Inside CoCoMan on the Color Computer 3

Replacing a PPU with Generated Code, Banked Bitmaps, and a Decoupled 6809 Render Pipeline

Brian K. Holman
BS, Computer Science, Brigham Young University
MBA, Marriott School of Management, Brigham Young University
me@brianholman.com
VCC Plus Technical Papers · November 20, 2026

Abstract

I built CoCoMan as an eight-world, 32-course action platformer for a stock-speed 512 KB Color Computer 3. The Color Computer and the Nintendo Entertainment System ran nominally similar 1.79 MHz 8-bit processors, but they did not divide graphical work the same way. The NES Picture Processing Unit fetched tiles, applied scrolling, composed sprites, resolved priority and generated the raster while the CPU supplied game rules and small updates. The CoCo 3's GIME scanned a bitmap, supplied a palette and supported display offsets. It did not supply a sprite list, tile fetcher, collision hardware or the same fixed pipeline. I therefore had to replace that work with a software renderer running on the same 6809 that simulated the game. [1, 2]

I built the game around that problem. I separated field-rate input and rules from variable-rate presentation, gave each drawing page its own sprite backups and terrain history, compiled repeated drawing work into 6809 routines, and paged the campaign across four disks. The game does not reproduce the NES PPU in software, nor does it reproduce NES frame rates everywhere. It replaces the PPU's division of labor with a different one. Input and rules advance every vertical field; pictures are committed only when a complete drawing page is ready. The simulation can therefore stay ordered while the renderer falls behind. In three measured passages, Release Build #3 presented 11, 11 and 21 pictures per second, with longest complete holds of 183, 117 and 50 ms. A joystick press became a visible lift after nine NTSC fields, or 150 ms. Those measurements show a playable schedule under named loads, not a game-wide frame rate.

The result I set out to achieve is narrower and more useful than saying the CoCo 3 matched a console built around a dedicated video processor. I built an original campaign with suits, swimming, assisted flight, moving machinery, projectiles, backtracking and side rooms by making presentation a scheduled software subsystem. The difficult part for me was keeping that subsystem inside the 6809 cycle budget and the Color Computer's banked-memory budget while preserving the timing and visible consequences of play.

1. Introduction: replacing a workload, not just a chip

The NES and Color Computer 3 invite a misleading specification comparison. Both machines ran at about 1.79 MHz in their NTSC configurations. The CoCo 3 could address far more memory, display a sharper bitmap and use a 68B09E with 16-bit registers and a multiply instruction. On paper, it can sound as though the home computer should have the easier time with an action game. The comparison breaks down because the two machines gave their CPUs different jobs.

The NES PPU is best understood as a workload engine. Its pattern and nametable memory hold tile graphics and a tile map. During each visible scanline the PPU fetches background tiles, applies horizontal and vertical scrolling, evaluates the sprite list and composes pixels according to sprite priority. The program controls those structures through registers and uploads a compact sprite list through OAM. The PPU still imposes limits, including 64 OAM entries and eight sprites on a scanline, but it performs the repeated per-tile and per-sprite work automatically. [2, 3, 4]

The CoCo 3's GIME is a different contract. It converts a region of system RAM into a bitmap and scans that bitmap through a palette. It provides display offsets that I used for smooth camera movement. It does not know what a game tile is, which pixels belong to a character or which object should appear in front of another. Every changed tile and every moving sprite must be written into the bitmap by the 6809. The GIME is essential to the result, but its role is closer to a display controller attached to a software compositor than to the NES PPU.

That distinction is the subject of this paper. I wrote CoCoMan to make an action platformer run on that bitmap-only path. The design question was where the game spent its scarce cycles, how I could fit a 512 KB machine into a 64 KB address space, and what the measured results would and would not establish. The answer is not that the 6809 out-draws the PPU. The answer is that I changed when and how drawing happens so the player can continue to act while the renderer works.

2. Design and verification

CoCoMan has its own characters, art, courses and course logic. This is a technical comparison of platform-game techniques, not a claim of hardware equivalence. The game page and player's manual identify the edition and controls described here. [5, 6]

I built the game in about 36,000 lines across 250 C, header and assembly files. My display implementation is about 4,200 lines and owns the display pipeline, tile stamping, sprite transfer, page restoration, mechanisms and presentation handoff. My shared rules implementation is about 3,200 lines and owns movement, collisions, supplies, suits, patrols, checkpoints and course transitions. I divided supporting modules by bank, and the build and link checks enforce fixed address and size boundaries.

I organize the account in three layers. First, I describe the mechanism I built and the limits it was designed to satisfy. Second, I report Release executions that establish cadence and input-to-visible-change for named passages on a stock-speed emulated MC6809 with 512 KB and Disk Extended Color BASIC. Third, I mark what was not tested. I keep design, emulated execution and unmeasured physical-hardware behavior distinct throughout.

I built the normal edition with CMOC `-O2`, LWTools and the VCC Plus bundled toolchain. The program links at `$1000` with a `$7C00` resident ceiling and a 2,048-byte stack allocation. Release and Debug are separate configurations. Build `#3` identifies Release disks with edition `9c1c5fc457a84323` and Debug disks with edition `6860ea15a4a0eb7b`; the player measurements in this paper use Release.

The design also reflects a constraint that matters to the paper's claim. A screenshot can show that one frame eventually looked correct. It cannot show that the other drawing page retained valid scenery, that a quick button press survived a long redraw, or that collision and sound ordering continued while presentation was held. I designed and verified an interaction through the interrupt queue, the rules step and the page handoff rather than treating the renderer as a standalone image generator.

3. Two machines, two divisions of graphical labor

The useful comparison is not a contest between two clock ratings. It is a comparison between fixed-function video work and a bitmap path made programmable by the CPU.

Attribute	NES	Color Computer 3 used by CoCoMan
Processor	Ricoh 2A03, 6502 core, about 1.79 MHz NTSC	MC6809E, about 1.79 MHz in high-speed mode
CPU-visible memory	2 KB work RAM plus cartridge memory	512 KB physical RAM, 64 KB addressed at once through eight 8 KB MMU windows
Video memory	2 KB PPU VRAM, 256-byte OAM, separate pattern and nametable storage	Shared with system RAM; two drawing surfaces occupy different physical banks
Playfield construction	Tile fetch from nametables and pattern tables	GIME scans a packed bitmap; the 6809 paints 16 x 16 tile pictures into it
Moving objects	Up to 64 OAM entries, eight sprites on a scanline	No hardware sprite list; software saves, blends and restores rectangles on each page
Scroll support	PPU-controlled scrolling of tile maps	GIME horizontal/vertical display offsets over a wider software-maintained bitmap
Typical CoCoMan display	Not applicable	320 visible pixels from a 512-pixel-wide surface, 16 colors, two pixels per byte

The table explains why the CoCo's larger memory does not automatically make it the easier game machine. The NES spends transistors on the repeated video work. The CoCo spends RAM and CPU time on a framebuffer while GIME handles scanout. The PPU can be told that a tile at nametable position 12 should use pattern 37.

