NES 101

•••

Travel back to the 80's and learn how to create your own NES games with 6502 assembly!

presented by Emily & Grayson from Sleepy Bits Games at Fan Expo San Francisco 2025

I have been in the games industry since 2007, mostly focusing on graphics and optimization.
I have dedicated the past 2.5 years to the NES specifically.
There are a tremendous number of resources online, some of which I will highlight towards the end of this talk.

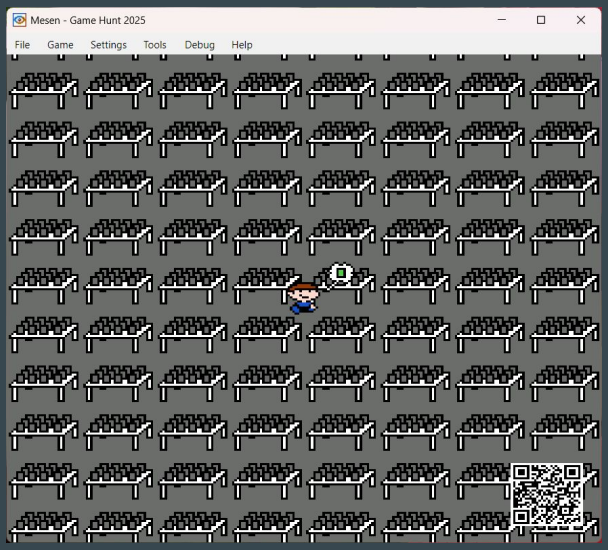
Everything in this presentation is open source!

- Feel free to use this project to get a running start on your NES project!
- No need to take notes or try to remember every detail.

- All you need is this repository:

  https://github.com/grendell/gamehunt2025

Example Project: Game Hunt 2025



Let’s jump to the emulator for a quick demonstration.

So why learn 6502 assembly?

It’s not the only way to make NES games in 2025.
Great alternatives exist.
It’s definitely not the easiest way.

Apple][, Atari 2600, VIC-20, Commodore64

Simplicity in Limitation

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

Here are all the commands available in 6502 assembly!

That doesn’t look simple…

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

By the time you’re through the sample project, you’ll have example usage of all the highlighted commands.

Speak Directly to the Hardware

Matt Hughson (NES & Game Boy D
@matthughson

With some help from the #nesdev folks, Super Sunny World got a huge performance boost today!

Highlighted in the screenshot on the left is how much of the "frame time" was dedicated to moving 4 smashed blocks. On the right is the exact same effect taking 3x less time!

3:40 PM · March 1, 2023

Matt Hughson (NES & Game Boy D
@matthughson

As you might know, most of the game is written in C. Moving things around the screen involves lots of code like this (adding 2 16-bit values)

This looks simple enough, but compiles to ~50 assembly instructions. If written by hand in assembly, this would be about 10 instructions.

3:40 PM · March 1, 2023

Matt Hughson (NES & Game Boy D
@matthughson

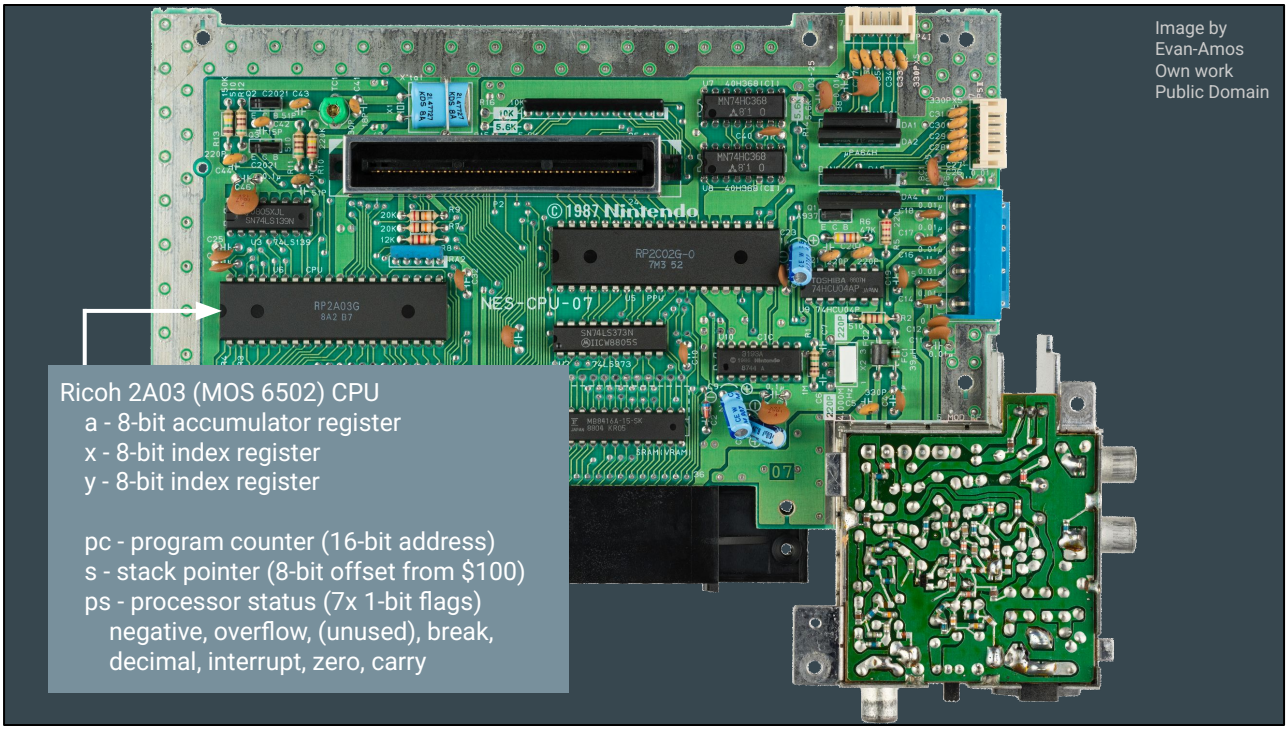
Often when working in C on the NES, we can coax the compiler to generate decent assembly, but in this case we couldn't figure out how.

