

KIKENDO ENTERTAINMENT SYSTEM

CREATING A NES EMULATOR*

GAMES INCLUDED!



SUPER
MARIO BROS.

MARIO BROS.

POPEYE

DONKEY
KONG

Galaga

KiKendo®

Kike Alcor @SuperKeeks // Apr 2018

*: Audio sold separately

Motivation

- **Curious about how old console games were made**
- **Emulators felt like black magic**
- **Refresh my HW knowledge (and C++!)**
- **Learn Unit Testing**

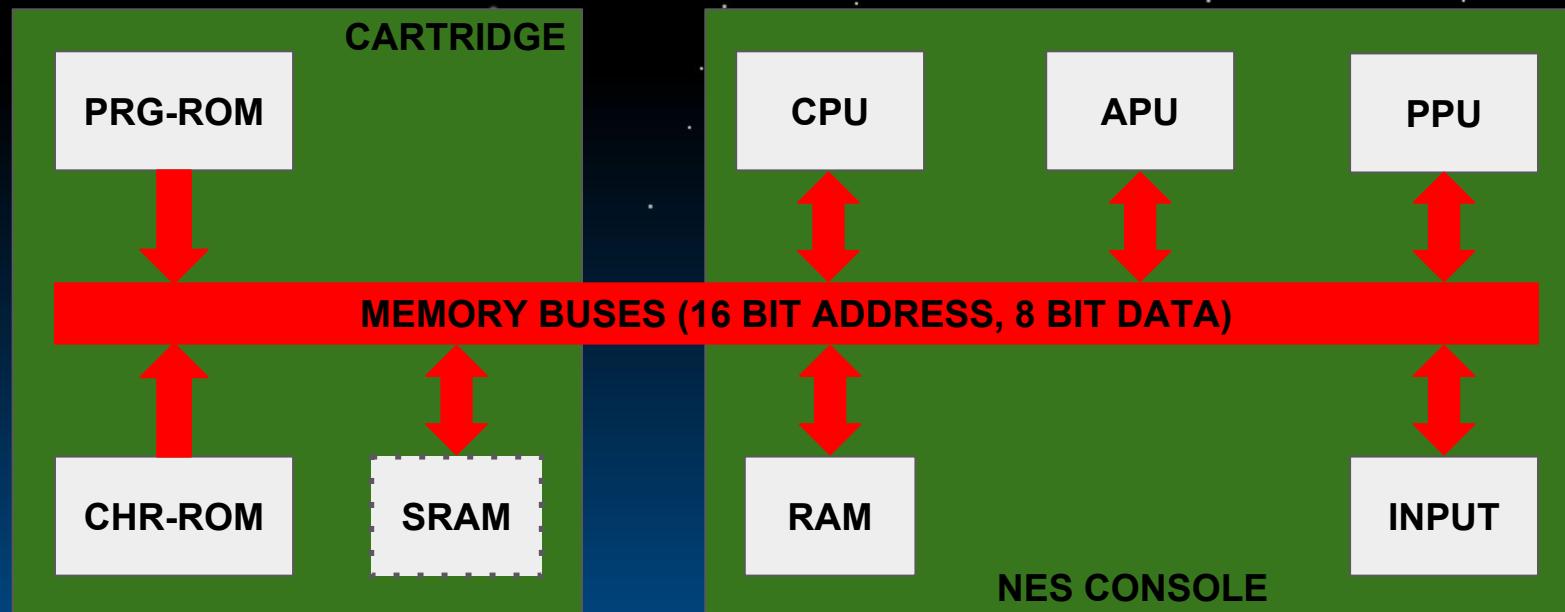
Technology & Tools

- **C++ 11**
- **SDL 2 (Video, Input and Audio)**
- **Visual Studio Community 2017**
- **Git (Bitbucket + Sourcetree)**
- **Trello**

NES Specs

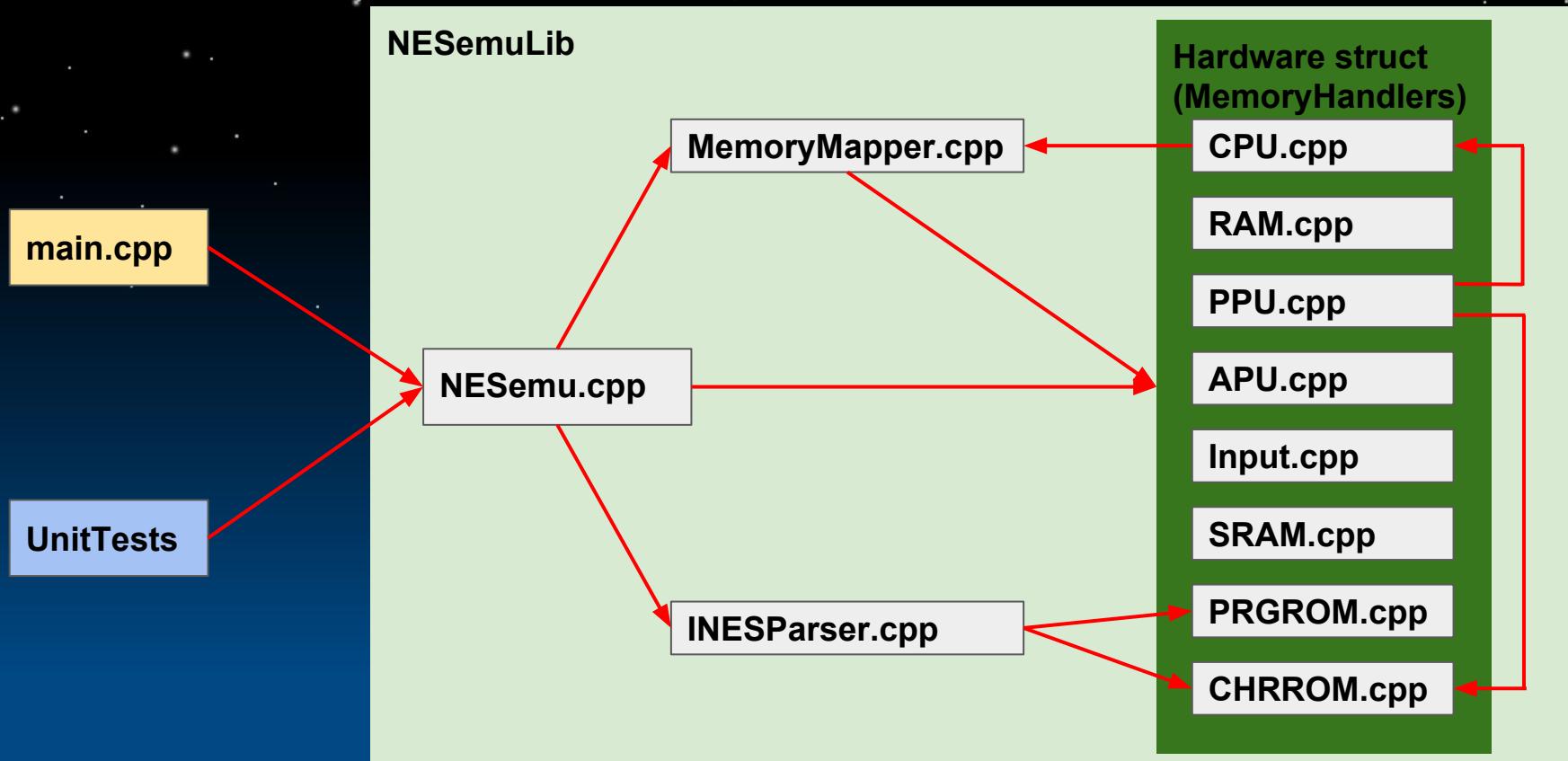
- **8 bit (8 bit registers / data, 16 bit address space)**
- **CPU: MOS 6502 @ 1.789 MHz (NTSC)**
 - + pAPU (Pseudo-Audio Processing Unit)
- **PPU (GPU): RICOH 2C02 @ 5.369 MHz (NTSC)**
- **RAM: 2 KB**
- **VRAM: 2 KB**
- **Palette: 54 Colours**
- **Resolution: 256x240 pixels**

