ChatGPT

Using Mesen-S Lua Scripting to Read Sprite Data and Map to ROM Graphics

Overview of Mesen-S Lua API for SNES Debugging

Mesen-S provides a Lua scripting API that can read various SNES memory regions in real-time, including CPU memory, VRAM, OAM (sprite memory), and CGRAM (palette memory) [1] [2]. This allows us to inspect sprite data on the fly during emulation. The core functions include `emu.read(address, memType)` (and its 16-bit/32-bit variants) for memory reading, and callback hooks to run code each frame [3] [4]. We can leverage these to extract sprite OAM data every frame, determine which graphics tiles are in use, and then trace those back to their location in the ROM. In addition, we can read the sprite palette data from CGRAM to identify the colors used.

Reading Sprite OAM Data in Real Time

On the SNES, **Object Attribute Memory (OAM)** holds the sprite display list. Each sprite uses 4 bytes of OAM data: Y position, tile index (also called "Name"), attributes, and X position (low byte) [5]. There are 128 sprite slots, so OAM is 512 bytes for the main entries plus 32 bytes of extra data (which hold the high bit of X positions and other flags). Using Mesen-S's Lua API, we can read OAM as follows:

- **Select the memory region:** Mesen-S defines `emu.memType.snesSpriteRam` for OAM (sprite RAM) [2]. This treats OAM as a 544-byte address space (0x000–0x21F in this memType).
- **Read each sprite's data:** Iterate over the 128 sprite entries (index 0–127), reading 4 bytes per entry from `snesSpriteRam`. Also read the **high X bit** from the extra OAM space (the 9th bit of the X coordinate).

Below is a Lua snippet that hooks into the end of each frame (during VBlank) and logs Kirby's and enemies' sprite data (for simplicity, it reads all OAM entries and you can filter by specific indices if known):

```
-- Callback each frame (end of frame during VBlank) to read OAM
emu.addEventCallback(function()
    local OBSEL = emu.read(0x2101, emu.memType.snesDebug)  -- Sprite size/VRAM base register (no side-effects) [6]
    local base_index    = OBSEL & 0x07                     -- ccc bits: OBJ tile VRAM base index
    local name_select  = (OBSEL >> 3) & 0x03               -- Name select bits (for tile index offset) [7]
    local obj_vram_base = base_index * 0x4000              -- Base VRAM address for sprite tiles (16KB-aligned) [8]

    for i = 0, 127 do
        local addr = i * 4
        local y    = emu.read(addr,     emu.memType.snesSpriteRam)
        local tile = emu.read(addr + 1, emu.memType.snesSpriteRam)
```

```lua
        local attr = emu.read(addr + 2, emu.memType.snesSpriteRam)
        local x_lo = emu.read(addr + 3, emu.memType.snesSpriteRam)
        -- High X bit: each of 128 sprites has one extra X bit stored in the 16 extra bytes (0x200-0x20F)
        local high_x_byte = emu.read(0x200 + math.floor(i/8), emu.memType.snesSpriteRam)
        local x_hi = (high_x_byte >> (i % 8)) & 0x1        -- i%8-th bit of the byte is the high X bit for sprite i
        local x = x_lo + (x_hi << 8)                       -- full 9-bit X coordinate

        -- If this sprite is not being used (e.g., Y == 0xF0 which is often off-screen), skip
        if y < 0xF0 then
            local pal = attr & 0x7          -- 3-bit palette index (0-7) [9]
            local h_flip = (attr >> 6) & 1
            local v_flip = (attr >> 7) & 1
            local priority = (attr >> 4) & 0x3

            -- Map OAM tile index to VRAM tile address:
            local tile_addr
            if tile < 0x100 then
                -- In first 256-tile block
                tile_addr = obj_vram_base + tile * 32                                -- each 4bpp tile is 32 bytes
            else
                -- In second 256-tile block, offset by (NameSel+1)*8KB [10] [8]
                tile_addr = obj_vram_base + ((name_select + 1) * 0x2000) + (tile - 0x100) * 32
            end

            -- Read tile graphics data from VRAM (32 bytes) and output info
            local tileData = emu.read(tile_addr, emu.memType.snesVideoRam)

-- (In practice, use emu.readRange or loop to get all 32 bytes if available)
            emu.log(string.format(
              "Sprite %d: X=%d,Y=%d Tile=$%03X VRAM=$%04X Pal=%d FlipH=%d FlipV=%d",
               i, x, y, tile, tile_addr, pal, h_flip, v_flip))
        end
    end
end, emu.eventType.endFrame)
```