Luckily the compiler allows me to *inline* raw assembly (16bit add by lidnariq) in C. Here's what that 1 line of C becomes (6x perf improvement)

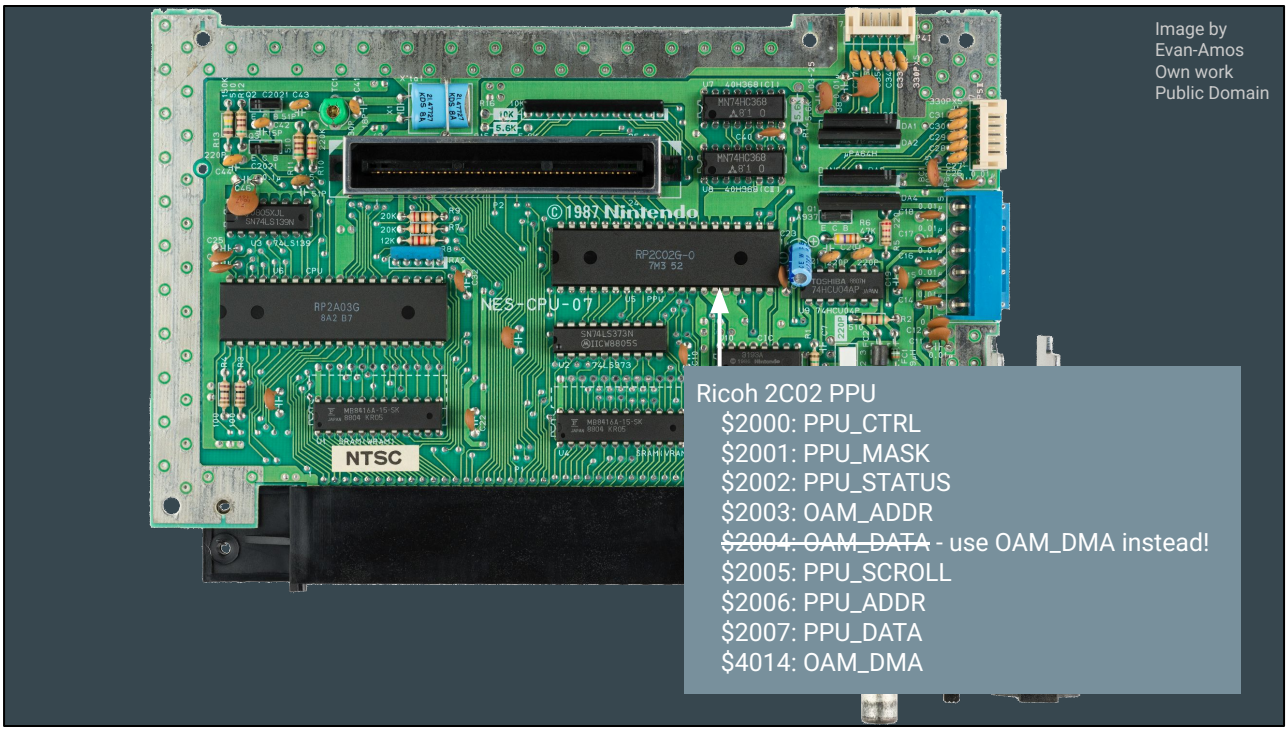
```
//fx_objs.pos_x[i] += fx_objs.dir_x[i];

__asm__ ("lda %v", i);
__asm__ ("asl");
__asm__ ("tay");
Uncommitted changes
__asm__ ("clc");
__asm__ ("lda %v+%b,y", fx_objs, offsetof(fx_actors, pos_x));
__asm__ ("adc %v+%b,y", fx_objs, offsetof(fx_actors, dir_x));
__asm__ ("sta %v+%b,y", fx_objs, offsetof(fx_actors, pos_x));

__asm__ ("iny");
__asm__ ("lda %v+%b,y", fx_objs, offsetof(fx_actors, pos_x));
__asm__ ("adc %v+%b,y", fx_objs, offsetof(fx_actors, dir_x));
__asm__ ("sta %v+%b,y", fx_objs, offsetof(fx_actors, pos_x));
```

