2012

I Am Error

Nathan Altice
Virginia Commonwealth University

Follow this and additional works at: https://scholarscompass.vcu.edu/etd
Part of the Interdisciplinary Arts and Media Commons

Downloaded from
https://scholarscompass.vcu.edu/etd/405

This Dissertation is brought to you for free and open access by the Graduate School at VCU Scholars Compass. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of VCU Scholars Compass. For more information, please contact libcompass@vcu.edu.
I Am Error

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University.

by
Nathan Daniel Altice
B.A. Sociology, University of Virginia, 2001
M.L.A., University of Richmond, 2005

Director: David Golumbia
Assistant Professor - Media, Art & Text

Virginia Commonwealth University
Richmond, Virginia
August 2012
Acknowledgement

I would like to thank several people that made this work possible. First, I want to thank my wife Amanda for keeping me sane, happy, and healthy during seven months of intense writing and research. I want to thank my chair, David Golumbia, for his support, guidance, and enthusiasm. Until he arrived at VCU, I did not think I could ever get a videogame-focused dissertation off the ground. I also want to thank the remainder of my committee—Joshua Eckhardt, Ryan Patton, Bob Paris—for their thoughtful revisions.

I wish to thank Neal Wyatt for her camaraderie, conversation, and coffee breaks throughout the writing process, Aria Tanner for her excellent (and affordable) Japanese translations, Vera Brown for her friendship (and Dendy), Matt Schneider for his long-distance reading, Scott Benson for his Game & Watch, and Drs. Steven Jones and George Thiruvathukal for first planting the seed for this project.

And finally, I want to thank Zac Price, a great friend and a superior Mega Man 2 player.

Thank you, everyone.
# Table of Contents

**Abstract** .............................................................................................................................................. v

**0: I Am Error** ..................................................................................................................................... 1  
  Methodology........................................................................................................................................ 4  
  *Game Over* ...................................................................................................................................... 7  
  A Note on Proper Names...................................................................................................................... 7  
  Organization....................................................................................................................................... 9

**1: The Family Computer** ............................................................................................................. 12  
  An Unconventional Stone.................................................................................................................... 14  
  Red, White, and Gold.......................................................................................................................... 18  
  Close Playing..................................................................................................................................... 24  
  Central and Picture Processing.......................................................................................................... 36  
  Setting the Tables.............................................................................................................................. 40  
  Of Rasters and Regions...................................................................................................................... 50  
  Sprite Nouns and Adjectives.............................................................................................................. 56  
  Scraped to the Die.............................................................................................................................. 60

**2: Ports** ............................................................................................................................................ 67  
  ‘Novelty Games’................................................................................................................................. 70  
  Bringing *Kong* Home......................................................................................................................... 80  
  The Ape in the Background............................................................................................................... 85  
  Girder Fonts and Missing Pies........................................................................................................... 92  
  Manic Compression............................................................................................................................ 96  
  Beyond Jumpman............................................................................................................................... 100

**3: The Toaster** ................................................................................................................................. 104  
  American Crash................................................................................................................................. 105  
  Advanced Video System..................................................................................................................... 107  
  Zero Insertion Force........................................................................................................................... 113  
  10NES Racket................................................................................................................................... 116  
  Operating Buddy............................................................................................................................... 121  
  ‘AFTER THE EYES FLASH.’ .......................................................................................................... 131
Abstract

I AM ERROR

By Nathan Daniel Altice, Ph.D. Media, Art + Text

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University.

Virginia Commonwealth University, 2012.

Director: David Golumbia
Assistant Professor - Media, Art & Text

I Am Error is a platform study of the Nintendo Family Computer (or Famicom), a videogame console first released in Japan in July 1983 and later exported to the rest of the world as the Nintendo Entertainment System (or NES). The book investigates the underlying computational architecture of the console and its effects on the creative works (e.g. videogames) produced for the platform.

I Am Error advances the concept of platform as a shifting configuration of hardware and software that extends even beyond its ‘native’ material construction. The book provides a deep technical understanding of how the platform was programmed and engineered, from code to silicon, including the design decisions that shaped both the expressive capabilities of the machine and the perception of videogames in general. The book also considers the platform beyond the console proper, including cartridges,
controllers, peripherals, packaging, marketing, licensing, and play environments. Likewise, it analyzes the NES’s extension and afterlife in emulation and hacking, birthing new genres of creative expression such as ROM hacks and tool-assisted speed runs.

*I Am Error* considers videogames and their platforms to be important objects of cultural expression, alongside cinema, dance, painting, theater and other media. It joins the discussion taking place in similar burgeoning disciplines—code studies, game studies, computational theory—that engage digital media with critical rigor and descriptive depth. But platform studies is not simply a technical discussion—it also keeps a keen eye on the cultural, social, and economic forces that influence videogames. No platform exists in a vacuum: circuits, code, and console alike are shaped by the currents of history, politics, economics, and culture—just as those currents are shaped in kind.
0: I AM ERROR

オレノナハ
エラー ダ・・・
- エラー, リンクの冒険

I AM ERROR
- Error, Zelda II: The Adventure of Link

My name is Error...
- Error, Zelda II: The Adventure of Link

Far from being an aberrant error, the glitch is a central part of the experience of the NES, an era where the games frequently existed on a spectrum between function and breakdown.
- Philip Sandifer, The Nintendo Project

During his quest to find the elusive Island Palace, Link, protagonist from the 1987 Nintendo Entertainment System (or NES) videogame Zelda II: The Adventure of Link, visits a small house in the Town of Ruto. When Link approaches its sole resident, a portly, bearded fellow in purple attire, the man declares, ‘I AM ERROR.’ Until Link speaks to another character further along in his quest, any interaction with Error yields the same curious result. Though the dialogue is simply a mistranslation of a programmer joke—a look-alike named Bagu, or ‘bug,’ makes this more apparent—the encounter became an infamous part of the NES’s cultural legacy. For years, the cryptic message was thought to be a programming flaw that replaced the character’s name with a diagnostic message. It was as if the game’s code mistakenly found its way to the graphical surface.

I Am Error is meant to evoke a number of material realities of the Nintendo Entertainment System—and its elder sibling, the Family Computer—that characterized its cultural reception, expressive output, and hardware design. The NES’s distinctive front-loading cartridge slot, for instance, partly caused the console’s infamous blinking screen, leading millions of players to blow into game cartridges as a quick ‘fix.’ Such game-disrupting hardware imperfections regularly spell commercial failure for consumer electronics, but players, developers, and software partners alike absorbed the Famicom’s/
NES’s flaws into the fabric of gaming culture. Hardware limitations that governed the complexity of graphics and the number of digitized sound channels—or worse, caused sprites to flicker or slowed on-screen action to a crawl—are now part of the living legacy of videogames, referenced by fans and repeated in game design. Contemporary games that aim for retro appeal still mimic the console’s shortcomings, since they provide quick visual cues to a past era of gaming.

The book title also alludes to the larger cultural and economic dialogue that took place between Nintendo, a Japanese company, and their newfound international audience. Nintendo has a long history, originating decades before the invention of videogames. But from their beginnings as a playing card company in 1889 to their stint as a toy manufacturer in the 1960s and 1970s, Nintendo has always had an interest in games. However, until the early 1980s, the company was largely unknown outside Japan. The success of their first handheld LCD games, called Game & Watch, and the unexpected phenomenon of an arcade game called Donkey Kong propelled them to international attention. Since then, the NES, Super Nintendo, Gameboy, Wii, Nintendo DS, and a host of landmark videogames have made Nintendo a household name equivalent to Disney or Apple. But this process was fraught with challenges. As they grew, Nintendo had to adapt to unfamiliar markets, transforming their success at home into success abroad.

The theme of this book is translation—not only in the linguistic sense, manifested in Error’s dialogue, but in a material sense as well: translation has real social, economic, and cultural consequences beyond simple misinterpretation; translation takes place between circuits, cartridges, code, and cathode rays just as it does between human actors; and translation is inexorably and inevitably riddled with errors. As Derrida wrote in his ‘Letter to a Japanese Friend,’ translation is not ‘a secondary and derived event in relation to an original language’—in other words, not merely a supplement. 

But this book aims to push Derrida’s notion further, beyond language. Applied to the production of technological objects that must enter cultures, markets, and domestic spaces, that must be made by bodies and touched by bodies, that must be made from rare earths and precious resources, translation does not simply derive meaning from prior sources—translation produces new meanings, new expressions, new bodies, and new objects as well. 

Nintendo’s software and hardware alike were subject to translations and mistranslations, from the design of the Family Computer console—radically altered before export—to the censorship of content thought objectionable to Western audiences. In mild cases like Zelda II, mistranslation was quirky and benign; in circumstances like the NES-exclusive lockout chip, meant to wall off piracy and unlicensed developers alike, mistranslation led to hardware malfunctions that plagued millions of consumers.

Translation also takes place in emulation, where certain allowable thresholds of error are often necessary to feasibly replicate one platform on another. The fidelity, speed, and
ease of translation through emulation can make or break the success of new platforms, especially those meant to lure customers away from an entrenched software ecosystem. And the translation of bodily movements—the sequence and timing of hands manipulating controllers—into text-based ‘movies,’ afforded by emulation, has led to new possibilities for videogame play.

It is hard to overstate the NES’s importance to both videogame history and culture at large—twenty-five years after its introduction, the NES remains an iconic console. For gamers and developers, part of the NES’s importance as a platform is its design legacy. Though the ‘platforming’ genre has fallen out of prominence in the wake of advanced 3D hardware, the emergence of low-overhead digital distribution networks (e.g. Steam, XBLA, PSN, Flash portal sites like Newgrounds) and mobile gaming, both viable avenues for small or single-person development teams, has spurred a resurgence of the genre. Platformers are both homage to the games developers played in their youth and economically-feasible alternatives to big budget 3D titles. The NES’s legacy also continues in emulation, both through grassroots homebrew efforts that port emulators to each new generation of consoles, PCs, and mobile devices and Nintendo’s own efforts to mine its back catalog, most recently via the Wii’s Virtual Console.

At the peak of their console dominance, the NES and Famicom were in one out of every three homes in the United States and Japan. In the late-1980s, Nintendo’s sales surpassed those of the nascent personal computer market, despite, under the guise of an ‘entertainment system,’ stealthily embedding a capable computational device into millions of households worldwide. Although Nintendo is now one of three large players in the home console race, the legacy of the NES still looms large in gaming, art, music, graphic design, and popular culture. There are copious popular and scholarly texts that focus on the NES and its influence, though they tend to fall into a few broad categories: part of the larger history of videogames; Nintendo’s corporate history; Nintendo’s individual game designers; sociological studies of game players; and iconic video game characters. However, the NES’s importance as a computational platform is widely overlooked.

The NES, of course, was neither the first nor the most technologically advanced home console, but it did mark a transition point in the types of videogames that they could proffer. Early Famicom games were either direct ports of or callbacks to arcade games, designed for short-burst, single-screen play. However, within two years, the platforming genre, largely pioneered by Nintendo’s own arcade hit Donkey Kong, emerged as the dominant genre of the ‘second generation’ of consoles. The titular platforms were obstacles or structures the player-character had to traverse in order to reach a goal, like the girders that Jumpman scaled to save Pauline from Donkey Kong or the pits and alligators Pitfall Harry had to swing across to reach the hidden gold bars. Later platformers built upon these early prototypes, expanding traversal beyond single screens to elaborate scrolling and/or
non-linear spaces. Platformers encouraged progressive, long-term play, coherent worlds, and narrative development, characteristics antithetical to the arcade’s quarter-consuming economy.

Nintendo’s first home console was primed to capitalize on this transition in gameplay style. Unlike most of its peers, the NES was engineered with hardware-based scrolling, plentiful on-screen sprites, dedicated VRAM, and ample cartridge program ROM. None of these individual technical specs were revolutionary, but in combination they served as the foundation for the tile-based worlds tailored for character-based platforming. Even the Famicom controller, with its patented plus pad and dual action buttons, was geared for vertical and horizontal movement through 2D space. *Super Mario Bros.*, one of the best-selling games of all time, became the archetype of the genre and encapsulated the Famicom’s unique strengths on a single NROM board. It had a distinctive world (The Mushroom Kingdom), a memorable protagonist (Mario, the plumber), and a clear narrative goal (rescue Princess Peach from Bowser)—all novel features for console games at the time. Legacy consoles meanwhile struggled to adopt new play styles that their hardware was not built to suit. Nintendo never looked back. Popular Mario successors like *Castlevania, Mega Man, Kid Icarus, Metroid, Contra,* and *Ninja Gaiden* underscored the sophistication of console gaming in the new Famicom era. In short, the Famicom was materially suited to produce the games that drove the industry into new expressive possibilities.

**Methodology**

The Famicom shares its 6502 CPU architecture with numerous other machines, including the Atari VCS and the Commodore 64, but its expressive capabilities are radically different. And those differences are more significant than graphics or character types. Historically, media scholars have overemphasized the visual aspect of digital media, a bias that Montfort, Kirschenbaum, and others call ‘screen essentialism,’ as though graphics and gameplay are largely the whims of the game’s designer, rather than a creative negotiation with the hardware’s unique constraints. *I Am Error*, in line with the platform studies series, takes a ‘bottom-up’ approach, locating the code and hardware decisions that shaped the console’s creative possibilities, cultural reception, and styles of play. The book argues for the Famicom’s material importance along two trajectories: first, as a pivotal platform in the evolution and popularization of the platformer genre, for which the Famicom’s hardware was distinctly suited; and second, as a platform of ideal simplicity and popularity to engender the console emulation scene in the 1990s, when both PCs and the Internet reached the necessary maturity to support an emulation ecosystem.

Years after the NES’s dominance and eventual replacement by newer hardware, the
platform had a marked resurgence in emulation. Console emulation mimics the target platform on another, typically more powerful, platform, ideally permitting users to play game software with the closest approximation to the original experience as possible. Though computer emulation was in practice since the 1960s, it typically involved a combination of software and hardware to carry the computational load.

Higher accuracy comes with a concomitant increase in processor demands, especially if the emulation is purely software-based. Similarly, emulation is not solely a matter of replicating the target console’s CPU, but also any additional co-processors, input/output devices, lower level instruction sets, and so on. Beyond its CPU, for instance, the NES has customs picture and audio processors that equally contribute to the look and sound of its games. Each of these processors is a necessary component of emulation.

Until the 1990s, few personal computers were powerful enough to emulate even the simplest consoles. However, the NES’s 6502-based architecture hit the sweet spot of manageable complexity, popular appeal, and PC storage/network limitations. Indie outfits like Bloodlust Software, who released the popular Nesticle emulator in 1997, could easily distribute their software online. Likewise, dumped ROM images were small enough to transfer via modem or floppy and store locally in a handful of megabytes. This was not feasible with contemporary platforms like the Sony PlayStation. Its 3D capabilities were too computationally costly for mainstream PCs and disc-based media was not yet economical to duplicate or store.

The NES stood at a unique turning point in the emulation capabilities of mass-market PCs and the emerging distribution channels of the World Wide Web. The NES was popular enough to attract enthusiast programming interest, its limited CPU, video, and audio processors were manageable for software emulation, and its compact programs were ideal for PC storage and transmission. These crucial features sparked the rapid proliferation of console emulators and allowed the NES platform to ‘live on’ beyond its hardware life cycle. And though the NES’s emulation afterlife augmented many of its shortcomings—via save states, net play, and the like—its hardware quirks were dutifully translated into software.

* I Am Error explores how these quirks are ingrained in the NES’s computational architecture and advances the concept of platform as a shifting configuration of hardware and software that extends even beyond its ‘native’ material construction. The book provides a deep technical understanding of how the platform was programmed and engineered, from code to silicon, including the design decisions that shaped both the expressive capabilities of the machine and the perception of videogames in general. The book also considers the platform beyond the console proper, including cartridges, controllers, peripherals, packaging, marketing, licensing, and play environments. Likewise, it analyzes the NES’s extension and afterlife in emulation and hacking, birthing new genres of creative expression such as ROM hacks, chiptunes, and tool-assisted speed runs.
The book also examines the key games and programmers that either defined the NES's hardware or cleverly worked within its constraints to exceed its expected limitations. *Super Mario Bros.*, despite its mainstream influence, has never had a close analysis of how its successful design was tied intimately to the Famicom's PPU and cartridge ROM. The game's metatile structure (a collection of multiple sprites assembled into a single computational object) adheres strictly to the limitations of the PPU's attribute tables, which govern how many colors can be displayed within a given number of tiles. As a result, *Super Mario Bros.*'s blocks and pipes are constructed according to a strict 16x16 and 32x32 pixel geometry, respectively. Even the distinctive clouds and bushes dotting the landscape are identical sprites repurposed via a palette swap. In short, the Mushroom Kingdom is artfully built around a hardware palette restriction.

This book adopts the methodology of platform studies, first outlined in Montfort and Bogost's *Racing the Beam*, but the particular object of study diverges from that model, expanding and critiquing the notion of a platform as a ‘stable’ configuration of hardware and/or software. First, unlike the Atari VCS, the Nintendo Famicom is a Japanese product that was later exported to the United States. As such, it is born in a vastly different cultural context, shaped by considerations ranging from the smaller size of the Japanese household (and body) to the legacy of suspicion that Americans felt toward Japan post-World War II. Thus hardware and software alike underwent a number of translations that the VCS never experienced, from the shape and color of the console to the censorship of potentially sensitive religious or political imagery that might be unsuitable for American audiences.

Second, the NES hardware became obsolete at a pivotal moment in the history of personal computing. Console emulation became feasible for PCs in the mid-1990s, allowing another ‘translation’ to take place, from hardware to virtual hardware. Though the Atari VCS was also emulated during this time (few vintage console were not), it did not spur the nascent emulation scene like the NES did. The features built into NES emulators spawned new forms of play, performance, and videogame archiving. Suddenly, players could record movies of their play, save games at any point, play online, alter graphics, play foreign titles, and so on. The NES platform blossomed beyond the constraints of its hardware and became more capable than its designers ever intended.

Finally, *I Am Error* considers videogames and their platforms to be important objects of cultural expression, alongside cinema, dance, painting, theater and other media. It joins the discussion happening in similar burgeoning disciplines—code studies, game studies, computational theory—that engage digital media with critical rigor and descriptive depth. But platform studies is not simply a technical discussion—it also keeps a keen eye on the cultural, social, and economic forces that influence videogames. No platform exists in a vacuum: circuits, code, and console alike are shaped by the currents of history, politics, economics, and culture—just as those currents are shaped in kind.
David Sheff’s *Game Over* is indispensable to any study of the Nintendo Famicom (or Nintendo in general). Its influence since its publication in 1993 is pervasive in print and online. Any proper history of videogames, scholarly or journalistic, cites Sheff’s work, and rightfully so. It is a well-written and well-researched book, offering rare insights into the inner workings of Nintendo and its employees.

However, I have chosen to lean lightly on *Game Over*, for reasons worth outlining. First, there are a number of minor technical mistakes or misunderstandings that, over the years, have unfortunately disseminated to secondary sources. These are forgivable oversights. Sheff’s aim was not technical accuracy (though he got a lot right), nor was he an engineer or computer scientist (nor am I, for that matter). Nonetheless, there are enough discrepancies in description that scholars cannot rely solely on Sheff’s account. I wholeheartedly recommend *Game Over* as a primary source, but we must dig deeper.

Second, I now have the benefit of nearly twenty years of fervent historians, journalists, programmers, and fans dissecting and discussing one of the most beloved consoles in videogame history. Wikipedia, YouTube, NintendoAge, and the NesDev community are indispensable in this regard. An astounding database of technical specifications, gameplay videos, television commercials, marketing ephemera, emulators, box scans, password databases, translations, magazine archives, and the like are available to those with the time and effort to investigate. This book would not be possible without those efforts—often anonymous, rarely compensated.

Finally, there is one aspect of research where Sheff had a considerable advantage that is now impossible to replicate. Sheff had unprecedented access to Nintendo’s American and Japanese employees. Nintendo rarely grants such unmediated insight into their corporate practices today. I faced an impenetrable brick wall of PR automata, insistent to defer my research to their sanctioned FAQs and publicity materials. I may have fared better if I were a Japanese speaker. I don’t know. But this did inspire me to dig deeper in the trenches below canonical books like *Game Over* to unearth new insights on a familiar console. I hope there are surprises in store for even the most grizzled disciples of NES lore.