NES Architecture



Optional

Emulator architecture



main.cpp

```
InitVideo();
InitInput();
emu.Load("rom_name.nes");
while (!quit)
{
    UpdateInput();
    emu.Update(deltaTime);
    PresentFramebuffer();
}
CleanUp();
```

NOTE:

All the SDL code in the project is here (i.e. the emulator itself has no 3rd party code dependencies)

.nes (iNES) parser

Header (16 bytes)
Trainer (0 or 512 bytes)
PRG-ROM data (16384*x bytes)
CHR-ROM data (8192*x bytes)
PlayChoice data

0	'N'
1	'E'
2	'S'
3	EOF
4	PRG-ROM page count (16 KB)
5	CHR-ROM page count (8 KB)
6	Flags (Mirroring, SRAM...)
7	Flags
8	PRG-RAM page count (8 KB)
9	Flags
10	Flags (unofficial)
11	0
12	0
13	0
14	0
15	0

.nes (iNES) parser



Memory Mapping

ADDRESS RANGE	MEMORY HANDLER
0x0000 - 0x1FFF	RAM
0x2000 - 0x3FFF, 0x4014	PPU
0x4000 - 0x4013, 0x4015, 0x4017 (W)	APU
0x4016, 0x4017 (R)	Input
0x6000 - 0x7FFF	SRAM
0x8000 - 0xFFFF	PRG-ROM

```
uint8_t ReadMem(uint16_t address)
```

```
void WriteMem(uint16_t address, uint8_t value)
```

RAM, SRAM, PRG-ROM, CHR-ROM

```
uint8_t _ram[2048];
```

```
uint8_t _sram[8192];
```

```
uint8_t _prgRom[16384 * 2];
```

```
uint8_t _chrRom[8192 * 4];
```

Input

Reads of 0x4016 (P1) and 0x4017 (P2) return state of buttons in this order:

A, B, Select, Start, Up, Down, Left, Right

Possible values are:

- **0x40 (Button Released)**
- **0x41 (Button Pressed)**



Input

To read `input` again, write 0, then 1 to `0x4016`.

NOTE: These writes are the reason why `0x4X` values are returned when reading `input` (high bits are retained in the bus)

CPU: Registers

- **Accumulator (A)**
- **X**
- **Y**
- **Status/Flags**
 - **Carry, Zero, InterruptDisable, Decimal (N/A), Break, Unused, Overflow, Sign**
- **Program Counter (16 bit)**
- **Stack Pointer**

CPU: Assembly intro/review

_ram[0x1234] = 2 + 11;

```
CLC      // Clear carry Flag
LDA #$02 // Loads 2 into accum.
ADC #$110x0B // Adds 11 to accum.
STA $1234 // Stores result into 0x1234
```

CPU: Addressing modes

Immediate

LDA #\$44 _a = 0x44;

Zero Page

LDA \$44 _a = Read(0x44);

Zero Page X & Zero Page Y

LDA \$44,X _a = Read(0x44 + _x);

Absolute

LDA \$1234 _a = Read(0x1234);

Absolute X & Absolute Y

LDA \$1234,X _a = Read(Read(0x1234) + _x);

CPU: Addressing modes

Indirect X:

```
LDA ($44,X)    int temp = Read(0x44) + _x;
                int final = Read(temp) + Read(temp + 1) << 8;
                _a = ReadMem(final);
```

Indirect Y:

```
LDA ($44),Y    int addr = Read(0x44) + Read(0x44 + 1) << 8 + _y;
                _a = Read(addr);
```

CPU: Assembly intro/review



CPU: CLC

```
■ int CPU::CLC(AddressingMode mode)
  {
    SetFlag(Flag::Carry, false);
    return 2;
  }
```

CPU: LDA

```
int CPU::LDA(AddressingMode mode)
{
    int cycles = 1;
    const uint8_t value = GetValueWithMode(mode, cycles);
    SetAccumulator(value);

    return cycles;
}
```

CPU:ADC

```
int CPU::ADC(AddressingMode mode)
{
    int cycles = 1;
    const uint8_t value = GetValueWithMode(mode, cycles);
    const bool isValueNegative = IsValueNegative(value);
    const bool wasAccumulatorNegative = IsValueNegative(_accumulator);
    uint16_t result16 = _accumulator + value + (GetFlag(Flag::Carry) ? 1 : 0);
    SetAccumulator((uint8_t)result16);
    const bool isResultNegative = IsValueNegative(_accumulator);

    SetFlag(Flag::Overflow, (isValueNegative == wasAccumulatorNegative) &&
        (isValueNegative != isResultNegative));
    SetFlag(Flag::Carry, result16 != _accumulator);

    return cycles;
}
```