Explanation: In this script, we use `emu.addEventCallback` with `emu.eventType.endFrame` to run our code at the end of each frame [11]. We read OAM bytes via `emu.read(..., memType.snesSpriteRam)` and reconstruct multi-byte values like the 9-bit X coordinate. The SNES PPU register `$2101` (OBJSEL) is read in debug mode (to avoid side-effects) to get the sprite **VRAM base** and **name select** values. These are used to calculate the VRAM address of the sprite's tile graphics. We then read the tile's data from VRAM via `memType.snesVideoRam`. The script logs each sprite's coordinates, tile ID, VRAM address, palette, and flip attributes. You could refine the script to specifically track Kirby's sprite (if you know which OAM index he uses, e.g., usually sprite 0 or a fixed range) or certain enemy indices.

Mapping Sprite Tiles to ROM Offsets

Once we have the VRAM address and raw tile pixel data, the next step is to find where that tile came from in the **ROM**. In some games, graphics are stored uncompressed in the ROM, meaning the 32-byte pattern for an 8×8 tile in VRAM would directly appear in the ROM (which you can search for). Mesen-S allows reading ROM data via `emu.memType.snesPrgRom` [1] [2]. For example, you could fetch bytes from the ROM file with `emu.read(address, emu.memType.snesPrgRom)`. If the tile data is uncompressed, you could read the 32 bytes from VRAM and scan through the ROM to find a matching sequence.

However, **Kirby Super Star uses compression for graphics data** in most cases (as was common in late SNES games). The game employs a custom HAL Laboratory compression format (documented by Parasyte) [12], so the tile data you see in VRAM is likely the result of decompression. In other words, you **won't find a verbatim 32-byte match in the ROM** for most sprite graphics. Instead, the data is stored compressed, and the game (with help of the SA-1 chip) decompresses it into WRAM/VRAM at runtime [13] [14].

How to locate the ROM source of a sprite:

- One approach is to use **breakpoints or memory write callbacks** to catch when VRAM is being filled with sprite data. For example, you can set a write callback on `memType.snesVideoRam` for the address range of interest. When the callback triggers, inspect the CPU memory (or SA-1 memory) to see where the data came from (e.g., check DMA source registers or track reads from ROM). Mesen-S's Lua has `emu.addMemoryCallback` which can watch for writes [15] [16]. By registering a callback for VRAM writes, you can log what code or data is writing those tiles. Often, games use DMA transfers from ROM or from a decompression buffer in WRAM to VRAM during vblank.

- Another approach is to consult community documentation. In the case of Kirby Super Star, a fan disassembly project exists [17] and a **ROM map** is documented on Data Crystal. These can tell you where certain graphics or sprites are stored in the ROM and how they're compressed. For example, tools like **KompreSS** or **exhal/inhal** were created to decompress KSS graphics [18], indicating the compression algorithm and offering a way to unpack the ROM's graphics for analysis.

In summary, **if the sprite graphics are compressed, you will need to identify the decompression routine or use specialized tools** rather than a simple direct memory search. Mesen-S can still help: by logging memory accesses around the time a sprite is loaded, you might catch the ROM reading sequence or identify the format.

Accessing and Mapping Palette Data

Each SNES sprite uses one of 8 palettes (0–7) for its colors, as indicated by 3 bits in the OAM attribute byte [9]. Kirby Super Star uses the SNES's **CGRAM** (Color Generator RAM) to store palettes. Object palettes typically reside in CGRAM indices 128–255 (the second half, as BG palettes use 0–127) – palette 0 corresponds to entries 128–143, palette 1 to 144–159, and so on. Mesen-S exposes CGRAM via `emu.memType.snesCgRam` [2], where each color is 2 bytes (15-bit BGR format).

