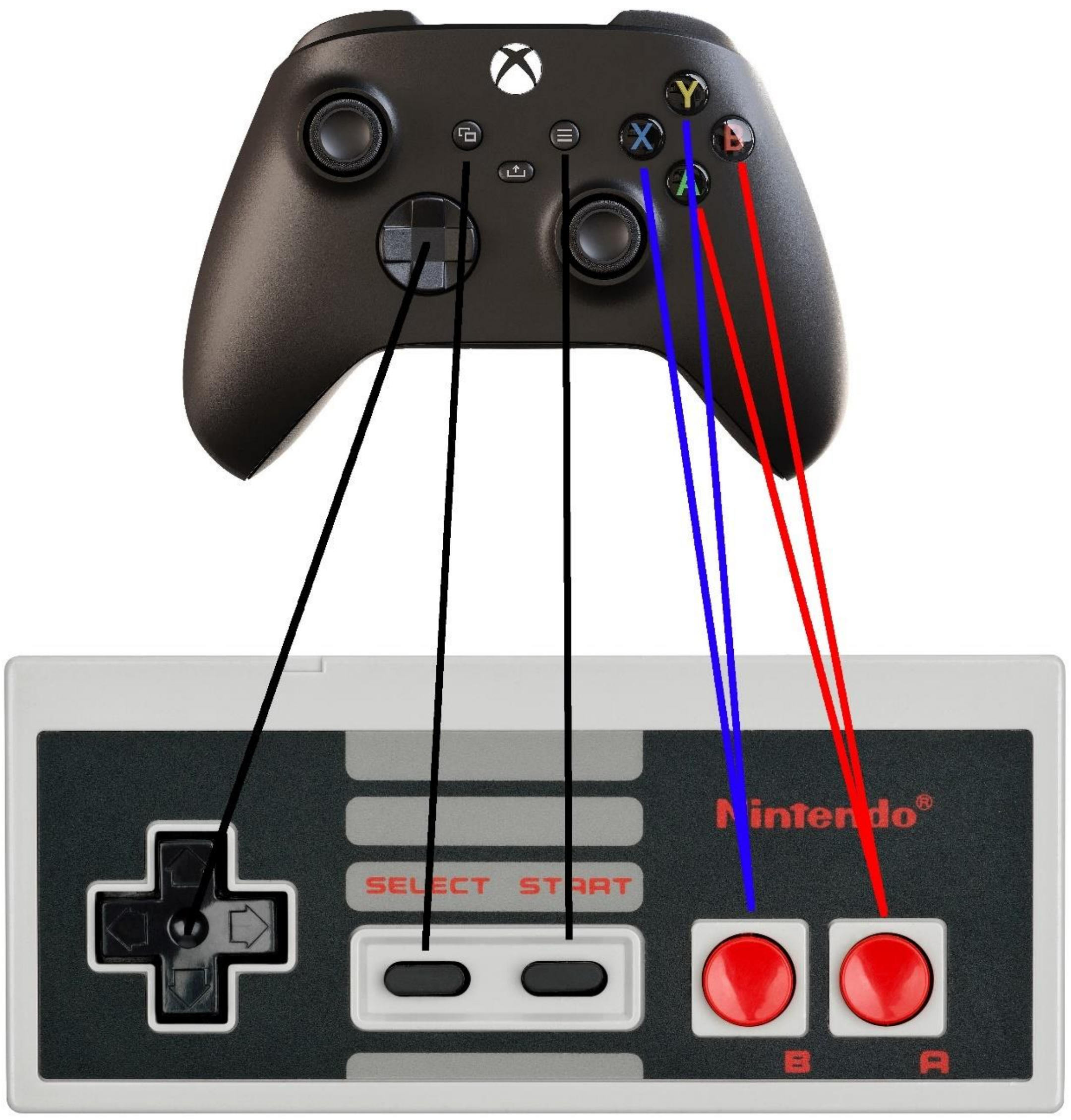


Figure 92. User Manual: Button mapping

The image above contains the button mappings for a modern controller, they are as follows:

- The d-pad is mapped to the d-pad.
- Start is mapped to start.
- Select is mapped to select.

- B is mapped to face up (Triangle or Y).
- B is mapped to face left (Square or X).
- A is mapped to face down (Cross or A).
- A is mapped to face right (Circle or B).

