KIKENDO ENTERTAINMENT SYSTEM

CREATING A NES EMULATOR*

GAMES INCLUDED!

SUPER MARIO BROS.

DONKEY KONG

MARIO BROS.

Galaga

POPEYE

KiKendo®

Kike Alcor @SuperKeeks // Feb 2018

*: Audio sold separately

Motivation

- Curious about how old console games were made
- Emulators felt like black magic
- Refresh my HW knowledge (and C++!)
- Why the NES?

Objectives

- Get Super Mario Bros. running
  - Side quest: Implement Audio
- Try SDL 2
- Get used to Unit Testing
- Meet deadlines

The process

1. Read some NES documentation
   - nestech.txt
   - 6502 Programming guide
2. Choose technology
3. Start coding

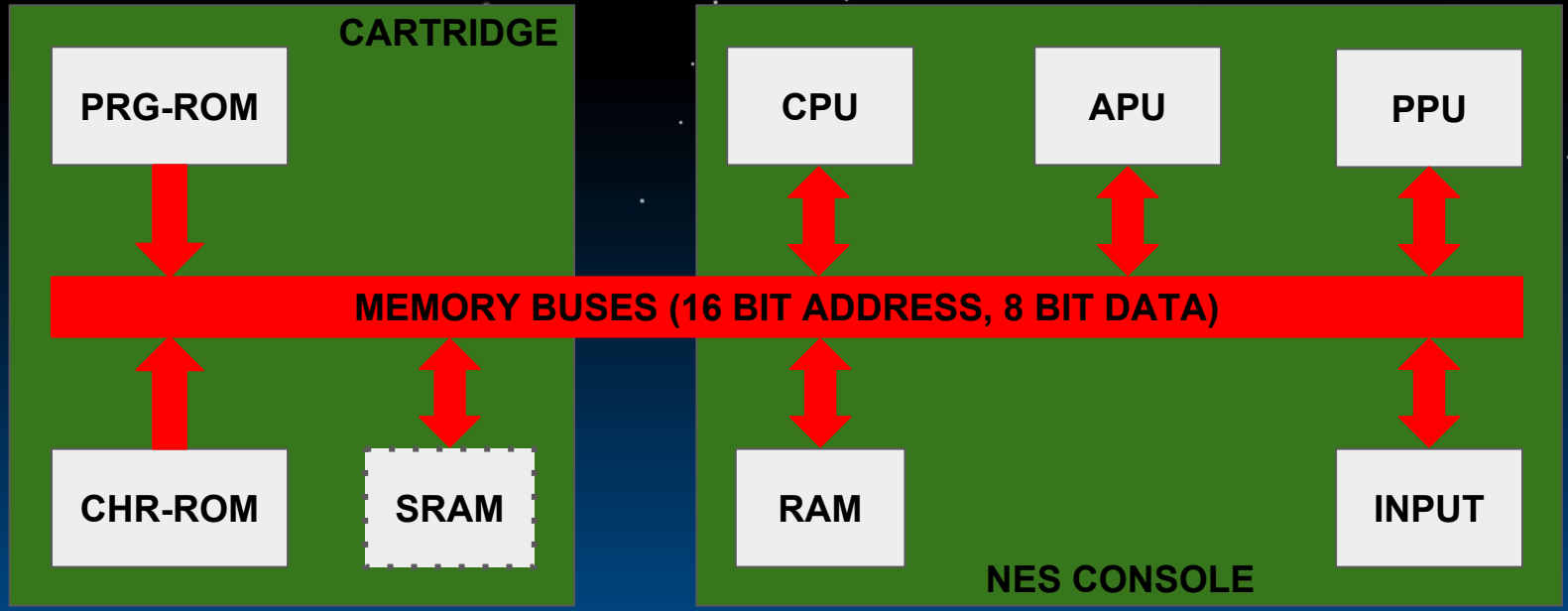
Technology & Tools

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

NES Specs

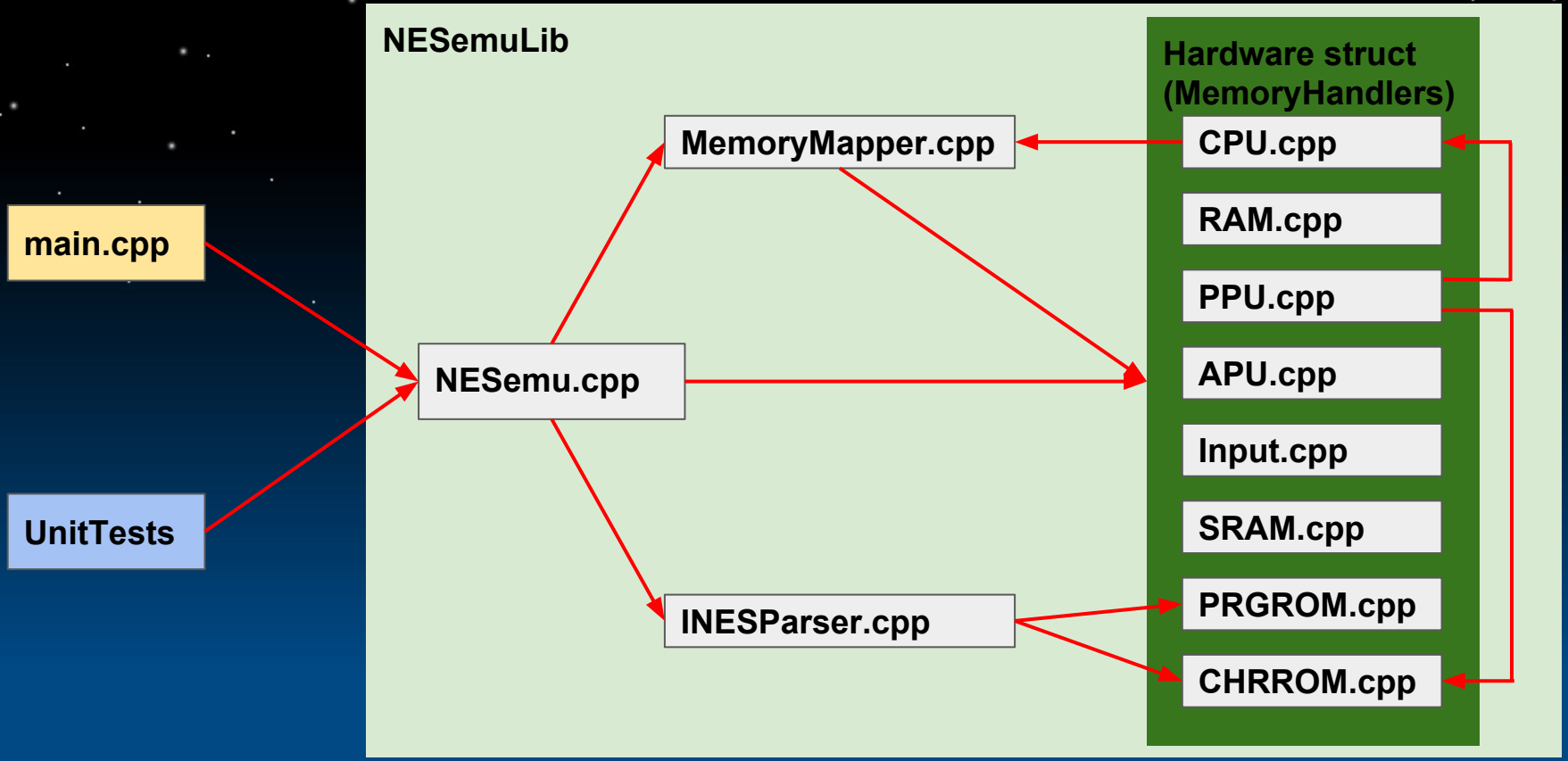
- 8 bit (8 bit registers / data, 16 bit address space)
- CPU: MOS 6502 @ 1.789 MHz (NTSC)
  - + pAPU (Pseudo-Audio Processing Unit)
- PPU (GPU): RICOH 2C02 @ 5.369 MHz (NTSC)
- RAM: 2 KB
- VRAM: 2 KB
- Palette: 54 Colours
- Resolution: 256x240 pixels