3:40 PM · March 1, 2023

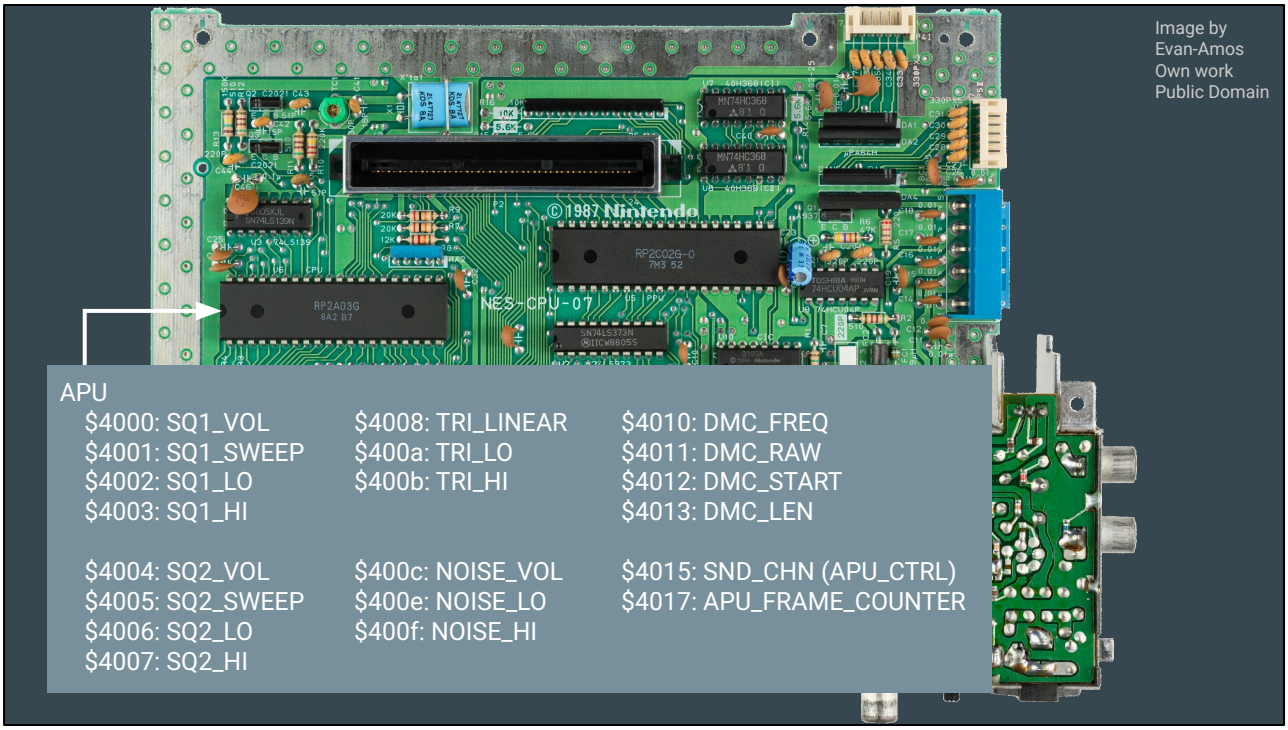
Matt Hughson, developer of Super Sunny World, Witch n' Wiz, and many others.
Give him a follow to see all the cool things he's doing on the NES.
Reading Rainbow: You don't have to take my word for it!