8. Conclusions and Future Work.

8.1. Conclusions

The main objective of this project was to learn and show what it takes to build a simple emulator, in the end, even the simplest emulator is a brutal thing to build, and this project excels in showing that, I also think that I was able to show how the NES works on the surface.

This project has been an incredible learning experience for me, thanks to undertaking it, I learnt about emulation, serialization, graphics programming, shader programming, user interface design, low level programming, optimization… It also has been a gateway to learn new code libraries and code techniques, it has changed the way I manage projects and the way I view coding.

There were less than stellar ideas in this project, the main one was a custom rendering solution that this project does not need or require, a problem that could have been fixed with a couple lines of SDL code, nonetheless, I'm happy I did it, even as it was the most painful and annoying part of this system.

I'm very happy with the overall architecture of the system even though is all over the place as I was trying new things while developing.

Personally, I think taking this project head on was one of the best things I have done in my short life as a developer, it was a theme that I found fascinating, and I was able to learn a lot while developing it.

8.2. Ampliations

This project is nowhere near close to being done, there are a lot of NES features left to implement and some quality-of-life things to do.

8.2.1. NES emulation

I would like to extend the system in the near future with the following features:

- Audio, the third and last major component would be to emulate the APU and add an audio system.
- Mappers, currently only mapper 0 is supported, and I'd like to add at least MMC1, MMC3 and VRC6.
- Cycle accuracy, as expected before, the CPU is not cycle accurate, I'd like to make sure it is.
- Colour correctness, as stated before, the current system makes use of a colour palette instead of the original composite signal, the NES dev wiki provides an algorithm to work it out, so I want to implement it.

There are probably more features that are missing, but these are the most important.

8.2.2. Quality-of-Life

As it is now, the application is a bit rough, here are some ideas to make it better.

- A way to rebind input, right now the inputs are hardcoded, that includes the controller and the key binds for the application, allowing the user to modify them would be great.
- Feedback when saving states, right now there is not feedback when saving state, so the user does not have a way to make sure the state was created.
- The application should not stop when resizing or dragging it, it stops right now and I don't like it.
- Multiplatform application, some steps have been taken already to ensure a multiplatform application, and in order to make it work I'd need to recompile some libraries and change the way the file manager works, but it's on the realm of possibility.
- Make the renderer better, as I went with the fully custom renderer, I'd like to make it thinner and better, the current iteration is my first take on it and I already have ideas on how to make it better.

9. Annex

9.1. Risks

9.1.1. Risk Identification

ID	Risk	Category	Description
1	Lack of experience	Organizational	The author has little experience in the field of emulation, which can lead to time underestimation since he has no time frame in which an emulator can be finished
2	Over scoping	Organizational	As the author does not know what is needed for the emulator to be considered finished, some unnecessary features can be added, which will led to more time
3	Internship	Managerial	At the time of developing this project, the author is currently looking for an internship, which can lead to work stoppage.
4	Nintendo	External	Even though emulators are legal, Nintendo has been known to go after emulators with shady legal practices to make them stop development
5	Incorrect estimation	Organizational	Some sections of the project can be wrongly estimated since the author is not the best at project planning and management

6	External libraries	Technical	This project makes use of external third-party libraries, of which the author has no control, these libraries can have bugs or have incorrect documentation.
7	Hard concepts	Technical	Low Level Emulators are made emulating the hardware of consoles, to do that, it is crucial to have a correct and accurate breakdown of the hardware, if some part of the hardware is not correctly documented, the author will have to look for that information elsewhere.

Table 155. Risk identification

9.1.2. Risk Impact

Using the values defined in table 26. We can assess the probability of a risk occurring.