NES Architecture



Optional

Emulator architecture



main.cpp

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

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

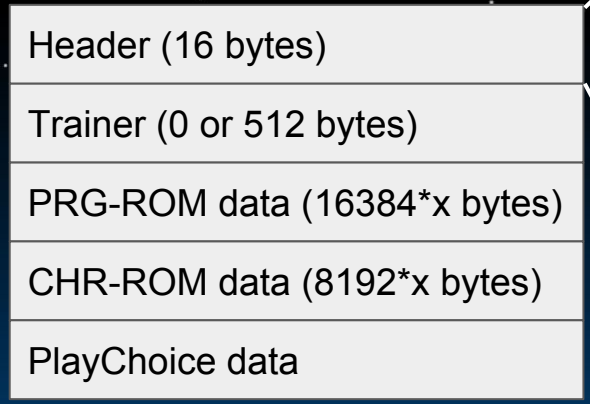
NESemu::Update(double delta)

```
double masterDelta = kMasterClockSpeed * delta;
int ppuDelta = round(masterDelta / 4);

for (int i = 0; i < ppuDelta; ++i)
{
    if (i % 3 == 0)
    {
        cpu.Tick();
    }

    ppu.Tick();
}
```

kMasterClockSpeed = 21.477272 MHz = 21477272 Hz

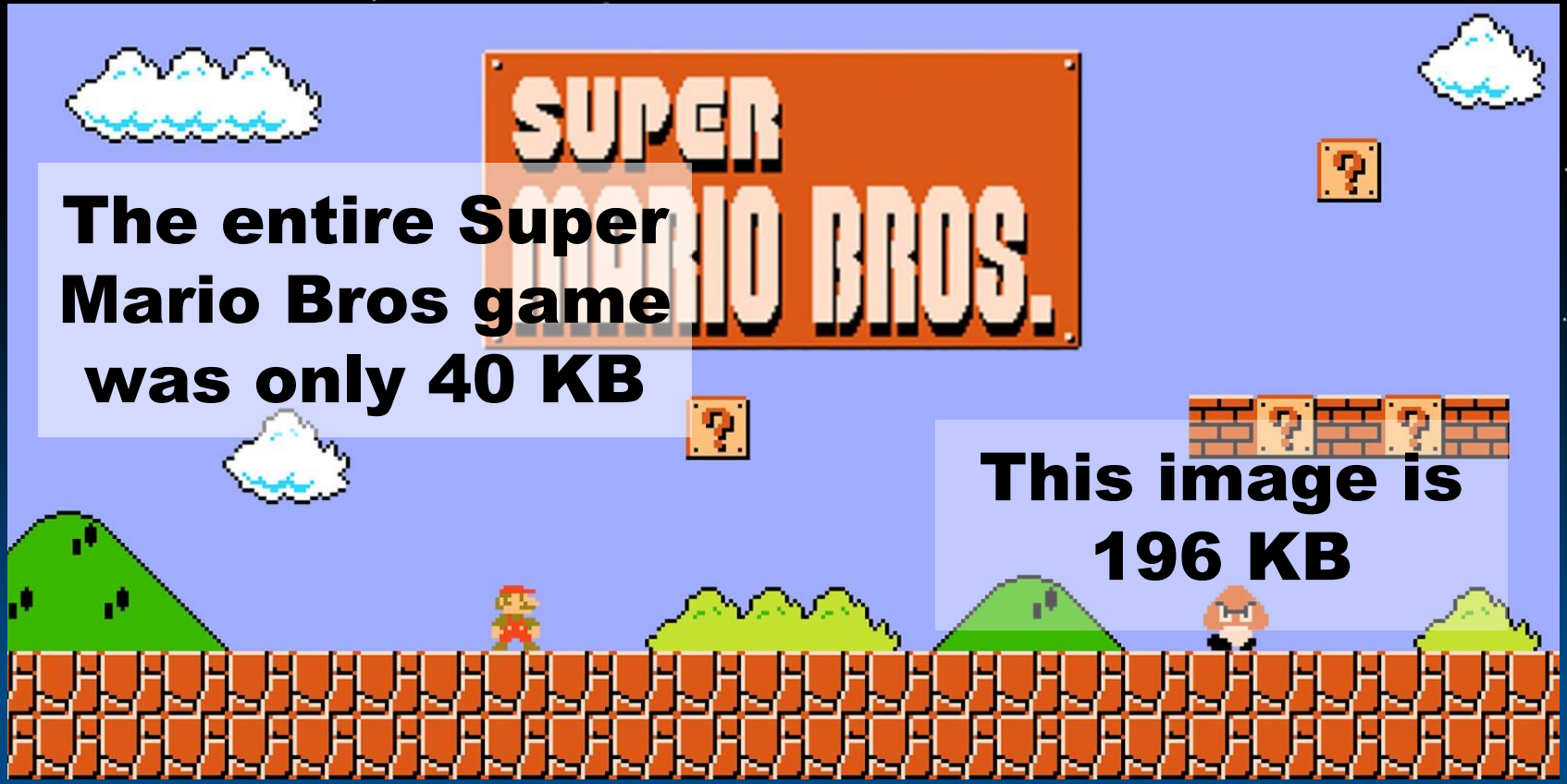
PPU Clock: kMaster / 4
CPU Clock: kMaster / 12

.nes (iNES) parser



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



CPU Memory map

0x0000 - 0x1FFF	RAM (2 KB) + Mirrors
0x2000 - 0x3FFF, 0x4014	PPU (8 registers + Mirrors)
0x4000 - 0x4013, 0x4015, 0x4017 (W)	APU
0x4016, 0x4017 (R)	Input
0x6000 - 0x7FFF	SRAM
0x8000 - 0xFFFF	PRG-ROM

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 #$11      // Adds 11 to accum.
STA $1234     // Stores result into 0x1234
```

0x0B

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

BPL PLP BIT BCS

TYA BEQ CLV

ROL ROR LDX

STY STX PLA RTI

CPU: Timing

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

CPU: ExecuteNextInstruction()