**A Note on Proper Names**

Since part of this book’s project is to pay close attention to the intricacies of hardware and software, a problem arises concerning the naming of objects. The Family Computer and the Nintendo Entertainment System are not interchangeable consoles. They share numerous affinities—a CPU, a PPU, two controllers—but they are not identical. Even the NES is not a
single object. Its US and European models interface with televisions differently—creating important timing consequences for programmers—and they have different external branding.

For consistency’s sake, I chose to designate the Famicom as the ‘default’ console. It is meant to stand in for the entire ecology of 8-bit Nintendo consoles we might alternately call Family Computer, NES, Twin Famicom, NES-101, and so on. In contexts where I need to specifically isolate the Family Computer, I will make that distinction clear. Likewise, when I refer to the NES, I do so intentionally, designating a particular configuration of hardware distinct from the Japanese Famicom. I wish there were a better word to sum up the myriad consoles subsumed under the label ‘Famicom,’ but non-specific terms like ‘the Nintendo’ (despite its common vernacular usage in the 1980s and 1990s) only serve to muddle the corporation and the console. Similarly, an invented term feels unnecessarily academic.

I will also be specific about software when necessary. Some games survived their trans-Pacific voyage largely unmolested. Others were edited heavily, either internally, textually, or for content. In the cases where these distinctions are important, I will designate the game’s region of origin. When I reference a Japanese game, I tend to choose the name most commonly referenced by the online collecting community. Japanese titles are tricky—frequently they contain a mixture of English and Japanese alphabets. Other titles are better known by their romanized names rather than their proper Japanese characters. When possible, I will include both the (non-Romanized) Japanese name and its translation. Since I do not speak or read Japanese, I will only insert Japanese names when I am certain of their proper titles, such as when I have compared the characters to those on the game’s label or consulted my translator.

A final note on the use of Nintendo as a sentient pronoun. I am aware, despite current American policy, that a corporation is not a person. There is no monolithic Nintendo hive mind that exerts its will over the videogame industry. Since its inception, Nintendo has been and continues to be a literal corporation of men, women, engineers, designers, consoles, toys, hanafuda cards, circuit boards, warehouses, office buildings, legal documents, and plumbers (real and imagined). Please grant an exception to this premise when I say that Nintendo does or does not do particular things. We can lay equal blame upon the videogame companies, the passage of time, and lazy authors and scholars who have contributed to the cloud of anonymity lingering over the early history of the videogames. Many of the engineers and programmers responsible for Nintendo’s diverse and substantial hardware and software catalog are unknown or otherwise obscured by nicknames, particularly in Japan. In cases where credits and attribution are extant, I will do my best to name names. When that information is unknown, I must substitute the vagaries like ‘Nintendo’ or ‘the programmers’ when needed.
**Organization**

*I Am Error* is not a chronological review of the Famicom’s hardware and games nor a history of Nintendo. Larger surveys of videogame history are better-equipped to furnish a sequential list of names, dates, and events. While I do provide some necessary historical context, chapters are more generally geared toward specific hardware innovations and the software they supported. Since Japan, the United States, Europe, and the rest of the world experienced concurrent but staggered trajectories of Famicom/NES development, this hardware focus will often require chronological backtracking. Chapter 2, for instance, covers the development of arcade *Donkey Kong*, which precedes the events of chapter 1. Chapter 4, on *Super Mario Bros.*, should chronologically precede chapter 3, which outlines the development of the Nintendo Entertainment System. Despite these non-linear jumps, the book’s plan does follow a roughly chronological order, beginning with the debut of the Japanese Famicom in chapter 1 and ending with the ‘rebirth’ of the NES via emulation in chapter 5.

Similarly, software examples are rarely chosen based on their gameplay merits or review metrics. Games are highlighted based on their relevance to the platform-specific topic, not on whether they are subjectively good, bad, or fun. Fortunately, some of the best and most critically-acclaimed games are also the most interesting to examine. *Super Mario Bros.*, for instance, is not only one of the most critically-lauded and bestselling videogames in history, it is also one of the best illustrations of the strengths and limitations of the Famicom platform.

The book is organized as follows:

Chapter 1 ("The Family Computer") introduces Nintendo’s first cartridge-based console, the Family Computer (or Famicom). The chapter begins with the development of the hardware, proposed by Nintendo president Hiroshi Yamauchi and spearheaded by Masayuki Uemura and his team, R&D2. Thanks to the unlikely choice of the MOS Technology 6502 microprocessor over a more commonplace competitor, Nintendo’s engineers were faced with the challenge of translating their arcade hits to a unfamiliar CPU architecture. The chapter includes detailed descriptions of the Family Computer’s external and internal hardware—from chips to case—and its impact on the domestic spaces for which is was designed. The chapter concludes with an overview of the Famicom CPU’s and PPU’s operation, as well as a technical discussion of the Famicom’s neglected other half: the CRT television.

Chapter 2 (‘Ports’) backtracks a bit to Nintendo’s first forays into the US arcade market via a mistimed failure (*Radar Scope*) and the triumphant phoenix that arose from
its ashes (Donkey Kong). The title ‘Ports’ is used here in two senses: first, as a metaphor for the international culture exchange taking place between Eastern and Western shores; and second, as the term designating the challenging conversion process necessary to translate a game from one platform to another. Donkey Kong is used as a case study for the strange nexus of Western popular culture influences refracted through an Eastern lens and sold back to the West as an unlikely arcade sensation. The chapter highlights the broader cultural context of Japan’s entrance into the Western videogame industry and specifically their software’s categorization into the catch-all ‘novelty games’ genre, then concludes with a comprehensive technical comparison of Donkey Kong and its subsequent Famicom port. A number of significant cuts and revisions were made based on the limitations of both the Famicom’s console and cartridge technology, which are explained in exhaustive detail.

Chapter 3 (‘The Toaster’) details the challenging launch of the Nintendo Entertainment System, a hardware ‘translation’ of the Family Computer suited to the demands of a troubled US videogame market. Two of the console’s initial marketing gimmicks—the Robot Operating Buddy and the Zapper—are examined in-depth. Despite their limited use in the Nintendo software catalog, both are clever toys that reflect the company’s gaming legacy and interface with the television in remarkable ways. The chapter concludes with an overview of inconsistent, and sometimes inexplicable, translations used to transition content from a Japanese to a worldwide audience, from the design and marketing of box artwork to the censorship of ‘offensive’ in-game content.

Chapter 4 (‘Platforming’) is devoted to a technical exegesis of the seminal Famicom game, Super Mario Bros. The chapter delves into the game’s source code and analyzes the Famicom’s peculiar hardware and software constraints that helped shape the game’s design. Many of the hardware programming concepts introduced in prior chapters—scrolling, metatiles, data compression, attribute tables, palette swaps, sprite 0 hit—are expanded and explicated through Super Mario Bros.’s software engine. The chapter concludes with an analysis of the game’s unique and sometimes unintended innovations, including player movement beyond ‘world’ boundaries, exploits, and glitches (e.g. the ‘minus world.’)

Chapter 5 (‘Tool-Assisted’) tracks the Famicom’s ‘rebirth’ through emulation. The chapter begins with a discussion of tool-assisted speedruns, a specialized play style devoted to completing games as quickly as possible while using software assistance, i.e., emulators. NES emulation is situated within a greater history of computer emulation, which I argue is ubiquitous and persistent in the history of computing. The so-called ‘conversion problem’ faced by IBM’s engineers in the 1960s is a translation problem faced time and again during transitions between successive competing platforms. This historical contextualization is followed by a history of the early development of NES emulators and their eventual evolution into the modern forms used in tool-assisted play. The chapter
concludes with a look at the surprising new forms of play afforded through human/software collaboration and their explicit challenge to the notion of platforms as stable objects of study.

Appendix A (‘Famicom/NES Bibliographic Descriptions’) is a practical call for better enumerative and descriptive bibliographies for videogames (and digital objects in general). The appendix includes guidelines for both types of bibliographies, including practical models for scholarly and critical use.

Appendix B is a glossary of terms used throughout the book.

A list of Sources concludes the book. I mention it explicitly since, in lieu of citing each Famicom/NES videogame mentioned in the text and bloating the endnotes unnecessarily, all such examples may be found listed alphabetically (by English translation).

³Tanner, “Adventure of Link - Retranslation.”
⁴Sandifer, “Am Error.”
⁶Donovan, Replay: The History of Video Games.
⁷Sheff, Game Over.
⁸Chaplin and Ruby, Smartbomb.
⁹Provenzo, Video Kids: Making Sense of Nintendo.
¹⁰Ryan, Super Mario: How Nintendo Conquered America.
¹²Kirschenbaum, Mechanisms.
¹³W., Dan et al. “Reference - Oops!”
¹⁴See Chaplin and Ruby, Smartbomb; Dillon, The Golden Age of Video Games; Donovan, Replay; Gamespite, GameSpite Quarterly 5; Goldberg, All Your Base Are Belong To Us; Herman, Phoenix; Kent, The Ultimate History of Videogames; Kline et al., Digital Play; Kohler, Power-Up; Ryan, Super Mario; Sellers, Arcade Fever; Sheff, Game Over.
1: The Family Computer

The Nintendo way of adapting technology is not to look for the state of the art but to utilize mature technology that can be mass-produced cheaply.

- Gunpei Yokoi

Turn on the Control Deck power switch. A colorful game display should appear on the TV screen.

- NES Control Deck Manual

In October 1981, encouraged by the dual successes of the breakout arcade hit Donkey Kong and the Game & Watch LCD handheld games, Nintendo president Hiroshi Yamauchi approached Masayuki Uemura, head of the hardware-focused 任天堂開発第二部 (Nintendo Research & Development 2, or R&D2), about the feasibility of a home videogame console.² Yamauchi knew that arcade games and cheap portables were excellent for short-term profits, but an inexpensive console with interchangeable cartridges could generate profits for years.³ Atari had proven the razor/razor blade model with their longstanding Video Computer System (VCS), the wood-paneled wonder that continued to dominate the US market well beyond the console’s expected shelf life. Yamauchi reasoned that Nintendo could manufacture their own home system, leveraging their popular arcade titles to entice consumers.

Uemura, alongside young engineer Katsuya Nakakawa, a recent addition to R&D2, researched the feasibility of the console’s technical requirements at the budget price Yamauchi demanded—9800 yen, roughly $75.⁴ Thanks to Nintendo’s recent experience with Donkey Kong, it was:

The conclusion [Nakakawa] came up with was that a domestic game console looked to be a possibility if they IC’d the Donkey Kong arcade machine’s circuits and used them as a base. In the spring of 1982, a concrete development project had begun. The code name of the game console they set out to develop was the GAMECOM.⁵

GAMECOM was the internal name for Nintendo’s first cartridge-based home console, the Family Computer (ファミリーコンピュータ), released in Japan on July 21, 1983. Though Nintendo did not officially market it as such, the Family Computer soon gained an
an affectionate nickname. Lopping off a few trailing characters resulted in ‘Famicom’ (ファミコン), an abbreviation that resonated with the Japanese shorthand for personal computer:

In April, 1983, Uemura brought up the topic of the GAMECOM at home. Uemura’s wife then said the following: “If it’s a domestic computer that’s neither a home computer nor a personal computer, perhaps you could say that it’s a family computer. In Japanese, ‘personal computer’ is shortened to ‘pasokon’, so why don’t you nickname it the ‘Famicom’?” These words served as the trigger for what was to come. Several prospective titles had come up prior to the console’s manufacture, but “Family Computer” survived because of how well it expressed the nature of the product.

Thanks to Uemura’s wife, the GAMECOM was renamed Family Computer a mere three months prior to its release. And besides its linguistic affinities to the popular abbreviation of ‘personal computer’ (パーソナル・コンピュータ), the portmanteau of ‘family’ and ‘computer’ described how the machine was meant to fit into the lives of those who purchased it. ‘Family’ designated the console’s range of social functions: Nintendo was bringing its popular arcade titles into the home, to be shared with the family, to become part of the family, and to be played in the family’s social space. It would be a ‘domestic computer’ in the most familiar sense. But the console would also be a powerful and capable computing device—more than a simple machine that played variations of ball-and-paddle electronic games. From the outset, Yamauchi had ambitions to expand the Famicom’s capabilities beyond reproducing Nintendo’s arcade roster.

Of course, this linguistic idealism was not all social goodwill on Nintendo’s part. Nintendo was selling a game console and game consoles were seen as toys. Despite its clever name, marketing the Family Computer to children would find limited success without the entire family’s economic input. Katayama’s 1996 profile of Nintendo explains:

As the Japanese name, ‘family computer,’ shows, the designers had the family market in mind. The product had to be priced so that parents would buy it. No matter how great the games, if mothers thought they were too expensive the machine would never take off. Nintendo therefore aimed for prices that children themselves could afford or at least would be able to convince their parents to lay out.

Juggling Yamauchi’s demands of affordability, approachability, and power posed significant challenges for the R&D2 team. Not only was Uemura expected to produce a console for 9800 yen, but it had to be future-proofed. Yamauchi expected the Famicom’s underlying hardware to grant Nintendo a three-year competitive advantage.
Family Computer’s spec software was Donkey Kong, a videogame supported by bleeding-edge hardware (see chapter 2). Arcade machine were custom built to suit the needs of individual games and cost hundreds of thousands of yen to produce. A cartridge-based machine had to be more flexible; Donkey Kong would be one of many arcade titles the Family Computer could support. And Nintendo’s official home version of Donkey Kong had to be at least as good as the exceptional ColecoVision port. Nintendo could not allow a competitor’s arcade reproduction to outshine their own efforts.

An Unconventional Stone

Uemura grappled with the problem for many months, consulting with the company’s arcade engineers to figure out how they might transition Donkey Kong’s powerful hardware to an inexpensive home console.¹¹ Nakakawa’s solution to ‘IC the arcade machine’s circuits’ was not a simple plug-and-play operation. At the heart of any arcade game was a microprocessor CPU. And the choice of CPU had important ramifications on cost and capabilities. Choosing an underpowered CPU might hamstring the number of sprites available onscreen, the game’s palette, or the options for sound output. Alternately, if a CPU was too complex, it would be cost-prohibitive to manufacture and difficult to program.

In the early 1980s, there were two major players vying for the low-cost microprocessor market, MOS Technology’s 6502 and Zilog’s Z80. Both 8-bit processors were cheap but powerful, capable of driving a range of videogame consoles, PCs, and arcade games. Japan’s arcade industry leaned heavily toward Zilog’s microprocessor. The massive hits of the era—Pac-Man, Galaxian, Galaga—and Nintendo’s own stable of early 1980’s arcade titles were all powered by the Z80.

The Family Computer nearly had a Z80 too. In fact, prior to Nintendo’s decision to forge ahead with their own console, the Family Computer was nearly a ColecoVision.¹² In the US, Atari had maintained a near-deadlock on the emerging home videogame industry with the Atari VCS. However, the unlikely Connecticut Leather Company (Coleco for short), who had previously dallied in derivative Pong clones and electronic handheld games, forged a short-term exclusive licensing agreement with Nintendo to bring Donkey Kong to their new console, a Z80-based machine that technologically trumped the elder VCS. The Kong partnership benefited both parties, spurring the ColecoVision to impressive first year sales and expanding Nintendo’s market reach beyond the arcades.

Nintendo had great admiration for Coleco’s port of Donkey Kong. Licensing the ColecoVision in Japan would allow Nintendo to bring its arcade hits to the home videogame market without a massive upfront investment in manufacturing. In The Golden Age of Videogames, Dillon indicates that the two companies were close to a console licensing deal:
Due to this successful [licensing] partnership and to ColecoVision’s unique capabilities to render Nintendo’s arcade games properly, Coleco and Nintendo were quite closely tied at a certain point. So close, in fact, that Nintendo proposed to Coleco an agreement to distribute and sell ColecoVision in Japan. The two companies, though, couldn’t reach an agreement on the economic terms and negotiations were abandoned when Nintendo declared it would design its own system instead.

Coding for the ColecoVision’s Z80 core certainly would have smoothed software conversions from arcade to console. Nintendo’s Radar Scope, Donkey Kong, Donkey Kong Jr., Popeye, Mario Bros., and Donkey Kong 3 were all Z80-based, so translating them to Coleco’s console would have been much simpler than porting to an unfamiliar microprocessor architecture. There was also a built-in risk to introducing new proprietary hardware. The less friction there was for third parties (i.e., publishers not directly affiliated with the console manufacturer) to port their games across multiple systems, the better. Without third party support, a new console was dead in the water.

Clearly, the Nintendo ColecoVision never came to fruition. Once negotiations broke down, Nintendo decided not only to design their own superior console, but to forgo the Z80 microprocessor altogether in favor of the 6502. This surprising decision ultimately came down to a mixture of corporate politics, managerial mandate, hardware licensing, manufacturer supply, and competitive strategy.

President Yamauchi had a long-standing reputation as a shrewd but imperious leader. According to his employees, he possessed an incisive but opaque business sense. Prior to the Famicom’s development, Yamauchi unexpectedly forbid any collaboration between Sharp and Nintendo related to the new console. This was a jolt for Uemura—Sharp was both his former employer and Nintendo’s close hardware partner for the Game & Watch. But Yamauchi insisted that Sharp’s attention would be divided if they continued the Game & Watch line and diverted resources toward Famicom research and development.

Uemura was left to look for partnerships elsewhere, but with Sharp out of the picture, Uemura found little support from other electronics suppliers. Officially, vendors told Uemura that parts were scarce due to a recent surge in demand for PCs and word processors, but he suspected that they were either reticent to wager on a risky videogame product (and Nintendo themselves) or had no idea how to produce the machine that Nintendo required.

Uemura and semiconductor manufacturer Ricoh found one another at a fortuitous time. Ricoh had the advanced facilities Nintendo required and were currently only producing at 10% capacity, an unsustainable shortfall for a large manufacturing operation. Uemura, along with engineers Nakakawa and Masahiro Ootake, visited the semiconductor factory, where they were met with enthusiasm about a potential partnership. Hiromitsu
Yagi, a Ricoh supervisor, had worked at Mitsubishi when they had partnered with Nintendo to produce the Color TV Game 6 in 1977. Yagi had been in charge of the console’s chip design, so he knew how to deliver up to Nintendo’s expectations. Uemura pitched the idea of making Donkey Kong for a console and, thanks in part to their employees’ desire to ‘take the game home,’ Ricoh agreed to take on the challenge. However, the per-chip price necessary to comply with Yamauchi’s 9800 yen target was not feasible at videogame console production numbers. Chips prices drove down at volumes of millions, not tens of thousands. So Nintendo had to make an extraordinary gamble. They guaranteed Ricoh a three-million chip order within two years. Ricoh happily agreed to the terms, but feared that Nintendo was headed for an economic catastrophe.

Though Donkey Kong was the hardware target, Ricoh lacked a manufacturing license for the Z80. They proposed a suitable alternative: the MOS Technology 6502. Despite the microprocessor’s capabilities, it was a peculiar choice. The chip was popular in US and European consoles and PCs, but it was relatively unknown among Japan’s engineers. Both Uemura and Yamauchi saw this as an advantage: the tradeoff in engineering complexity would pay off in hardware obfuscation. In a recent interview, Uemura and current Nintendo president Satoru Iwata reflected on Nintendo’s non-conventional choice of microprocessors:

Uemura: The one Ricoh suggested was a CPU called 6502. It wasn’t widely used at the time, it was said that the only people in Japan who could understand it were a few people at Tokyo University and Kyoto University. The reason we used it was that it would be hard to analyze... Normally, in transplanting Donkey Kong, the quickest way would have been to use the CPU in the arcade version. But Ricoh wanted us to use the 6502, which they had the license for. When I said I wanted to use the 6502 at Nintendo, the staff told me that I make such decisions because I didn't make video games.

Iwata: Maybe the reason you experienced in-house resistance was that even people at Nintendo had never used that CPU. The most common 8-bit CPUs back then were what were known as the 80s, CPUs like the 8080 and Z80. It was most common to use those for arcade games like Donkey Kong.

Uemura: That turned out lucky for us, though. After the Famicom went on sale, if someone from another company opened it up, they wouldn't be able to make sense of it.

Iwata: Even if another company wanted to make a game for the Famicom, they wouldn't understand the CPU, and wouldn't be able to do anything.