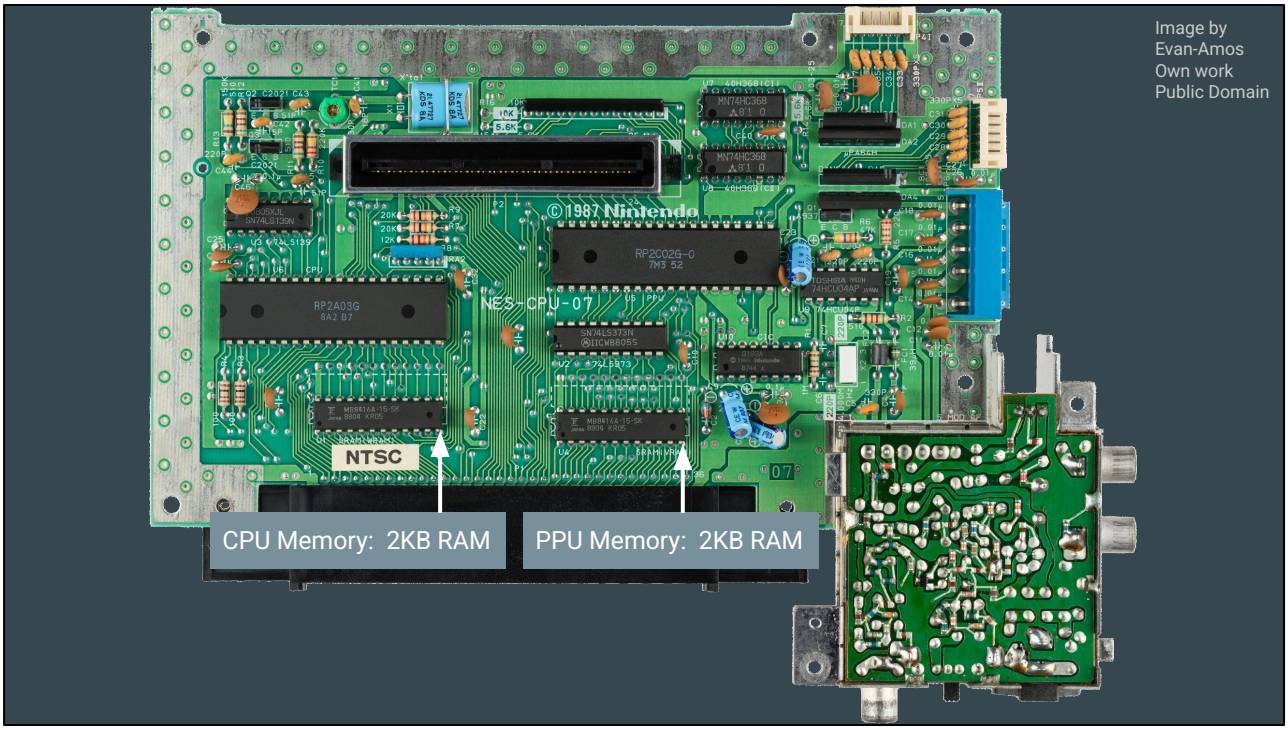
Here are the registers, or how we'll interact with, the NES's CPU.
a is your only general purpose register!
x and y are most often used for indices and/or counters.



Here are the registers, or how we'll interact with, the NES's PPU.
Picture Processing Unit (Graphics)
Each register has a specific function. We'll see an example of several towards the end of the talk.
Most data is given to the PPU by the CPU setting the destination with PPU_ADDR and then the content with PPU_DATA, byte-by-byte.
OAM_DATA can corrupt your sprite data! Use OAM_DMA (Direct Memory Access) instead.

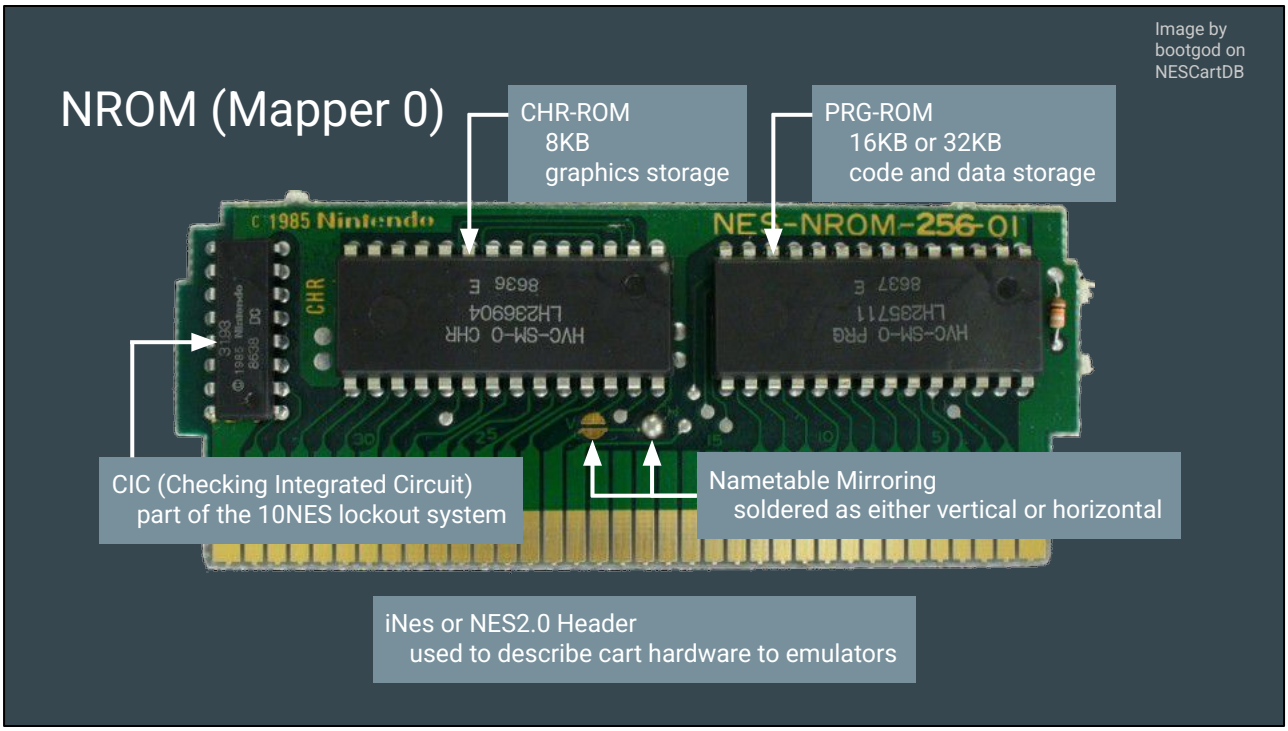


Audio Processing Unit
Wait a minute, isn't that the CPU? Ricoh removed some functionality from the MOS 6502 and replaced it with audio capabilities.
Five sound channels: two pulse or wave channels (bleeps and bloops), one triangle wave channel (lower, sustained notes),
the noise channel (explosions, buzzers, percussion), and a Delta Modulation Channel for sample playback.