```cpp
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);
}
```

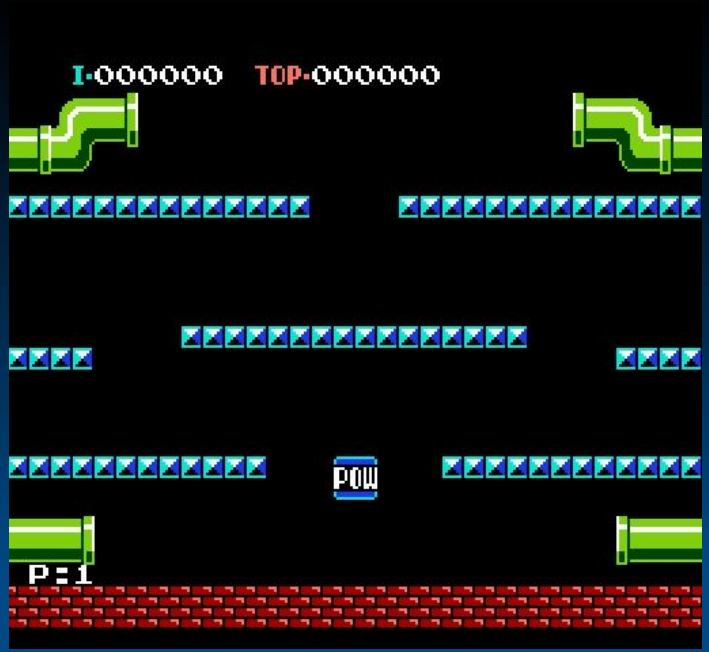
PPU: Intro

2 types of renderable objects:

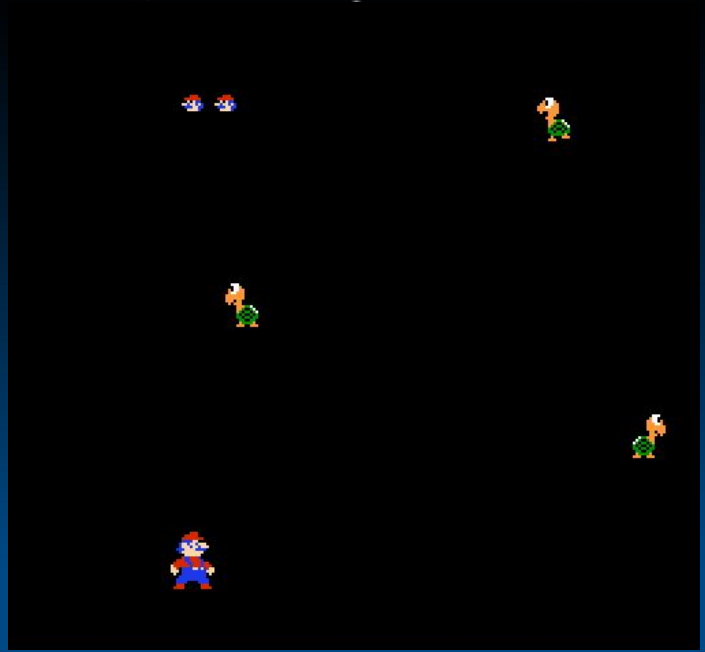
- Background: Mostly static objects, scrollable. Stored in nametables.
- Sprites: Moving objects (player, enemies, projectiles...), but unaffected by scroll. Stored in OAM. Can be above or behind the background.

PPU: Background vs Sprites

Background



Sprites



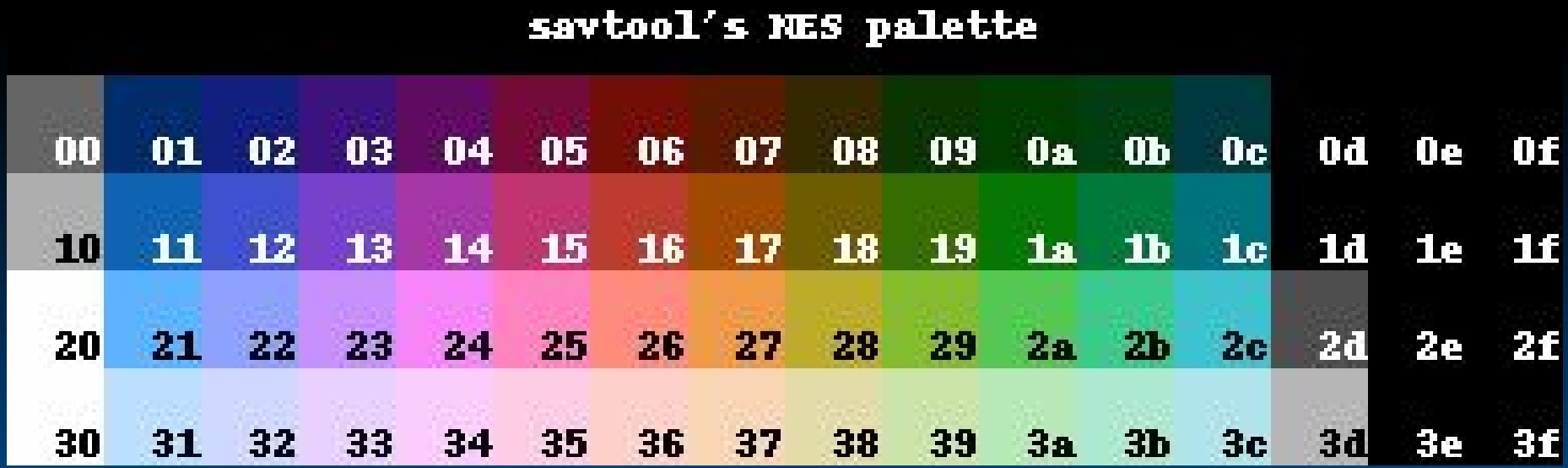
PPU: Registers & Data

- **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



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: Nametables

Nametables: Arrays that contain indices of patterns (tiles)

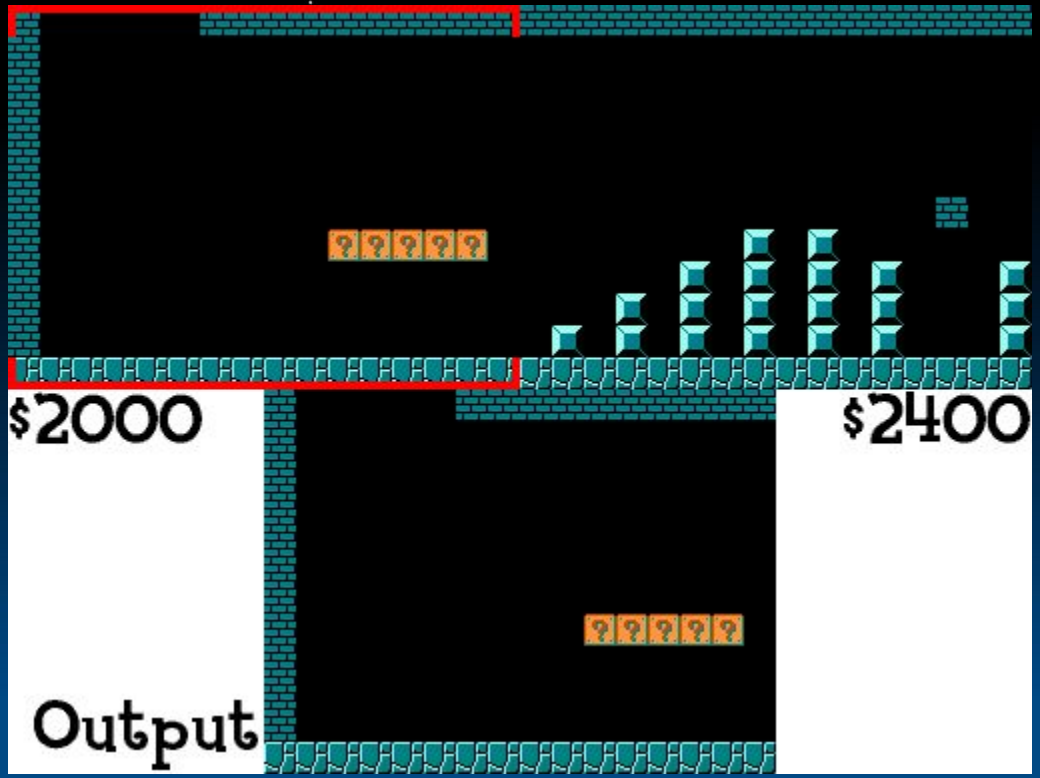
Each nametable contains 30x32 tiles (256x240 pixels = 1 fullscreen)

0x2000	0x2400
0x2800	0x2C00

Each nametable is 1 KB, however there are only 2 KB of VRAM! Solution: mirroring:

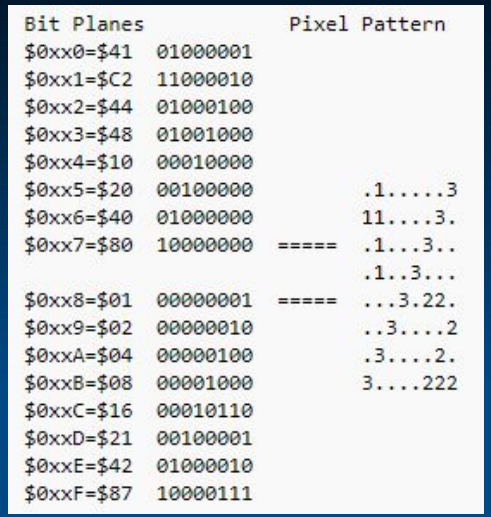
- **Vertical mirroring: 0x2000 equals 0x2800 and 0x2400 equals 0x2C00**
- **Horizontal mirroring: 0x2000 equals 0x2400 and 0x2800 equals 0x2C00**

PPU: Scrolling



PPU: Pattern/Tile tables

Pattern table: Area of data (part of CHR-ROM) that defines tiles (each tile = 16 bytes) Each pixel is represented by a 2 bit value (4 colours, 1 being transparent) made up of 2 planes.



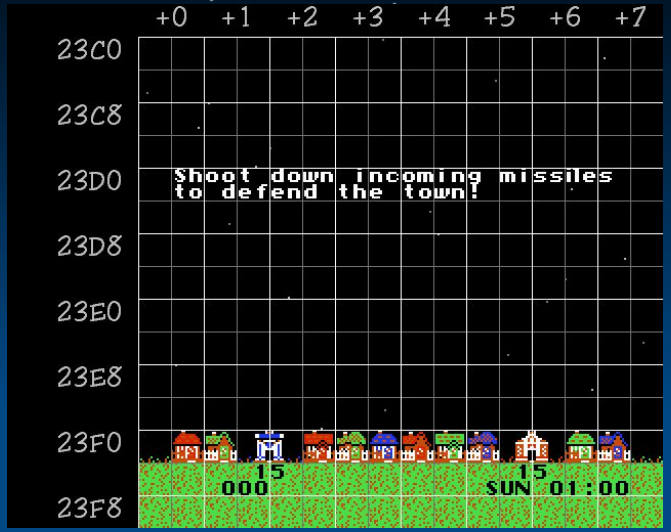
PPU: Pattern/Tile tables

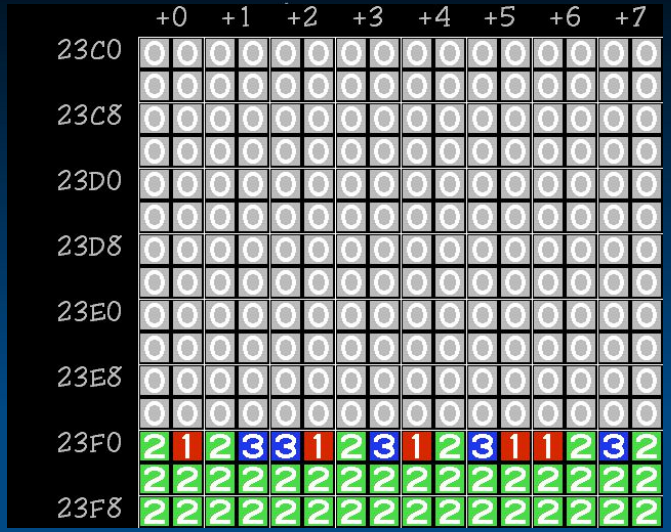
```
Bit Planes                Pixel Pattern
$0xx0=$41  01000001