Uemura’s team was right; it was difficult to transition from the commonplace Z80 to the
unknown 6502. Since they had no available development tools, they had to build their own. And because their console prototype no longer shared an architecture with their arcade titles, they had to tediously reconstruct the games from scratch. As a result, ‘the work required a lot of patience, including tasks such as watching the game screen and measuring the timing of animations with a stop watch.’²⁰ In April 1983, Nintendo’s development team were granted reprieve when they hired Shuhei Kato, a young engineer who specialized in the 6502. The ‘Living 6502 Manual’ spurred on the final surge of software development.²¹

In the end, the Ricoh partnership satisfied all of Yamauchi’s stipulations. Thanks to an unprecedented parts order, the console would be cheap (though not on target—the Famicom debuted at 14,800 yen). And thanks to an unconventional choice of ‘stones,’ as semiconductors were called in Japan, Nintendo would have an edge on their competitors. Reverse engineering the Famicom would prove as troublesome to competitors’ engineers as its had for Nintendo’s. From the outset, Nintendo designed their console with proprietary control in mind. And despite the risk of alienating third parties, they wanted to be the ultimate arbiters of who could and couldn’t produce software for the Famicom—a predilection that would intensify as Nintendo gained control of the worldwide videogame market (see chapter 3). Yamauchi wagered that third parties would bend to brute market dominance.

The 6502 had one other significant advantage: its die was one-quarter the size of the Z80. As a result, Nintendo’s and Ricoh’s designers were able to shrink the Famicom’s body considerably and further slash the cost of the machine. Compared to contemporaries that used the Z80—ColecoVision, Sega’s SG-1000, MSX—the Famicom was more compact and less expensive. The design had both marketing and cultural advantages. On one hand, the smaller Famicom, garbed in bright red and white plastic, had a toy-like appearance, sure to grab the attention of young children. On the other hand, the diminutive size fit the tastes of Japanese consumers and the size of their domestic spaces.²² The compact profile made good on the Family Computer name.

With the microprocessor question settled, Nintendo began to work in earnest on the Famicom prototype in late 1982 and soon after courted another US licensing partner. Ever since their entry into the arcade business, Yamauchi had had his sights set on America. Donkey Kong, its arcade successors, and the Game & Watch were strong starts, but in America, Nintendo was still a minor Japanese player with a funny name. As a result, Nintendo initially decided to license the Famicom hardware to an established US partner—a reversal of their aborted negotiations with Coleco.

In April 1983, Atari executives flew to Kyoto to inspect demo versions of Donkey Kong Jr. and Popeye running on early emulated hardware.²³ According to Atari’s Don Teiser, the prototype ran the arcade ports with ‘only minor display glitches.’²⁴ Teiser’s memo indicated Atari’s interest in the console along with a sizable list of stipulations that President
Yamauchi demanded (e.g., a minimum two million console order, Atari’s limited access to Nintendo’s hardware specifications, etc.). But the memo also indicates that he and the other executives present were withholding their true intentions from Nintendo. Atari was shopping for a successor to the aging VCS and its disastrous follow-up, the Atari 5200. While they openly courted Nintendo, internally they were weighing a competing prototype, codenamed MARIA, developed by General Computer Company in Cambridge, MA. This alternative looked to be the ‘superior machine,’ but uncertainty regarding the chip’s large-scale manufacturing costs kept them in talks with Nintendo. In other words, Atari was purposely delaying their decision so they could pick the better hardware.

Once again, the partnership was not meant to be. Although Atari ultimately opted to use MARIA in the Atari 7800, due in part to Nintendo’s impatience with the waffling American company, grander mitigating circumstances intervened on Nintendo’s behalf. By the end of 1983, the US videogame industry collapsed catastrophically (see chapter 3). Had Nintendo partnered with Atari, the US Famicom would have likely been among the collateral damage. Even the promising ColecoVision was one of many casualties in the videogame market fallout. Nintendo would have to forge ahead alone.

Red, White, and Gold

Readers unfamiliar with the Family Computer might be surprised by its size: it measures 22cm long, 15cm wide, and 6cm deep and weighs approximately 620g. Its curious profile looks more like the torso of a plastic robot than a serious game machine. In short, the Famicom looks nothing like the personal computers with which it shared a name.

Nintendo’s early consoles—all simple variations of tennis, tabletop, racing, and brick-breaking games—had already experimented with novel, colorful designs. The deep oranges, reds, and yellows of the Color TV games were far afield from the faux wood-paneled furniture style of US consoles, thanks in part to the influence of a young designer named Shigeru Miyamoto. The cherubic Miyamoto is now one of Nintendo’s most prominent representatives and universally hailed as one of the greatest innovators in videogame history. He has designed, produced, or directed the lion’s share of Nintendo’s most prized franchises, from Donkey Kong and Super Mario Bros. to Nintendogs and Pikmin. But prior to Miyamoto’s industry ascension, he worked as an industrial artist. His first jobs at Nintendo included designs for mahjong labels, playing card stencils (hanafuda, or Japanese playing cards, were Nintendo’s original gaming industry), arcade cabinet exteriors, and the 1979 console game Color TV Game Block Breaker (カラーテレビゲームブロック崩し). ⁵⁵ Miyamoto brought a playful sensibility to his industrial designs, emphasizing simplicity, accessibility, and fun. The bold colors and compact form factors of Nintendo’s earlier consoles would carry over into the look of the Family Computer.
While Miyamoto was not directly involved with the Famicom’s external design, Uemura was, and the latter established a list of seven specification guidelines to help shape the console’s look. These included the necessity of two controllers, the ability to store them on the console (another legacy of the Color TV consoles), the number of controller buttons, the various ports and power connectors, and the desire to have the cartridge dimensions ‘be about the same as an analog cassette tape.’ Curiously²⁶, in spite of the console’s name, Uemura did not want the Famicom to look like either a computer or a toy, but something wholly different. Ricoh’s designers concurred. The console should not be judged on looks alone:

If the system’s exterior resembled an audio device, for example, consumers would make judgements on the product’s price and value based on preconceptions. [Ricoh] instructed the team to design the exterior in such a way that people wouldn’t be able to make snap judgements about it.²⁷

Despite Uemura’s concession that he failed to realize his design goals, there is no question that the final console stands out from its consumer electronics peers. The Famicom body is replete with ridges and angles. Numerous recessed surfaces, buttons, levers, hinges, and vents combine to form a unique plastic topography (figure 1.1).

1.1 The Nintendo Family Computer. (Source: Evan-Amos, Wikimedia Commons)

The front of the console slopes forward slightly to display a slender aluminum plate printed with the console’s name (in English) and the Nintendo logo. Behind the angled surface there are three mechanical switches: a reset button, a power button, and a large
mechanical slider to help children lever the cartridges out of the console’s interior. The two buttons have explanatory stickers pointing toward them, explaining their use. The left sticker reads, in Japanese, ‘When removing a cartridge, please be sure that the system is off,’ and the right, ‘Pressing the reset switch will cause the score you’ve obtained to be deleted.’

Cartridges are inserted vertically into a narrow slot behind the lever. Since the slot exposes the cartridge card edge connector (and the console’s interior), a hinged plastic flap covers the hole when no cartridge is present. Though Uemura hoped to match the cartridges’ dimensions with cassettes so they could be stored in standard tape cases, the final ROM PCB ultimately proved too large, though only by a few millimeters. Famicom carts measure 11cm x 7cm x 1.7cm in comparison to the 10.9cm x 6.9cm x 1.7cm dimensions of a cassette case.

Uemura’s guidelines did result in some successes: the sides of the Famicom have recessed edges cut to house the console’s wired controllers. Both controllers are rounded along the edges but have an additional raised molding around their perimeters that allow them to nest within their respective cradles without falling out. Cords emerge from the controllers on either side—from the upper left on controller I, upper right from controller II—rather than the top. Though the placement looks sleek when the controllers are stored, as there are no cords sprouting from the top, it makes the controllers awkward to hold, since the cords emerge where the hand naturally grips the joypad.

As they had with their Color TV consoles, Nintendo chose to connect the controllers directly to the motherboard. Again, to accommodate the controller’s storage position, the cords enter the Famicom from the rear. As a result, the Famicom cords are incredibly short, since a bulk of their length is actually inside the console. From the rear of the Famicom’s interior, the cords are wrapped around a plastic post, then run the length of the machine and attach to the front of the motherboard. This design proved troublesome; players could not easily replace faulty controllers, so they had to send their entire console to Nintendo for repair.

A 15-pin expansion port is the sole feature of an otherwise barren front edge. On the motherboard, the expansion port and controller connections share the same edge, so Nintendo could have run the cords directly into the front of the Famicom. (In fact, detachable controllers were initially considered, but ultimately axed to cut costs.) The expansion port is evidence of the Famicom’s aspirational design. Yamauchi originally requested that the Famicom support a number of computer peripherals, including a cassette disk drive, a keyboard, and a modem. Yamauchi eventually told his engineers to nix these add-ons in the interest of cost reduction. Fewer peripherals made the Family Computer appear less intimidating to new users. Excessive hardware additions would dampen its ‘family’ aspect in favor of ‘computer’ traits, making the console less appealing to
children. Still, Nintendo had the forethought to include the expansion port. As the Famicom gained popularity, the peripherals excluded from its initial launch were eventually added by third parties. The expansion port supported keyboards, all manner of controllers and joysticks, light guns, 3D glasses, and more. A 1987 profile of the Famicom in *Best of Japan*, for instance, noted that Nomura Securities ‘announced plans to develop a system allowing investors to use their famikons to read market information and to buy and sell stocks at home.’

The rear of the console includes several connection ports and switches. From left to right, there is a port for connecting the 4-watt power supply, a TV<>GAME switch to allow the Famicom to serve as a TV pass-through when not in use, a CH1<>CH2 switch for selecting one of two TV channels for game display, and an RF (radio frequency) switch to connect to the television. The RF terminal on an analog television was used to connect an antenna, the standard means to receive broadcasts at the time. In our contemporary world of HDMI, component, and DVI inputs, the RF switch occupies the bottom of the totem pole in signal quality. But for televisions of the late 1970s and early 1980s, RF was the reigning analog standard.

The bottom of the unit has a vented plastic base affixed (with six Phillips head screws) to the larger upper portion of the console body. The base has four rubber feet and angled edges that create a footprint smaller than the full width of the console body. Directly above the vent is an embossed rectangle with the Family Computer’s model number ‘HVC-001,’ copyright information (‘©Nintendo Co., Ltd. 1983’), and ‘MADE IN JAPAN.’

Nintendo used the HVC, or Home Video Computer, abbreviation for all manner of hardware related to the ‘Famicom Family,’ as the unifying brand mark was known (figure 1.2).

1.2 The ‘Famicom Family’ mark meant to unify the Family Computer brand. The logo did not appear on Family Computer consoles until approximately 1986.

The AC adapter is HVC-002, the keyboard peripheral is HVC-007, the R.O.B. robot...
peripheral is HVC-012, the revised Family Computer body style is HVC-101, and so on.\(^{34}\) Nintendo reserved HVC labels for select games and internal components as well. *Super Mario Bros.* received HVC-SM, for instance, while its PCB was stamped with HVC-NROM-256K.\(^ {35}\) Nintendo used a similar labeling scheme for their international hardware, substituting NES for HVC (e.g., NES-001 was the Nintendo Entertainment System).

Once the bottom cover is removed, one can see that the bulk of the Famicom’s interior is occupied by the motherboard, again attached to the upper cover with six screws. The motherboard itself is tightly packed with the large shielded power supply, the cartridge card edge connector, controller/expansion port connections, and its myriad ICs.

The internal eject mechanism, a wide plastic bar with two protruding arms, is attached to the interior of the console’s upper cover. The mechanism has a thin metal bracket that attaches to the red slider used to hoist cartridges out of the system. When the player pushes the slider, the metal bracket, itself attached to a small spring, rolls the mechanism along its angled arms until they collide with the cartridge. With a slight bit of pressure, the arms pop the cartridge off the card edge connector. Since the slider is spring-loaded, once the player releases pressure, it moves back to its initial position. It is a handy mechanism, but largely for show. Players could also grip the cartridge and pull it out manually, but Nintendo R&D1’s lead engineer Gunpei Yokoi thought the eject lever added a nice toy-like feel, delighting children by popping the cartridge out: ‘Even when they weren’t playing games, they could entertain themselves by clattering around.’\(^ {36}\)

While the bulk of Famicom’s body is white, the switches, logo plate, cartridge slot cover, expansion port plug, controllers, and bottom plate are all painted a rich maroon. The controllers feature two additional accent colors. Each of the buttons, their labels, and two thin horizontal decorative lines are painted black, all surrounded by a brushed gold face place. The red, white, and gold triumvirate are as iconic in Japan as the grey, black, and red of the NES are abroad. Many products since 1983 feature the Family Computer color scheme, including a special Japan-exclusive edition of Nintendo’s own Game Boy Advance SP, painted to mimic its elder console.

There are several accounts of how and why Nintendo settled on the Family Computer’s trademark color scheme. The red, for instance, has roots in Japanese culture, where historically color could indicate social rank and hierarchy. 小豆色 (or *azuki-iro*) is a deep red derived from a bean used in many Japanese dishes and treats.\(^ {37}\) In 2010, ITmedia reported that Nintendo chose *azuki* red for reasons unrelated to cultural allusions. Like the hardwired cords, white and *azuki* red were simply the least expensive options for colored plastic. However, a *Nikkei Electronics* retrospective profile of the Family Computer’s development reports that President Yamauchi picked the colors from an advertisement. While commuting with Uemura, the president pointed toward a billboard for DX Antenna
and said, "That’s a good color."³⁸ Despite the apocryphal tone of such an offhand decision, the similarities between DX Antenna’s package design and the Famicom’s color scheme are striking (see figures 1.3 and 1.4). Even the DX logo’s diagonal stripe is reminiscent of the ‘pulse line’ design used on many early Famicom cartridges.

1.3 The ‘Silver Ribbon’ DX television antenna (left) and the DX Antenna logo (right).

1.4 Two Famicom controllers (left) and the ‘pulse line’ cartridge design (right). (Source: Earthbound Wiki / Gay Gamer)

The Family Computer box included a form-fitted styrofoam insert cut to house the remainder of its contents: the console, AC power adapter, RF adapter, several single-sheet safety warnings, a console instruction manual, and a 34-page manga called ‘This is Family Computer!’³⁹ The comic, a companion to the ‘adult’ instruction manual, is a remarkable document considering its intended audience. It uses a short story, centered around three elementary school children, to instruct young Famicom owners on how to properly use the console.⁴⁰ One of the children, Konkichi, possesses powers (along with a tail) that allow him, for instance, to magically dry the Famicom after he spills his drink on it. He explains, ‘The Famicom is weak against water you see. Inside the main unit there are LSIs and from these goes many connector traces.’⁴¹ Konkichi asks whether they ‘get it’ and is surprised when the other children have never heard of Large Scale Integrations. Konkichi then uses his ‘psychokinesis’ like an x-ray, showing the children the interiors of the console and cartridges, naming their various components and how they function (figure 1.5).
1.5 Two pages of the Family Computer pack-in manga ‘This is Family Computer!’ show its phenomenally detailed explanation of the console’s underlying technology. (Source: Famicom World)

The children use a number of apt broadcast analogies, comparing the cartridge’s PRG- and CHR-ROM to the script and actors, with the CPU serving as director and the PPU as the camera. Though we will not have Kokichi as our guide, we too will discuss the inner workings of the Famicom in a later section.

Close Playing

The Family Computer had two wired controllers that ran directly into the rear of the console, threaded across the length of its interior, and connected to the front of the motherboard. The short connecting cable, only 75cm long (with an additional 22cm wound inside the Famicom), kept players in close proximity to the console. Smaller televisions and shorter cables meant that videogames were played close to the screen, an ideal spatial configuration for typical Japanese homes, which tended to be much smaller than their American counterparts. The Family Computer, as its name implied, was a console designed
for intimate domestic spaces. Two people playing simultaneously would be close to both one another and the console.

Despite its computer namesake, the Famicom was meant to rest on the floor or low table. In 1983, a PC was comprised of several bulky components: a monitor, a box to house the silicon internals, a keyboard, disk drive, and so on. A computer took up a lot of space. A desktop and chair were best suited for both the size of the machines and longterm computer use, especially typing. The linguistic legacy of the ‘desktop computer’ and the metaphor of the desktop as the default state of the graphical operating system indicate as much. As we have seen, the Famicom was originally planned to be more computer-like, with bundled peripherals like tape storage and a keyboard. Had the Famicom ultimately ended up looking more like a conventional PC, it would have been better suited to the ‘vertical’ orientation of desks and chairs. Uemura considered such a configuration, but he soon realized that most players would not want to use their consoles on desks. A ‘horizontal’ orientation better suited the tastes of Japanese consumers:

In the beginning, we wanted to add all sorts of bells and whistles to our game machines, but we realized that in Japan, anyway, most people would probably be lying on the floor or snuggled up inside the kotatsu (foot warmer with a quilt over it) when they played, not sitting in front of a solid, stable desk. ‘Game and Watch’ was a hit because you could play it in any position, and we decided to adopt that same form for the Nintendo Entertainment System... ⁴²

Uemura’s conclusion seems obvious now, but home videogames in the 1980s were still novel enough that their ‘proper’ position in domestic and social space was not yet codified. The physical forms of computational devices, from mobile phones to room-size server racks, are not benign objects; they participate in and structure social, personal, cultural, and economic spaces. Upright arcade cabinets, for instance, were played while standing. Their form facilitated fluid movements between machines. Players were meant to insert a quarter, play for a short period of time, and move along to the next game. Screens were commonly angled backward to allow players to lean into the cabinet, shielding them from the sights and sounds of other noisy videogames. When players sat at upright cabinets, they perched on high stools, the common furniture of the pubs and taverns that initially hosted such machines. Cocktail arcade cabinets likewise reflected their social milieu. Their flat, squared surfaces and low profiles resembled tables. Players sat on either side of the machine and looked down at the monitor, which was mounted flat on its back. Unlike the cave-like immersion of upright cabinets, cocktail cabinets encouraged social play. Accordingly, cocktail games were programmed either for cooperative play or to rotate between multiple competitors in turns. Players could sit for a while, rest their drinks on the glass, and enjoy a videogame together.
When videogames moved into the home, consumers had to be taught how, where, and with whom to play them. In single-television homes, the TV was usually located in the living room, where it could be shared by the family. Since the videogame console required the television, it resided there too. The earliest home consoles featured simple variations on the ball-and-paddle play pioneered by *Tennis for Two*, *Pong*, and their imitators. Due to both limited technology and their arcade heritage, these games normally required two players. The living room, already a site of family gathering, was conducive to social play. Early commercials for the Magnavox Odyssey, Fairchild Channel F, Coleco Telstar, and other contemporary consoles showed variations on the same themes: this is how the console connects to the television, this is how you select or insert different games, this is how you hold the joystick or paddle, this is how you and your friends and family gather around the television to play.\(^4\) None of these practices were taken for granted. The instructions for play, including the arrangement of bodies and machines, were built into the commercials.

By the early 1980s, Japanese consumers were more accustomed to computers’ presence in the home, but any added sense of familiarity or comfort could help the introduction of a new product. The Family Computer joystick was new compared to competitors’ controllers, but it was a familiar design for the millions of people who had played Nintendo’s handheld videogames.

In 1980, Nintendo introduced *Ball*, the first in a long series of Game & Watch titles. A Game & Watch portable was a single game, clock, and alarm built into a compact handheld enclosure. However, calling them videogames is something of a misnomer. Game & Watch titles were certainly games, but their simple circuits generated no video signal. Instead, simple animated graphics were built from segmented liquid crystal displays identical to those used in calculators. In *Ball*, a cartoon man attempts to juggle either two or three balls (depending on the Game A/B selection) between his left and right hands. Each individual position of each of three arcs that the balls may travel is an LCD segment that is either lit or not. The same is true for the man’s body and moveable arms. Current Nintendo president Satoru Iwata and Game & Watch developer Takehiro Izushi described the limitations of the calculator LCDs in a 2010 interview:

Iwata: Each number from 0 to 9 is made of seven parts called segments. In other words, it’s a way to display numbers using seven component parts.

Izushi: Right. So if a chip can calculate eight digits, that’s 7 segments times 8 digits for a total of 56 segments. And there’s the decimal point and symbols like the minus sign. We made the Game & Watch: Ball game using a chip that could display 72 segments.

Iwata: You could turn each of those 72 segments on or off, and used them to
The Family Computer’s PPU can position a sprite object at various positions along a fixed coordinate grid. Updating these positions over time produces motion. In the Game & Watch, screen objects are discrete entities, so no deviation beyond their fixed course is possible. Similarly, objects can not overlap. Only one object can occupy a given portion of the screen, even when unlit, so animation appears jerky as objects shift from position to position. The monochromatic LCD graphics are thus more akin to the grid of bulbs used to animate sports stadium displays or traffic signs than any console video processor.