Impact	Range	Value
Very Low	[0%..20%]	10%
Low	(20%..40%]	30%
Medium	(40%..60%]	50%
High	(60%..80%]	70%
Very High	(80%..100%]	90%

Table 156. Impact Probability Definitions

The following tables contains the probability of a risk given the impact given to it

Probability

Very High	0.9	0.05	0.14	0.27	0.50	0.81
High	0.7	0.04	0.11	0.21	0.39	0.63
Medium	0.5	0.03	0.08	0.15	0.28	0.45
Low	0.3	0.02	0.05	0.09	0.17	0.27
Very Low	0.1	0.01	0.02	0.03	0.06	0.09
		0.05	0.15	0.3	0.55	0.9
		Very Low	Low	Medium	High	Very High

Negative Impact

Table 157. Negative Impact Probability Matrix

Probability

Very High	0.9	0.81	0.50	0.27	0.14	0.05
High	0.7	0.63	0.39	0.21	0.11	0.04
Medium	0.5	0.45	0.28	0.15	0.08	0.03

Low	0.3	0.27	0.17	0.09	0.05	0.02
Very Low	0.1	0.09	0.06	0.03	0.02	0.01
		0.9	0.55	0.3	0.15	0.05
		Very High	High	Medium	Low	Very Low
		Positive Impact				

Table 158. Positive Impact Probability Matrix

Positive impact risks are considered opportunities, that is, risks that will benefit us, instead of being a ‘risk’ to development.

Table 29 shows the impact assigned to each risk based in the previous two tables.

ID	Risk	Probability	Impact				Impact
			Budget	Planning	Scope	Quality	
1	Lack of experience	High	High	High	High	Critical	0.68
2	Over scoping	Low	Medium	Medium	Low	Very Low	0.06
3	Internship	Medium	Low	Medium	Very Low	Very Low	0.07
4	Nintendo	Very Low	High	Low	Critical	Low	0.04
5	Incorrect estimation	High	High	High	Low	Low	0.25
6	External libraries	High	Low	Low	Medium	High	0.20
7	Hard concepts	High	Medium	Low	Low	Medium	0.16

Table 159. Risk Impact

The following table contains the strategy and response given to each risk.

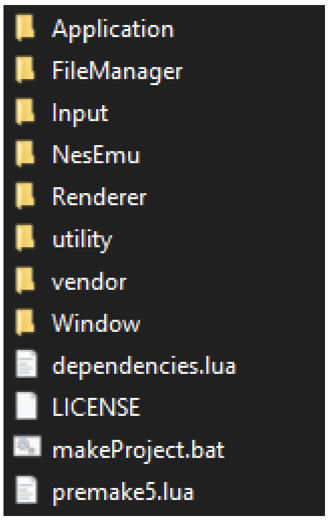
ID	Risk	Strategy	Response
1	Lack of experience	Mitigate	The only way to get experience it's through work, and planning, so the planning will be estimated taking the lack of experience into account
2	Over scoping	Avoid	To avoid over scoping, the author will study other emulators, and the original hardware.
3	Internship	Accept	The author needs an internship to get his degree, so the risk will be gladly accepted
4	Nintendo	Accept	If Nintendo decides that this project is worthy of being c&d'd, then there's nothing that I can do, hence, the risk will be accepted.
5	Incorrect estimation	Mitigate	To mitigate incorrect estimation, a study of similar projects led by developers of similar skill may be needed.

6	External libraries	Mitigate	The author will only use libraries that have been already tested, this will have led to a steep decline in bugs, but they can't be avoided completely
7	Hard concepts	Mitigate	The author will keep an eye in the forums of the NES dev wiki to cleanse any kind of hardware question that may arise

Table 160. Risk Response and Strategy

9.2. Project File Structure

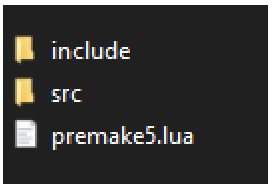
The structure of this project is very simple



```
Application
FileManager
Input
NesEmu
Renderer
utility
vendor
Window
dependencies.lua
LICENSE
makeProject.bat
premake5.lua
```

Figure 93. Project root

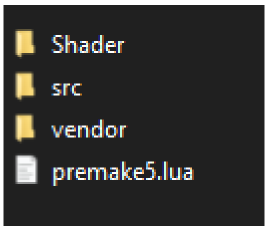
Each different project has its own branch, in this case the different branches are Application, FileManager, Input, NesEmu, Renderer and Window; the utility folder contains scripts to help compilation, and the vendor folder contains project wide dependencies like imgui or sdl2. Then each project can contain an additional vendor folder if they contain specific dependencies like in the case of the renderer with vkbootstrap.



```
include
src
premake5.lua
```