I had to make the renderer find the corresponding 16 x 16 picture, select its palette pixels, write those pixels into a bitmap at the correct stride and later restore the background when the picture moves. [1, 2]

The NES also has specialized object metadata that CoCoMan does not. OAM stores positions, tile numbers and attributes for sprites in a compact list. I stored a saved rectangle for each visible actor: its page column and row, width, height and pointer to the background bytes it replaced. My renderer paints current poses over the bitmap with transparent pixels left untouched. The absence of OAM removes a fixed 64-entry ceiling, but it replaces that ceiling with a cycle budget. More actors can be visible, but each actor consumes background-copy, mask and redraw time. [3]

The same inversion appears in scrolling. The GIME's horizontal offset is genuinely useful: it moves the scan window across a wider bitmap without the CPU copying an entire screen for every camera step. It does not make the world an automatically scrolling tile map. I used a 512-pixel-wide surface under a 320-pixel viewport, divided the world into 16-pixel columns and kept a 32-column ring cache for each drawing page. When the camera advances, only the newly exposed columns are painted. When a tile changes, the corresponding dirty cell is repaired. The hardware offset supplies the motion; software supplies the content that enters the view. [1]

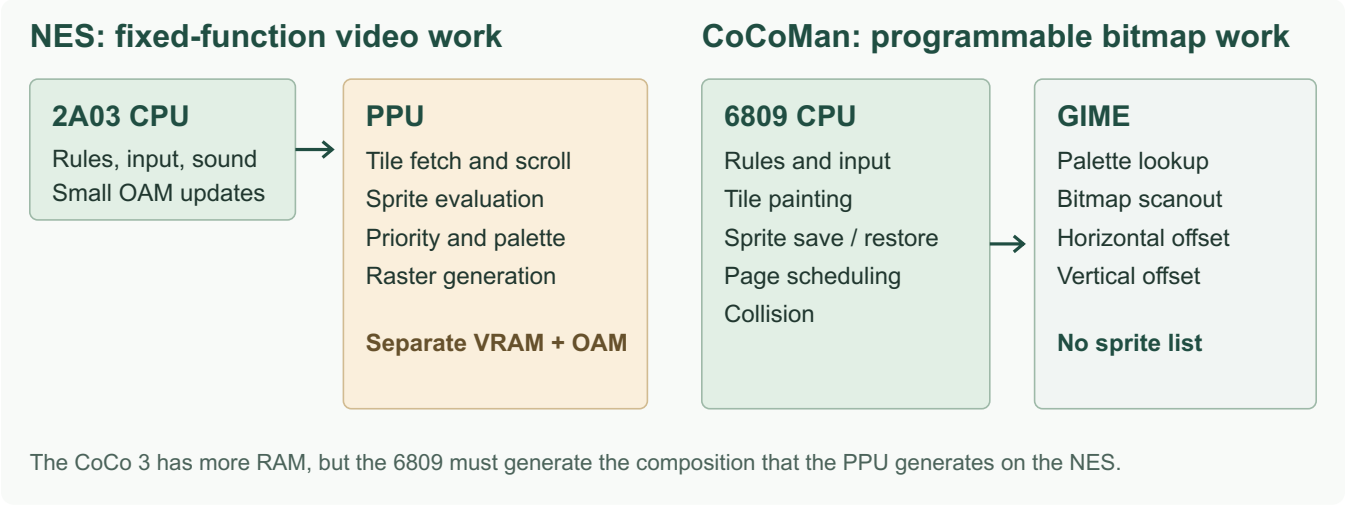


Figure 1. The NES gives repeated tile, sprite and priority work to the PPU. CoCoMan's GIME scans a bitmap, so the same 6809 that advances collisions and platform rules also maintains tiles, sprites, page history and banked artwork.

I chose a display mode that favors game readability rather than the CoCo's maximum resolution. In the ordinary screen, the GIME shows 16 colors from its 64-color palette. Two packed pixels occupy each byte. A visible 320-pixel line therefore contains 160 bytes, while the drawing surface advances 256 bytes between rows so the camera has another 192 pixels of surface on each side. This packed format is efficient, but it is still a framebuffer: every tile stamp and sprite pixel is a CPU store. The 16-color format reduces the bytes relative to a one-byte-per-pixel representation, not the number of decisions the renderer must make.

Some courses use the GIME's 225-line mode, with the same 256-byte row pitch and 16-color packing. A full-height drawing page then occupies roughly 57,600 bytes, more than seven 8 KB banks; two pages use banks `$00-$07` and `$10-$17`. The ordinary 192-line pages, sprite backups and art use the upper bank region. The display mode is therefore only one part of a larger residency design. A wider or taller bitmap increases the number of physical banks that must remain valid while the CPU renders and GIME scans.

This is where the PPU comparison becomes concrete. The NES did not merely have a video chip; it had an architecture in which the CPU could delegate the per-scanline visual workload. I could not delegate that workload to GIME. I had to reduce it, schedule it around the simulation, prepare repeated work in advance and keep enough page-specific state that the renderer never mistakes a background saved at one camera position for the scenery at another. I organized the rest of the implementation around those four requirements.

4. A renderer built around time, not around a frame

My most important design decision was that a displayed picture would not be the unit of game time. I decoupled the field-rate simulation from the variable-rate presentation. This is how a software renderer can miss a drawing deadline without flattening a quick jump into a later held button state or changing collision order.