The Game & Watch platform limited the sophistication and density of animations possible on a small screen, but Nintendo devised a remarkable number of gameplay variations from that limited palette. Early portables like Ball, Fire (1980), Manhole (1981), Parachute (1981), and Octopus (1981) were fun if rudimentary games. As the platform matured, the games grew in complexity. Nintendo even sold streamlined conversions of their arcade and console games, such as Donkey Kong (1982), Mario Bros. (1983), and Zelda (1989). To increase the range of gameplay options, Nintendo eventually expanded the Game & Watch to two connected screens (figure 1.6).
Gunpei Yokoi, Nintendo’s head of R&D1, led the development of both the Game & Watch concept and its hardware. Yokoi had been a mainstay of Nintendo’s games division since the late 1960s, when he designed the ウルトラ ハンド, or Ultra Hand (1966). Yokoi’s first toy was a blue plastic lattice that expanded and contracted like an accordion bellows. A child could grip and squeeze the handles to extend the plastic arm, then grasp small objects with a pair of rubber cups fastened to the end. The simple mechanical device, handpicked by President Yamauchi, was a big hit for Nintendo, resulting in Yokoi’s move from maintenance man to game designer. During his tenure at ‘pre-videogame’ Nintendo, Yokoi invented many weird and inspired gadgets: the Ultra Machine (1967), an automatic baseball pitching mechanism; an electric Love Tester (1969); the Light Telephone (1971), a short-range walkie-talkie using photo cells; the Custom Lion (1976) light gun target shooting game; the Chiritorie (1979), a radio-controlled mini-vacuum; and the mechanical
puzzle *Ten Billion* (1980).”

Yokoi was instrumental to Nintendo’s success, not only due to his inventiveness, but also his influential approach to product design. In a series of interviews published in Japan in 1997, Yokoi articulated his design philosophy as 枯れた技術の水平思考, which translates to English clumsily as ‘lateral thinking for withered technology.’ In English, ‘lateral thinking’ denotes a creative, unexpected approach to problem solving. This is the approach Yokoi brought to ‘withered’—i.e., outdated, inexpensive, or otherwise off-the-shelf—technology. Yokoi famously devised the Game & Watch concept after noticing a train commuter whiling away the time on his pocket calculator. If a simple calculator could engross the man, why not a pocket-sized videogame?

In the late 1970s and early 1980s portable calculators were a consumer electronics sensation in Japan. The so-called ‘Calculator Wars’ were sparked by the introduction of the Casio Mini in 1972, which catalyzed developments in electronics miniaturization and low-cost liquid crystal displays. What was formerly a vestige of Japanese office culture transformed into a mainstream computing device. Everyone wanted one:

> The Casio Mini sent a shockwave through every company that had a stake in the calculator market. It was a quarter the size of other calculators at the time, and its price had been kept low—at 12,800 yen, it was about a third as expensive as its competitors. It was an explosive hit, breaking the million-unit barrier within ten months of its introduction and eventually going on to sell ten million units. It recast the terms of the conflict in the direction of lower prices and miniaturization.

The two main competitors Sharp and Casio battled for years, introducing progressively cheaper and smaller models. According to Inoue, Casio struck the death blow in 1983 with a solar-powered calculator measuring 0.8mm thick. By the time Yokoi saw the salaryman engaged with his pocket calculator, LCDs and miniaturized ICs were cost-effective enough to use for an affordable handheld gaming system. As the earlier quote from Izushi illustrated, segmented displays were originally devised for displaying ten digits, not game graphics. In the case of the Game & Watch, the withered technologies were low-cost calculator components and the lateral thinking was using their simple segments to build games.

Thanks to Yokoi’s influence, Nintendo has consistently shown an ability to mine engaging games and hardware from obsolete and outdated technologies: *Donkey Kong* arcade units were reworked versions of unsold *Radar Scope* boards (see chapter 2); the clunky 3D of the Virtual Boy resurfaced in the Nintendo 3DS; the everyman Mario has worked in professions ranging from doctor to chef; the mechanical *Duck Hunt* rifle resurfaced as the Famicom light gun game *Duck Hunt*; and the gripping cups of the *Ultra
Hand are echoed in the grasping arms of the Robotic Operating Buddy (see chapter 3). Unsurprisingly, Yokoi had his hand in each of these projects.$^{52}$

The Game & Watch featured another Yokoi innovation that would live beyond the simple LCD toy. Nintendo’s US patent for the ‘Multi-Directional Switch’ illustrates the eponymous device as a component of the Game & Watch (figure 1.7), but it reached iconic status as the ‘D-pad’ (‘D’ for digital or directional, rather than a shape reference) on the Famicom/NES controller.

![Figure 1.7: Nintendo’s US patent illustration for their ‘multi-directional switch,’ seen lower right.](image)

Yokoi first devised the D-pad (originally called the ‘Plus’ button) for the Game & Watch port of Donkey Kong (1982). The gameplay of many titles took place on a single horizontal axis and thereby required only two buttons for control. In Ball, for instance, the player could press a ‘<LEFT’ or ‘RIGHT>’ button, each on its respective side of the handheld, to move the juggler’s hands into position. The same button configuration controlled the stretcher in Fire, the rowboat in Parachute, and the turtle-hopping man in Turtle Bridge (1982). In instances where gameplay required an extended range of motion, the buttons multiplied. In Egg (1981), two stacked pairs of buttons on either side controlled the four possible diagonal positions of the wolf’s arms. In fact, nearly all Game & Watch titles prior to Donkey Kong used four buttons for diagonal movements rather than the four cardinal directions. The exceptions either used the buttons for non-directional purposes (Flagman [1980]) or two-player simultaneous play (Judge [1980] and Lion [1981]).
As we will see in chapter 2, arcade Donkey Kong’s run-and-jump gameplay required movement along both axes. Jumpman ran horizontally along girders, climbed ladders vertically, and jumped with a dedicated button. To replicate this movement on the Game & Watch using the standard button configuration would have required five individual buttons — four to control movement and one to control jumping. Stacking the directional buttons on either side would make the controls unnecessarily confusing, much less trying to position the jump button so it could be reached comfortably during play. Donkey Kong’s novel gameplay required an equally novel control solution.

![How to Play Donkey Kong](image)

1.8 The Game & Watch Multi Screen Donkey Kong insert card included in the handheld’s box introduces the player to the ‘PLUS’ controller.

The D-pad design is so simple, it seems self-evident, but no joystick design up to that point had cracked the problem as well as Nintendo did. Atari’s iconic VCS/2600 joystick was ergonomic and easy to use, but its simplicity limited the input complexity of its games. The successor 5200 controller added buttons but regressed in reliability and ease-of-use. Intellivision and ColecoVision controllers, featuring an overwhelming array of buttons arranged in keypad style, looked more like tortured remote control designs than hardware meant for gameplay.
These early attempts and others like it were influenced by arcade controls, adopting some variation of the standard joystick. But arcade hardware had more leeway in their controller designs compared to home consoles. Since arcade hardware catered to the needs of a single game, their control interface could be built to suit. *Tapper’s* (1983) controller was a literal beer tap. *Crystal Castles* (1983) provided an embedded track ball to navigate its axonometric structures. *Super Off Road* (1989) used three conjoined steering wheels and accompanying accelerator pedals. *Skydive* (1978) featured ripcord parachute handles as controllers. Home consoles had to play generic, sculpting their controllers to permit a range of game types. When stock joysticks were unable to accommodate software, designers substituted specialized controllers: paddles, keypads, wheels, gloves, guns, cockpits, goggles, microphones, and even balance boards.

Controller design dictated the range of possible play options. In the D-pad patent’s ‘Background’ section, it is evident that Nintendo’s engineers were thinking through a usability problem. How would the addition or exclusion of buttons affect the possibilities of play? They wrote, ‘In using conventional character moving switches, there is a disadvantage that either variety of a game or simplicity in operation be disregarded.’ In other words, if designers limit a game’s input to a single button, it may be easy for the user to understand, but prove boring to play. On the other hand, if the designers increase the inputs by assigning buttons to all possible character movements, one suddenly faces a control monstrosity—imagine individual keys to control each appendage of an in-game character. Likewise, excessive buttons can inhibit the shape of the hardware. If buttons are packed too tight, Nintendo asserts, players will ‘encounter another problem in that two or more than two character moving switches are often pressed simultaneously.’ However, leaving adequate space between buttons yields another compromise: ‘the content of a game organized in the liquid crystal display has to be limited.’

Nintendo offered the D-pad as a solution, stating that its primary objective was to ‘provide a multi-directional switch which can be operated with efficiency in a simplified manner and does not occupy much space for fixing.’ It was an elegant design for the Game & Watch’s cramped body. An arcade-style joystick was not practical for a portable system. A jutting protuberance would make the portable difficult to stow in a pocket or backpack without the risk of snapping off. Once the handhelds shifted to the dual-screen format, there was an added limitation: the unit would not fold closed with a joystick in the way.

The Famicom’s flagship titles from the 8-bit era were platformers, whose dominant trope is movement through 2D space, i.e., running past, jumping across, and climbing past obstacles. The D-pad was the ideal hardware for platform-style movement. It fit comfortably under a single thumb, freeing the right hand to control the action buttons. Until the explosion of 3D videogames, the D-pad was the standard method of character control.
Since Nintendo designed the Family Computer to play home versions of their arcade hits—specifically Donkey Kong—it made sense for them to adopt the controller design best suited for those games. Uemura had disassembled a number of US arcade joysticks and commissioned a handful of prototypes to see how they might transition to a handheld controller. He worried that it would be difficult to fix the joystick in place sturdily and that the protruding controller might potentially injure children. Takao Sawano, a member of Yokoi’s R&D1 engineering team, suggested that they transplant the Game & Watch plus pad to the Famicom. The rest of the team was initially resistant, so Sawano, ‘pulled a lead line out of a Game & Watch and connected it to a Famicom prototype, then invited the development staff to give it a try.’ Once the team felt how responsive the controller was, they decided to ‘port’ the Game & Watch D-pad to the Famicom, albeit in a slightly larger version. Thus the Famicom received its own bit of withered technology: Yokoi’s plus pad.

Curiously, the Family Computer did not feature a matched pair of controllers. The leftmost Controller I included, from left to right: plus pad, Select, Start, B, and A. Controller II omitted the Select and Start buttons in lieu of a small grilled microphone and its accompanying volume slider. The microphone was a late addition suggested by Uemura. While the microphone input translated to a simple binary signal internally, it did have a connecting line to the Famicom’s audio output, allowing sound to pass through to the television speaker. In other words, controller II acted as a makeshift megaphone. Uemura was convinced that ‘there was no way players wouldn’t be entertained simply by hearing their own voices come out of the television set.’ Regardless of its amplification effects, programmers rarely used the microphone in-game. Its two most famous implementations come from the Family Computer Disk System versions of The Legend of Zelda and Kid Icarus. In the former, players may yell into the microphone to defeat the rabbit-eared Pols Voice enemies; in the latter, the microphone is used to ‘bargain’ for shop discounts. Due to limited developer support, the microphone was omitted from both a later Famicom revision and the Nintendo Entertainment System.

The asymmetrical controller arrangement created two consequences peculiar to the Family Computer. First, the repair of faulty or broken controllers, already difficult due to being hardwired, was further confounded by their functional specificity. In other words, one could not swap a spare controller II for controller I since it lacked the full range of inputs. Second, the leftmost player was always the arbiter of menu selection and pausing, the standard domain of the Start and Select buttons. This minor detail actually had cultural ramifications for how games were played socially. For instance, in Japan, a common bit of Famicom lingo related to Super Mario Bros. was the スタート殺し, or ‘Start Kill.’ When player two, controlling Luigi, would leap over a gap, player one would pause the game mid-jump in hopes of breaking player two’s concentration. Once play resumed, player 2 would
be unable to re-acclimate and consequently fall to their death. Today, videogame players call this type of behavior ‘griefing.’

The Famicom controllers were gently rounded on the edges and sized to fit comfortably in a child’s hands. The size also permitted easy access to all the buttons with two thumbs, while the remainder of the hand gripped the controller from behind. The left thumb could drive the D-pad and Select button, while the right thumb bounced between A, B, and Start.

One of Uemura’s stated design goals was to make the controller easy to use while looking at the screen. The goal seems obvious now, but if we think of the Famicom controller as a successor to legacy hardware, we must consider the perceptual rift between controller and screen that a home console creates. Arcade cabinets and handheld videogames have their controls and monitor embedded in the same physical housing. Increasing the distance between screen and input device increases the need for the player to be able to look at the former without shifting their gaze to the latter. As controllers have become more complex, multiplying buttons, joysticks, bumpers, and triggers many times over, this input/output divide has grown more significant. New players unaccustomed to the evolution of controller design face a steep barrier to play (explaining part of the appeal of simplified control schemes seen in mobile touchscreen devices, the Nintendo Wii, and Xbox Kinect). By striking a balance between too few and too many buttons, Nintendo designed a controller that favored tactile control and gameplay complexity.

However, if the Family Computer was meant primarily for arcade ports, why did the joypad include controls beyond the plus pad and a single action button? Part of the answer is certainly engineering foresight. Conceivably, if the Famicom turned out to be a success, Nintendo and their licensees would want to offer a range of games beyond those found in arcades. A single button would turn out to be an unnecessary limitation. Of course, based on Nintendo’s multi-directional patent, ergonomics were also a consideration. Both the Intellivision and ColecoVision controllers had far more buttons than the Family Computer controller, but the former were unwieldy to hold and hard to operate without constantly looking down at the controller to see what one was pressing. But there is also an underlying technical reason why the D-pad, Select, Start, B, A pattern made sense.

All computer platforms have architectural structures that govern the size (or width) of their data units. The Famicom/NES is a member of the 8-bit generation of videogames, alongside the Sega Master System, Atari 7800, and a few others. The \( n \)-bit designator, although sometimes used misleadingly in the history of videogame marketing, tells us a lot about a console’s underlying architecture. Computers make calculations based on binary numbers, a simple numerical abstraction meant to mirror the physical states of a semiconductor gate—0 for ‘off,’ 1 for ‘on.’ The 8 in 8-bit refers to the range of possible values the 6502 CPU can handle in any given chunk of data. An 8-bit binary number, known
as a byte, can have eight individual digits of either one or zero, such as %01001011. Binary is a base-2 number system. As such, an 8-bit CPU can represent $2^8$ possible values, or 256. However, the individual bits of a byte do not necessarily have to represent numbers. In Famicom programming, bits are equally likely to represent the status of a particular object. For any bistable element—that is, something that is either off or on—a single bit is an ideal means to store its current state.

What does this have to do with a Famicom controller? Consider the range of possible inputs that the controller allows. In any given frame, a player may press up, down, left, right, B, A, Select, and Start. There are eight distinct inputs that may have one or two states: pressed or not pressed. Consequently, the status of the entire Famicom controller may be stored in a single byte. It is truly an 8-bit controller.

The initial Famicom manufacturing run included controllers with square, rubber A and B buttons. Nintendo soon discovered that, after prolonged use, avid Famicom players had buttons that were either worn away or stuck in the depressed position. Uemura was shocked, since he recounted that the buttons underwent a 'one-million-punch test' to ensure their reliability. Afterward, Nintendo switched to sturdier circular buttons made from plastic. For Uemura, it was another lesson in the changing context of play:

I don’t think we really understood how people play home video games. We solved the button problem by making the square buttons round, but I think players were using more force with their fingers than when using Game & Watch.⁴⁴

Iwata added that the physicality of Famicom play, such as players swinging their entire body with the controller in hopes of giving Mario’s jump a bit more inertia, likely compounded the wear and tear. Again, that kind of concerted body/controller movement would not be possible if a player had to look at the controller while playing. Jerking the Game & Watch to either side would make it difficult to track the action on the screen.

Another more serious flaw threatened to derail the Family Computer’s launch. Sheff reports that just prior to the Japanese New Year, six months after the Famicom’s debut, there were widespread reports of games causing the console to freeze, what he describes as ‘trouble with one of the integrated circuits that got locked when certain information traveled on certain pathways.’⁴⁵ More recently, however, Uemura explained a different malfunction:

Uemura: Problems were appearing one after the other. One example was the "disappearing ball"...

Iwata: That was a bug which caused the ball or white lines in Baseball to disappear under certain circumstances.
Uemura: We had stretched to make it, so the thermal design was insufficient and it would heat up fast, causing the sprite for quickly displaying the graphics to disappear."

Regardless of whether one or both happened, Nintendo faced a serious problem, one that could sway public opinion away from Nintendo just as they had managed to sell over half a million units. Yamauchi riskily decided to recall all of the faulty consoles and have them repaired to the tune of several million yen. Like Yamauchi's unprecedented Ricoh manufacturing order, the gamble paid off. Nintendo recovered and the Famicom reached the million unit mark eleven months into its life. And, more importantly, the hardware debacle marked the first of many times that the Famicom would succeed in spite of its errors.

**Central and Picture Processing**

As Kokichi illustrated in ‘This is Family Computer!,’ One of the simplest ways to understand how a console functions is to peer beneath the hood and look at the underlying components. Though we can grasp the basic relationships of inputs and outputs externally — controllers plug into ports on the front, cartridges are inserted into the console, the RF connector is attached to the television—the internal arrangement of chips and circuits can help us conceptualize the unique division of labor that takes place when a game is powered on.

Removing the Famicom's plastic exterior reveals several rows of neat, but crowded integrated circuits. The two most prominent ICs are the Ricoh RP2A03G (or 2A03) and the Ricoh RP2C02G (or 2C02). The first IC contains both the Central Processing Unit (CPU) and the Audio Processing Unit (APU) in a single package. The second is the Picture Processing Unit (PPU), the graphical workhorse responsible for translating the palette, pattern, attribute, and name table data stored in memory into video signals that display onscreen. These three processors form the core of the Famicom, handling computational tasks, audio, and graphics. They also administrate both internal connections to hardware (i.e., cartridges) and external interfaces, like the controllers and television. The important point is that the 2A03 and 2C02 are independent entities. Accordingly, they are allotted their own designated regions of memory. The CPU has a small number of control registers that allow data exchange with the PPU and APU (despite sharing the same package as the latter). Doling out tasks to individual processors in such a manner is a wise division of labor. CPUs at the time were extremely limited compared to today's microprocessors. If the CPU had to handle controller input, movement calculations, waveform generation, and graphics display on its own, one or more of those processes would inevitably suffer.
The 2A03 CPU has a 16-bit address bus, meaning that it can access up to 64KB of memory. The address space of a CPU is similar to a geographic address, though much more rigid and uniform. We use a combination of street numbers, city names, and zip codes to locate individual houses in a neighborhood. In CPU space, all houses are equal in size and numbered in sequence from zero to the maximum possible number the address bus can handle—in this case, $FFFF. In decimal notation, the number of available addresses between $0000 to $FFFF equals 65,536, which when divided by 1,024 equals 64. Alternatively, we can express the 16-bit address bus in base-2 nomenclature: $2^{16}$ equals 65,536. Keep in mind that the address bus size does not make the Famicom a 16-bit machine. Despite the 2A03’s ability to access 64KB of memory, it must still send addresses in single byte (or 8-bit) chunks, the upper limit for a single machine instruction. Thus the width of the data bus, rather than the address bus, defines the machine.

The Famicom’s 64KB of CPU memory space is arranged in a specific, unchanging order. Charting that arrangement of addresses and their intended contents in tabular form is called a memory map. The map permits the programmer to locate specific memory locations and understand their designated function. Returning to our geographical analogy, a mail carrier ordered to deliver a parcel to one of 64,000 available mailboxes without an adequate street map would be madness. Similarly, a programmer must know where to deliver his or her data parcels. Furthermore, a CPU’s mailboxes can often perform special functions or return parcels from other regions beyond the CPU’s map. Part of 2A03’s memory map, for instance, includes the 16K of PRG-ROM located on a cartridge, which is physically external to the CPU.

When a player inserts a cartridge, a sophisticated series of communication networks is established between connected but independent hardware components, all dedicated to supervising and implementing their designated tasks. Game code, for instance, cannot execute directly on the PPU; only the CPU runs the source. But the program data is located in a ROM on the cartridge. Similarly, the television is the PPU’s domain. It stores, arranges, and displays the graphics that end up on the screen, scanline by scanline. Similarly, the CPU can not produce sound on its own; it must send data via designated hardware registers in order to instruct the APU.