$0xx1=$C2  11000010
$0xx2=$44  01000100
$0xx3=$48  01001000
$0xx4=$10  00010000
$0xx5=$20  00100000          .1.....3
$0xx6=$40  01000000          11....3.
$0xx7=$80  10000000  =====   .1...3..
                             .1..3...
$0xx8=$01  00000001  =====   ...3.22.
$0xx9=$02  00000010          ..3....2
$0xxA=$04  00000100          .3....2.
$0xxB=$08  00001000          3....222
$0xxC=$16  00010110
$0xxD=$21  00100001
$0xxE=$42  01000010
$0xxF=$87  10000111
```

PPU: Attribute tables

Attribute table: 64-byte array at the end of each nametable that controls which palette is assigned to each part of the background.

Each byte controls a 4x4 tile area. Each 2 bits of this byte controls a 2x2 tile sub-area.





PPU: CalculateBkgColourAt()

1. Calculate pattern index (Base nametable + tile position within nametable)
2. Get pattern value (2 bits) -> Index within palette.
   If 0, return transparent colour
3. Get attribute value, select area bits (2 bits) -> Palette number.
4. Final colour index = Palette number + Index within palette.

PPU: Sprites

- Up to 64 sprites (OAM)
  - Each sprite is 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)

PPU: Sprite evaluation

- For each scanline: find (up to 8) sprites in scanline, copy to secondary OAM
- For each pixel within scanline
  - Find first sprite that overlaps pixel within secondary OAM
  - Evaluate colour for pixel (apply flipping if necessary)
    - If colour = transparent, check next sprite
    - If colour != transparent, return colour index

PPU: RenderPixel()