At 60 NTSC fields per second, a 1.79 MHz 6809 has roughly 29,800 processor cycles per field before accounting for interrupts, audio, input and rendering. A full-screen bitmap update is much too large to treat as an unconditional per-field task. I instead reserved a small, persistent input interface. The ordinary Disk BASIC vertical interrupt samples the keyboard and right joystick and writes each processed sample into a 128-byte circular queue at `$0F00`. Each held-control sample is paired with a newly pressed play-aid byte in a parallel table. Producer and consumer indices stay in the signed-A addressing range of the 6809, and I explicitly accounted for repaints that can span more than 63 input samples. [1]

My foreground loop drains those samples in order. For every sample it services audio, calls `game_step`, dispatches the resulting sound cues and handles any course transition. It asks the video subsystem for a new picture only when `video_ready()` reports that the previous completed page has been published. If the renderer is still working, the loop continues consuming field samples and advancing the game. The player's jump press, collision order, game clock and action cues therefore retain their field-level sequence even when several simulation updates contribute to one visible picture.

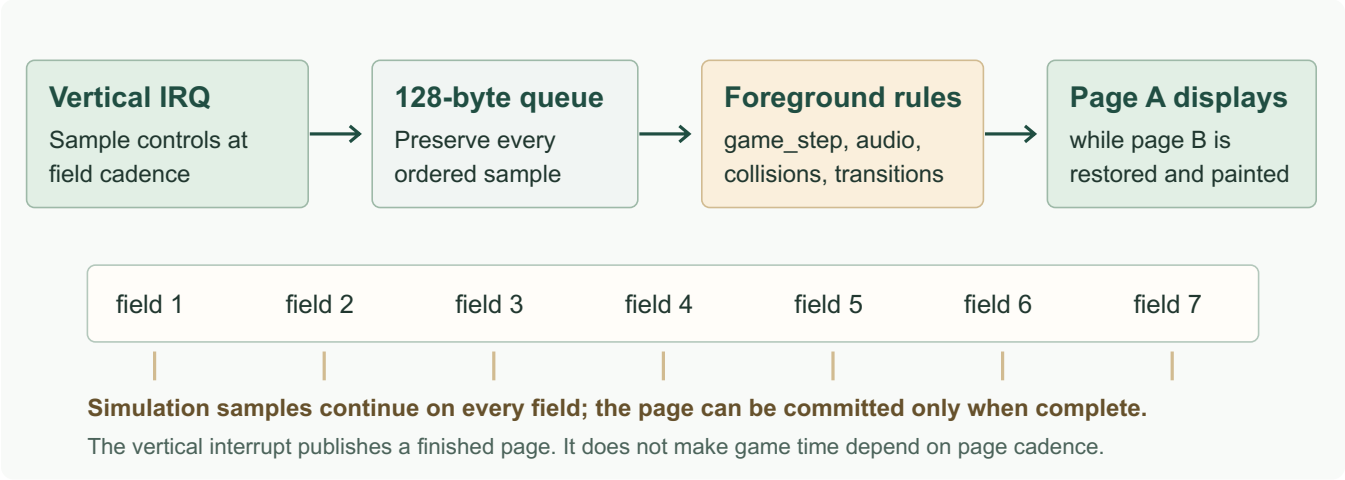


Figure 2. Input and rules run at field cadence while rendering runs at page cadence. The interrupt keeps sampling during a long draw; the foreground drains ordered samples and advances the game even when no new picture is ready. A completed page is committed at vertical interrupt.

This is not an attempt to disguise a slow renderer as a 60-frame game. It is an explicit contract about which parts of the experience must remain continuous. The player can release a button during a long redraw without the release disappearing. A projectile can move and contact an actor according to the sampled field that produced it. The audio scheduler can react to an event before the corresponding object is visible. The presentation can lag, but it cannot rewrite the order of play that produced the next page.

I used two complete drawing surfaces. While GIME scans page A, the 6809 erases old object footprints and paints the next state into page B. When page B is complete, `pending_base`, `pending_pan` and `pending_camera` describe the next page and viewport. The vertical interrupt publishes those values by changing the GIME display registers. A one-field hold flag prevents the same page from being republished immediately and gives the foreground at least one complete field interval between page commits. The display surface and its camera origin then move together, rather than combining a new address with an old horizontal offset.

I gave each `Surface` state that cannot safely be shared between pages: its 32-column ring history, current top, display bank, terrain revision, room identity, link state, exchange state, HUD revision, visible-left and visible-right bounds, enemy count, camera and saved rectangles for every visible actor. I stored the two surfaces in otherwise idle low-memory regions of Disk BASIC. Their terrain history and sprite backups live in physical banks, while the compact page descriptors remain addressable by the resident renderer. I therefore did not treat a screenshot as a complete renderer test: the other page may hold a different camera, different saved backgrounds and different terrain history.

I also carried the HUD state through the page commit. The renderer can restore and repaint moving objects while keeping the score, lives, suit and notices consistent with the page being presented. If the camera has moved, the page records its new left and right terrain extents. If a course is re-entered, a teleport crosses rooms, a terrain revision changes or a new room invalidates the cache, the page can be retired and rebuilt rather than trusting a stale ring slot that happens to contain the same tile number.

The result is a three-stage schedule:

1. The interrupt samples controls and advances bounded audio work at field rate.
2. The foreground consumes ordered input samples and advances rules, collisions, animation and course transitions.
3. The renderer prepares a complete page in its own time and publishes it only at an eligible vertical interrupt.

The schedule is the software substitute for the PPU's fixed relationship between CPU updates and raster output. The PPU could accept a small OAM update and perform the composition itself. My renderer instead accepts a queue of input and game-state changes and composes a full bitmap when the CPU can finish it. The price is variable picture cadence. The benefit is that game time does not stop when the renderer is busy.

This separation also explains the measured input delay. A joystick press can be accepted at a field boundary while the previous page is still being prepared. The simulation can begin the jump immediately, but the first picture showing the lift cannot appear until the new page has been completed and committed. The delay is therefore not simply the time to execute the jump rule. It includes the renderer's queue, the remaining work on the current page and the next eligible vertical interrupt. I could not omit this distinction: a renderer with a good average frame rate can still fail the interaction the player is trying to read.

5. Paying for pixels: how the PPU's work became software

I did not write a generic bitmap library with actors layered on top. I wrote the renderer as a collection of decisions about where repeated work could be removed from the hot path. I eliminated some work with caching, compiled some before play, and handed repeated stage patterns to specialized assembly loops. The common theme is that the 6809 should not make the same decision twice when the course's authored structure makes the result predictable.

5.1 Sprites without an object processor

The NES receives a compact OAM list and composites sprites during scanout. CoCoMan has no object processor. My renderer keeps a `SavedSprite` for each visible object: a page column, row, packed width, height and pointer to the background bytes that the object will cover. To paint an object, I first copy the untouched bitmap bytes into that object's page-specific backup and then blend the sprite's colored nibbles over the canvas. Transparent nibble values leave the background alone. Erasing a later object restores its saved bytes.