Famicom cartridges contain their own hardware dedicated to each half of the console’s processing team. Even the simplest cartridges contains at least two ICs fixed to the slender PCB concealed within the plastic case. One—PRG-ROM—contains the program code executed according to the 6502’s set of 256 documented instructions. The other—CHR-ROM—contains the character tiles the PPU uses to populate the screen (figure 1.9).
1.9 The (rotated) PCB for *Family Computer title Nuts & Milk* features one of the simplest cartridge IC layouts. The 16KB PRG-ROM IC is on the left and the 8KB CHR-ROM IC is on the right. The thin unsoldered line visible between the ICs also designates the game’s name table mirroring as ‘H’ for Horizontal. (Source: bootgod, NES Cart Database)

The ‘This is Family Computer!’ manga uses helpful television analogies of script and actor to characterize each chip. Building on that analogy, we can say that the CPU (director) coordinates the script (PRG-ROM) and sends instructions to the PPU (camera) to choose which tiles from CHR-ROM (actors) will appear onscreen. Without the script or actors, the director and camera have nothing to do. In an obvious but important sense, a valid connection between cart and console completes the holistic system necessary to produce a videogame.

Novice programmers who aim to dabble in Famicom/NES code are often surprised that its architecture does not support the high-level languages taught in high school and university. The Famicom has no operating system or firmware to boot into. It relies solely on the instructions stored in PRG-ROM. Code must be written in 6502 assembly language, whose syntax is quite different than modern compiled programming languages like C++ or Java. Those languages bear some resemblance to everyday grammar and even non-programmers can locate recognizable words and even basic commands.

6502 assembly, which is a single abstraction layer above machine code, is cryptic in comparison. Each line of code is composed of a three-letter mnemonic, representing an instruction, followed by an associated numeric value or address. Though I have kept the code examples in the book to a minimum, it is worthwhile to understand the basic structure of 6502 assembly along with a number of common addresses the Famicom uses to get work done.

There is no simple ‘Hello World’ program for the Famicom, since it has no straightforward command to output text to screen, much less a built-in character set.
Words are tiles like any other graphic onscreen. Displaying text requires creating custom bitmap letterforms, waiting for the Famicom to initialize, clearing the contents of RAM, routing the background graphics to the screen, setting the palettes, and so on. We do not have the time or space to devote to such a lengthy process, but a small routine will help us understand the general look and feel of Famicom programming. Displaying a background color is one of the simpler tasks one can program on the Famicom, so we will walk through a few steps of that process. The following code sets the PPU background color to green:

```
LDA #$3F
STA $2006
LDA #00
STA $2006
LDA #$2A
STA $2007
```

Again, the three-letter ‘words’ on the left are instructions that the Famicom CPU understands, while the numbers on the right designate addresses or specific numeric values.

If the dollar signs seem strange, rest assured that they indicate hexadecimal values, not the cost of the instructions. Hexadecimal is a base 16 numbering system. In our common base 10 (i.e., decimal) system, we count from 0 to 9, then move to the tens place, repeat until we require a hundreds place, and so on. Binary, which counts with only 1s and 0s, is a base 2 system. Base 16, as expected, counts each digit place up to sixteen. However, since we do not have single-digit symbols to represent numbers above 9, the six digits needed between 10 and 15 use letters A through F. The $3F value above is equivalent to 63 in base 10 notation. (If you are curious why we only count to 15 in a base 16 system, most computer counting starts at 0.) Learning hex counting is not necessary to read further, as I regularly convert hex values to decimal.

As mentioned above, all data in the Famicom is moved via registers, or hardware memory locations. In-game math, physics, AI, graphics, and sound are all controlled by moving data to and from a handful of registers, including three of the CPU’s special registers: the accumulator, x, and y. The code snippet above uses the accumulator—the primary register used for adding, subtracting, and comparing numbers—to shift a number of values to the PPU. $2006 is one of the 16-bit hexadecimal addresses of the Famicom’s PPU I/O control registers. Changing the palette (or color) of the background is the picture processor’s job, so the CPU must send data via designated addresses and allow the PPU to act on that data appropriately.

The first four statements do the following: LDA stands for ‘Load Accumulator’ with the value that follows. In this case, it is the hexadecimal value $3F (the ‘#’ symbol tells the
assembler to load the literal value, rather than the value stored at that address). $3F$ is the first half of an address mapped in the PPU space, so we have to load the value in two chunks of one byte each. Once we STore the Accumulator (STA) at address $2006$, we then load it with zero and send that value to the same control register. Via four commands, we have told the PPU that we want to send some data to address $3F00$ in the PPU’s memory. And $3F00$ happens to be where background palette information is stored.

With our data destination set, we can then pass along information via the PPU data port, located at register $2007$. To do so, we load the accumulator with a color value ($2A$, or green) then store that value at $2007$. In sum, the CPU tells the PPU, ‘I have the value for green and I would like you to store it in your palette at the following location: address 3F00.’

As this short example illustrates, trivial procedures in today’s programming languages require significant legwork in 6502 assembly. Everything must be coded by hand using a limited set of instructions that shift data between hardware memory locations. Modern programmers might think coding in such a manner is lunacy. But assembly language’s close relationship to the underlying hardware grants programmers extraordinary control over that hardware. There is little abstraction between instruction and physical process. And once one understands the locations and behaviors of a handful of circuits, extraordinary things can happen onscreen.

**Setting the Tables**

While the 2A03 CPU is a minor alteration of the stock 6502 (see ‘Scraped to the Die’ below), the PPU is a top-to-bottom custom chip design. And, since its handles the graphics processing for the system, its peculiarities govern the visuals of Famicom games. When modern game designers aim to give their work a retro, 8-bit Nintendo style, they are typically mimicking the hardware limitations of the 2C02. More than any other component, the PPU defines the look of all Family Computer games.

Today’s PCs and console have luxurious graphical control compared to those of the 1980s. Programmers can manipulate the screen pixel by pixel and pick from millions of possible colors. Hundreds of screens worth of graphics data may be buffered in advance of its appearance on screen. Backgrounds are composed of multiple layered planes that can scroll independently. Raycasting, blitters, and shaders provide realistic lighting, texture, and volume to millions of polygons. Even rudimentary browser-based Flash games have access to impressive particle effects that throw hundreds or thousands of independent elements on screen simultaneously.

The Famicom is far more limited, with a meager 2KB of video RAM (or VRAM)—enough to hold two screens worth of graphics. In 1983, memory was expensive (both
economically and computationally), so getting a convincing image onscreen required judicious use of both the CPU's horsepower and the PPU's capabilities. Displaying Famicom graphics on a television is much like laying tile in a bathroom floor. Tiles are arranged side by side in a two-dimensional gridded array. Joined properly, the tiles may form larger, more complex images. In fact, tile is the common term used to describe the 8x8-pixel 'mini-mosaics' that characterize many early computer graphics systems. If a pixel (short for 'picture element') is the smallest unit of measurement that the PPU can output, the tile is the smallest indivisible element of Famicom graphics that a programmer can manipulate to build the screen.

Of course, the colorful arrangement of gridded tiles used to construct Metroid's Samus Aran, Mother Brain, and Brinstar are not as rudimentary as they appear. Each aspect of an individual tile, from its position to its color, is the result of a tight coordination of data from several disparate locations in PPU memory. There is no single tile repository that contains the final colored sprites and backgrounds one sees onscreen. Sequestering each tile's constituent bitplanes, colors, positions, arrangement, mirroring, and priority allowed Uemura's team to both cut costs and optimize graphics performance. In the end, the PPU must fetch, assemble, and arrange the scattered contents of video memory, both internally and on the cartridge, according to a specific set of rules coordinating which tiles will appear where, which tiles will display above or behind others, how many tiles may appear simultaneously, what colors the tiles will display, and how many tiles will appear on a single horizontal scanline. It does this job relentlessly, according to the CPU's instructions, and synced carefully to the ebb and flow of the television refresh rate. To get a better picture of how a Famicom screen comes together, we will take a look at each region of the PPU memory map in turn.

Unlike the CPU's 64KB of addressable memory (16-bit), the PPU has just 16KB (14-bit). Half of this space is reserved for the CHR-ROM/-RAM that resides on the cartridge, split into two 4KB chunks. Another 2KB, as noted above, is reserved for the contents of VRAM. Technically, the name and attribute tables that comprise this 2KB are known as CIRAM, or 'Character Internal RAM,' which despite their name are external to the PPU but still connected to the video memory bus. The remaining space includes the palettes for backgrounds and sprites, along with mirrors of both the palettes and name/attribute tables.

Sprite and background tiles reside in two pattern tables, the first two segments of PPU memory located between addresses $0000$ and $1FFF$—i.e., the contents of the cartridges CHR-ROM. The PPU has memory allotted for two pattern tables, each 4KB in size, comprised of 256 tiles apiece (figure 1.10).
1.10 The contents of Super Mario Bros.’s CHR-ROM. Each pattern table contains 256 8x8 tiles. In Super Mario Bros., pattern table #0 (left) contains sprites and pattern table #1 (right) contains background tiles. (Emulator: FCEUX 2.1.5)

These are the ‘racks’ of tiles used to pattern our metaphorical bathroom floor (rather than the floor itself). Each tile has its own unique pattern, which may be flipped, repeated, and combined with other tiles to build the game graphics. Visually, sprite and background tiles are identical. Their distinctions lie in how they are used, which pattern table they belong to, and what limitations those locations impose. Therefore, one table must be designated for sprites only and the other for background tiles only. However, the choice of which tile type goes in which pattern table is up to the programmer.

Each tile in the pattern table is described by sixteen bytes of memory. Those sixteen bytes are divided into two 8-byte bitplanes. We can imagine each bitplane as an 8x8 matrix of binary digits. Each digit represents a single pixel of the final tile. If the PPU could only output monochrome graphics, a 1 might designate white and a 0 black. However, Famicom tiles can access up to four colors in an 8x8 pixel matrix: one background (transparent) and
three opaque. To denote four possible colors, a tile’s two bitplanes must be ‘stacked’ atop one another. Consider the example below:

<table>
<thead>
<tr>
<th>Bitplane A:</th>
<th>Bitplane B:</th>
<th>Composite Tile:</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>01111110</td>
<td>00000000</td>
<td>01111110</td>
</tr>
<tr>
<td>01000010</td>
<td>01111100</td>
<td>01222210</td>
</tr>
<tr>
<td>01011010</td>
<td>01111100</td>
<td>01233210</td>
</tr>
<tr>
<td>01011010</td>
<td>01111100</td>
<td>01233210</td>
</tr>
<tr>
<td>01000010</td>
<td>01111100</td>
<td>01222210</td>
</tr>
<tr>
<td>01111110</td>
<td>00000000</td>
<td>01111110</td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
</tbody>
</table>

Each row of Bitplane A and B is a single byte. When the two bitplanes stack and their individual binary values sum, four discrete values are possible: \( A0+B0 \), \( A0+B1 \), \( A1+B0 \), \( A1+B1 \). If we use each of the resulting values to encode a color reference, we can build a four-color tile. In the Composite Tile above, value 0 would be transparent and values 1, 2, and 3 would be three opaque colors. The final tile shows a central square with two borders, each with a different color.

If the Famicom only had a **single** palette of four colors, the two stacked bits from the pattern table bitplanes would be sufficient to describe the color of the tile. However, the Famicom allots memory for eight individual palettes—half devoted to sprites and the other half to background tiles—containing four colors each. In figure 1.10 above, the eight palettes are shown along the bottom edge of the window: background palettes on top, sprite palettes on the bottom. Notice that the first color of each four-color block is identical: the rich blue sky used for the background of *Super Mario Bros.*’s famous World 1-1. **All** palettes must share this same background color (stored at PPU address $3F00 if you recall the previous code example), effectively limiting the color options of individual tiles to just three. In total, the Famicom can display twenty-five individual colors onscreen simultaneously—thirteen in background tiles and twelve in sprites.

Despite its obvious constraints, the shared background color serves as a clever shortcut to ‘erase’ portions of sprite tiles that are meant to be transparent. When a sprite is on top of a background tile, any color bit set to the shared background color will permit the underlying background tile to show through. Mario’s constituent sprites in *Super Mario Bros.*, for instance, use only three colors, drawn from the bottom left sprite palette seen in figure 1.10: red for his hat and overalls; a muted brown for his hair, eyes, sleeves, and boots; and burnt orange for his skin and overall button. All other pixels in the Mario tiles are set to sky blue, but they render transparent, permitting the background to pass behind his body as he moves. Without this palette limitation, any graphic that did not conform to a square shape would have distracting colored borders. Every sprite would look like a literal
tile moving around the screen.

Observant Famicom players may notice that many game graphics appear to violate the four-color rule for sprites and backgrounds. The title character of *Mega Man 2*, seen below in figure 1.11, clearly uses five colors for his head and helmet: black outlines his head, helmet, eyes, and mouth; two distinct shades of blue paint his helmet; his face is a pale peach tone; his irises are white; and an unseen transparent color permits the brown wall to show through.

![Mega Man from Mega Man 2. His character sprite appears to violate the NES's palette limitations. (Emulator: Macifom 0.16)](image)

The secret is sprite stacking. *Mega Man 2*'s programmers reserved a single sprite for the whites of Mega Man’s eyes, which sits atop the sprites that comprise his head. Numerous Famicom/NES games use this trick, though it has disadvantages. Only eight sprites may be visible on a single scanline and sixty-four onscreen simultaneously. Using an additional sprite for the eyes sacrifices a slot from both in favor of a more colorful Mega Man.

In many modern videogame systems, the values stored in palettes reference actual RGB (red green blue) color values that may be mixed in combination to create a given color. The Famicom, however, uses a hue saturation value (HSV) model based on the NTSC color wheel (roughly, blue to red to green to cyan). The byte stored for an individual palette slot is actually a lookup index for sixty-four possible HSV combinations (figure 1.12).
Four bits of the byte encode the hue (tracking from left to right in the image above), two bits encode the value (tracking top to bottom), and the final two are unused. Note that a few values in the black range are duplicates. Value $0D$ is a reserved special case that programmers are warned against using in their games. Its low voltage output is considered 'blacker than black' and NTSC monitors can misinterpret the color as a sync signal, causing the display to malfunction.

Beyond the sixty-four color values above, the Famicom also has a few other hardware tricks that can increase (or decrease) the range of colors. Three bits in the PPU control register located at $2001$ (PPUMASK) are known as the 'color emphasis bits.' Setting each bit causes the entire palette to intensify in either red, green, or blue, while darkening the other colors. In general, the color emphasis bits are used to create special lighting effects or subtle transitions between screens. Another bit in PPUMASK sets the Famicom PPU to grayscale mode, although the name is a misnomer. When the bit is set, all four lower bits of a palette color entry are treated as zero, effectively narrowing the range of colors to three: two identical whites, gray, and light gray. Thus the Famicom grayscale has only two gray values and no black. Similarly, when a color emphasis bit is set while in grayscale mode, the overall tint still changes.

Between the pattern tables and palettes, beginning at PPU address $2000$, reside 2KB of name tables. A name table is the actual 32x30 grid of tiles, selected from the pattern table, used to build the backdrop of graphics seen onscreen. This is the bathroom floor where we arrange the tiles selected from our pattern table racks. Each name table occupies 960 bytes, one for each tile in the grid. However, the name table is not an image stored in memory corresponding to the graphics we see on screen. Rather, each byte of the name table stores the reference to the appropriate tile in the background pattern table. In practice, this saves an enormous amount of memory. With only 256 tiles available to build the name table, many tiles are repeated in large blocks—think of the long stretches of bricks in *Super Mario Bros.*

Each name table is followed by its own 64-byte attribute table. Attribute tables are
one of the trickiest aspects of the Famicom to grasp, but they are one of the most important features of the PPU, since they restrict the number of colors available to the background tiles. Unlike sprites, which can individually choose from among the four available sprite palettes, background tiles are assigned their palettes in groups of four. The pattern tables described above only contain two bits of the necessary four to describe any tile’s color. The pattern table bits tell the PPU which of the four colors within a palette a tile will use, but not which of the four palettes those colors reside in.

The attribute tables supply the final two bits of palette information for the background tiles. Each attribute byte defines the palette values for a 32x32 pixel area of tiles. Since each palette value requires two bits to use in conjunction with the pattern table bits, only four palette values may be assigned within that area. In other words, background palettes may only be assigned to tile regions 16x16 pixels in area, or 2x2 tiles.

Though this technique saves a significant amount of memory, it imposes significant constraints on the shape and granularity of background graphics. In general, games are designed to align their background metatiles, or groups of tiles, along attribute table boundaries. Thus the building blocks of Famicom games are literally square blocks. Shapes that rely upon circular or diagonal edges risk attribute clashes, where the transition between borders of two attribute table entries creates abrupt shifts in color. Snake Rattle N Roll, for instance, uses an isometric perspective uncommon on the NES, making its faux three-dimensional terrain appear as if it is diagonal to the player’s perspective. The off-axis perspective poses problems for background palettes, since none of the terrain aligns along 16x16 pixel attribute table boundaries.
In figure 1.13 above, you can see significant attribute clashes along the edges of terrain objects. The pyramid that abuts the vine-covered platform near the center of the screen has a portion of its point colored green. The transitions between the tops and faces of waterfalls exhibit sharp transitions between varying shades of blue. Likewise, the gold mechanism near the lower right corner transitions to green near its base.

Avoiding attribute clashes requires either careful planning near attribute boundaries or making concessions to color and terrain structure. *Solstice* adopts the same isometric perspective as *Snake Rattle N Roll*, but limits its level palettes to variations of a single color (figure 1.14).
Rooms in Solstice are limited to a few shades of a single color to avoid attribute clashes. (Emulator: Macifom (0.16))

Doing so sacrifices the overall vibrancy of its levels, but avoids gaudy attribute clashes. It also allows the sprites to appear that much more vivid against the subdued backdrops.

Sprites similarly derive the final two bits of their palette values from a separate source—sprite OAM—but, as mentioned above, each sprite is assigned its own palette. Subsequently, sprites are often used in atypical ways to bolster the colors capabilities of background tiles. The most common use of this technique is seen in title screens and cinematic cutscenes, which require minimal object interactions (freeing up sprites) but formidable graphics skills. The title screen, after all, is the player’s first impression of a game, so it is regularly reserved for the game’s best and most complex artwork.
1.15 The Contra title screen uses combined sprite and background tiles to draw more colorful character portraits. Note that the red pause button is part of the emulator GUI, not the in-game graphics. (Emulator: FCEUX 2.1.5)

In figure 1.15 above, interleaved sprites and background tiles are used in conjunction to build the characters portraits on Contra’s title screen. Without using layered sprites for the hair, eyes, cigarette, and tank top, Contra’s programmers would not have been able to
create such smooth transitions between colors. They would have either had to reduce the overall color detail of the two characters or leave distracting attribute clashes between their skin and the remainder of the portraits.

As we will see in later chapters, Famicom programmers regularly used background and sprite tiles ‘against type.’ What we often expect to be background is composed of sprites, and vice-versa.

**Of Rasters and Regions**

Allow me to reiterate an obvious fact: the Family Computer requires a television.

Though I prefer the concatenated form of *videogame*, there is a reason that the original construction ‘video game’ has its adherents: the words succinctly describe the medium’s dual components. A videogame is literally a game *generated from* and *played on* a video display. And any console-based platform study that neglects the ‘video’ in favor of the ‘game’ is shirking half of the equation. The carefully orchestrated interplay between television and console is crucial to the Famicom’s function.

If we keep in mind both the etymological root and the original intent of television, to *see at a distance*, the videogame is something of a parasite. Television was designed for broadcast, to transmit in real time a two-dimensional image from one location to another. The videogame hijacks this purpose—the image is not produced remotely then captured on camera, but generated internally by a graphics processor then rendered to the display. In either case, the process of getting an image on screen takes place via an elegant synchronicity of optics, chemistry, and electrical and mechanical engineering.

Until the recent rise in popularity of HDTVs, the dominant display technology of arcade and console videogames was the CRT monitor, named after its functional core, the Cathode Ray Tube. The CRT is similar to a vacuum tube—during manufacture, its interior oxygen is burnt away, creating a highly pressurized interior seal.⁷⁴ At one end of the tube is a barium-coated cathode that, when heated, emits negatively-charged electrons. Positively-charged anodes attract, accelerate, and focus the electrons into a narrow beam. The side of the tube opposite the cathode, which widens considerably to a large curved surface, is the portion we call the television screen. The *rear* (or internal) side of the screen is coated with a luminescent phosphor material. When the gun fires its accelerated beam of electrons, they strike this material, causing the phosphor’s electrons to momentarily become unstable, then settle down. The settling down process emits a photon at a wavelength that human eyes perceive as light.