The CPU cannot directly access the PPU's memory.
It must issue commands via the PPU to transfer data.

To read button state, the controller must first be “strobed” to fill the buffer.
The buffer is then read one button at a time by the CPU.
Accessories like the zapper and multitap work in the same way, but we won’t be covering them today.



This configuration is used by Balloon Fight, Donkey Kong, Duck Hunt, Excitebike, Ice Climbers, Super Mario Bros. and many more.
To encapsulate this information for emulators, the iNes or NES2.0 header is used. This has even been used in official products, like Animal Crossing and the NES Classic!

Number Systems

- Decimal
  - base 10: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - no ca65 prefix
  - example: 240
- Hexadecimal
  - base 16: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f
  - ca65 prefix: $
  - example: $f0
- Binary
  - base 2: 0, 1
  - ca65 prefix: %
  - example: %11110000

Any number preceded by # represents a constant.

#240, #$f0, #%11110000

You may have noticed letters in the APU registers. In fact, all addresses are typically represented in hexadecimal.
Hexadecimal is just a shorthand for groups of four binary digits.

6502 Instructions

binary and:

- and
  - a = a & param
- useful to clear bits

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

binary or:

- ora
  - a = a | param
- useful to set bits

A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1

binary xor (exclusive or):

- eor
  - a = a ^ param
- useful to flip bits

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

All of these instructions can accept a constant, variable name, an address, or what's called an indirection/pointer.
And is useful to clear or isolate bits, or is useful to set bits, and xor is useful to flip bits.

6502 Instructions

branch:

- bcc
- bcs
- beq
- bne
- bpl
- bmi
- bvc
- bvs
- checks specific status flags to make decisions

jump to address:

- jmp
- unconditionally jumps to a specific address
- can't go back

jump to subroutine:

- jsr
- updates the stack and jumps to a specific address
- can go back (rts)

Branches are how we make decisions, based on the various status flags we saw earlier, just like "if" clauses in more modern languages.
Because jmp doesn't update the stack, there's no automatic way to return from our destination.
jmp can be useful for chaining together subroutines that always execute in order and for "else" clauses.

Program Flow

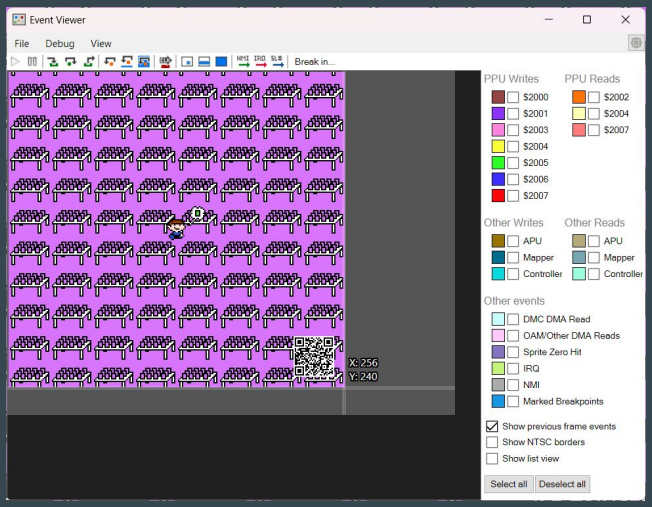
- reset vector
  - starting point for code execution
  - program counter jumps here during power-on and reset
  - responsible for game initialization
- game loop
  - typically an infinite loop after initialization
  - usually waits for a graphics update before iterating
  - responsible for per-frame game logic updates
  - runs while the PPU is drawing the previous frame's graphics

Program Flow

- NMI vector
  - interrupt signaled at the beginning of Vertical Blank
  - typically graphics are only updated during VBlank
  - runs while the TV prepares for the next frame
- IRQ vector
  - entry point for all other interrupts
    - brk instructions
    - APU DMC playback
    - advanced game cartridge hardware

Interrupts are responsible for saving and restoring register values and status flags, as we will see.