The order matters because objects overlap. If a pilot stands in front of a deck and a mechanism passes behind the pilot, the later object's saved rectangle may include pixels already changed by the earlier object. Erasing the earlier object first and restoring the later rectangle second would paint a fragment of the earlier object back into the scene. I therefore store overlapping objects in drawing order and unwind the saved rectangles in reverse order on the page being prepared. The visible result is correct only when both the order and the page-specific backup are correct.

Each drawing page owns its own backup storage. The alternate page may have been drawn at a different camera position, so a rectangle saved on page A cannot be used to restore page B after a scroll. The ordinary page backup bank is `$1C`; the second page uses `$2E`. I deliberately gave each page descriptor its actors, camera, terrain history and HUD state together, because they are invalid as a set when the page is retired. When I tested the renderer, I alternated both pages because a trail or scenery repair on one page could be missed on the other.

I also had to make the sprite path respect interrupt latency. Sprite pieces are split at horizontal wrap boundaries and at an eight-scanline interrupt bound. My ordinary masked transfer uses paired bytes where possible, precomputes mask bytes, and keeps the timer FIRQ path available while restoring or painting a bounded piece. A large object is not one unbounded CPU loop; it is a sequence of pieces whose duration and bank crossing are controlled. This is the software equivalent of a PPU sprite budget, except the game chooses the budget from the rectangle sizes and the current page rather than from an OAM register format.

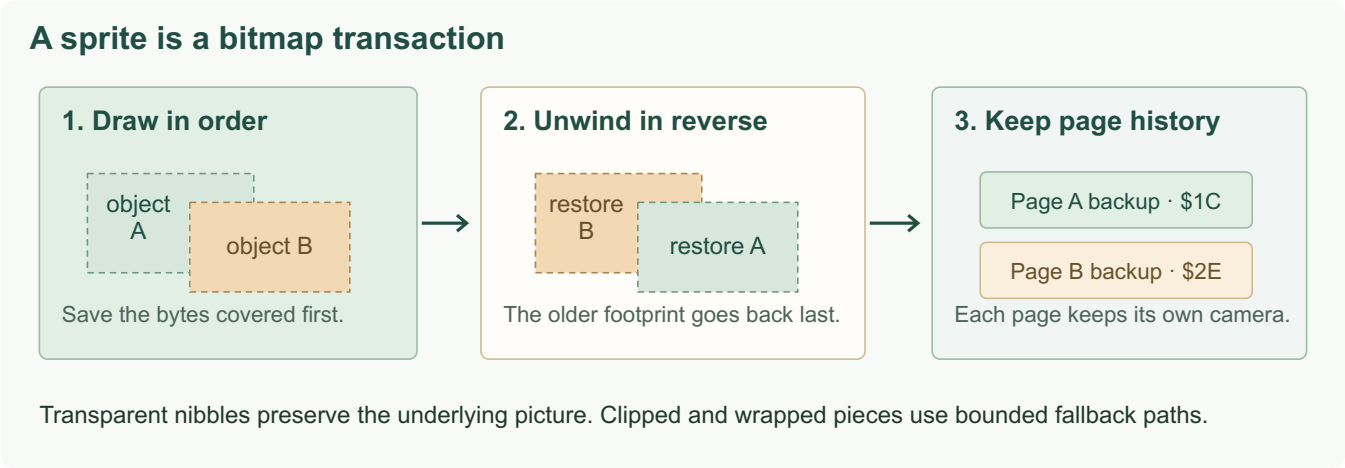


Figure 3. CoCoMan saves the background under each visible object, blends the current pose and later restores the page's own bytes in reverse order. Two pages therefore preserve two independent histories of the same world.

The absence of hardware sprite collision is equally important. Collision is resolved in world space, not by asking the video device whether two colors touched. The visible object and the collision object remain related but separate; an actor can be drawn with ornaments outside its contact rectangle, or clipped at the viewport edge, while the rules continue to use the authored body bounds. I could therefore make a sprite visually large without making its whole envelope hazardous, and keep collision stable across suit changes or camera movement.

5.2 A ring of world columns

I laid out the playfield as 16 x 16 tiles and kept a 32-column ring for each drawing page. A horizontal camera move at 320 pixels of viewport covers 20 tile columns; the ring gives the renderer enough history to compare the old camera interval with the new one. If the camera moves right, columns leaving the old view are already represented and only newly exposed columns need painting. If the camera reverses, the same cached columns can be reused if their tile content still matches. My renderer compares the old and new left/right bounds rather than repainting the entire viewport.

A tile is not a pointer to a reusable hardware pattern. I write its 16 scanlines into the bitmap at the 256-byte row stride. The course data supplies compressed tile dictionaries or stage-local pictures. For several courses I compile the dictionary into straight-line 6809 stamps at startup. A worst-case tile emits constant-color loads and 16-bit stores and occupies at most 381 bytes. The resulting routine still writes 16 rows, but it has already resolved the tile's sparse rows, repeated colors and byte offsets. My frame loop executes code rather than walking a general-purpose decoder for every tile.

I applied the same idea to dirty terrain in the opposite direction. A brick break, foil pickup or mechanism change records a dirty cell and a terrain revision. On the next page preparation, the renderer checks whether that cell is in the page's column ring and repaints the affected cell. If a room change or a large revision invalidates the cache, it retires the page and rebuilds both pages before play resumes. I wrote targeted paths for rooms whose scenery occupies a ring slot but is not implied by the collision ID alone. A tile cache that reused only by tile number would be smaller, but it would paint the wrong landmark when two distant rooms shared the same collision value.