A beam shot directly down the cathode tube would generate a single illuminated point, not an entire image. In order to draw a complete picture, the electron beam must be diverted from its single vector in a predictable, consistent pattern. Along the exterior neck
of the tube, magnetic coils deflect the beam’s path based on alternating voltages. Two electrical signals, each synced to a separate oscillator, play complementary roles in this process: one guides the beam horizontally, the other vertically. The television image is drawn line by line as the electron beam sweeps left to right and top to bottom, much like a hand composes a letter, repositioning each time it reaches the right margin and ‘resetting’ to the top when it fills the full page.75 Thus the beam starts at the upper left, draws the first line, drops down a line while the horizontal position resets to the left, then draws the next line. Once all possible scanlines are drawn, the vertical position of the beam is reset to begin the next image. The electron hand pens its letter dozens of times per second. Displays that use this technique are called rasters, based on the Latin rastrum, or rake, which describes the overall pattern of lines drawn across the screen.

Color television requires a more elaborate mechanism. The electron gun multiplies by three, each assigned to a single color—red, green, and blue. The guns do not shoot color or light; instead, three individual electron beams are deflected at slightly different angles by a small perforated plate called a shadow mask. Each beam strikes a grouped array of phosphors, known as triads, that emit their assigned color. Television uses triads as additive primaries, meaning that the three colored light sources are blended to produce the desired hue.76 When all three guns fire at once, for instance, the result is white; a single gun may light green alone.

You can clearly see color triads by ignoring your parents’ advice and sitting directly in front of a CRT screen. In figure 1.16, the triads are visible within dozens of miniature columns of rectangles, like so many vertical bricks stacked one after another.
The columnar configuration is a result of a PIL, or precision-in-line, tube. In a PIL tube, the electron guns are mounted in a straight line, rather than in a triangle, and the shadow mask is perforated with vertical slots rather than circles.

Triads do not produce the clean, squared pixels we associate with 8-bit videogames. In fact, even at the focal length depicted above, it is impossible to discern pixels at all. The illuminated bleed of CRT triads both softens the edges of pixels and creates pleasing blends of colors that derive from the display’s inaccuracy rather than the tile’s designated palette. The ‘Xs’ and numerals above, drawn from The Legend of Zelda’s inventory icons, appear to have a golden tint around the upper edges and blue shadows beneath. On an LCD screen or in an emulator, the same tiles are simply white pixels on a black background.

The flaws of the CRT contribute to a richer final image. Carefully choosing pixel color and placement creates perceived hues and shades that the PPU does not actually output. A pixel is the fundamental graphical unit of the PPU, but not of the display. CRT televisions have their own subatomic particles—phosphor triads—that do not adhere to the strict gridded geometries associated with pixel graphics. That is why Famicom/NES games look
so poor on modern televisions: CRTs do not have a fixed resolution. Multiple triads can constitute a single pixel dependent on the size of display. LCDs, DLPs, plasmas and the like have a fixed resolution, but their individual pixels are so small that if we mapped the 256x240 pixel matrix of the NES PPU to a modern HDTV, the resulting screen would be miniscule. Therefore, individual pixels must be scaled according to the TV’s resolution, resulting in a blurry, distorted image.

Converting a two-dimensional image into electrical impulses then reproducing them via variations in luminance on phosphor-coated glass is impressive in itself, but getting that picture to move relies on tremendous electromagnetic speed and a particular ‘drawback’ of the human eye. Cinema, television, and animation all rely upon persistence of vision, a trait of human optics that causes images to stick around, ghost-like, after we have seen them. Though this optical residue is brief—about a tenth of a second—it facilitates a useful illusion. If enough images are strung along in sequence before that residue dissipates, human perception physiologically reassembles them into movement. Cinema (and various proto-cinematic forms) figured this out in the nineteenth century, so the basic concept was understood by the time television emerged. But the young upstart medium required a bit of extra technological legwork to pull off the same visual feat.

Despite the ambitions of some contemporary videogame designers, there is a reason that videogames are not called cinema games, as the means of capturing and reproducing images in either medium are fundamentally different. A film camera captures its images like rapid-fire photographs. Entire frames are sampled from a scene many times per second. Proper playback requires those frames to be strung together in sequence at the exact rate they were captured. Raster-based videogames, in contrast, do not output a full-frame image; instead, they are constructed from the rake of individual scanlines described above. Television and videogame producers also speak of frames (often in relation to their rapid and consistent rates, a gold standard of modern game design), but it’d be a misnomer to label them as the atomic unit of either medium. Frames are more of a practical human convention, a unit that makes sense to the speeds that our senses can handle. At the microprocessor level, units shift in scale; the frame is suddenly a monstrous amount of time, so it makes more sense to speak of individual scanlines, pixels, or clock cycles.

Before television was a mature technology, the number of times a full vertical scan, or field, took place per second was not standardized. When single-digit field rates were the norm, television had noticeable flicker; the electron gun simply could not generate enough fields per second for persistence of vision to kick in. Field rates progressively improved, but early on a rudimentary form of compression called interlacing helped alleviate noticeable flicker. Interlacing is analogous to ‘shoelacing’—in the same way that one might alternate laces along the tongue of a shoe in order to secure it tightly, an interlaced image is drawn with alternating scanlines during a single field. Only half the image needs to be drawn at a
time (thus: compression), but it then takes two full fields of interlaced scanlines to compose a single frame of video. Again, due to phosphor’s luminous residue, human eyes do not perceive two alternating sets of scanline window shades, but a single unbroken image. And since the CRT’s phosphor luminance bleeds around the edges, it smooths the gaps between scanlines, effectively masking the raster on early television sets. The drawback, however, is that rapid motion can often happen between fields, impacting the clarity of the composite frame.⁷⁹

Once early television experimenters settled on the minimum threshold of fields necessary to overcome flicker,⁸⁰ standard field rates began to emerge around technological, economic, and political interests. The three established standards were NTSC, PAL, and SECAM.⁸¹ The first, a monochrome system named for the National Television System Committee, was adopted in the United States in 1941, then later revised to a color standard in 1954 (called NTSC-2). However, NTSC had color instabilities (necessitating manual hue controls) that European interests sought to improve. Europe adopted the Phase Alternating Line color encoding standard in 1963, eliminating NTSC’s hue inefficiencies—with the added side benefit that it kept non-European sets out of the domestic television market. The French forged their own path in the early 1960s (also correcting NTSC’s color limitations), introducing SECAM, or Séquentiel couleur à mémoire.

Though these standards branch into myriad subdivisions that describe precise technical differences, the details relevant to the Famicom involve geographic divisions and their associated field rates. Japan adopted the NTSC standard, which eased the Famicom’s later export to the United States. The bulk of Europe and Australia adopted the PAL standard, with pockets of SECAM sprinkled throughout (obviously in France, but also in other parts of Eastern Europe). The discrepancies required a hardware revision for Nintendo’s European console launch, so the PAL-compatible NES debuted after the US version with a number of important differences that affected how video displayed.⁸²

One modification handled the television standards’ differing vertical scan rates. An NTSC television draws its frame at approximately 60Hz, or sixty individual top-to-bottom electron beam trips per second. The PAL standard is slightly lower—approximately 50Hz—but the discrepancy has a huge impact on graphics rendering and program timing. Much of the important work of displaying a videogame is done while the electron beam is ‘at rest,’ either resetting from drawing a horizontal scanline or resetting to the top after it reaches the bottom. By the time we see the results on screen, most of the preparation work has been done. Graphics updates are ideally queued and ready prior to the beam’s incessant drawing duties. Those horizontal and vertical ‘reset periods,’ or blanks, are key to the Famicom’s operation.

Achieving a stable image on screen is a significant feat. The synchronization involved is a precision art, as the electron beam is not only deflecting at speeds the human eye
cannot perceive, but the voltages controlling horizontal and vertical positioning must operate at separate frequencies. Those of us alive in the 1980s and prior may be more attuned to the delicate choreography required to keep a television’s synchronization consistent. ‘Rolling’ images were as common to the CRT generation as compression artefacting is to the flatscreen generation. When the vertical syncing frequency fell out of lockstep, the TV image wrapped around the top and bottom borders of the screen, separated by a moving black bar—the vertical blank made visible. Similarly, when horizontal sync went awry, the image ‘tore’ along jagged diagonals.

Though the vertical and horizontal blank (VBLANK and HBLANK) describes distances, their importance to programmers is temporal. In other words, it takes time for the gun to travel—time necessary to update or move the proper graphical elements into place before the screen starts to draw again. But it is a slender margin that demands careful preparation. Updating the PPU while the scanline is actively being drawn is a Famicom programming no-no, resulting in glitches on screen. Any violations to this rule must be done deliberately and with precise timing. The PAL NES’s lower refresh rate grants programmers additional affordances: since the gun resets ten fewer times per second, it can make its vertical ascent at a more leisurely pace. In fact, the PAL VBLANK is a full fifty scanlines longer than NTSC.³⁸

A PAL television also has more graphical real estate than its NTSC brethren. This is not a discrepancy of the PPU—on either PAL or NTSC systems, the processor outputs a full 240 scanlines of graphics. The difference stems from overscan, a variation in picture visibility common to CRT displays. Overscan describes the area around the edge of the video frame that is not visible based on variations in the individual monitor. This may be due to a television’s physical bezel or the curve of the cathode ray tube itself. Due to the incredibly high pressure inside the CRT, early televisions were tiny, curved, and typically circular, all necessary constraints to keep the tube from imploding. As manufacturing tolerances improved, shapes resolved into the more familiar rectangular 4:3 aspect ratio, curves decreased, and bezels receded. But this process took decades. Television and videogame producers learned to keep important content away from the edges of the screen, lest it be invisible to some fraction of viewers. On NTSC sets, up to eight pixels of the upper and lower borders of the PPU’s output are lost to overscan, reducing its visible scanline count to 224. PAL sets lose a single upper scanline and two pixels on the left and right.

Regional refresh rate differences create an ironic kinship between consoles that share few external similarities. The Family Computer and the US Nintendo Entertainment System do not appear to be blood relatives, but internally they have identical CPUs. Famicom systems play fine on US televisions. The only modifications necessary are stepping down the voltage for the power supply and finding the proper broadcast channel for playback
(Channel 2/3 in Japan is not equivalent to channel 2/3 in the US). In contrast, PAL NES consoles, which are externally identical to their US brethren, contain a CPU variant known as the Ricoh 2A07. To account for the 10Hz discrepancy in refresh rates, the 2A07 has a slower clock speed (1.66 MHz) than the 2A03 (1.79 MHz). In practical terms, PAL game cartridges played on an NTSC console will both play back faster and have their sound pitched up in frequency (the reverse is also true). In some cases, the speed difference is bearable; in others, it renders a game unplayable. If the game code relies upon the increased VBLANK time to run game logic or make graphics updates, PAL games will glitch or freeze entirely. So despite the apparent visual hardware compatibility of PAL consoles and cartridges, their internal differences have a meaningful impact on gameplay.

**Sprite Nouns and Adjectives**

The PPU includes a distinct memory area, with its own address space, reserved for sprites. Object Attribute Memory (OAM) is 256 bytes wide—large enough to hold sixty-four sprites, the maximum the PPU can render onscreen simultaneously. But if the pattern tables hold the actual 8x8 or 8x16 bitplanes that define a sprite tile, what is stored in OAM? And why, if we have 256 bytes of available space, do we only have room for 64 sprites? Why do they occupy more than one byte each?

Due to their onscreen capabilities, sprites require more computational overhead to maintain. Mario’s constituent tiles are more agile than a static bit of pipe. Consequently, the PPU requires additional information to define a sprite beyond its constituent on and off bits. This is why OAM is called ‘Object Attribute Memory’ rather than ‘Object Memory.’ The visible sprites are not housed here—rather, the memory is occupied by a display list of descriptive qualities. We might say that if the pattern tables contain ‘nouns,’ the OAM contains ‘adjectives’ that complement the nouns. Without adjectives, sprites are simply tiny mosaics with no distinguishing characteristics.

Within OAM, each sprite is allotted exactly four bytes worth of adjectives. One byte is devoted to the tile index value, which designates which sprite from the appropriate pattern table will occupy that slot. This is similar to the name table, since a single pattern tile may be indexed multiple times in OAM. This is useful in cases where the same tile is needed multiple times. Each of the four fielders visible during pitching and batting in Famicom Baseball, for instance, look identical since they use the same pattern tiles. However, each of the four tiles comprising each of the four players occupies its own position in OAM—a total of sixteen out of sixty-four slots.

Two additional bytes are devoted to coordinate positioning. In addition to knowing which pattern tile we will use, we also need to know where it is. Again, a byte can hold unsigned values between 0 to 255, so our theoretical x- and y-ranges cover 256 pixel units.
However, the Famicom PPU, regardless of region, produces a 256x240 pixel display. Thus the visible range for the y-coordinate is $00$ to $EF$ (or 240 scanlines). The PPU plots its coordinates like a Cartesian graph flipped along its x-axis; as the y-coordinate increases, sprites move down the display. A sprite positioned beyond $EF$ is out of the PPU’s bounds.

In practical use, the visible area may be slightly tighter, depending on the region. As we saw in the previous section, overscan may clip up to sixteen total scanlines from the upper and lower borders of an NTSC CRT but only a single scanline from a PAL display. The discrepancy creates divergent programming concerns according to the region: for NTSC games, important player information such as status bars should not be placed too close to the edges, lest they are cropped by the television; for PAL games, any sprites that need to be temporarily hidden should be properly placed beyond the PPU’s boundaries, not simply within the presumed overscan area.

In modern programming environments like Flash, visible objects may be created, cloned, moved ‘onstage’ and off, and destroyed when needed with minimal programming effort. In OAM, however, all sixty-four slots are filled at all times, whether with the sprites the programmer intends or simply residual data from powering the console on. This poses a problem when the programmer does not want a sprite to be seen. Sprites may be toggled on and off with a write to PPU register $2001$, but doing so is an all or nothing affair—sprites are either all visible or all invisible. There is no picking and choosing. In lieu of turning sprites off, programmers commonly use the safe areas around the borders of the display. As explained above, updating a sprite’s y-coordinate byte to the range between $EF$ and $FF$ makes the sprite effectively invisible, since it is beyond the PPU’s reach. A common initialization step in Famicom games involves zeroing out OAM, often by literally inserting zeroes into memory, conveniently assigning all tile indices and coordinates to $00$.

However, misunderstanding the peculiarities of OAM can lead to display errors. Beyond knowing the sprite’s x- and y-coordinates, a programmer must also understand its registration. Though a sprite is positioned as if it is a one-dimensional point, the visible object has a two-dimensional area. Imagine placing an 8x8mm stamp in a collection book that happens to be organized as a grid. When you want to place a stamp at position 0,0 (the upper left), what portion of the stamp are you specifically referencing? Its center point? Its edge? The sprite’s registration describes the coordinate reference point used for positioning. In our conceptual stamp collection, this might be the position of our thumb and pointer finger that we use to grasp and affix the stamp. The PPU acts like a lefty—it ‘grasps’ its sprites by their upper left pixel.

Why is this important? If a programmer updates both of a sprite’s coordinate bytes to $00$, the sprite will shift to the upper left of the screen. Or more accurately, its upper left pixel will reside at that point, while the remainder of the sprite extends to coordinate 8,8 or 8,16. Depending on the television’s overscan, the sprite may still be visible to the player.
Some careless programmers used the 0,0 coordinate to hide unused sprites from view, thinking they would be covered by overscan. Tengen’s NES port of Pac-Man, for instance, has a conspicuous yellow sprite fragment floating above the upper left of the maze (see figure 1.17).

1.17 In Tengen’s Pac-Man, the unused sprite stack is visible in the upper left corner of the screen. Note that in some emulators that simulate CRT overscan, this portion of the screen may be clipped. (NES-101 CRT capture).

Though the errant graphic appears to be a single tile, it is actually a stack of all OAM sprites not currently in use for gameplay. Since OAM governs the tile ID and coordinates, the sprite stack is shifted to the upper left origin and pattern tile $00$—the upper left corner of Pac-Man’s body—is displayed. The proper technique would have been to set the vivisected
sprites’ y-coordinates to a value below scanline 240.

*Baseball’s* programmers, in contrast, moved all unused sprites to a reliable safe location (figure 1.18).

**Figure 1.18** In *Nintendulator’s* PPU Debugger, the bottom left area labeled ‘Sprites’ displays the contents of *Baseball’s* OAM. On the bottom right, you can see the ‘Details’ of an unused sprite whose y-coordinate is set to F0. The last thirty sprites in OAM are positioned offscreen. (Emulator: *Nintendulator* 0.975)

While the tile ID and x-coordinate are zeroed out, the y-coordinate is assigned value $F0$, well beyond the PPU’s rendering range.

OAM’s remaining byte controls multiple attributes. Two bits are devoted to the sprite’s palette, one bit controls sprite priority, and another two bits control horizontal and vertical flipping.¹⁶ Sprite priority determines the sprite’s location in OAM. Higher priority sprites have a lower slot number. Sprite 0 (a special case that we will discuss in chapter 3 and 4) has highest priority while sprite 63 has the lowest. In practical terms, that means that sprite 0 will always appear on top of all other sprites. In figure 1.18 above, the first
four OAM slots (0-3) are occupied by the umpire's tiles. The catcher's four tiles directly follow. As expected, when these sprites are stacked near home plate, the umpire appears behind (i.e., on top of) the catcher, consistent with the player's point of view.

Also notice that the umpire's tiles appear to be duplicates of only his right side, while onscreen we see the proper mirrored body. The umpire's body does use only two pattern table tiles (SF4 and SF5), but the first two sprites in OAM have their horizontal flip bit set. Once positioned properly, the two flipped tiles form the proper mirrored image.

**Scraped to the Die**

The Famicom CPU core is a modified version of the 6502, an 8-bit processor first produced in 1975. Eight years prior to the Famicom's introduction, the 6502 transformed the microprocessor market—it was powerful, easy to program, and cheap, debuting for an astounding $25. MOS Technology's bargain basement chip was overwhelmingly attractive to PC and videogame manufacturers looking to reduce costs and bring PCs to the mass market. Atari, Commodore, Apple, and Nintendo all launched successful platforms based on the 6502. MOS Technology, of course, did not directly manufacture the Famicom CPU. As we have seen, Ricoh, a so-called 'second source' manufacturer, licensed the rights to produce and sell the 6502. And, as its name suggests, the 2A03 was not simply a stock 6502.

In custom microprocessor production, clients are able to cut features in order to reduce individual chip costs. Atari, for instance, used a 6502 variant called the 6507 in the VCS. The microprocessor had a streamlined package that reduced its addressable memory to 8KB and eliminated the interrupts. The modification was a worthwhile cost concession in the late 1970s, but proved limiting as the console aged and competitors arose. Nintendo opted for minor revisions. The sole omission from the 2A03's design was the 6502's binary coded decimal mode, or BCD. The significant modifications were the addition of the onboard APU.

In 1976, Commodore acquired MOS Technology and entered the US PC market with gusto, leveraging their ownership of the inexpensive chip to undercut competitors' prices. Commodore's CEO Jack Tramiel was highly defensive against the threat of Japanese companies 'invading' the US PC and gaming markets. His stance was a mixture of post-WWII xenophobia and competitive sour grapes. The Commodore 64 had unsuccessfully challenged the popular MSX standard in Japan, despite being a cheaper and technologically superior machine. Commodore simply could not compete in the face of the third-party groundswell supporting MSX. Tramiel decided that if Commodore could not break into the Japanese market, they would shore up US borders instead.

Facing such staunch opposition could have caused significant problems for Nintendo.
Had Commodore known that Nintendo would bring the Famicom to the US in 1985, they might have blockaded the 6502 license, or otherwise inflated the manufacturing price. It is unclear whether Commodore caught wind of Nintendo’s console plans, but after the NES’s stateside release, Commodore’s engineers were convinced that Nintendo had illegally skirted their microprocessor patents. As Bagnall reports, it took a bit of reverse-engineering to discover Nintendo’s ‘modifications’:

[Commodore 64 programmer] Robert Russell investigated the NES, along with one of the original 6502 engineers, Will Mathis. “I remember we had the chip designer of the 6502,” recalls Russell. “He scraped the [NES] chip down to the die and took pictures.”

The excavation amazed Russell. “The Nintendo core processor was a 6502 designed with the patented technology scraped off,” says Russell. “We actually skinned off the top of the chip inside of it to see what it was, and it was exactly a 6502. We looked at where we had the patents and they had gone in and deleted the circuitry where our patents were.”