Frame Timing



Event Viewer inside Mesen

Frame timing on the NES is based on how CRT televisions worked.
The electron gun has to be repositioned after every row (HBLANK), and every full screen (VBLANK).
We can only interact with the PPU while it isn't drawing, so all graphics updates must happen in one of these blanking periods, usually VBLANK.
"Racing the Beam"

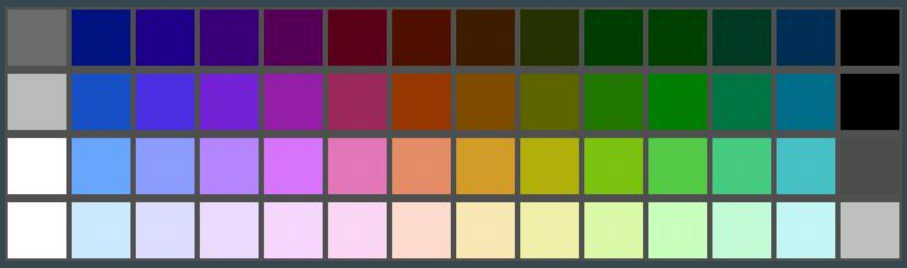
Memory Map

- Zero Page: $00 - $ff
  - fast-access variables
- Stack: $0100 - $01ff
  - program flow
- OAM Shadow: $0200 - $02ff
  - next sprite update
  - copied via DMA (Direct Memory Access) during VBlank
- Unused: $0300 - $07ff

OAM - Object Attribute Memory

Palettes

- The NES can produce 54 total unique colors

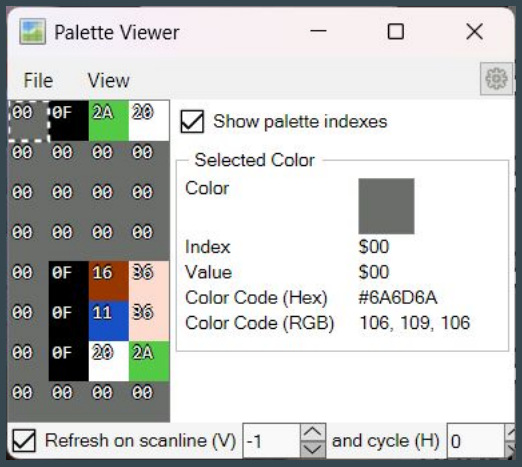


"Smooth (FBX)" palette definition and image by FirebrandX

The PPU also has emphasis bits for red, green, and blue, to offer full-screen tinting, as well as grayscale.
As a side-note, the NES does not define RGB values for its colors!
A ton of work as gone into identifying and replicating its color output for emulators and modern devices.
Embrace the unpredictability!

Palette Usage

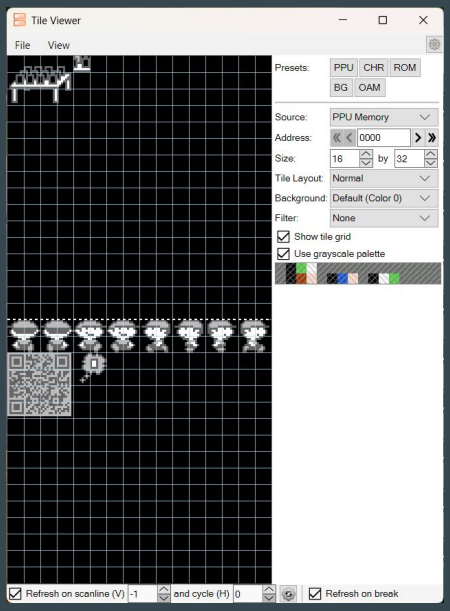
- Backgrounds
  - 4 palettes of 3 colors + a shared background color
- Sprites
  - 4 palettes of 3 colors + transparency



Palette Viewer inside Mesen

Pattern Tables

- 1KB array of tiles
- stored as packed indices into a palette
- 2 pattern tables are available to store graphics on NROM

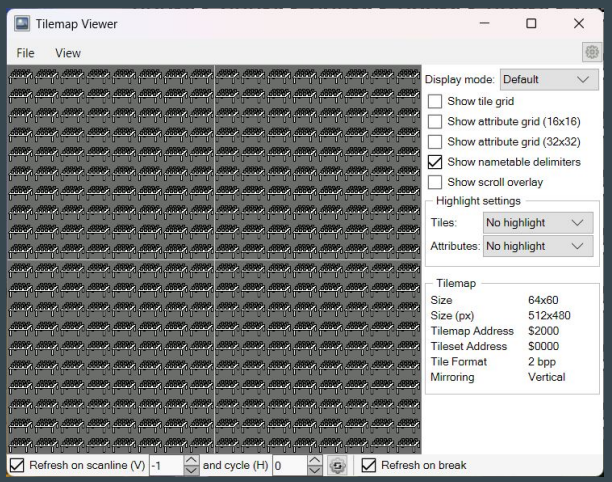


Tile Viewer inside Mesen

Backgrounds and sprites can reference the same or separate pattern tables. There are various tools available to help format the tile data, which will be shared towards the end of this talk.