CPU: STA

```
int CPU::STA(AddressingMode mode)
{
    int cycles = 1;
    SetValueWithMode(mode, _accumulator, cycles);

    return cycles;
}
```

CPU: Opcodes

STA

SBC

CPY

RTS

TAX

CPX

56 INSTRUCTIONS
151 OPCODES

TYA

BPL

PLP

STY

BEQ

CLV

BIT

BCS

ROL

ROR

STX

PLA

LDX

RTI

CPU: Timing

```
void CPU::Tick()
{
    --_ticksUntilNextInstruction;
    if (_ticksUntilNextInstruction <= 0)
    {
        _ticksUntilNextInstruction = ExecuteNextInstruction() - 1;
    }
}
```

CPU: ExecuteNextInstruction()

```
int CPU::ExecuteNextInstruction()
{
    // Fetch next opcode
    const uint8_t opcode = _memoryMapper->ReadMem(++_programCounter);

    switch (opcode)
    {
        // LDA
        case 0xA9: PrintOpcodeInfo(opcode, "LDA", AddressingMode::Immediate); return LDA(AddressingMode::Immediate);
        case 0xA5: PrintOpcodeInfo(opcode, "LDA", AddressingMode::ZeroPage); return LDA(AddressingMode::ZeroPage);
        case 0xB5: PrintOpcodeInfo(opcode, "LDA", AddressingMode::ZeroPageX); return LDA(AddressingMode::ZeroPageX);
        case 0xAD: PrintOpcodeInfo(opcode, "LDA", AddressingMode::Absolute); return LDA(AddressingMode::Absolute);
        case 0xBD: PrintOpcodeInfo(opcode, "LDA", AddressingMode::AbsoluteX); return LDA(AddressingMode::AbsoluteX);
        case 0xB9: PrintOpcodeInfo(opcode, "LDA", AddressingMode::AbsoluteY); return LDA(AddressingMode::AbsoluteY);
        case 0xA1: PrintOpcodeInfo(opcode, "LDA", AddressingMode::IndirectX); return LDA(AddressingMode::IndirectX);
        case 0xB1: PrintOpcodeInfo(opcode, "LDA", AddressingMode::IndirectY); return LDA(AddressingMode::IndirectY);

        // LDX
        case 0xA2: PrintOpcodeInfo(opcode, "LDX", AddressingMode::Immediate); return LDX(AddressingMode::Immediate);
        case 0xA6: PrintOpcodeInfo(opcode, "LDX", AddressingMode::ZeroPage); return LDX(AddressingMode::ZeroPage);
    }
}
```

CPU: Unit Tests

```
TEST_METHOD(DEXINX)
{
    NESemu emu;
    CPU& cpu = *emu.GetCPU();
    uint8_t rom[PRGROM::kMaxPRGROMSize];
    rom[CPU::kResetVectorAddressL - PRGROM::kStartAddress] = 0;
    rom[CPU::kResetVectorAddressH - PRGROM::kStartAddress] = PRGROM::kStartAddress >> 8;
    int cycles;
    int codeIndex = 0;

    rom[codeIndex++] = 0xA2; // LDX Immediate
    rom[codeIndex++] = 0x00; // Value to load
    rom[codeIndex++] = 0xE8; // INX
    rom[codeIndex++] = 0xCA; // DEX
    rom[codeIndex++] = 0xCA; // DEX
    rom[codeIndex++] = 0xE8; // INX

    emu.Load(rom, PRGROM::kMaxPRGROMSize);

    cpu.ExecuteNextInstruction();
    Assert::AreEqual((uint8_t)0x00, cpu.GetX());
    cycles = cpu.ExecuteNextInstruction();
    Assert::AreEqual((uint8_t)0x01, cpu.GetX());
    Assert::AreEqual(false, cpu.GetFlag(CPU::Flag::Sign));
    Assert::AreEqual(false, cpu.GetFlag(CPU::Flag::Zero));
    Assert::AreEqual(2, cycles);
    cycles = cpu.ExecuteNextInstruction();
    Assert::AreEqual((uint8_t)0x00, cpu.GetX());
    Assert::AreEqual(false, cpu.GetFlag(CPU::Flag::Sign));
    Assert::AreEqual(true, cpu.GetFlag(CPU::Flag::Zero));
    Assert::AreEqual(2, cycles);
    cycles = cpu.ExecuteNextInstruction();
    Assert::AreEqual((uint8_t)0xFF, cpu.GetX());
    Assert::AreEqual(true, cpu.GetFlag(CPU::Flag::Sign));
    Assert::AreEqual(false, cpu.GetFlag(CPU::Flag::Zero));
    Assert::AreEqual(2, cycles);
    cycles = cpu.ExecuteNextInstruction();
    Assert::AreEqual((uint8_t)0x00, cpu.GetX());
    Assert::AreEqual(false, cpu.GetFlag(CPU::Flag::Sign));
    Assert::AreEqual(true, cpu.GetFlag(CPU::Flag::Zero));
    Assert::AreEqual(2, cycles);
```

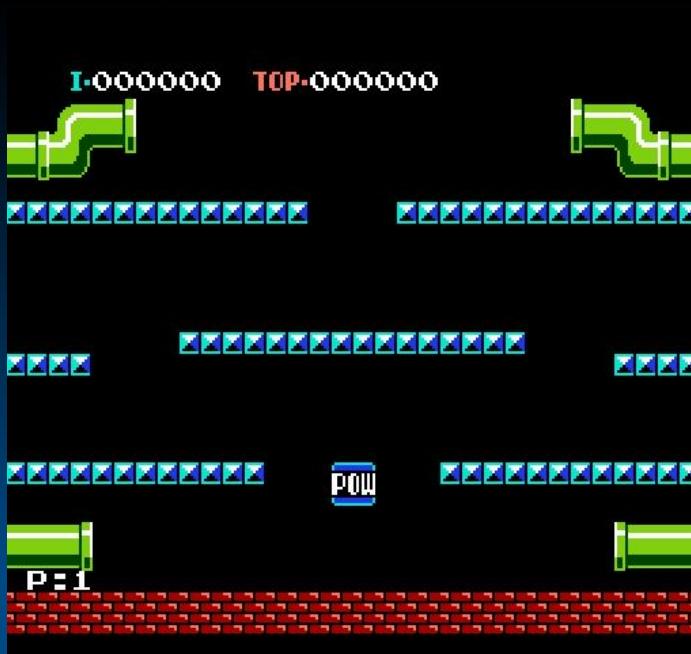
PPU: Intro

2 types of renderable objects:

- **Background: Mostly static objects, scrollable. Stored in nametables.**
- **Sprites: Moving objects (player, enemies, projectiles...), but unaffected by scroll. Stored in OAM. Can be above or behind the background.**

PPU: Background vs Sprites

Background



Sprites



PPU: Registers & Data

- **Control (W): Various PPU config values: interrupt enable, sprite height (8/16), bkg/sprite pattern table...**
- **Mask (W): More config values affecting colour and visibility of sprites and background.**
- **Status (R): Various flags indicating current PPU status: VBlank, sprite 0 Hit, sprite overflow...**

PPU: Registers & Data

- **PPU Address (W): For setting VRAM address.**
- **PPU Data (R/W): VRAM data read/write.**
- **OAM Address (W): Used for initialising OAM address.**
- **OAM Data (R/W): Sprite data read/write. Each write increments the address.**
- **Scroll (W): 2 writes: 1 for X, 1 for Y.**
- **OAM DMA (W): Fast RAM->OAM transfer.**

PPU: Palettes

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

PPU: Palettes

The output colours aren't accessed directly but by palettes:

- **Universal background colour**
- **4 Background palettes (3 colours each)**
- **4 Sprite palettes (3 colours each)**

Address	Purpose
\$3F00	Universal background color
\$3F01-\$3F03	Background palette 0
\$3F05-\$3F07	Background palette 1
\$3F09-\$3F0B	Background palette 2
\$3F0D-\$3F0F	Background palette 3
\$3F11-\$3F13	Sprite palette 0
\$3F15-\$3F17	Sprite palette 1
\$3F19-\$3F1B	Sprite palette 2
\$3F1D-\$3F1F	Sprite palette 3

PPU: Palettes



	Mario	Luigi
Color 1		
Color 2		
Color 3		

PPU: How are backgrounds drawn?

Imagine we want to render this small section of the background:



M A R I O

PPU: How are backgrounds drawn?

**To save memory and computing power,
backgrounds are drawn using 3
elements:**

- Pattern tables**
- Name tables**
- Attribute tables**

PPU: Pattern tables



0	A	9	J	18	S
1	B	10	K	19	T
2	C	11	L	20	U
3	D	12	M	21	V
4	E	13	N	22	W
5	F	14	O	23	X
6	G	15	P	24	Y
7	H	16	Q	25	Z
8	I	17	R		

Tiles don't specify final colour for each of their pixels, but (1 to 3) index within a palette or transparent (0)

PPU: Pattern tables

0	0	1	1	1	0	0	0
0	1	1	0	1	1	0	0
1	1	0	0	0	1	1	0
1	1	0	0	0	1	1	0
1	1	1	1	1	1	1	0
1	1	0	0	0	1	1	0
1	1	0	0	0	1	1	0
1	1	0	0	0	1	1	0

In our letters example, all letters are only using 2 values (1 for opaque, 0 for transparent)

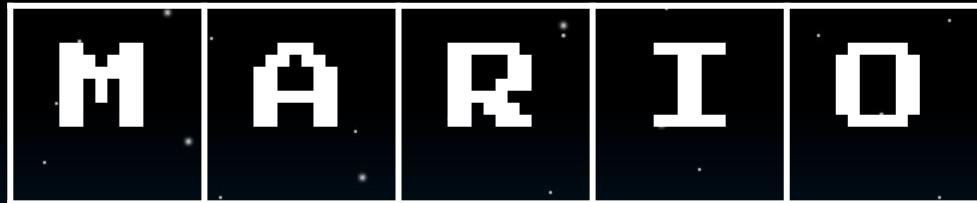
PPU: Nametables

Nametable: 32x30 array containing indices of the tiles that make a full screen (256x240 pixels = 32x30 tiles)

PPU: Nametables

Pattern table

0	A	9	J	18	S
1	B	10	K	19	T
2	C	11	L	20	U
3	D	12	M	21	V
4	E	13	N	22	W
5	F	14	O	23	X
6	G	15	P	24	Y
7	H	16	Q	25	Z
8	I	17	R		



Nametable

12	0	17	8	14
----	---	----	---	----

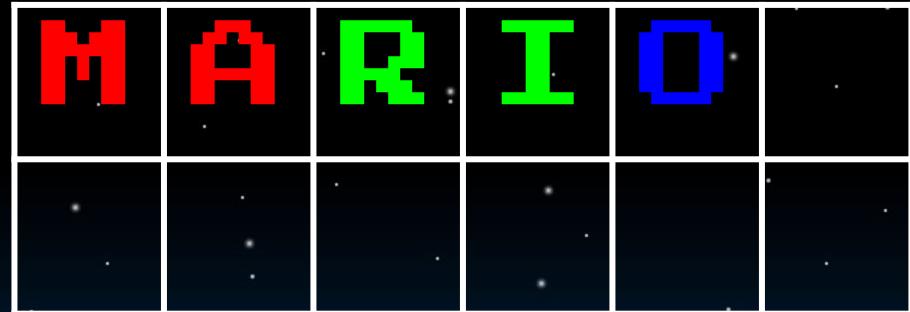
PPU: Attribute tables

Attribute table: Array that specifies what palette to use for each 2x2 tiles (16x16 pixels) section of a full screen.