Using Lua, you can read sprite palette data like this:

```
-- Suppose 'pal' is the palette number (0-7) from the sprite’s OAM attributes
local cgram_start = 128 + pal * 16    -- starting color index in CGRAM for
this palette
local colors = {}
for j = 0, 15 do
    local color_addr = (cgram_start + j) * 2  -- each color index is 2 bytes
in CGRAM
    local color_word = emu.read16(color_addr, emu.memType.snesCgRam)
    local r =  (color_word       & 0x1F) * 8  -- 5-bit red
    local g = ((color_word >> 5) & 0x1F) * 8  -- 5-bit green
    local b = ((color_word >> 10)& 0x1F) * 8  -- 5-bit blue
    colors[j] = string.format("(%d,%d,%d)", r, g, b)
end
emu.log("Palette " .. pal .. " colors: " .. table.concat(colors, ", "))
```

This snippet calculates the CGRAM address for the chosen sprite palette and reads 16 color entries (each 2 bytes). We convert the 15-bit BGR values into 0–255 RGB components (by multiplying each 5-bit channel by 8, which is a common approximation for SNES colors). The result is a list of the actual RGB colors the sprite uses.

Understanding Sprite Tile Mapping in Kirby Super Star

To successfully map runtime sprite data to ROM, it’s essential to know how the game organizes sprite graphics:

- **VRAM Tile Arrangements:** The SNES can only display tiles that have been loaded into VRAM. Kirby Super Star dynamically loads sprite graphics into VRAM as needed (often during transitions or when new enemies appear). The `$2101` OBSEL register’s *Base* and *Name Select* fields determine where sprite tile numbers point within VRAM [10] [8]. In KSS, typically the sprite VRAM base might be set to use a portion of VRAM for sprite tiles. The **tile index** from OAM is an offset within that region. For example, if OBSEL’s base is 0, sprite tile 0 refers to VRAM $0000; if base is 2, it refers to VRAM $8000 (since base index 2 = 0x4000 * 2) [6] [8]. The Name Select adds a second block offset for tile indices 0x100–0x1FF (i.e. it selects whether the second 256-tile bank is contiguous or offset) [10]. In practice, this means you might see tile numbers >=256 for certain large sprites or animations – those get an additional VRAM offset defined by Name Select.

- **Compression and SA-1:** Kirby Super Star leverages the SA-1 coprocessor for fast decompression of graphics [19]. Sprite graphics (for Kirby, enemies, etc.) are stored compressed in the ROM and decoded to VRAM on the fly. For instance, when Kirby uses a new ability or an enemy spawns, the game may fetch the compressed sprite data from ROM into SA-1 pack RAM or Work RAM, decode it, then DMA the raw tiles into VRAM. Knowing this, our script’s output of `tile_addr` in VRAM can be correlated with when and what data was written there. Using the Mesen-S script, you might observe that at the moment an enemy appears, a range of VRAM addresses gets populated with new tile data – that’s a clue to where the compressed graphics came from. By setting breakpoints on SA-1 memory or tracing the DMA source, you can pinpoint the ROM location (and then use a tool like **exhal** to decompress it outside the emulator if needed).

- **Disassembly and documentation:** For deeper research, you can refer to the **Kirby Super Star disassembly** project [17] and community documentation. These resources might have identified the format of sprite mappings or even symbol names for Kirby's graphics. For example, a "CharMaps" file in the disassembly suggests mappings of character (sprite) IDs to graphic data. Such documentation can greatly assist in linking an OAM tile index to a ROM asset. If available, also check if the disassembly labels the VRAM buffering routines or sprite table locations.

Putting It All Together – Script Workflow

1. **Setup Frame Hook:** Use `emu.addEventCallback` with `endFrame` (or `startFrame`) to run your code every frame [11]. This ensures you capture dynamic changes in sprite data.

2. **Read OAM Entries:** Within the callback, loop through OAM. Identify the entries corresponding to Kirby or the enemies of interest. (Kirby might occupy specific OAM slots; you could also filter by certain tile indices or positions that correspond to Kirby).

3. **Calculate VRAM Tile Address:** Read the OBSEL register ($2101) to get base index and name select [10]. Compute the VRAM address for each sprite's tile number. Each 8×8 sprite tile is 32 bytes (4bpp), and OBSEL's base gives the 16KB block of VRAM where tile index 0 starts [8]. If the tile index ≥256, add the name-select offset of (NameSel+1)*8KB for the second tile bank [8].