Nametables

- 960 byte array of tile indices
- references the tiles in the background pattern table
- used to layout backgrounds into 32 x 30 screens
- 2 nametables are available to support scrolling on NROM

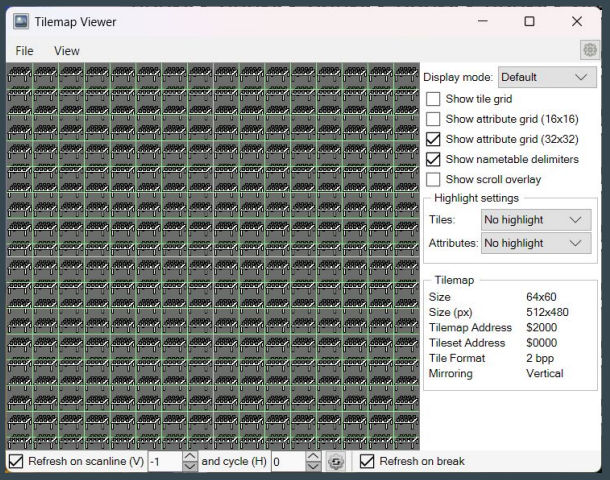


Tilemap Viewer inside Mesen

There are various tools available to help format the tile data, which will be shared towards the end of this talk.
We will be using vertical mirroring, which is typically used for horizontal scrolling, such as Super Mario Bros.
Horizontal mirroring is typically used for vertical scrolling, such as Ice Climbers.
More advanced mappers can support changing the mirroring during gameplay, such as Metroid.

Attribute Tables

- 64 byte array of packed palette indices in an 8 x 8 grid
- used to control a group of 4 x 4 tiles for backgrounds
- palettes can only be specified in groups of 2 x 2 tiles



Tilemap Viewer inside Mesen

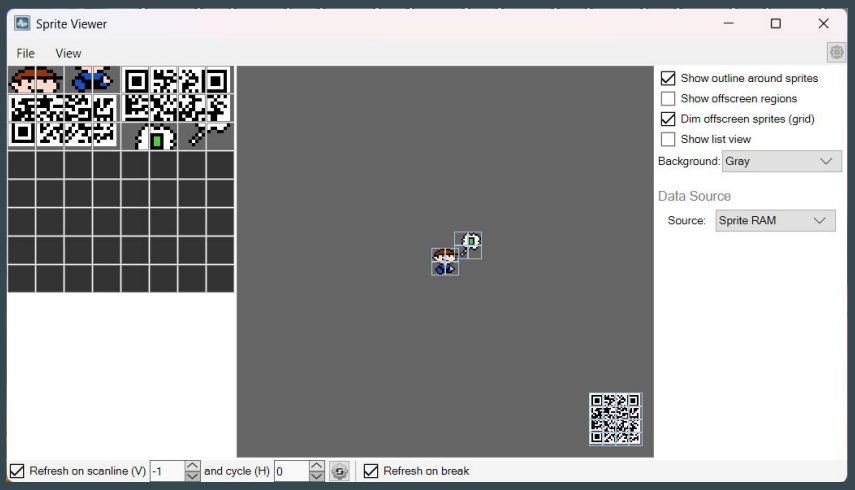
This is the cause of those strange screen-edge colors while scrolling in so many NES games!

Sprite OAM (Object Attribute Memory)

- 256 byte array of data for 64 sprites
  - y position
    - starting from the top of the screen
    - offset by one pixel!
  - tile index
    - references the sprite pattern table
  - attributes
    - palette index
    - priority
    - horizontal and vertical flipping
  - x position
    - starting from the left of the screen

It should be noted that it is impossible to place a sprite beyond the left edge of the screen, or above the second scanline of the screen!
Priority controls whether the sprite appears above or below the background graphics layer.
If more than 8 sprites appear on a single scanline, only the first 8 defined in memory will be drawn. This is why sprite flickering is utilized often on the NES!

Sprite OAM (Object Attribute Memory)



Sprite Viewer inside Mesen

Game Loop

```
game_loop:
  ; wait for frame to be completed
  inc waiting_for_nmi
: lda waiting_for_nmi
  bne :-

  jsr handle_input
  jsr update_animation

  jmp game_loop
```

NMI will eventually store 0 into waiting_for_nmi.
After that, we can continue to handle_input, which immediately calls read_joypad.

Check Cart - Part 1

```
; are we in the right nametable?
lda ppu_ctrl
and #1

cmp #HIDDEN_CART_NT
bne play_failure
```

There are several checks to see if we've succeeded in finding our dream cart. Recall that nametables can be thought of as screens, so we first check if we're on the correct "screen."

Check Cart - Part 2

```
; are we close enough horizontally?
; a = player_x - cart_x
lda scroll_x
sec
sbc #HIDDEN_CART_X

; a = |a| (absolute value)
jsr abs

; is a < SEARCH_DISTANCE?
cmp #SEARCH_DISTANCE
bcs play_failure
```

Checking the actual distance from the player to the cart would require an expensive, handwritten square root operation, so we will only check Manhattan distance.
Taking the absolute value of the horizontal distance uses two's complement, which we unfortunately don't have time to get into today, but the source code includes a link to wikipedia's explanation if you'd like to dig into it.
Finally, we use cmp and check the carry bit to determine if horizontal distance is less than the constant SEARCH_DISTANCE.

Check Cart - Part 3

```
; are we close enough vertically?
; a = player_y - cart_y
lda scroll_y
sec
sbc #HIDDEN_CART_Y

; a = |a| (absolute value)
jsr abs

; is a < SEARCH_DISTANCE?
cmp #SEARCH_DISTANCE
bcs play_failure

jmp play_success
```

Same check, only in the vertical direction this time. Nothing new!
If all of these checks pass, we unconditionally jump to play_success.