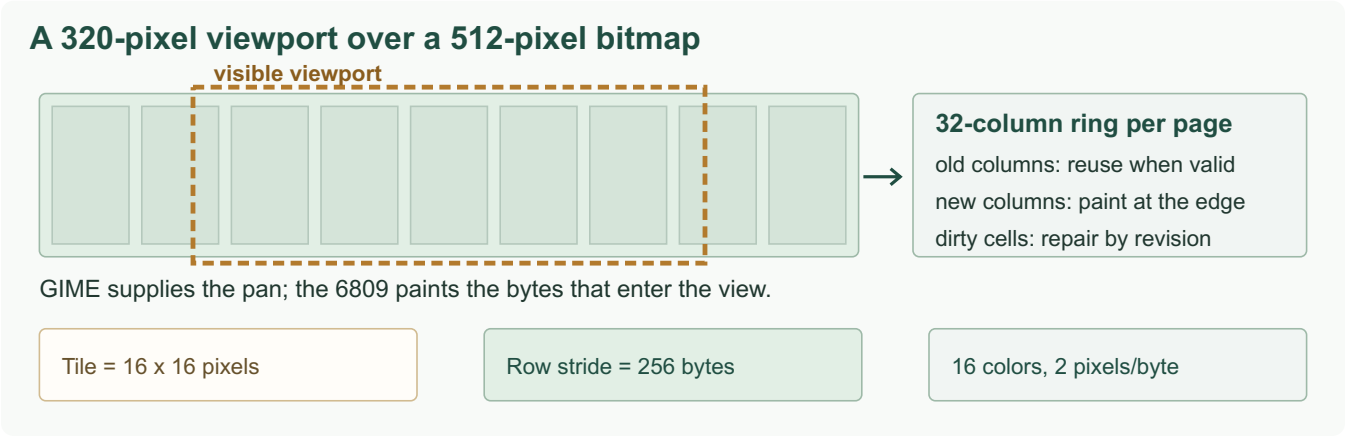


Figure 4. The renderer first restores each page's old footprints, then repairs dirty terrain and paints newly exposed columns. Only the changed edges of a pan require new terrain.

Full-height mode made this design more demanding. A 225-line page spans more than seven 8 KB banks. I had to know which page bank contains each row and which physical bank was currently mapped at the drawing aperture. The same logical address can refer to different physical memory after a bank switch, especially around stage overlays and sprite code. I use explicit bank and offset pairs for drawing operations, mask ordinary interrupts while changing the map, and restore BASIC's windows before the next interrupt can run. The visual consequence is a camera that can show the highest dish or platform while the 6809 continues to service a Disk BASIC environment that expects its ROM and RAM windows to exist.

5.3 Compiling the repeated cases

My most distinctive optimization was not a faster generic blitter. I compiled specific visual structure into 6809 code before the player needed it.

`prepare_cave_stamps()` reads the compressed cave tile dictionary and emits straight-line `LDD`/`STD` operations into banks `$08` through `$0A`. Constant colors remain in the D register across adjacent words and identical rows. A fully changing tile has a bounded code size; a repeating tile produces much less work. My frame loop sees a compact offset in a directory, maps the correct bank and calls the stamp. The atlas does not need to keep a second uncompressed copy in memory or reload pixels from disk for every repaint.

I made the sparse-sprite compiler go further. During room preparation it reads the packed pixels of a pose, hashes and deduplicates identical rows, emits a paint routine and an erase routine for each distinct row, and records the row list in metadata. Banks `$08` and `$09`/`$0A` hold the records and generated code; up to 64 poses can be prepared. The compiled metadata is accessed through `$A000`, the generated code through `$8000`, and the background still follows the renderer's existing save-and-restore convention. If the compiler cannot fit a pose, it abandons the prepared path and the renderer uses the ordinary transfer. Preparation is not allowed to silently overwrite a page.

I handled other stage-specific cases with handwritten batch kernels. The refinery, kiln, pump and bead drawing units enumerate several rotors or beads in one assembly frame. They calculate common origins and clipping once, prepare the saved rectangles, then call a dense paint or erase path for the repeated bead shape. These paths avoid a C function call and repeated argument setup for every small repeated object. Their comments make the safe cases explicit: an ordinary eight-row bead inside the horizontal ring can use a fast path; a clipped, wrapped or differently shaped object falls back to the generic sprite routine.

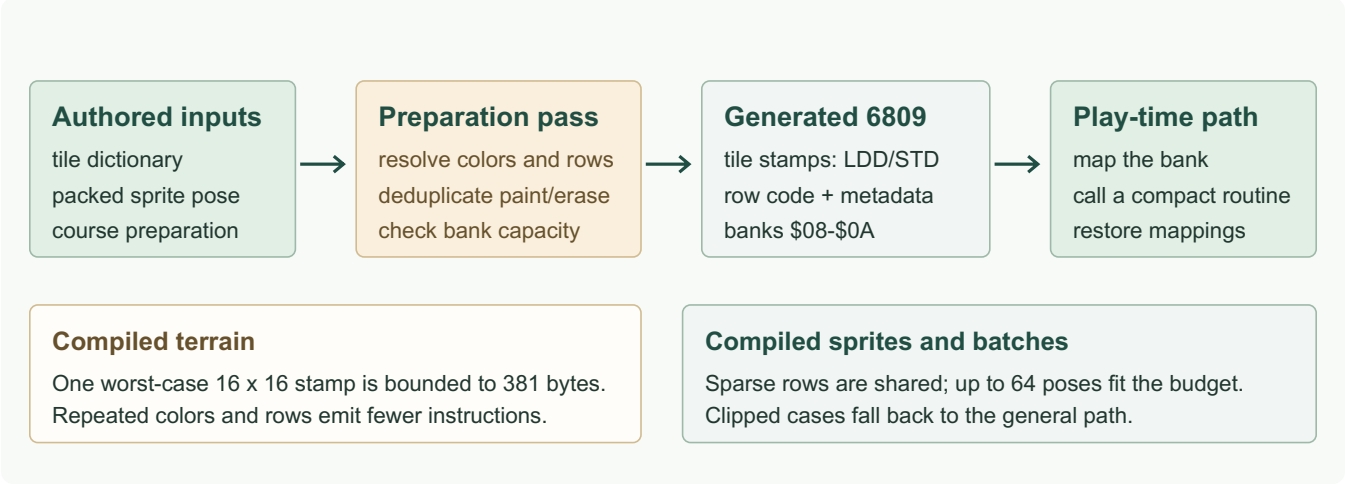


Figure 5. Authored art enters a preparation pass instead of a per-pixel interpreter. Tile stamps, sparse-sprite rows and stage-specific batch kernels become bounded 6809 instructions before play, and the runtime dispatches to the right banked routine.

The cost of this strategy is complexity in the build and in the fallback boundary. A compiled routine must fit its bank, preserve the caller's mapping, leave the timer FIRQ available and agree with the ordinary path on clipping and transparency. The payoff is that the expensive cases are concentrated in preparation while the visible interaction uses compact code. I therefore built the renderer as a small code generator and linker, not only as a collection of blitters.