PPU: Attribute tables

Background palettes

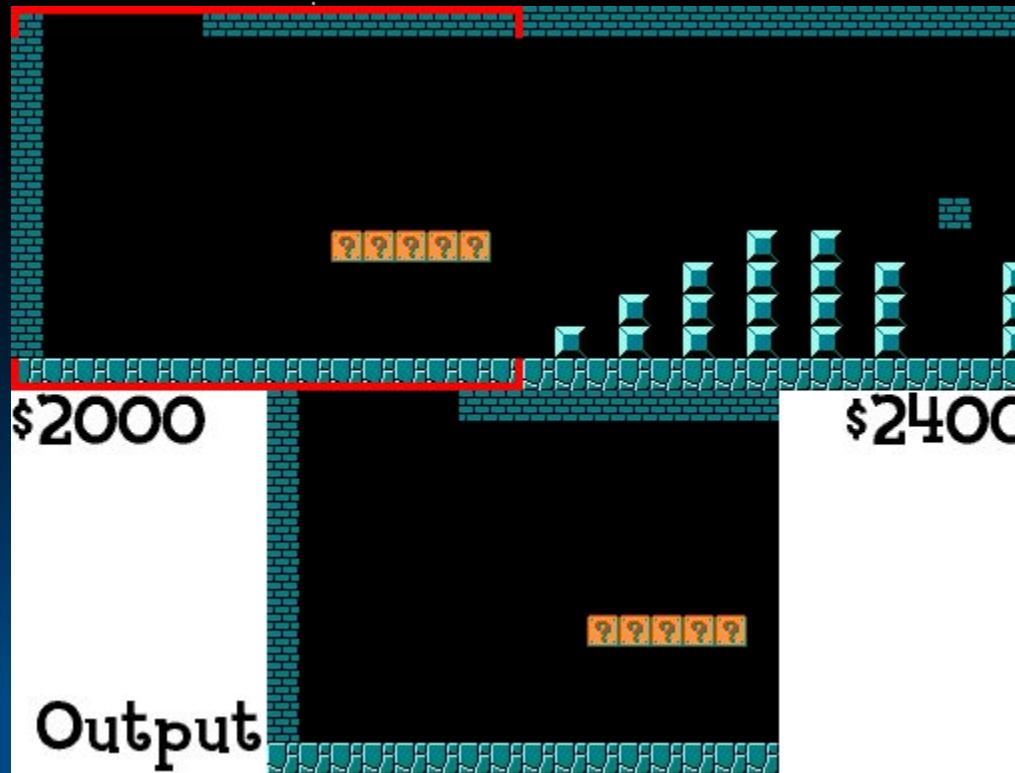
0	Red	Magenta	Magenta	Magenta
1	Green	Magenta	Magenta	Magenta
2	Blue	Magenta	Magenta	Magenta
3	Magenta	Magenta	Magenta	Magenta



Attribute table

0	1	2
---	---	---

PPU: Scrolling



PPU: *Sprites*

- **Up to 64 sprites (OAM) made of 4 bytes:**
 - 0: Y position
 - 1: Pattern index
 - 2: Attributes (2 palette bits, priority vs background, horizontal flip, vertical flip)
 - 3: X position
- **Up to 8 sprites per scanline (Secondary OAM)**

TODO

- **Implement more memory mappers to support more games (i.e. SMB 3)**
- **APU**



Lessons learned + advice

- **CPU Unit Testing has been invaluable, almost a must.**
- **Asserts have also been super useful for checking my assumptions were right.**
- **Most important part to get right is probably memory mapping. All communication between components depends on it.**

Lessons learned + advice

- **Bugs require A LOT of patience given you didn't code the game.**
- **Start with simpler/older games like Mario Bros. Don't start with *Super* Mario Bros, as it's more challenging.**
- **Writing a NES emulator requires discipline and patience, but it's not difficult.**

Resources

Best NES wiki and forums:

nesdev.com

6502 opcode info:

6502.org/tutorials/6502opcodes.html

My NES emu repo:

bitbucket.org/gamezer0/nesemu

Project timeline

 **Kike Alcor**
@SuperKeeks

Today I officially started my new pet project. A NES emulator (my first ever emulator) using SDL2 (also first time I use it)

SET UP GIT REPO AND VS PROJECT FOR MY NEW PET PROJECT



SO I GUESS YOU CAN SAY THINGS ARE GETTING PRETTY SERIOUS

11:09 PM - 9 Oct 2017

12 Likes 

Project timeline



Kike Alcor @SuperKeeks · 3 Dec 2017

:D

 C:\Users\Kike\Documents\NESemu\Debug\NESemu.exe

CPU: 151 of 151 opcodes implemented



1



8



Kike Alcor @SuperKeeks · 25 Nov 2017

Half way there!!