We interrupt this talk… for graphics!

NMI - Part 1

```
; retain previous value of a on the stack
pha

; clear vblank flag
bit PPU_STATUS

; update sprite OAM via DMA
lda #>OAM_SHADOW
sta OAM_DMA
```

Since we don’t know when the interrupt will occur, we must preserve all the registers we will affect during the interrupt.
The processor status flags are automatically preserved for us.
The entire OAM is updated via Direct Memory Access by writing the high-byte of the OAM shadow’s address to OAM_DMA.

NMI - Part 2

```
; show backgrounds and sprites, including leftmost 8 pixels
;            0 - disable grayscale rendering
;           1  - show background in left-most 8 pixels on screen
;          1   - show sprites in left-most 8 pixels on screen
;         1    - draw backgrounds
;        1     - draw sprites
;       0      - disable red emphasis
;      0       - disable green emphasis
;     0        - disable blue emphasis
lda #%00011110
sta PPU_MASK
```

Enabling the backgrounds and sprites during each NMI ensures that no partial frames are drawn during initialization.

NMI - Part 3

```
; update background scroll position
lda scroll_x
sta PPU_SCROLL
lda scroll_y
sta PPU_SCROLL

; update base nametable
lda ppu_ctrl
sta PPU_CTRL

; allow game loop to continue after interrupt
lda #0
sta waiting_for_nmi
```

Scroll position is updated via a pair of writes to PPU_SCROLL, similar to PPU_ADDR.

NMI - Part 4

```
; restore previous value of a before interrupt
pla
rti
```

Finally, the contents of a is restored before a Return from Interrupt instruction.
rti is necessary here, so that the processor status flags are restored correctly and code execution returns to the point when the interrupt took place.

We made it!

Required Tools

- cc65
  - https://cc65.github.io
- emulator
  - Mesen
    - https://www.mesen.ca
  - FCEUX
    - https://fceux.com/web/home.html
- text editor
  - https://imgs.xkcd.com/comics/real_programmers.png

cc65 is a development package for the 6502, containing an assembler, linker, and C compiler.
Mesen and FCEUX both have good tools built in for NES development. The event viewer in Mesen is invaluable!
Of course you'll need your favorite text editor to write your code. If in doubt, please reference this XKCD page.

Optional Tools

- make
  - https://gnuwin32.sourceforge.net/packages/make.htm
- graphics editor
  - NEXXT: https://frankengraphics.itch.io/nexxt
  - NES Lightbox: https://web.archive.org/web/20230929133500/https://famicom.party/neslightbox/
  - YY-CHR: https://www.romhacking.net/utilities/958/
- Famistudio
  - https://famistudio.org
- NESmaker
  - https://www.thenew8bitheroes.com

A Makefile has been included with the example project for simple building. Simpler rebuild scripts are also available, but don't offer incremental building.
All of these graphics editors are compatible with the graphics formats presented in this project. It's also possible to process your own PNGs! Use what works for you.
Famistudio is an absolutely amazing music editor and playback engine for the NES. It has unmatched functionality and tutorials to get you up to speed.
This talk has been focused on 6502 assembly, but if you're not ready to take that plunge, NESmaker is a great alternative.

Additional Resources - References

- 6502 OpCodes
  - http://www.6502.org/tutorials/6502opcodes.html
- 6502 Math
  - https://codebase64.org/doku.php?id=base:6502_6510_maths
- NesDev.org
  - https://www.nesdev.org/wiki/Nesdev_Wiki

This has been a whirlwind tour of an introduction to NES programming, but thankfully, there are so many online resources available.
Whether you are looking for a deeper explanation of something, or just want to hear someone else's take, these should have you covered.
This list is by no means exhaustive and is always open to suggestions!

Additional Resources - Online Courses

- Pikuma
  - https://pikuma.com/courses/nes-game-programming-tutorial
- Nerdy Nights
  - https://nerdy-nights.nes.science

Pikuma is a set of premium online courses taught by Gustavo Pezzi, a computer science lecturer from London.
Nerdy Nights is a series of online tutorials written by Brian Parker.

Additional Resources - Technical YouTube Channels

- NesHacker
  - https://www.youtube.com/@NesHacker
- Displaced Gamers
  - https://www.youtube.com/@DisplacedGamers
- explod2A03
  - https://www.youtube.com/@explod2A03

Ryan at NesHacker does a great job introducing concepts vital to NES, and lately Game Boy, programming.
Displaced Gamers routinely takes thorough deep-dives through the code of retail NES games.
explod2A03 is a great resource for understanding the audio capabilities of the NES and historic accomplishments.

Questions, Suggestions, and/or Gratuitous Praise?

- https://sleepybits.com
- feedback@sleepybitsgames.com

- @chimbraca and @SleepyBitsNES on Twitter
- @grendell and @SleepyBitsGames on Bluesky
- @SleepyBitsGames on YouTube for game teasers and trailers

Thank you!!

Most importantly, remember to have fun!
Thanks so much for attending this talk today, and have an awesome weekend.
Thank you!