I also made composition part of the optimization. I used palette zero as transparency and kept the pilot's contact anchor stable across suit changes. I gave moving decks a continuous bright top so a landing edge was visually separable from the background. I used open masks for wheels, forks and rotors so negative space did not become an expensive full rectangle. These are not merely art preferences. They reduce the number of pixels that must be saved and restored and make the visible result legible at the lower picture cadence the renderer can afford.

5.4 What the software renderer does not do

I did not give the renderer a NES-style nametable, OAM shadow list or hardware pattern fetch. It does not assume that a tile number identifies a pixel pattern across pages, redraw the complete world each animation tick, use the GIME offset as a substitute for updating newly exposed tile columns, or promise a new bitmap every vertical field.

Those omissions define the engineering claim. I preserved the observable action-game contract through a different implementation: ordered input and simulation, camera-following terrain, page-local background histories, generated drawing code and measured page commits. The visual result can be slower than a PPU-driven game, but the game rules and the player's ability to read and act on the scene remain the design target.

6. Fitting a campaign into a 64 KB window

A 512 KB Color Computer 3 does not give the 6809 512 KB of simultaneous address space. The CPU still sees 64 KB. Eight 8 KB windows are mapped into 64 physical banks, and a write at a logical address has meaning only in the context of the mapping active at that instant. I designed the campaign with the same constraint as the renderer: I had to keep common rules and interrupts addressable while bringing authored courses, mechanism code and artwork through a small number of windows.

The resident program links at `$1000` with a `$7C00` ceiling. The Release map leaves 170 bytes below that ceiling and the packed boot image has a seven-byte margin. Debug leaves 94 resident bytes and 26 packed bytes. I did not have generous margins for a convenient abstraction. They forced fixed addressing, compact modules, bank-aware call gates and compile-time assertions that fail the build when a generated routine crosses a bank boundary.

The ordinary campaign is four 35-track, 18-sector disks of 256-byte sectors, or 161,280 bytes each. There are 32 courses served by 31 stage packs because 3-1 and 3-2 share one terrain owner. A stage pack is bounded to 2,304 bytes including its 12-byte header and expands to 8,160 bytes. The four images divide the route by

world rather than by a generic file cache: the boot disk carries World 1 and the writable flight, the first data disk carries Worlds 2-5, the second World 6 and the third Worlds 7-8. A spare granule on a later disk cannot absorb an oversized file on an earlier volume.

My loader does not casually map a file into RAM and begin using it. It finds the file and its allocation chain, checks the sector extent, streams sectors through a flight workspace and calculates a checksum while copying. It maps the destination bank into the `$8000` aperture only for a bounded copy and restores BASIC's previous mapping before an interrupt or disk call can enter the Disk BASIC chain. A file can share a bank with another owner, so the affected cache identity is retired before a partial replacement begins. Only after the complete length and checksum validate does the loader publish the new stage or overlay identity.

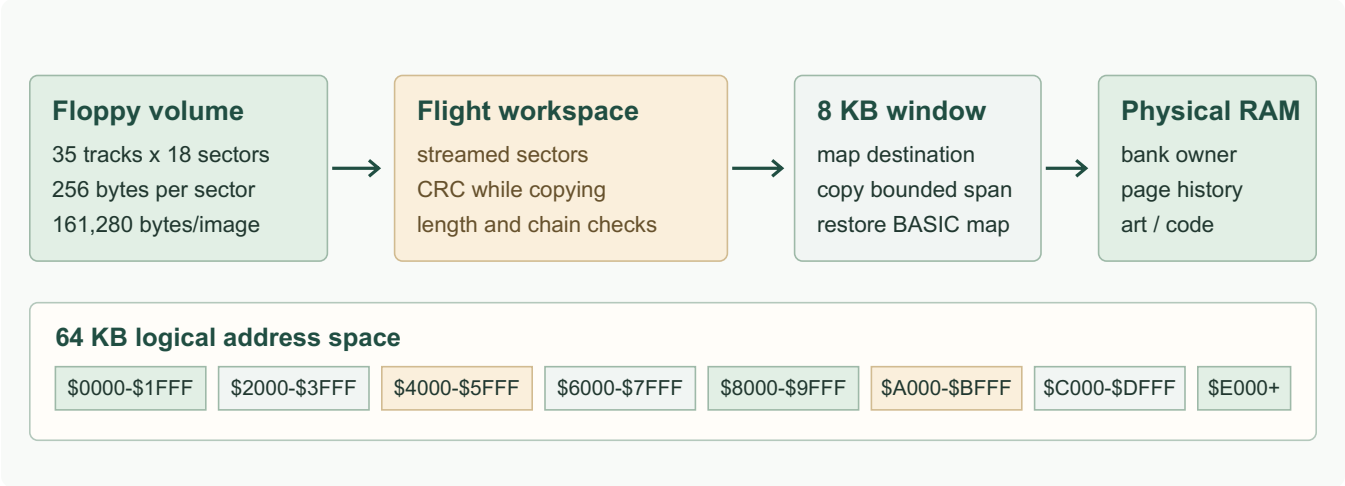


Figure 6. The game uses temporary MMU apertures for bounded copies, then restores Disk BASIC's windows before returning to interrupts. A stage or overlay is published only after its sector extent and checksum validate.

I divided resources by lifetime: some are resident, some are loaded once per course and some share a bank with a different owner. Common rules, the HUD and the interrupt gateway must remain available before the first stage. The active terrain bank may contain a compressed dictionary and scenery. Later course code can share a physical bank only if the loader authenticates the complete replacement and invalidates any page history that depended on the old owner. Artwork and mechanism-specific code have their own windows. I encoded these boundaries in the implementation and linker checks because a logical address alone would be ambiguous in a debugger.

I also made the course catalog part of the rendering design. A room can contain a side room, a wet route and a return path that share terrain but keep distinct mutable state. My renderer needs enough room identity and revision information to know when a ring slot can be reused. The camera uses world space for collision and encounter activation, a logical camera value for route progress and a wrapped physical bitmap position for drawing. I deliberately kept those three coordinate systems separate. Conflating them produces either a visible camera jump, a collision mismatch or a wrong restore after a backtrack.