Although there were changes, the NES microprocessor ran 99% of the 6502 instruction set. “Some things didn’t work quite right or took extra cycles,” says Russell. […]

The tenacity of the Japanese was obviously formidable. Russell offers an opinion on why the Japanese elected not to purchase chips from North American sources. “They looked at the patents and realized that we weren’t going to let them come over and sell against us,” he says.⁹¹

Ed Logg, veteran arcade programmer for Atari, made a similar observation while working on the ill-fated Tengen port of Tetris. Asked about the ease of coding for the NES, he said:

Yeah, it was pretty similar...well, [Nintendo] basically used our patents. They violated Atari’s patent while they were suing us, so it was the basic same scrolling algorithm and such. So it was pretty much identical to what we were dealing with. Most of the difficulty came from figuring out what registers and bits did what, and when.⁹²

In either case, their discoveries were too late. By the time Russell, Mathis, and Logg could take a close look at its silicon, the NES had already arrived on Western shores. Whether Nintendo actively stole patented technology or not is speculation. Yamauchi-led Nintendo certainly had a reputation for getting what they wanted at the right price. But claiming that Nintendo used the ‘same scrolling algorithm’ is hardly a damning indictment. The pace of innovation in videogame technology was as tumultuous then as it is now. It is equally likely that Uemura and team simply got caught up in a highly-competitive microprocessor business where engineers would inevitably overlap in technique and design.
The truth is in the details. Calling foul on the Famicom/NES CPU’s violation of US patents both overestimates the extent of the 2A03’s cooperation in generating the final videogame image and misunderstands the Famicom’s close relationship to the ColecoVision. The CPU is crucial to a console’s function, of course, but as we have seen its relationship to graphics rendering is indirect. The PPU has an equal stake in the Famicom’s design. If anyone had a feasible bone to pick with Nintendo, it was Coleco.

Despite their ultimate choice to forgo partnerships with Coleco and the Z80, Nintendo clearly drew inspiration from the former’s video display processor (VDP) design. The ColecoVision’s VDP TMS9918 (also standard in the popular Japanese MSX PC standard) has several key features that resurfaced in the Famicom Picture Processing Unit. The TMS9918 acts as a mediator between the CPU and VRAM, permitting the former access to the latter via eight write-only control registers (0-7) and a single read-only status register. The control registers perform a number of graphics-related tasks, such as setting the sprite size (8x8 or 16x16), enabling the VDP interrupt, selecting among text or graphics modes, and designating the name table base address.

The ColecoVision VDP’s read-only status register functions remarkably similar to the Famicom’s PPU Status Register ($2002). Bit 7 of the VDP Interrupt Flag triggers ‘at the end of the raster scan of the last line of the active display,’ effectively marking the beginning of VBLANK. The same bit in $2002 monitors whether the PPU is currently in VBLANK or not. The VDP status register’s bit 6 is the Fifth Sprite Flag (5S), signaling when five or more sprites are concurrent on a single scanline. Bits 0-4 then store the sprite flag that triggered 5S. Bit 5 of $2002 also records sprite overflow, although the Famicom had a higher maximum (8) and kept no record of which sprite had exceeded the limit. Finally, bit 5 of the VDP status register records the ‘Coincidence Flag,’ a catch-all collision detection that triggers if any two sprites onscreen have an overlapping pixel. This limited functionality works for only a small range of gameplay types where coincidence events are discrete. In cases where multiple collisions might happen simultaneously, its use is less practical. The Famicom has a coincidence flag of a different sort, governed by the status of sprite 0, the first entry in Object Attribute Memory. Though we will discuss its use in greater detail later, it functions as follows: whenever a non-transparent pixel of sprite 0 overlaps with a non-transparent pixel of a background tile, bit 6 of $2002 sets. Like the VDP’s limited collision detection, sprite 0 has little practical import for conventional object collisions, but proves useful for scanline timing and raster effects.

Like the ColecoVision, the Famicom supports variable sprite size selection, though the available choices are 8x8 or 8x16. The Famicom also mirrors many of the ColecoVision’s key technical terms: Pattern Name Tables in the latter are simply name tables in the former; the Sprite Attribute Table is Object Attribute Memory; and the Pattern Generator Table is the pattern table. Compare these similarities to the C64 or Atari VCS. The former’s
VIC-II VDP share no such affinities with the Famicom’s PPU, beyond general concepts like sprites or VRAM. Likewise, the Atari VCS’s TIA is a wholly different beast, bearing an altogether different and more intimate relationship to the scanning electron beam than the Famicom or C64.97

The three consoles shared a similar CPU, true, but accusations that Nintendo simply made a clone system do not hold much technical water. Thanks to unlike VDPs, the Famicom, C64, and VCS are fundamentally different machines. The ColecoVision is the Famicom’s true elder step-sibling, related by looks if not by blood.

---

1 Quoted in Sheff, *Game Over*, 28.
4 Sheff, *Game Over*, 29.
6 The Famicom name is actually trademarked by Nintendo’s longtime hardware partner Sharp. See Masaharu (trans. Tanner), “Part 8 – Synonymous With the Domestic Game Console.”
7 Masaharu (trans. Tanner), “Part 8 – Synonymous With the Domestic Game Console.”
8 GAMECOM and Famicom have obvious similarities in English. However, in Japanese the former is rendered in four characters ゲームコン pronouned gamekomu (gah-may-ko-mu). To maintain the English-style pronunciation of ‘GAME-COM,’ Nintendo would have opted for the five-character ゲームコン, pronounced geemukomu (gay-mu-ko-mu). My thanks to Aria Tanner for pointing out the subtleties of the Japanese pronunciations.
10 Sheff reports Yamauchi’s admonition to stave off competition for ‘at least one year’ (*Game Over*, 29), while Uemura himself says three. See Nintendo of America Inc. “Iwata Asks - Volume 2: NES & Mario,” 1.
14 This was not an antagonistic move on Yamauchi’s part. Sharp and Nintendo continued to work in close partnership for many years. Sharp, for instance, manufactured the officially-licensed Family Computer / Family Computer Disk System all-in-one console called the Sharp Twin Famicom.
18 Sheff, *Game Over*, 32.
21 Masaharu (trans. Tanner), “Part 8 - Synonymous With the Domestic Game Console.”
24 Ibid.
25 W., Dan et al. “Exclusive Interview with Donkey Kong Creator Shigeru Miyamoto.”
"This design conceit would appear again in Lance Barr’s design of the Nintendo Entertainment System (see Chapter 3).

Masaharu (trans. Tanner), “Part 8 - Synonymous With the Domestic Game Console.”

Thanks to Aria Tanner for the translation.

Masaharu (trans. Tanner), “Part 8 - Synonymous With the Domestic Game Console.”

The expansion port also has an instructional sticker: ‘Please don't touch the terminals using fingers or metal objects. This will cause failures.’ Thanks to Aria Tanner for the translation

Ibid.

Sheff, Game Over, 33.

Kodansha Ltd. Best of Japan, 182.

See Maru-Chang. “Nintendo Hard Number: HVC.” and Famicom World, “Product Codes: HVC.”

Third party licensees adopted variations of Nintendo’s abbreviations: Hudson Soft’s Nuts & Milk (1984), for example, bore the code HFC-NM on its sticker label; Enix’s ドアドア (Door Door, 1985) was EFC-DR; Konami’s 悪魔城ドラキュラ (Akumajou Dracula, 1993) was KON-RV003; and so on. Bootgod lists these codes in the NES Cart Database under ‘Catalog ID.’

Masaharu (trans. Tanner), “Part 8 - Synonymous With the Domestic Game Console.”


Masaharu (trans. Tanner), “Part 8 - Synonymous With the Domestic Game Console.”


An equally technical manga came with the Family Computer Disk System.

Nintendo Ltd., This is Family Computer!, 12.


The dual-screen portable format would return twenty-five years later as the Nintendo DS handheld.

V., “Nintendo Ultra Hand (ウルトラハンド, 1966).”

For these and other Gunpei Yokoi gadgets, see V., “Label: Gunpei Yokoi.”

Crigger, “Searching for Gunpei Yokoi.”

Inoue, Nintendo Magic, 125.

Ibid., 123. Also see BOCTOK Co., Ltd., Bit Generation 2000 “TV Games.”, 58-9.

Inoue, Nintendo Magic, 123.

During Yokoi’s career, he oversaw the Donkey Kong arcade conversion, the Virtual Boy, Dr. Mario, Yoshi’s Cookie, Duck Hunt, and the Robotic Operating Buddy.

Shirai, “Multi-Directional Switch.”

The ‘no-button’ input of iOS games continually face this problem. On-screen overlays of virtual controls feel imprecise and block valuable screen real estate. The best and most successful iOS games (e.g. Angry Birds, Tiny Wings, Infinity Blade) manage to minimize input to the swipes, pinches, and taps native to the platform rather than relying on artificial console conventions.

The Flash/iOS game QWOP (2010) is a comically masochistic attempt to do just that. In the game, you use the four titular keyboard keys to control an Olympic runner’s thighs and calves. See http://www.foddy.net/Athletics.html

Shirai, “Multi-Directional Switch.”

Ibid.
"Ibid.
"Ibid.
The TurboGrafx-16, for instance, despite being touted as a 16-bit system, still contained an 8-bit Z80 CPU. The 16-bit described its dual GPUs.
Sheff, 35.
“If you are curious how 65,536 ‘rounds down’ to 64KB, this is a convention of byte counting in computational contexts, reflecting the asymmetry of base-2 and base-10 numbering systems. 1KB equals 1,024 bytes rather than 1,000.
Technically VRAM includes the CHR-ROM/-RAM present on the cartridge as well, so here we are using the term VRAM to describe the Famicom’s ‘built-in’ video RAM.
See “Clarification on OAM and palette locations,” NesDev.
Certain hardware tricks, such as timed mid-screen palette swaps, can push past the twenty-five color limit.
See Korth, “Everynes: Everything about NES and Famicom” and “PPU palettes,” NesDev.
Korth, “Everynes
Covell, “NES Technical/Emulation/Development FAQ [ver. 1.7].”
See Watkinson, Television Fundamentals, 10-4.
Whether this hand conforms to left-to-right or right-to-left writing systems is a matter of perspective. From the gun’s point-of-view, it is the latter; from the television viewer’s perspective, it is the former.
Watkinson, Television Fundamentals, 60.
The underlying technological challenge related to persistence of vision in television was understood quite early on: ‘Both television and the cinematograph depend entirely on persistence of vision; without this phenomenon both would be impossible... But in the case of television, as at present practiced, persistence of vision is relied upon to a far greater extent than in the case of the cinema...’ Dinsdale, First Principles of Television, 14.
See Watkinson, Television Fundamentals, 35-7.
See ‘Ferry-Porter law’ in Roberts, Dictionary of Audio, Radio and Video, 86.
The French also received the PAL NES, but it contained a unique internal PAL to RGB decoder to make it compatible with SECAM televisions. See alex, “Composite out of a french PAL NES.”
See NesDev Wiki. “Clock Rate.”
See ‘PPU OAM,’ NesDev Wiki.
See ‘Overscan,’ NesDev Wiki.
The remaining three bits are unused. See ‘PPU OAM,’ NesDev Wiki.
“As noted previously, the 6502’s low-cost competitor, the Zilog Z80, shared the dominance of the 8-bit era. The Z80 powered the ColecoVision, Tandy/RadioShack TRS-80 Model I, Sega SG-1000, Sequential Circuits Prophet 5, Roland Jupiter-8, Amstrad CPC, MSX, and the Sega Master System, among others.
In most Famicom game source code, one can find the assembly mnemonic CLD (CLear Decimal flag) in the system initialization routines. This disables the console’s decimal mode, despite there
being no apparent need to do so. This ‘safeguard’ is a vestigial holdover from coding practice on other 6502-based systems.
"Bagnall, On the Edge, 155, 297.
"Ibid., 296-7.
"Ibid., 467-8.
"Gifford, “Tetris... forever.”
"The TMS9918A could also fake a 32x32 sprite by toggling a magnification bit in VDP register 1. This mode ‘zoomed’ each pixel of the sprite into a 2x2 pixel area. See Texas Instruments, “TMS9918A/TMS9928A/TMS9929A Video Display Processors. (Microprocessor Series),” Sec. 2-26 to 2-27.
"Ibid., Sec. 2-5 to 2-7.
"Ibid., Sec. 2-11.
"Ibid.
"Montfort and Bogost, Racing the Beam.
Donkey Kong was Nintendo’s first arcade success in the United States, built literally on the foundation of a failure. Namco’s 1979 arcade hit *Galaxian* made an immediate impression on Nintendo’s Masayuki Uemura and his R&D2 team. Namco’s game was clearly indebted to the look and structure of Taito’s *Space Invaders* (1978), but it updated the formula with dive-bombing aliens, colorful sprites, and a backdrop of scrolling stars. Nintendo had attempted their own invading alien shooter in 1979 with the unremarkable *Space Fever*, but that game offered only minor variations on the *Space Invaders* formula. Uemura wanted Nintendo to match Taito and Namco’s successes, and based on *Galaxian’s* popularity, he thought the key was beefing up the underlying technology.¹ Throwing more hardware power at the game would make Nintendo’s shooter more realistic. Better realism would attract arcade players.

With an ambitious design in mind, R&D2 tasked Japanese television equipment manufacturer Ikegami Tshushinki to produce the sophisticated circuitry for Nintendo’s *Radar Scope*.² The results were impressive for their time. *Radar Scope* coupled a unique vanishing point perspective with alien sprites that appeared to grow as they dove toward the player. Nintendo upped the ante on *Galaxian’s* scrolling stars with a feature they advertised as ‘curvature of field.’ A dedicated EPROM drew the onscreen grid (so-called ‘3-Dimensional Vectors’) and twinkling stars (driven by the sound board’s noise generator) atop a gradient backdrop of black and deep blue.³ *Radar Scope’s* space setting had a shimmering depth unmatched by its competitors, aiming to immerse the player in the action in a way that a flattened planar perspective could not.⁴ The audio hardware was equally sophisticated, featuring ‘supernatural “Laser Sound.”’ The term was an obvious marketing ploy, but certain board versions did feature in-game speech.⁵ Of course, these technical marvels came at a significant expense. *Radar Scope* cost 1,000,000 yen, a hefty price tag for an arcade game.

---

2: Ports

4. Jump button makes Jumpman jump.

- *Donkey Kong* arcade cabinet instructions
It is unclear whether Radar Scope reached Galaxian-level popularity in Japan, but it was a success. According to Nintendo of America's (NOA) then-president Minoru Arakawa, it was second in popularity only to the monolithic Pac-Man. Whether true or not, Radar Scope seemed like enough of a sure bet for Arakawa to hedge Nintendo’s American future on the title’s success. The young president was eager to debut a Nintendo hit in the States—
Nintendo had recently established their first US office in New York and President Yamauchi was watching Arakawa’s decisions closely. Confident of the space shooter’s success, Arakawa requested a shipment of three thousand arcade cabinets from Osaka. Unfortunately, after the months-long shipping wait, any buzz preceding Radar Scope’s arrival had faded. Without the luster of cutting edge hardware, Radar Scope was left to rely on its derivative gameplay, a drawback that no amount of field curvature could mask. American arcade owners showed little interest, nor did players. Radar Scope was dead on the coast; NOA sold only one third of their order.

As surplus cabinets lingered in NOA’s warehouse, Arakawa scrambled for a solution. Radar Scope had sold enough to cover its manufacture and transportation costs, but Arakawa and his staff were working strictly on commission. Breaking even on costs meant no actual income for himself or his staff. As resources dwindled, Arakawa turned to Nintendo headquarters for aid. President Yamauchi quickly devised a plan to resurrect the unsold cabinets. He tapped R&D1 lead Gunpei Yokoi and junior industrial designer Shigeru Miyamoto to produce a Radar Scope conversion kit.

Conversion kits were common in the arcade era. A game’s popularity was subject to unpredictable player tastes, so lackluster titles might lose their appeal after a few weeks. When games no longer drew players’ quarters, they were removed from the arcade floor to make room for newer cabinets. Arcade space mirrored retail space: only a rare few products hung around for months or years. To combat the inevitable ebb in popularity, arcade manufacturers devised kits to update hardware without the need to replace the entire cabinet. Updated ROM chips reworked the gameplay and graphics while the existing CPU, sound hardware, and monitor were left intact. Slap on a fresh set of side decals, update the marquee, and you had a brand new game.

Kits were a boon to arcade owners. They no longer had to front the cost for new cabinets and could refresh their inventory without the hassle of shipping and installation. A single cabinet could last for several years with a regular supply of conversion kits. Namco’s 1981 hit Galaga, for instance, hosted multiple conversions: Bosconian (1981), Dig Dug (1982), Xevious (1982), and Super Xevious (1984). Similarly, Taito’s Qix (1981) was the foundation for numerous games, among them an unremarkable Donkey Kong imitator whose promotional flyer touted, ‘Now you can turn your QIX into a new money-maker with our exciting ZOO KEEPER conversion kit!’ By 1983, Nintendo offered ‘Nintendo-Paks,’ streamlined kits that promised to transform old arcade cabinets ‘into the next hot new games.’ Certified distributors could update the ‘side graphics, header, frontplex, FCC cage, complete control panel and all the electronics you need to give any Nintendo game new life.’ A mere two years after Donkey Kong’s 1981 release, a Nintendo-Pak would convert the title once again, into Donkey Kong 3.
‘Novelty Games’

Yokoi and Miyamoto had little time to design and engineer a *Radar Scope* conversion kit. They also shared different design motives than Uemura—rather than trying to compete with state-of-the-art technology while mimicking the gameplay of well-worn genre, they thought innovative style and rich characters would attract players, regardless of technical specs. They opted to use the scenario for a *Popeye* game they’d originally planned for a *Game & Watch* handheld. They drew on the cartoon’s familiar love triangle—Popeye, Bluto, and Olive Oyl—for both character motivation and spatial structure: Bluto would hold Olive captive at the top of the screen while Popeye would start at the bottom and work upward to rescue her.

Unfortunately, Nintendo failed to secure the license from Popeye’s rights holder, King Features, halting the designers’ plans in the pre-production phase.¹⁰ According to Yokoi:

> Pretty early on we had decided that Popeye would go on the bottom of the screen and Bluto would be on the top, thus establishing the framework for the game, but we would later discover that we wouldn’t be able to get the rights to use the characters after all. With no other options, we decided to keep the content of the game as it was and just change the characters. And so it was that those characters became Mario, Donkey Kong, and Princess Peach.¹¹

Miyamoto’s replacement designs were one-to-one translations of the copyrighted characters: Bluto became the sizable ape, Olive Oyl became the tall and slender Pauline, and Jumpman (as Mario was originally known) stood in for underdog hero Popeye. And though the characters changed, *Popeye* continued to be an inspirational resource, especially for the setting. Yokoi explained:

> There was an episode in the cartoon show for Popeye in which Olive was sleepwalking and wandered around a construction site. Whenever she was about to lose her footing, miraculously enough another platform would come out of nowhere and support her, and this left quite an impression on me. So we figured by using a construction site as the setting, there would be all kinds of things we could do, and thus chose that as the setting for our Popeye game.¹²

The referenced episode is a 1934 *Popeye* short called ‘A Dream Walking,’ wherein Olive Oyl falls asleep and wanders into a poorly-guarded construction site.¹³ As she sleepwalks across a series of perilous beams and girders, Popeye and Bluto pummel one another in a race to save her. The germ of a game idea was there, inspired by a veritable ‘platform cartoon’ (figure 2.2).
Yokoi’s and Miyamoto’s re-worked conversion became *Donkey Kong*, whose lighthearted cartoon design, inspired setting, and innovative gameplay were a marked departure from the sci-fi and military fare that populated arcades in the late 1970s. *Donkey Kong*, at heart, is a running and jumping game. The player takes the role of Jumpman, an apt descriptor for his and the player’s shared duties. A large ape named Donkey Kong has abducted Jumpman’s girlfriend, hauling her Fay Wray-style up endless elevated construction sites. Kong resides at the top, hurling obstacles downward as Jumpman makes his way to the top (figure 2.3).
When and if Jumpman reaches the screen's zenith, Kong snatches Pauline and climbs a ladder to the next level. The same scenario happens on two more levels. On the fourth and final screen, Jumpman must remove a series of rivets holding the center platforms in place. If he succeeds, the girders collapse, Kong plummets headlong to the bottom, and the lovers are reunited. After a brief respite, the game loops to a more difficult version of the opening level and Jumpman embarks once again on his Sisyphean task.
At the time, the four distinct screens and barebones tale of conquering adversity for love were unique. Unlike most arcade fare, players could work toward two parallel goals: higher scores and the culmination of the game’s micro-narrative. Additionally, each level required a new strategy. The opening ‘construction site’ level was jump-heavy, as barrels poured erratically over girder edges and down the same ladders that Jumpman clambered up. The ‘elevator’ level, in contrast, required patient timing to maneuver the rise and fall of the narrow elevators, since Jumpman was not a competent Fallman; any plunge larger than his height resulted in death. The ‘cement factory’ level threw Jumpman’s momentum into disarray, as ladders shifted height and conveyor belts changed direction beneath his feet. Each level also had a built-in risk/reward structure. The level timer counted down score rather than seconds, encouraging players to find the quickest route to the top. However, straying beyond the speediest route provided opportunities for point bonuses, either collecting Pauline’s scattered effects (her purse or umbrella) or grabbing an enemy-crushing hammer.