4. **Read VRAM Data & Map to ROM:** Fetch the 32-byte tile data from `snesVideoRam`. Attempt to find it in the ROM (`snesPrgRom`) if you suspect it's uncompressed. More likely, log when VRAM is written and which routine is running – this will hint at the ROM offset. You can also dump the VRAM tile data to a file for offline analysis (e.g., compare with known graphics in a tile editor).

5. **Read Palette (CGRAM):** Use the sprite's palette number to fetch the 16-color palette from `snesCgRam`. This gives you the actual colors. This is useful if, for example, you want to verify you're using the correct palette data from the ROM (palettes might also be stored in the ROM in uncompressed form even if tiles are compressed).

6. **Utilize Documentation:** Cross-reference the findings with known ROM maps or use tools like **KompreSS** or **inhal** to decompress graphics from the ROM [18]. If you know Kirby's sprite graphics are at a certain ROM address (from documentation), you can set a memory read breakpoint in Mesen-S at that address to see when the game reads those bytes – confirming the linkage between the ROM and the VRAM content.

References and Further Resources

- *Mesen-S Lua API Documentation:* The Mesen 2.0 API reference (which covers Mesen-S) details memory types and callback functions [1] [15]. Notably, `snesSpriteRam`, `snesVideoRam`, and `snesCgRam` correspond to OAM, VRAM, and CGRAM respectively [2]. The API also supports memory callbacks for reads/writes [15] and frame events [4].

- *SNES Sprite Hardware (OBJSEL Register):* Understanding $2101 is crucial. The SNESdev Wiki explains the `OBSEL` bits: sprite size, VRAM base (bBB bits), and name select (N bits) [7]. This tells how OAM tile indices map into VRAM addresses [10] [8].

- *SNES VRAM/Palette:* GeorgJZ's SNES tutorial provides a gentle introduction to SNES VRAM, CGRAM, and OAM [20] [5]. It notes that CGRAM holds actual color data and OAM tells the PPU "where to draw which sprite" [21] [9] – exactly what we're leveraging in the script.

- *Kirby Super Star specifics:* According to Data Crystal, KSS uses a custom compression (documented by Parasyte) and is an SA-1 game [22] [12]. Tools like **exhal/inhal** on ROMHacking.net can decompress KSS graphics [18]. The existence of an SA-1 indicates that extra processing was used for tasks like graphics decompression [19]. A community disassembly project is ongoing [17], which can be a goldmine for exact ROM addresses of assets and routines.

Using the above approach, you should be able to write a Lua script in Mesen-S that reports Kirby's and enemies' sprite IDs each frame, the corresponding VRAM tile data, and the palette colors. While direct mapping to ROM offsets is straightforward if data is uncompressed, in Kirby Super Star's case you will likely integrate this with further detective work (breakpoints and external tools) to trace the compressed data back to the ROM. The combination of **real-time OAM/VRAM inspection** and **static ROM analysis** will allow you to map in-game sprite data to the ROM offsets and palette info that you need. [3] [9]

---

[1] [2] [4] [11] [15] [16] Mesen Lua API reference_Mesen Lua 函数库 - Pastebin.com
https://pastebin.com/z9w7Dj5Q

[3] Lua Scripting (Mesen) - Page 2 - nesdev.org
https://forums.nesdev.org/viewtopic.php?t=16399&start=15

[5] [9] [20] [21] SNES Assembly Adventure 04: Display Your First Sprite – Machine Code Construction Yard
https://georgjz.github.io/snesaa04/

[6] [7] [8] [10] PPU registers - SNESdev Wiki
https://snes.nesdev.org/wiki/PPU_registers

[12] [18] [22] Kirby Super Star - Data Crystal
https://datacrystal.tcrf.net/wiki/Kirby_Super_Star

[13] HAL Laboratory compression - News
https://romhack.github.io/doc/halCompression/

[14] compression algorithm on SA-1 games - NESDev Forum
https://forums.nesdev.org/viewtopic.php?t=11602

[17] GitHub - Ankouno/KSS-disassembly: A repository dedicated to disassembling and documenting Kirby Super Star.
https://github.com/Ankouno/KSS-disassembly

[19] Quality Test - Kirby Super Star (SNES) - SDA Forum
https://forum.speeddemosarchive.com/post/quality_test__kirby_super_star_snes2.html