6.1 A campaign, not a single showcase level

I did not build a single showcase level. The campaign has eight worlds and four courses per world. Movement uses shared fixed-point rules: walking caps at 24/16 pixels per update, running at 40/16 and a grounded tall-suit crawl at 12/16. A fast takeoff selects an 80/16-pixel upward launch; slower takeoffs use a 64/16 launch, and releasing Jump selects a different gravity curve. The game preserves horizontal momentum in the air and snaps a descending crossed edge to the supported floor.

On top of that dry movement, I implemented pressure and Arc suit transformations, swimming strokes, spring compression, assisted jetpack thrust, scripted duct travel and moving decks. Encounters include stomped and kicked drums, flying forks that outlive their thrower, fixed darts, rotors, heat arms and patrols. Courses contain checkpoints, retries, side rooms, optional foil routes and backtracking. A save record stores the active room, suit, counters and course-specific moving state. These systems share the renderer but do not share a single visual primitive.

The PPU comparison therefore has two levels. At the low level, CoCoMan must draw every changed tile and sprite. At the course level, I had to keep the same software pipeline correct across many different rules, art

banks and room identities. A renderer that could draw one demo room would not establish the project's claim. The design therefore had to survive 32 playable course slots, 31 terrain packs, four media volumes, shared rules and per-course banked overlays.

I kept the save path complete for the same reason. A flight save writes four body sectors and a header into the alternate record slot, then rereads the record for verification. An unreadable slot aborts the write because it might contain the newest good generation. Preferences use an independent pair. This is not the central PPU argument, but I kept the game's storage design complete while the rendering pipeline was already using nearly every resident byte.

The memory story also explains why the game feels like a Color Computer production rather than a large-memory abstraction. The 512 KB machine is useful because the renderer can hold two canvases, generated code, sprite backups, terrain banks and overlays at once. It is still constrained because only 64 KB is addressable and because every mapping change can expose an interrupt to a banked assumption. Part of what I achieved was memory placement: the campaign fits because I gave each owner a declared lifetime and made the runtime refuse to publish an unauthenticated replacement.

7. Measured cadence and input response

The performance question has a tempting but wrong answer. If the renderer is software, it is natural to ask for one frame rate and compare it with an NES game. CoCoMan does not present a uniform rate, and the game's design does not depend on one. The meaningful measurement is a named passage: how often did the completed page change, how long was the longest complete gap and how long did a specific control action take to become visible?

I recorded the selected measurements on Release Build #3 in VCC Plus 3.7.1 build 67, with the stock-speed MC6809, 512 KB, Disk Extended Color BASIC and the normal four-disk set on disposable copies. The time base was 60 NTSC fields per second. I compared the displayed page at 61 consecutive field boundaries in each window: one initial observation and 60 opportunities for a change. I define the longest complete hold as the largest interval between adjacent changes inside the window, excluding partial intervals at the beginning and end. I used a page-identity method, not a claim that every visible pixel changed.

Passage and input	Guest fields	Presented pictures	Pictures/s	Longest complete hold
5-2, Right plus secondary and primary	10,810-10,870	11	11	11 fields, 183 ms
7-1, Right plus Z and X	8,177-8,237	11	11	7 fields, 117 ms
7-2, Right plus Down	8,065-8,125	21	21	3 fields, 50 ms

The numbers describe different demands. The 5-2 Rigger gantry combines moving decks, a pilot, a shell and a fork in a passage where the visible floor and takeoff point can be held constant. The 7-1 shore and 7-2 channel exercise different camera, terrain and actor loads. The 5-2 window and the 7-1 window both average 11 pictures per second, but the 5-2 window has a longer complete hold. That difference is exactly why the paper reports a longest hold beside an average rather than treating a frame count as a smoothness verdict.

The input test isolates a second question. In the 5-2 take, the pilot held Right and the secondary button. Primary was applied at the boundary between fields 10,818 and 10,819. The displayed helmet remained at screen row 192 at field 10,820, first rose at 10,827 and returned to row 192 at 10,838. The first visible rise therefore followed input application by nine guest fields, or 150 ms. I used the visible helmet and the floor edge, not merely the time at which the button entered the queue.

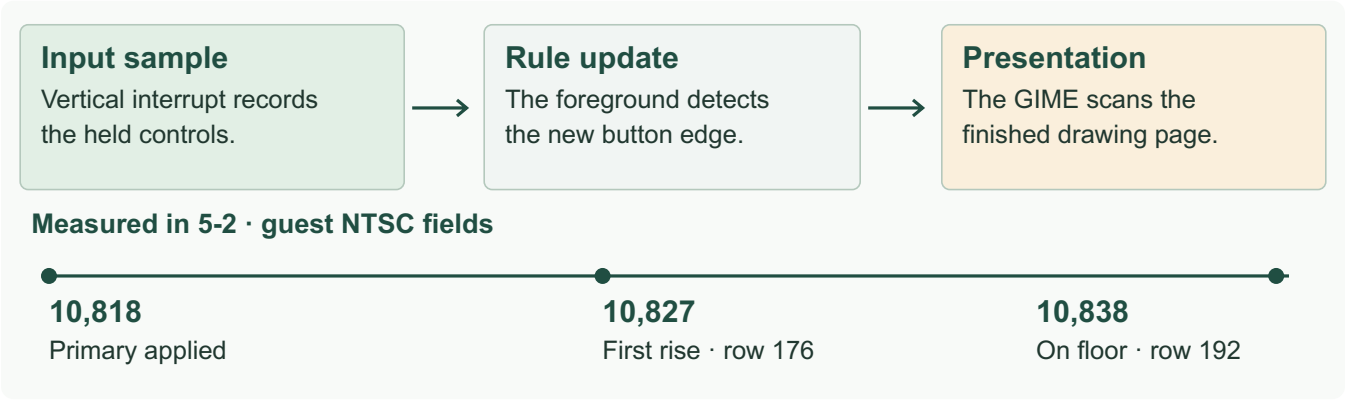


Figure 7. In the measured 5-2 take, the button was applied at field 10,818, the first visible rise appeared at 10,827 and the pilot returned to the floor at 10,838. The delay includes rendering and page publication, not only jump-rule execution.

At 60 fields per second, nine fields sounds like a long time for a button edge. It is shorter than the longest page hold in the same sample, which means the renderer was not waiting for a new jump animation to begin before accepting the input. The simulation could act on the press while the previous page was still pending. The visible rise appeared when a complete new page could be presented. This is the practical benefit of the input queue and the page handoff: a variable display cadence does not force the jump to be sampled at the display cadence.