 C:\Users\Kike\Documents\NESemu\Debug\NESemu.exe

CPU: 76 of 151 opcodes implemented



1



Project timeline

 **Kike Alcor**
@SuperKeeks

Let the fun begin!

```
void PPU::RenderScanline(int index)
{
```

2:16 PM - 31 Dec 2017

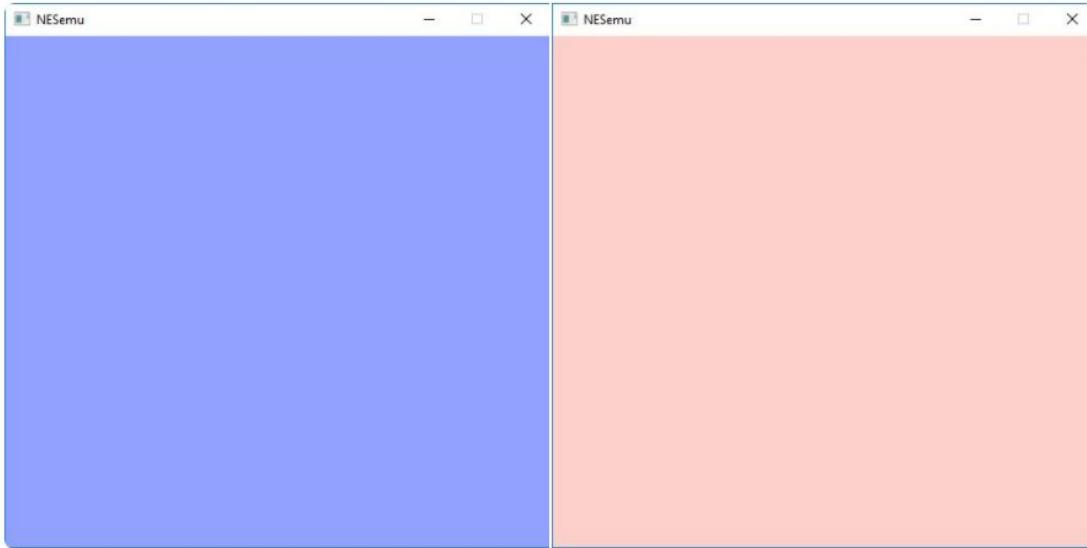
7 Likes 

Project timeline

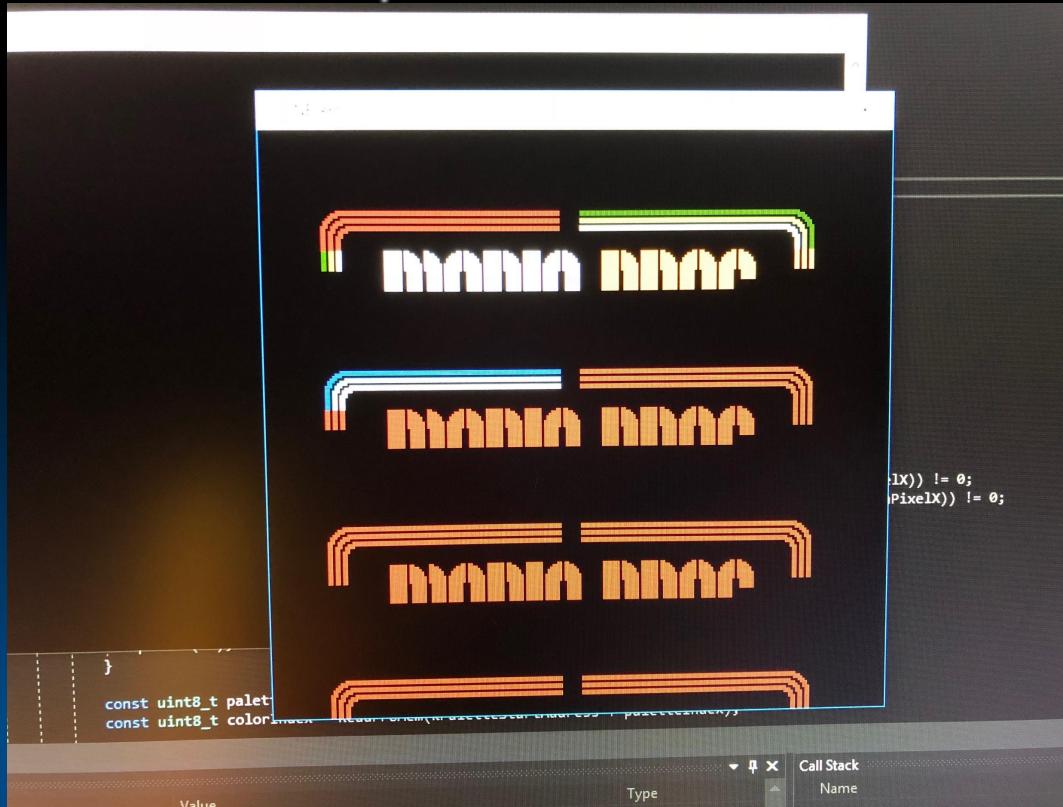


Kike Alcor @SuperKeeks · 31 Dec 2017

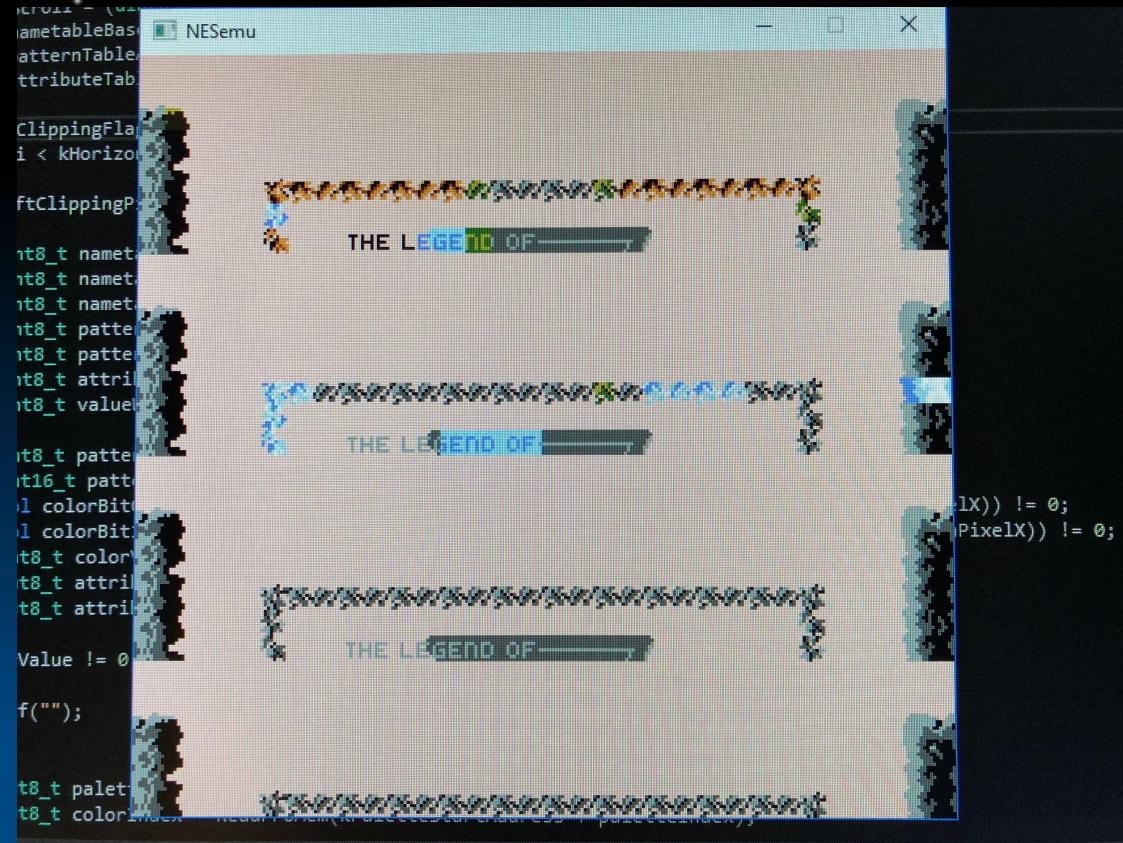
Well, this looks promising. Background colour seems correct so far. Here are Super Mario Bros and Zelda :D