```
void PPU::RenderPixel(int x, int y)
{
    SpriteLayer spriteLayer;
    const uint8_t spritePixelColour = CalculateSpriteColourAt(x, y, spriteLayer);
    const uint8_t bkgPixelColour = CalculateBkgColourAt(x, y);

    if (spritePixelColour != kTransparentPixelColour &&
        (spriteLayer == SpriteLayer::Front || bkgPixelColour == kTransparentPixelColour))
    {
        SetPixelInFrameBuffer(x, y, spritePixelColour);
    }
    else if (bkgPixelColour != kTransparentPixelColour)
    {
        SetPixelInFrameBuffer(x, y, bkgPixelColour);
    }
    else
    {
        SetPixelInFrameBuffer(x, y, ReadPPUMem(kUniversalBkgColourAddress));
    }
}
```

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

- Buts 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.
- Again, be PATIENT, DON’T RUSH

Resources

Best NES wiki and forums:

nesdev.com

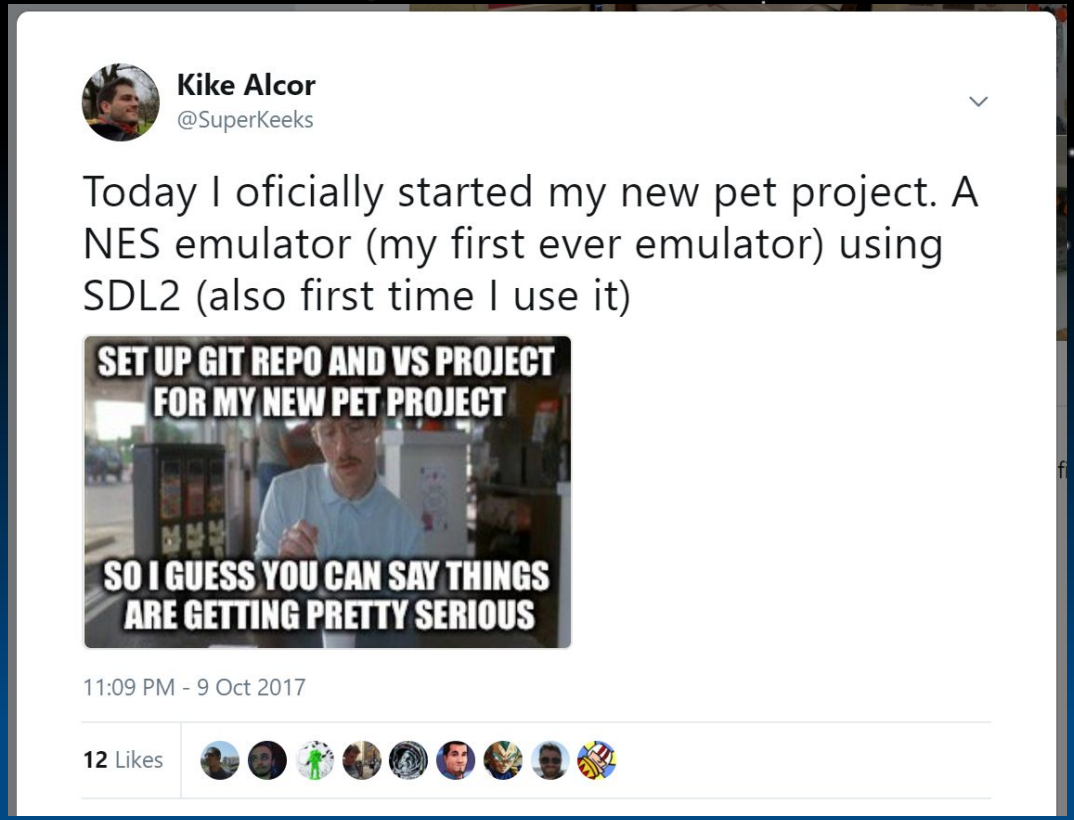