Figure 94. Project libraries file structure

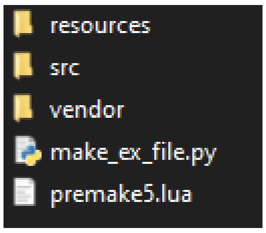
FileManager, Input and Window are meant to be used as libraries, so they contain an include folder and a src folder, the include folder then contains the name of the library as another layer in the folder structure so #includes have to contain the name of the library instead of just the file i.e #include "Input.hpp" would be "input/Input.hpp".



```
Shader
src
vendor
premake5.lua
```

Figure 95. Renderer file structure

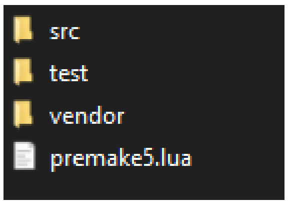
The renderer is an exception to the outlines defined above, it contains a Shader folder containing shader code, while the src folder contains the source code.



```
resources
src
vendor
make_ex_file.py
premake5.lua
```

Figure 96. Application file structure

The application project contains a resources folder that holds a collection of fonts, texts and the project icon.



```
src
test
vendor
premake5.lua
```

Figure 97. NesEmu file structure

NesEmu contains a test folder containing the project of the emulator tests

9.3. Licensing

All relevant licenses are stored next to the relevant licensed component, in the case of fonts, the license is stored in the font file and in the case of code libraries, in their respective LICENSE file.

The source code of this project is under the zLib licensing, the license is as follows:

```
Copyright (C) 2023-2025 Luis Vijande González

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:
```

9.4. Glossary

	Term	Description
C		
	CHR ROM	Character ROM, part of a ROM containing graphical data
	CRT	Cathode Ray Tube, a technology used by old televisions in which a tube sent rays to the screen in vertical lines called scanlines to display images.
D		
	DMA	Direct memory access, a period in which the PPU gets direct acces to CPU memory in order to copy sprite data
I		
	IRQ	Hardware interrupt, an interrupt raised by hardware, in the NES case, normally the APU or a mapper, in the context of this emulatur, this might not exist
M		
	MMIO	Memory mapped I/O, in the NES, these are PPU registers mapped in the memory range to be accessed by the CPU
N		
	NMI	Non-maskable interrupt, interrupt raised by the PPU at the end of a frame
	NTSC	National Television System committee, an analogic tv system using mainly in north America, pars of south America, and parts of Asia, notably Japan; different from the PAL system used in Europe, their main differences being framerate (60 vs 50), scanline count (525 vs 625) and resolution (720x480 vs 720 vs 576).
R		
	ROM	In the emulator world, a ROM is a dump of a games code in a computer understandable file format, NES ROMs also contain additional information like the required mapper and if they require persistent storage.
O		
	Opcode	An operation code refers to a portion of machine language, in 6502 they are 1-byte integers that refer to both the instruction and addressing mode of the instruction.
P		
	PRG ROM	Program ROM, part of a ROM containing code and data structures
T		
	Texel	Smallest unit of a texture, like a pixel on a screen

Table 161. Glossary

9.5. Bibliography

1. **Circuit, United States Court of Appeals for the Ninth.** Sony Computer Entm't, Inc. v. Connectix Corp., 203 F.3d 596 (9th Cir. 2000). [Online] 2000. https://www.copyright.gov/fair-use/summaries/sony-connectix-9thcir2000.pdf.

2. **NesHacker.** NES Graphics Explained. [Online] 06 2021. https://www.youtube.com/watch?v=7Co_8dC2zb8.

3. **Barr, David.** NES Emulator From Scratch. [Online] 07 2019. https://www.youtube.com/watch?v=nViZg02IMQo&list=PLrOv9FMX8xJHqMvSGB_9G9nZZ_4IgteYf.

4. **cmake.** Cmake. [Online] 2024. https://cmake.org/.

5. **Premake.** Premake. [Online] 2024. https://premake.github.io/.

6. **Google.** GoogleTest. [Online] 09 2024. https://github.com/google/googletest.

7. **Boost.** Boost. Test. [Online] 12 2020. https://www.boost.org/doc/libs/1_75_0/libs/test/doc/html/index.html.

8. **catchorg.** Catch2. [Online] 09 2024. https://github.com/catchorg/Catch2.