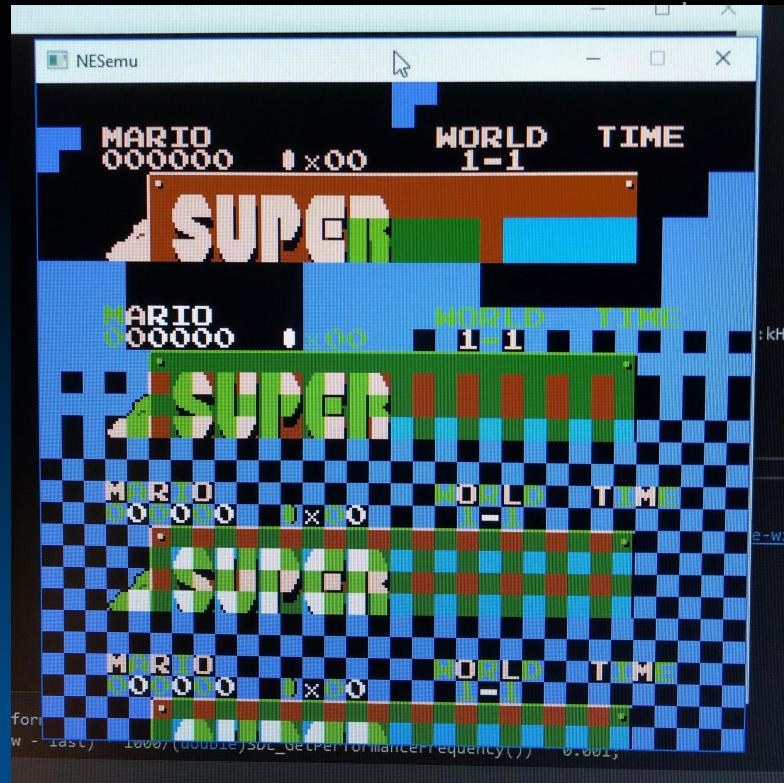
Project timeline



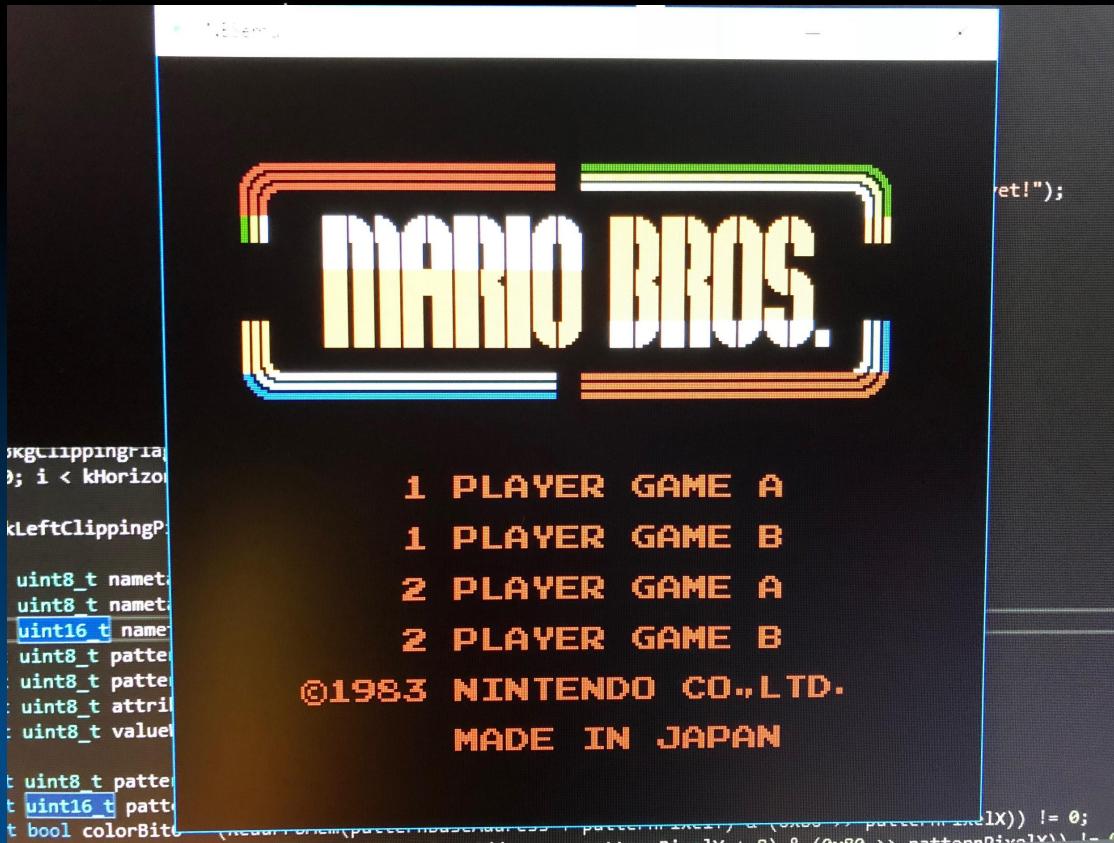
Project timeline



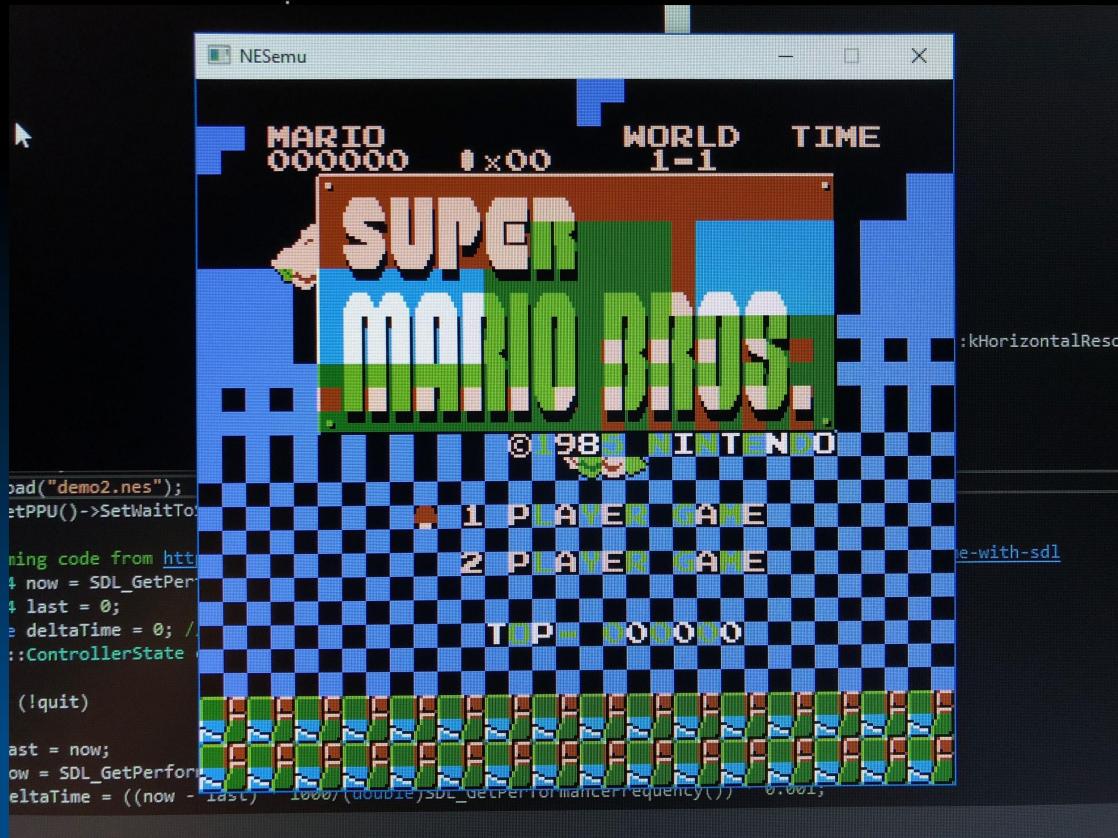
Project timeline



Project timeline



Project timeline



Project timeline

Kike Alcor @SuperKeeks · Jan 2
Background rendering is ALIVE!!



1 PLAYER GAME A
1 PLAYER GAME B
2 PLAYER GAME A