The table must not be generalized into a game-wide rate. A named opening, a still room and a crowded mechanism gallery can all differ. It also does not establish physical controller latency. The host input path, joystick polling, monitor scanout and human response are outside the guest-field measurement. What the numbers do establish is that the Release edition advanced a complete bitmap at the stated times while the simulation continued to consume field samples, and that the measured jump became visible and landed at the recorded fields.

The measured rates also clarify the cost of replacing the PPU. A PPU-driven game can spend a large fraction of its video hardware's fixed budget on composition while the CPU remains mostly free for rules. CoCoMan's CPU performs both jobs. In the 5-2 sample, the player sees roughly one new picture every five or six fields on average and waits up to eleven fields between page changes. The named interaction remained readable because I did not tie controls, collisions or the audio schedule to those page changes. The visible world can update less often than the simulation while the player's action remains ordered.

That is the result I consider defensible: the CoCo 3 did not become NES-equivalent because its 6809 was capable or because 512 KB was large. It became capable of this campaign because I moved the PPU's responsibilities into a measured pipeline with explicit failure modes. The frame rate is part of the evidence, but the input-to-visible-rise result and the two-page restoration checks are what make the cadence meaningful.

8. What replacing the PPU actually required

The easiest way to describe CoCoMan on a Color Computer 3 is to say that the 6809 drew everything in software. That statement is true but incomplete. A software bitmap renderer that tried to compose a full screen, every actor and every tile in one field would not have produced this campaign. It works because I changed the work into several bounded problems and scheduled them against the field clock.

8.1 The PPU supplied a time budget, not only pixels

The NES CPU does not need to know how the PPU fetches a background tile, selects a sprite pattern or resolves priority. It updates a compact game state and lets a separate circuit perform the repeated work during scanout. The PPU is therefore an allocator of video time. The 6809 has no equivalent circuit. I had to decide how much work belonged to terrain, how much to moving objects, how much to presentation and when all of it could fit around rules and audio.

I replaced that allocator with measurable software bounds:

- the input queue is independent of the display page;
- game updates run per field while page presentation runs per completed page;
- terrain is painted only for exposed or dirty columns;
- sprites are bounded pieces with explicit save and restore order;

- repeated art is compiled into banked 6809 routines during preparation;
- page histories and bank mappings are retired when their ownership changes;
- the loader validates an entire replacement before publishing it.

Those bounds are not a generic engine specification. They reflect the resources of a specific machine and the structure of specific courses. A future engine that copied the vocabulary but not the scheduling would still spend too many cycles in the wrong place.

8.2 The 6809 is fast enough only when it is not asked to rediscover the screen

The 6809 has useful 16-bit operations and a multiply instruction. That helps fixed-point movement, coordinate transforms and compact gameplay rules. It does not make a per-pixel interpreter free. My renderer gains time by knowing what does not need to change: the camera exposes only a few new tile columns; a sparse sprite has many empty rows; a tile has repeated colors; a rotor has repeated beads; a page's old background is still valid until an object overlaps it.

I made the distinction between a generic algorithm and a prepared one explicit in the code. A generic tile loop would decode every tile each time. `prepare_cave_stamps()` emits a specific sequence of loads and stores. A generic sprite loop would test every nibble for transparency. `prepare_sprite_masks()` creates mask bytes and specialized transfer paths. A generic mechanism loop would call a C drawing function for every bead. I made the batch kernels calculate common origins and clipping once. Each optimization trades resident code, bank space or preparation time for a smaller hot path.

That trade is the practical answer to the paper's central difficulty. The Color Computer 3 has enough total RAM to hold the campaign's art and enough CPU speed to run rich rules. It does not have a video processor that will absorb the composition work, and it does not have an address space that lets all of the campaign remain resident. I had to be selective about both cycles and banks.

8.3 Decoupling is a correctness property

CoCoMan's variable-rate presentation is often described as a performance technique. It is also a correctness property. If the game tied collisions and button edges to page presentation, a long redraw could cause a tap to be lost or two collisions to be evaluated in a different order. My input queue and foreground loop preserve the sequence of field samples. My renderer can be late; the rules cannot be rearranged to fit the late picture.

The two-page renderer extends the same principle to visual state. Page A and page B are not interchangeable buffers containing the same pixels. I gave each a camera, terrain history, actor backup pointers and HUD state. Publishing page A while its background belonged to page B would create trails, scenery from the wrong column or a HUD value from an earlier state. My renderer repairs and repaints the page it is about to publish, and the interrupt changes the display and pan only after that page is complete.

That is why I verified both mechanism and timing. Inspecting the mechanism showed how the queue and page handoff worked, but not the observed hold. Timing showed the page changes, but not whether the other page or the quick input edge remained correct. I accepted the architecture only when the mechanism and the measured consequence agreed.

8.4 A different result from a console

It would be inaccurate to conclude that the Color Computer 3 is generally better at action games than the NES, or that CoCoMan runs at NES rates. The Color Computer's bitmap path gives the programmer freedom and a larger palette and memory budget, but it also makes every moving object a software transaction. The NES PPU gives less memory and stricter object limits, but it removes a large, repetitive class of work from the CPU. Each architecture makes a different part of a platformer easy.

What I achieved is not a hardware victory. It is a software result under a specific set of constraints: a stock 6809, 512 KB of banked RAM, four floppy images, a 16-color bitmap and an independently authored 32-course campaign. I preserved field-rate rules and readable interaction by building a rendering schedule the hardware did not provide. The difficulty for me was not simply writing more drawing code. It was making every repeated operation small enough, every page history explicit enough and every memory transition safe enough that the CPU could still spend time on the game.

References

Primary documentation, implementation descriptions and selected evidence identify source contracts and authored fixtures. The selected evidence files retain the scope of recorded executions and distinguish them from source inspection.

Reference 1

Tandy Corporation. Color Computer 3 NTSC/PAL Version Service Manual, catalog 26-3334, printed pages 11-16. GIME video control, horizontal offset and memory mapping.

Reference 2

NESdev Wiki. PPU programmer reference: pattern tables, nametables, OAM, rendering and sprite evaluation.

Reference 3

NESdev Wiki. PPU OAM: the 64-entry sprite memory and sprite attributes.

Reference 4

8bitworkshop. Nintendo Entertainment System platform reference.

Reference 5

Holman, Brian K. CoCoMan: To Titan and Back, game edition and overview.

Reference 6

Holman, Brian K. CoCoMan player's manual: controls, movement, supplies and disk guidance.