9. **SDL.** SDL. [Online] 09 2024. https://www.libsdl.org/.

10. **GLFW.** GLFW. [Online] 02 2024. https://www.glfw.org/.

11. **Gasimzada, Gamis.** Managing BindlesS Descriptors in Vulkan. [Online] 05 2023. https://dev.to/gasim/implementing-bindless-design-in-vulkan-34no.

12. **Hector, Tobias.** Dynamic Rendering. [Online] Khronos Group, 10 2021. https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VK_KHR_dynamic_rendering.html.

13. **Khronos Group.** Vulkan. [Online] 09 2024. https://www.vulkan.org/.

14. —. OpenGl. [Online] 06 2021. https://www.opengl.org/.

15. **Chuchem, Yair.** The revolution in UI paradigms . [Online] 29 09 2021. https://yairchu.github.io/posts/ui-paradigms.

16. **ocornut.** Dear Imgui. [Online] 09 2024. https://github.com/ocornut/imgui.

17. **Qt.** Qt. [Online] 2024. https://doc.qt.io/.

18. —. Licensing. [Online] 2024. https://www.qt.io/qt-licensing.

19. **Microsoft.** Getting started With XInput. [Online] 31 10 2023. https://learn.microsoft.com/es-es/windows/win32/xinput/getting-started-with-xinput.

20. **GlassDoor.** Sueldos Desarrollador en España. [Online] 09 2024. https://www.glassdoor.es/Sueldos/desarrollador-sueldo-SRCH_KO0,13.htm.

21. **Fernández, Jorge Morales.** ¿Cuánto cuesta el kilovatio hora de luz (kWh) en España? [Online] https://tarifaluzhora.es/info/precio-kwh.

22. **Repsol.** ¿Cuál es el consumo de un ordenador? . [Online] 2024. https://www.repsol.es/particulares/asesoramiento-consumo/cuanto-consume-ordenador/.

23. **Movistar.** Tarifas de fibra. [Online] 2024. https://www.movistar.es/fibra-optica/.

24. **Various.** Nes Dev Wiki. *Mappers.* [Online] https://www.nesdev.org/wiki/Mapper.

25. —. Nes Dev Wiki. *PPU Palettes.* [Online] https://www.nesdev.org/wiki/PPU_palettes.

26. **Yerrick, Damien.** pineight. *Thwaite.* [Online] 2011. https://pineight.com/mw/page/Thwaite.xhtml.

27. **DawnBringer.** PixelJoint. *DawnBringer's 16 Col Palette v1.0.* [Online] 19 8 2011. https://pixeljoint.com/forum/forum_posts.asp?TID=12795.

28. **Various.** Nes Dev Wiki. *Emulator Tests.* [Online] https://www.nesdev.org/wiki/Emulator_tests.

29. **Unknown.** [Online] https://www.nesdev.org/wiki/PPU_registers#Master.2Fslave_mode_and_the_EXT_pins.

30. **Various.** Nes Dev Wiki. *PPU Pattern tables.* [Online] https://www.nesdev.org/wiki/PPU_pattern_tables.

31. —. Nes Dev Wiki. *PPU Palettes.* [Online] https://www.nesdev.org/wiki/PPU_palettes#Palette_RAM.

32. —. Nes Dev Wiki. *PPU nametables.* [Online] https://www.nesdev.org/wiki/PPU_nametables.

33. —. Nes Dev Wiki. *PPU Attribute tables.* [Online] https://www.nesdev.org/wiki/PPU_attribute_tables.

34. —. Nes Dev Wiki. *PPU OAM.* [Online] https://www.nesdev.org/wiki/PPU_OAM.

35. —. Nes Dev Wiki. *DMA.* [Online] https://www.nesdev.org/wiki/DMA.

36. —. Nes Dev Wiki. [Online] https://www.nesdev.org/wiki/PPU_rendering.

37. **Villa, Alberto.** Mapping IO Registers With Union And Structs. [Online] 7 1 2018. https://docbrown85.github.io/mapping-io-registers-with-c-unions-and-structs/.

38. **Unknown.** learn OpenGL. *Coordinate Systems.* [Online] https://learnopengl.com/Getting-started/Coordinate-Systems.

39. **Google.** Google C++ Style Guide. [Online] https://google.github.io/styleguide/cppguide.html.