2 PLAYER GAME B
©1983 NINTENDO CO.,LTD.
MADE IN JAPAN

2 7

Show this thread

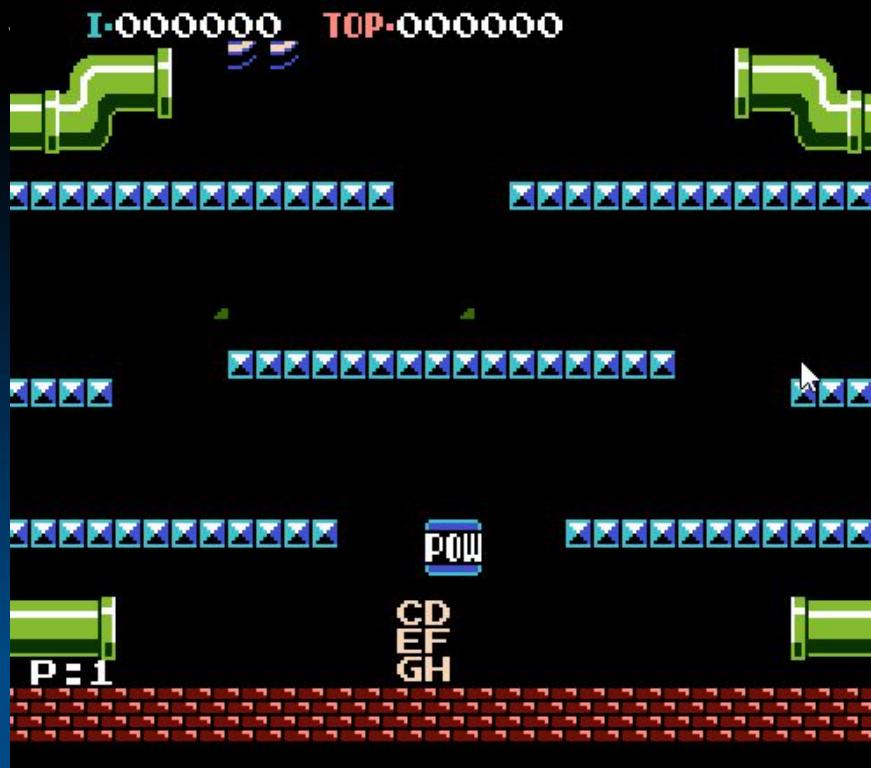
Project timeline



Project timeline



Project timeline



Project timeline



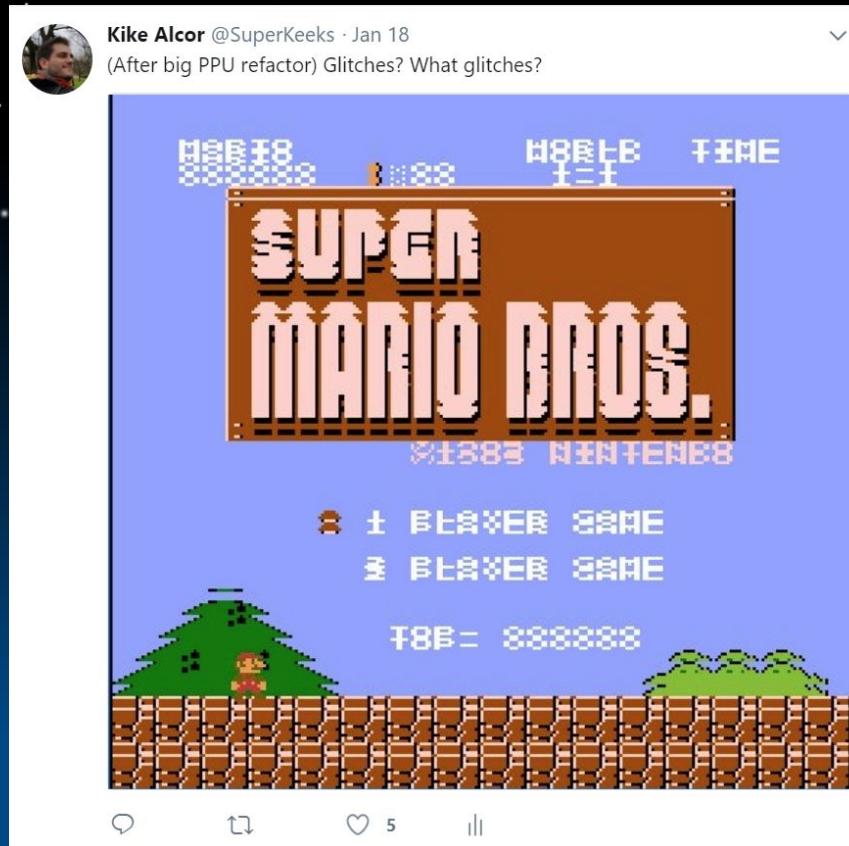
Project timeline



Project timeline



Project timeline



Project timeline



THANKS!



@SuperKeeks



ealcor@gmail.com