6502 opcode info:

6502.org/tutorials/6502opcodes.html

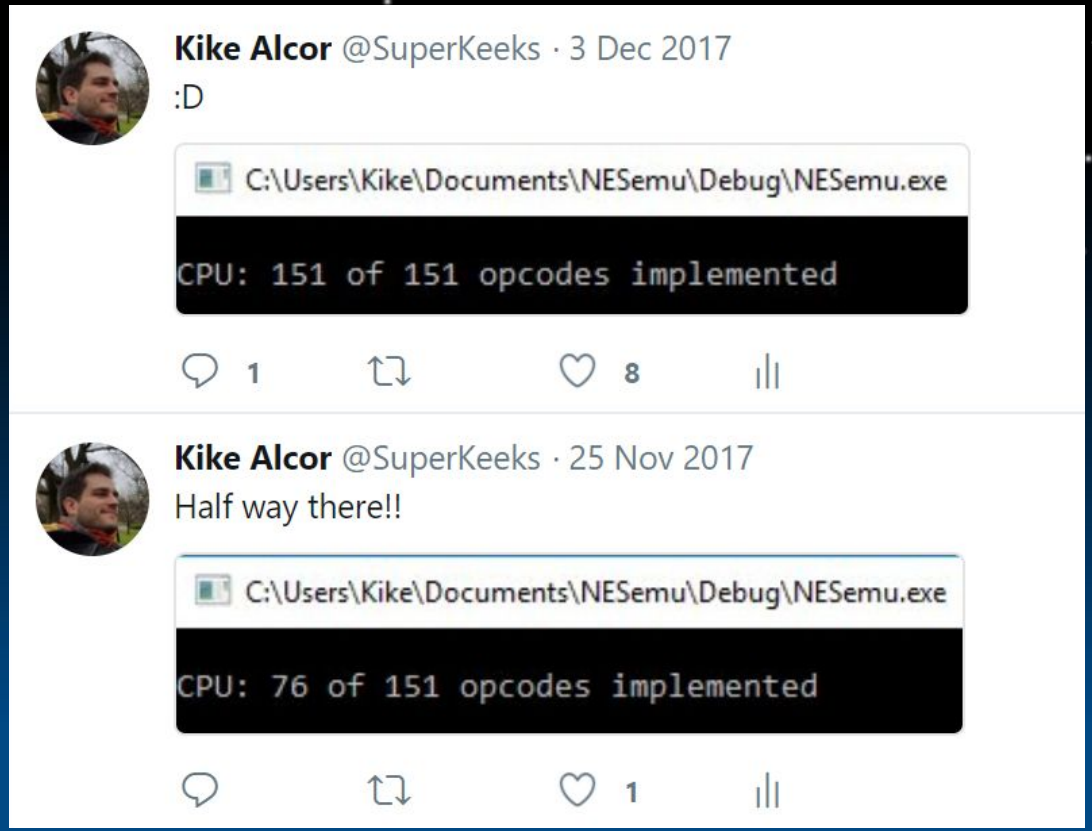
My NES emu repo:

bitbucket.org/gamezer0/nesemu

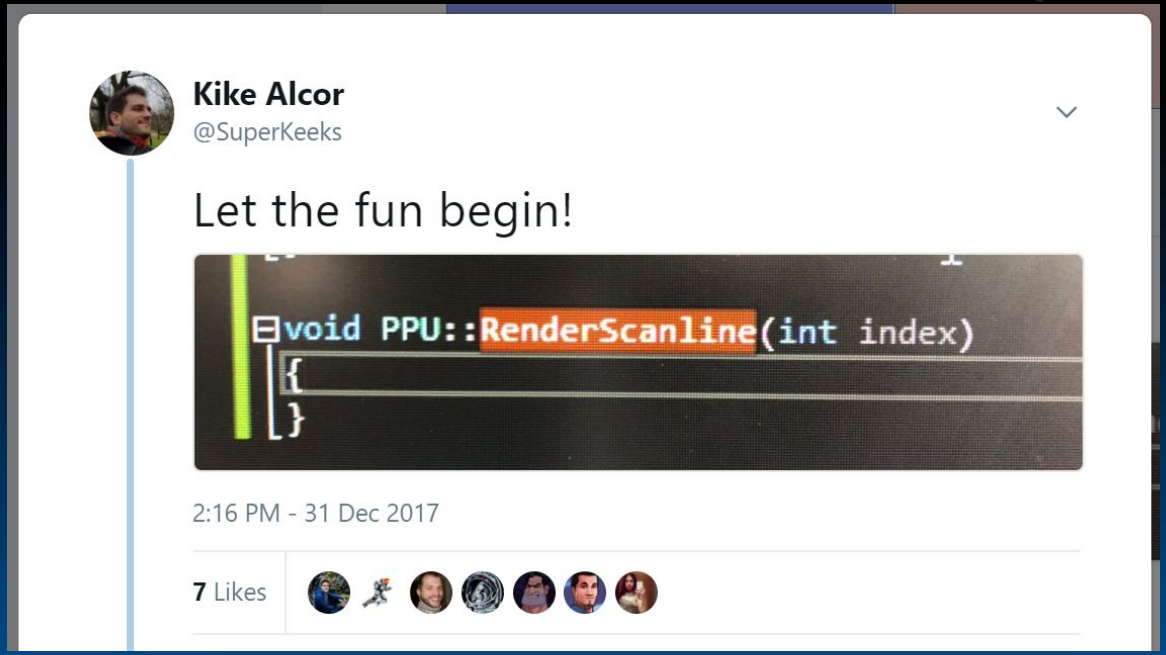
Project timeline



Project timeline

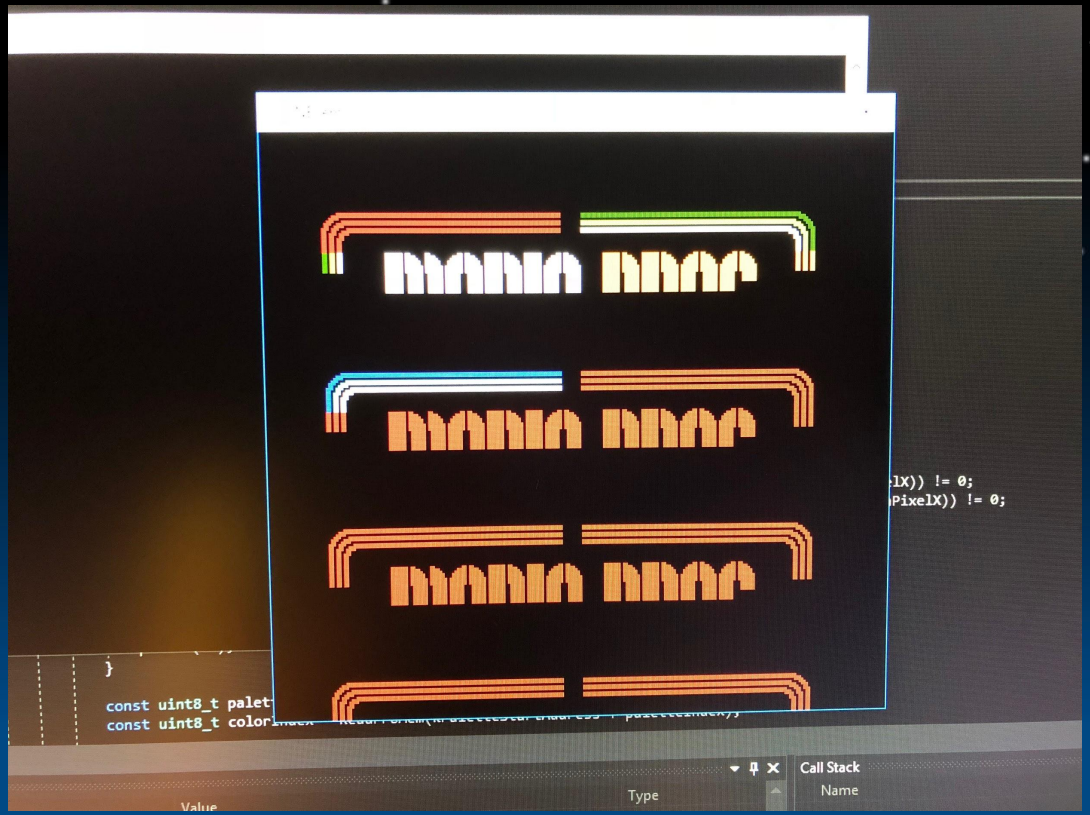


Project timeline

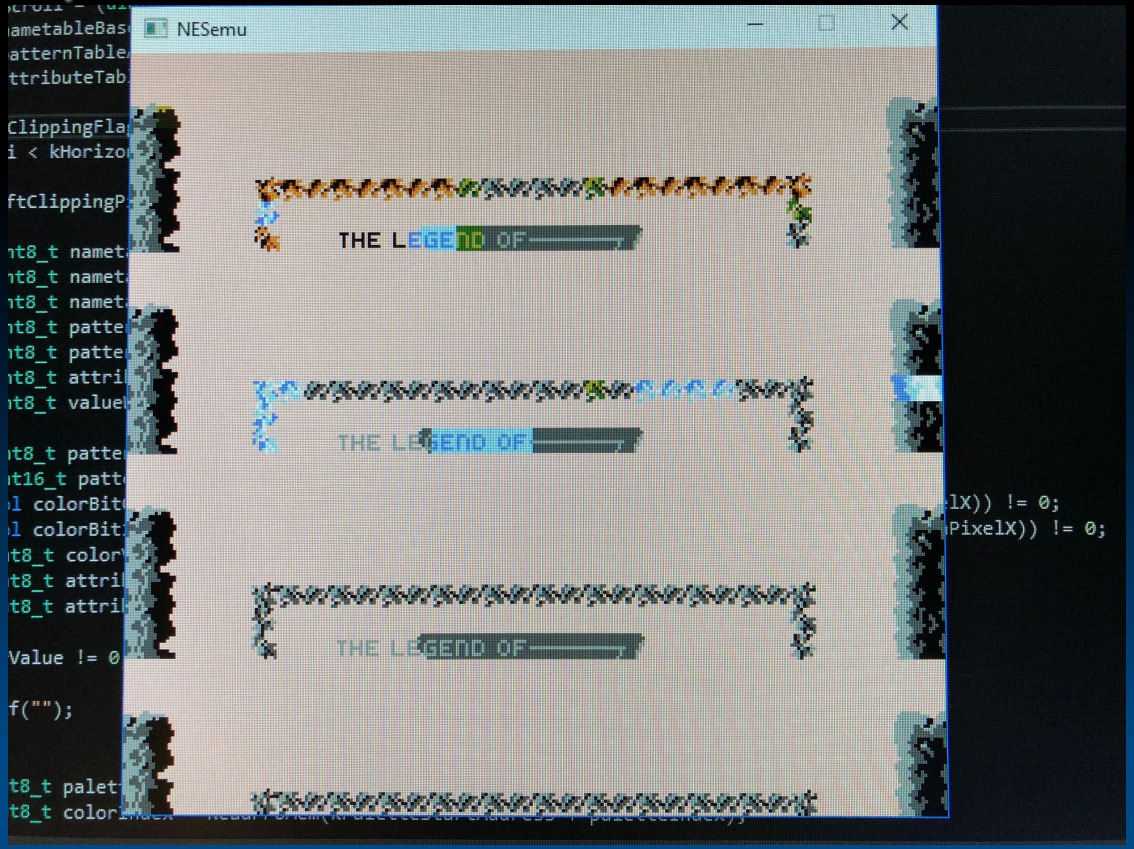


Project timeline

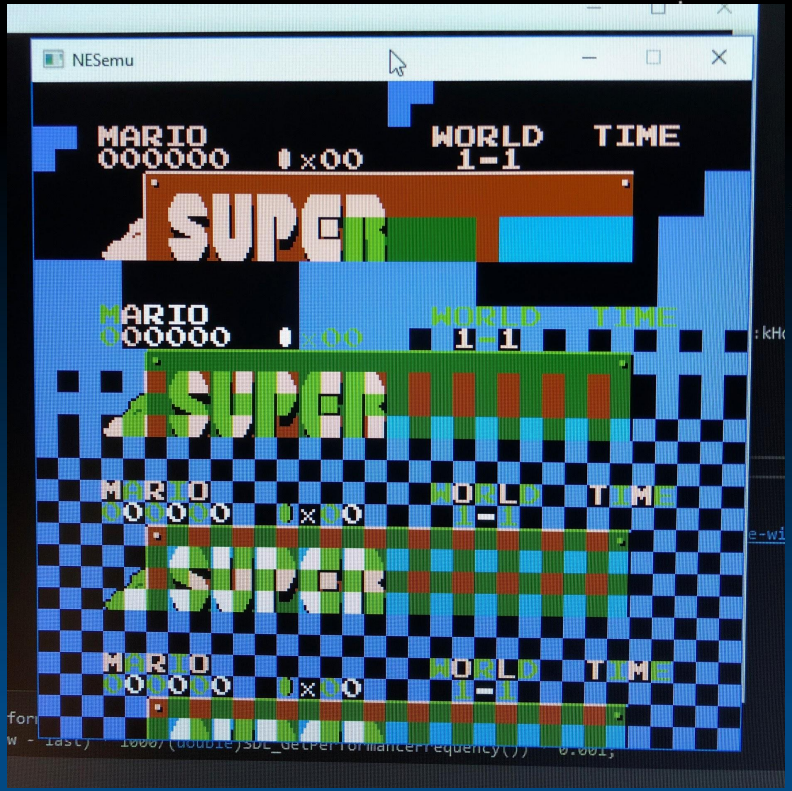
Project timeline



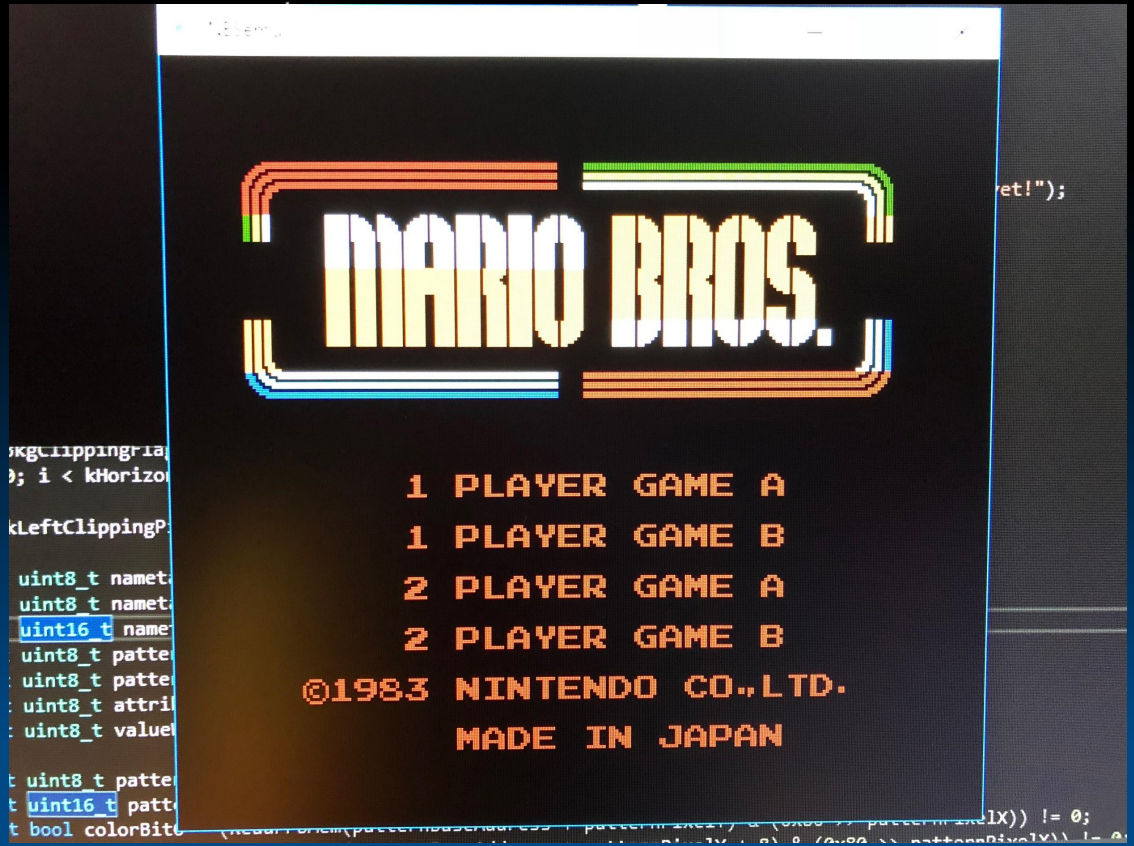
Project timeline



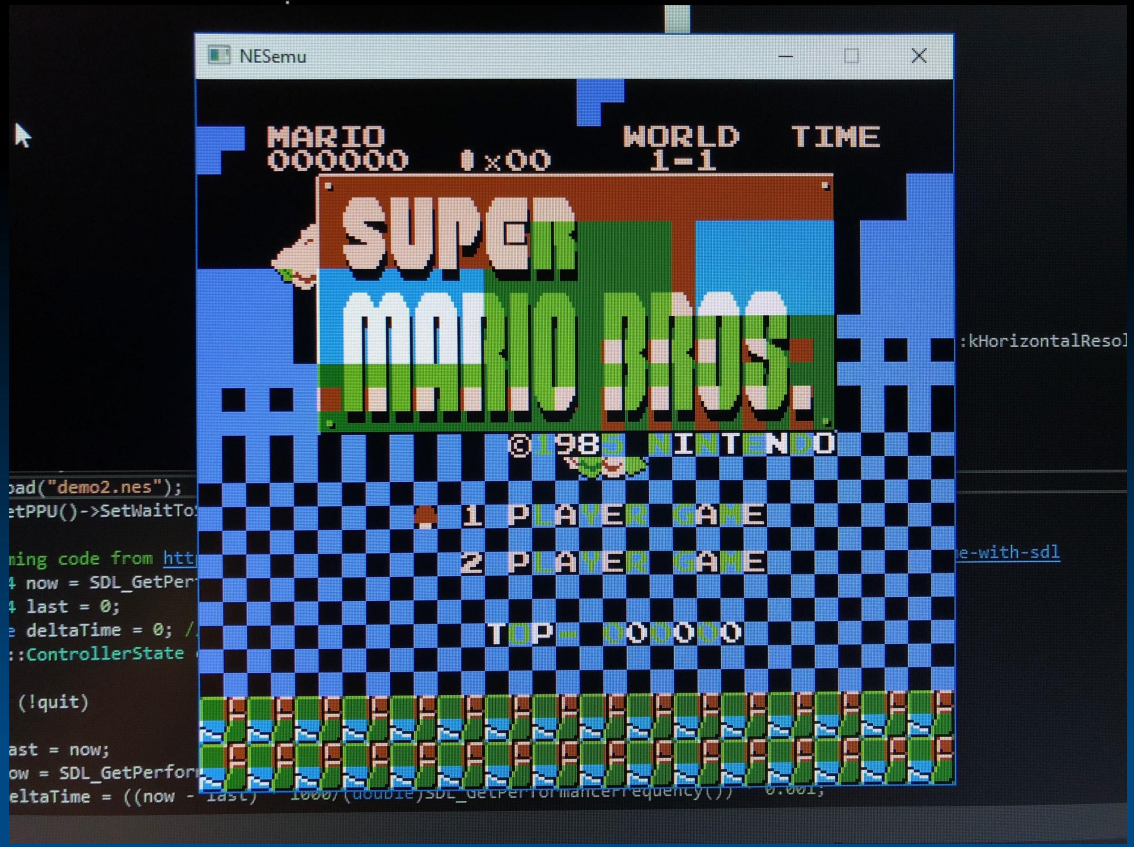
Project timeline



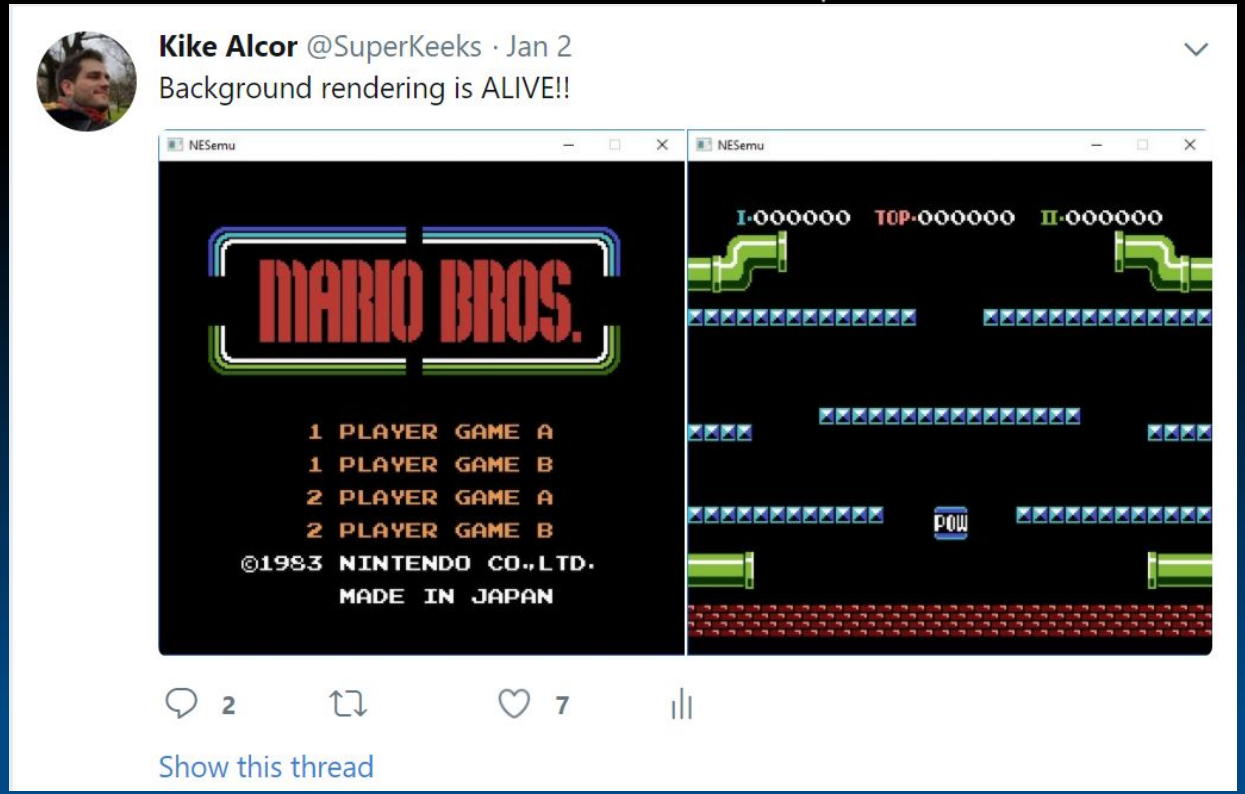
Project timeline



Project timeline



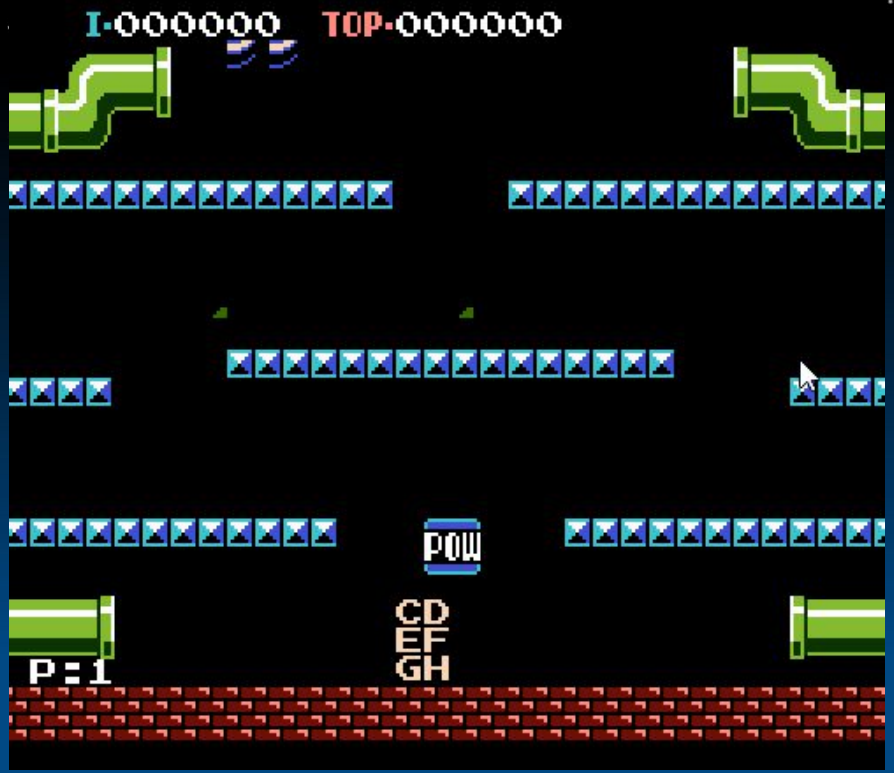
Project timeline



Project timeline

Project timeline

Project timeline



Project timeline

Project timeline

Project timeline

Project timeline

Project timeline

MISSION ACCOMPLISHED!!!

THANKS!

 @SuperKeeks

 ealcor@gmail.com