Like many innovative games of the arcade era, Donkey Kong’s play style demanded a new genre. In the early 1980s, videogame magazines waffled between a number of generic descriptors, from ‘climbing’ and ‘jumping’ games to ‘level’ and ‘ladder’ games. Many settled on the least inventive of the lot, ‘Kong-style games.’ Most curious was the ‘novelty games’ category, used to describe the influx of (most often Japanese) titles with colorful, cartoon-inspired sprites and gameplay that focused more on a character’s nimble movements than their fighting arsenal. The reviews supplement in the March 1983 issue of Computer & Video Games pegged two Japanese titles—Donkey Kong and Frogger—as key progenitors of a genre whose stylistic differentiator was ‘cuteness’: ‘Good graphics are by definition crucial to the success of novelty games. The characters must be cute or plausible, well defined, and above all central to the general theme of the game.’ The novelty adjective read more as a catch-all ‘non-category’ than a credible genre and it frequently described blatant rip-offs rather than games that were ‘novel’ in their own right. In the same issue, below the genre definition, Computer & Video Games lists a number of novelty PC titles like Pogoman and Hopper, whose lineages are not difficult to guess.

The ‘novelty’ designator carries two distinct and somewhat contradictory connotations. ‘Novelty’ is newness, a unique approach, a style gamers have not seen before. Donkey Kong and its ilk felt fresh in comparison to war, sports, and science fiction titles. Guiding a frog across a busy intersection or dining on ghosts with a sentient yellow circle was far afield from shooting Alien Invader X with Spaceship Y. But ‘novelty’ is also niche, an eccentric or even spurious outsider. This is the sense in which novelty games served as the miscellaneous bin for a number of games that resisted categorization. Frogger and Donkey Kong were not similar in the same way that Galaxian and Galaga were, so critics relied on a common visual style instead of concrete gameplay similarities. In Video Invaders (1982), an
early survey of the arcade field, the introduction to *Donkey Kong* highlights the conflicting senses of genuine excitement for the game’s uniqueness and the near-dismissal of its foreignness:

Donkey Kong is another bizarre cartoon game, courtesy of Japan. While we in America continue to invent new and improved methods of exploring outer space and obliterating all we find there, our Eastern rivals’ seemingly frivolous comic mentality keeps spilling over into their design of video games.¹⁵

Japanese videogames fulfilled both aspects of novelty in the US-dominated videogame market. They were outsiders in an industry they did not create, a sentiment echoed by the *Invaders* half of the book title quoted above. Yet the popularity of Japanese games was proving that they had appeal beyond their native country. For a nation only three decades removed from the catastrophes and antagonisms of World War II, it was no small feat to re-enter a market dominated by their former enemies.

In *Power-Up*, an insightful look into the Japanese revival of the American post-crash videogame industry, Kohler argues that Nintendo’s games, ‘are not products or models of our culture. They are products and models of Japanese culture, the “action and reaction” of the Japanese population, presented nearly unaltered for our consumption.’¹⁶ In other words, Japan was not selling American culture; Japan was selling Japanese culture, with little or no filter. The underlying assumption is that unmediated Japanese culture would be appealing to the West based on its exoticism, a reflection of a people wholly other from us. This is a long-held Western conception of Japan, what Napier calls ‘the embodiment of a variety of fantasies’ that throughout history have elicited fear, respect, admiration, fascination, desire, and suspicion.¹⁷

Part of Kohler’s assertion is formal: the exaggerated proportions of the manga (Japanese comic) style were well-suited to the limitations of early videogame hardware. Realism was tough to convey in a four-color, 16x16 block of pixels. Cartoons, however, where exaggeration and abstraction are the norm, made the transition easily. Simplified color and form conformed well to the crudity of palettes and pixels in the early videogame era. Jumpman’s visage and wardrobe, for instance, were largely determined by hardware limitations: a cap obviated the need for animated hair, a mustache defined an otherwise indistinct face, and overalls made fashion sense when one had to rely on only two colors to sell an outfit. Kohler also emphasizes how Japan’s distinct cultural interest in visual storytelling translated easily to the videogame medium. Japan was a society cultivated around images, from woodblock prints, Bunraku puppetry, and Noh theater to manga and anime. Unlike in America, where comic books were child’s fare, graphic storytelling in Japan ran the gamut of age and interests, from cartoons to pornography. In all regards, Japan’s was a culture primed for the transition to the vivid cartoon worlds and characters
of videogames.

Kohler is correct in the broad view. Nintendo undeniably helped ease the Western world into Japanese culture as a result of the monolithic popularity of their characters and consoles. And the exoticism of foreign products certainly played a part. Non-Japanese players could see, hear, and feel the novel influence of Eastern culture in the games’ designs, as the uncertainty of genre labeling suggests. But to say their products were ‘nearly unaltered for our consumption’ ignores the complexity of a decades-long cultural exchange taking place between Japan and the West, as well as the deliberate linguistic and cultural translations made in Japanese media prior to their export. In many cases, the Japanese origins of cartoons, comics, and videogames were obscured, or more generally labeled as ‘foreign’ rather than specifically Japanese. And the precedents for Japanese media in America were extensive enough that we can not credit videogames alone with initiating the cultural handshake between East and West.

Prior to the flood of videogames in the early 1980s, Japanese media was already gradually finding an audience on Western shores. Beginning in 1961, the first Japanese animated films—Magic Boy (MGM), Panda and the Magic Serpent (Globe Pictures), and Alakazam the Great (American-International Pictures)—were shown in American theaters. These ‘test cases’ for American audiences followed the Disney model of adapted folktales for children, featuring humans with a cast of adorable animal companions. Though their stories were sourced from Asian lore, any references to the films’ foreign origin were scrubbed. These and subsequent films were commercial failures, so American distributors shifted their efforts to television and 16mm rental markets. Throughout the 1960s and 70s, Japanese animated shows like Astro Boy, Gigantor, Kimba the White Lion, and Speed Racer debuted on American television in local syndications. Again, scrunched between children’s fare from Filmation and Hanna-Barbera, the shows’ foreign heritage was ambivalent. According to manga journalist Fred Patten:

To most Americans, these half-hour TV cartoons were indistinguishable from most American TV animation...[they] were not thought of by the public as “Japanese animation.” If their origins were realized at all, they were considered to be just part of a vague “foreign animation” category.¹⁸

Audience reaction was equally ambivalent. As the production of American kid-friendly cartoons increased and more stringent rules regarding violence in children’s programming appeared, Japanese animation was largely sequestered to networks catering to ethnic Japanese communities.

As Patten points out, a series of important events reversed Japanese animation’s success in America.¹⁹ New sci-fi series like Brave Raideen (1976) debuted, featuring teen heroes battling in oversized robots. These action- and drama-heavy cartoons appealed to
an audience older than the usual Hanna-Barbera set. The serendipitous arrival of the *Star Wars* phenomenon bolstered interest in any space adventure-themed animation, comics, and toys. There was also a concurrent shift in the tone and audience of American superhero comics in the 60s and 70s. The Golden Age of superheroes was making way for a new Silver Age, offering edgier interpretations of familiar favorites (e.g., Batman and Superman) and new, modern characters targeted for an older audience (e.g., Spider-Man and the X-Men).

Finally, the introduction of consumer video cassette recorders (VCRs) around 1975 created an inexpensive means to record, share, and distribute Japanese films and television shows among fans.

This synergy of social, cultural, and technological changes continued to coalesce. America’s youth counterculture movement sought edgier forms of art and expression that circulated outside mainstream norms. Racial tolerance was shifting. The generation that thirty years before had inaugurated atomic warfare against a Japanese enemy gave way to sons and daughters more receptive to cultural exchange. In short, the time was right for Japan’s cultural entrance in the West. By the early 1980s, Japanese media had seeped into all corners of pop culture: the Cartoon/Fantasy Organization formed in Los Angeles, the first fan group to cater specifically to anime; *Japanese* animated films and cartoons were screened at comic book and sci-fi conventions; fanzines devoted to ‘Japanimation’ and *manga* were published; an influx of Japanese sci-fi and robot toys were hitting American shelves; translations of popular *manga* were sold in US comic shops; and some of Japan’s artistic luminaries, like *Astro Boy* creator Osamu Tezuka, were meeting with fans in America.

By the time *Donkey Kong*’s debuted in 1981, there was already a modest but enthusiastic fan base for Japanese media, centered around the same adolescent, teen, and twenty-something demographics that would flock to videogame arcades. There was a reason the early crop of Japanese games were labelled ‘cartoon’ games—they resembled the first Japanese medium that made an impact on Western pop culture.

And just as anime’s foreign pedigree was hidden from American viewers in the early 1960s, Japanese creators (and their American distributors) modified, edited, or outright censored their media in accordance with Western tastes, even as their work gained increasing acceptance outside of Japan. In the 1980s, popular sci-fi anime series *Space Cruiser Yamato* and *Macross* were brought to American television, respectively, as *Star Blazers* and *Robotech*. Some revisions were benign, like the titles or character names—certainly Derek Wildstar was more palatable to American children than his original name, Susumu Kodai. Other more significant changes reflected divergent cultural attitudes toward sexuality or vice. Dr. Sano’s predilection for alcohol, for instance, was whitewashed in his transformation to the good-humored Dr. Sane, while Lance ‘Yellow Dancer’ Belmont’s transvestite tendencies were explained away through plot contrivances that cast him as a secret agent disguising himself in drag. Other revisions aimed to smooth historical
animosities: the space ship *Yamato* in the Japanese original, for instance, was a resurrected World War II battleship, a symbol of Japanese heroism whose historical forebear happened to be used to kill American soldiers.²² *Robotech* was subject to drastic structural changes:

The long, drawn out stills used to convey moments of extreme emotion or heroism were shortened, and the sound track was changed. Flaws and contradictions in the heroes’ personalities were softened. The deaths of some significant characters were covered up. And just to put the final kiss of death on the whole production, the networks persisted in scheduling them at times suitable for children.²³

As Levi implies, some of these changes were due to divergent audiences. In Japan, *Macross* was not a children’s show—in America, all cartoons were children’s shows, so they had to be cut accordingly.

The same attention to potentially problematic structure, content, or form applied to videogames. Kohler himself remarks on the ambivalent cultural status of Japanese videogames, many of which were designed with English text (reflecting Japan’s own exoticism of the other), writing, ‘Video games in general were a contemporary American invention, and there was nothing that clearly labeled newcomer Nintendo as Japanese.’²⁴ But he also reveals a litany of cultural translations necessary to soften videogames’ Japanese roots prior to their introduction to Western markets: among them, the original *manga* drawings from *Western Gun* arcade cabinets were not used for American release (despite the American-derived genre of the game); *Pac-Man’s* original onomatopoeic name *Puck-Man* was altered to discourage arcade vandals; and NES games were scrubbed of potentially racist or religiously inflammatory content prior to American release.²⁵ Time and again, Japanese videogame companies were surgically strategic in their modifications for non-Japanese audiences. Nintendo was no exception—they were acutely aware of their cultural product and its potential Western reception, from their industrial design to outright censorship (see chapter 3). This does not discredit Japan’s cultural contributions nor valorize ‘Western’ versus ‘Eastern’ design, whatever those broad designators might mean. It is simply that calculated (and miscalculated) translations were taking place on either side as part of a complex cultural exchange crossing thresholds of language, custom, technology, economics, and politics.

Of course, the flow of cultural commerce was not one-sided. Osamu Tezuka, widely considered the father of anime, was inspired in equal measure by both Japanese wartime propaganda films and Disney animation. Though Tezuka’s and Disney’s content and style were considerably different, Tezuka was dubbed ‘the Walt Disney of Japan.’²⁶ The same fans in the 1970s who were copying and trading shows amongst themselves were establishing international connections, swapping cartoons with Japanese science fiction fans hungry for
American fare like *Star Trek* and *Battlestar Galactica*. The shared NTSC standard between the US and Japan helped facilitate the exchange (and equally hampered anime’s expansion to PAL markets). Many of the first Japanese videogame consoles mimicked or licensed Western machines, like Nintendo’s own *Pong* derivative *Color TV-Game 15* (1977) or Atari’s Japanese version of the 2600, the 2800 (1983). The Japanese likewise picked and chose from a range of foreign inspirations for their game designs, from medieval knights to Western gunfights.

Consider the odd amalgam of Western influences in *Donkey Kong*. The most obvious is *King Kong*, the 1933 RKO film, directed by Merian C. Cooper and Ernest B. Schoedsack, featuring an enormous gorilla displaced from his native land for spectacular display in metropolitan New York. The original theatrical poster features a familiar pose: Kong perched atop a tall structure, clutching Fay Wray, the helpless damsel in the pink dress. Yet our in-game damsel is no short-cropped flapper. Her long hair, ankle-length dress, and blue heeled boots are distinctly 19th-century American Western, a sensibility already rehearsed by Nintendo in their early Wild West-themed arcade game *Sheriff* (1979). Pauline’s attire, coupled with the damsel-in-distress trope, also evokes the silent serials of the 1910s, like the tied-to-the-train-tracks classic *The Perils of Pauline*. Meanwhile, Mario (named after Nintendo’s American landlord) is a workaday anachronism, an unlikely Italian hero who relies on agility and carpentry tools to survive. When Mario dies, a symbol of Christian iconography—the halo—floats above his supine body. Today we tend to gloss the patent absurdity of *Donkey Kong* and its characters because they have been so tightly woven into our cultural fabric.

Some Americans were not so keen to forgive *Donkey Kong*’s similarities to a Western cinematic icon. Universal Studios famously threatened Nintendo of America with copyright infringement, claiming that the game’s name, ape, and premise were identical to *King Kong*. Universal tried to strong-arm Nintendo into a settlement, hoping the US newcomer would shirk in the face of Universal’s formidable legal team. Nintendo stood strong and Universal sued. In a grand twist of irony, it was revealed that Universal had no legal claim to King Kong—in fact in 1975, with plans to remake the 1933 film, Universal had successfully argued in court that RKO no longer owned the character. Kong was public domain.

Universal’s shaky ownership claims notwithstanding, Nintendo would have prevailed. The New York Circuit Court of Appeals ruled in 1984 that the game’s premise fell under parody, thanks again to its ‘cartoon’ sensibilities:

The district court conducted a visual inspection of both the *Donkey Kong* game and the *King Kong* movies and stated that the differences between them were “great.” It found the *Donkey Kong* game “comical” and the Donkey Kong gorilla character
“farcical, childlike and nonsexual.” In contrast, the court described the King Kong character and story as “a ferocious gorilla in quest of a beautiful woman.” The court summarized that “Donkey Kong creates a totally different concept and feel from the drama of King Kong” and that “at best, Donkey Kong is a parody of King Kong.” Indeed, the fact that Donkey Kong so obviously parodies the King Kong theme strongly contributes to dispelling confusion on the part of consumers.³⁰

For his part, Miyamoto deposed that the Donkey Kong name was a translation error, meant to convey a meaning closer to ‘stubborn monkey’ than the classic film ape.³¹ When Miyamoto looked up ‘stubborn’ in a Japanese/English dictionary, it read ‘donkey,’ while ‘kong’ was a general term for any large ape.³² Since then, the true origin of the game’s title circulates among numerous apocryphal tales. Regardless, the irony came full circle in 2010, when Nintendo attempted to trademark the phrase, ‘It’s on like Donkey Kong.’³³

Perhaps the bizarre compote of influences in Donkey Kong was an unfettered reflection of Japanese tastes, ripe from the minds of Yokoi and Miyamoto. Or perhaps they were honest translation errors. But Miyamoto’s early influences were already a fascinating mix of Western and Eastern sources. He idolized the Beatles, bluegrass music, and Disney cartoons, but he was raised in the culture of Japanese manga, cinema, and literature. These influences could surface in subtle ways. Regarding Donkey Kong’s four-screen design, Miyamoto said:

Thinking back, I would say that although it wasn’t done consciously, I ended up designing Donkey Kong like a traditional Japanese four-panel manga comic strip. That way of telling a story in four distinct parts seemed natural to me, so I created four separate screens from the opening to the conclusion.³⁴

Remarkably, Miyamoto and Yokoi were able to wed character archetypes from Popeye, a cartoon used at one point to propagandize anti-Japanese sentiment during World War II,³⁵ to a Japanese pop cultural form, built upon the silicon remains of a failed arcade game, and end up with a smash hit. Indeed, like its two talented creators, much of Nintendo’s appeal derived from an ability to recycle, reuse, and recontextualize, whether from its own hardware and software artifacts or from disparate cultural sources. This was true even prior to Nintendo’s entry into the videogame business. Their financial success in the late 1950s and early 1960s stemmed from a line of traditional hanafuda playing cards decorated with licensed Disney characters, a true mashup of East and West.³⁶ Miyamoto personified this impulse for the emerging videogame age. He fit the role so well that he was drawn, often unwittingly, into a public relations strategy that propelled him to the forefront of Nintendo’s corporate persona. Yamauchi, Arakawa, Uemura, and even Yokoi never managed to capture the hearts of the West like Miyamoto did.
Nintendo’s games had an uncanny familiarity to American audiences and the decades of pop cultural exchange leading up to Donkey Kong’s release paved the way for its success, though not without an initial measure of trepidation. The characters resembled Western cartoon favorites, but were not; the ape reprised the actions of King Kong, but was not; Pauline resembled the classic damsel archetype, but not. Nintendo’s translation of Western culture was a strange de-contextualized pastiche fashioned from the scraps of Western media, refracted through a foreign lens. There is no way to pinpoint the direction of cultural flow, the sophisticated patterns of influence that manifest as the novelty and cartoon games of the 1980s. The Japanese were adopting an industry from the West, infusing it with their own traditions, combining it with their own interpretations of Western culture, then translating it back again before export to its native land. This complex process continued to foster the same mix of attraction and puzzlement that struck players encountering Donkey Kong for the first time. And eventually, as the Japanese style became more and more familiar, works like Kong changed from ‘novelty games’ to simply videogames.

**Bringing Kong Home**

In the early 1980s, arcade machines were the lead platform for cutting edge videogame technology. Arcade games benefited from dedicated hardware: processors, memory, discrete sound circuits, cabinets, and control interfaces built to suit the needs of each game (or a handful of similar games). Marble Madness’s pseudo-3D graphics and control scheme demanded a different hardware configuration—from increased ROM space to a ‘rollerball’ input device—than, say, the two-dimensional, single-joystick play of Pac-Man. Home consoles and computers were underpowered in comparison, especially cartridge-based systems and consumer PCs that tried to provide flexible platforms for a range of software. By the time microprocessors were cheap enough to mass market inside an affordable console package, arcade technology had moved ahead by leaps and bound. Though cabinets were expensive to design, manufacture, and ship, they could make up for the large upfront costs through volume sales. Hundreds of players might cycle through a single game each day, eventually earning the arcade owner enough revenue to recoup costs. Everything else was icing. Of course, the economic flip side was an unpopular game that failed to attract quarters.

Despite the technology gap, arcade games still made their way to home consoles. The process of translating game software from one platform to another is commonly called porting. Today’s most popular games are no longer targeted for arcade cabinets, but they are still likely to live on multiple platforms in order to maximize their user (i.e., revenue) base. In comparison to the 1980s, today’s diversity of platforms is relatively narrow: we
have the big three console manufacturers—Sony, Microsoft, and Nintendo—along with OS X, Windows, Linux, the web, and a small range of handhelds. Thirty years ago there were multitudes of distinct platforms, confusing hardware forks within a single company (e.g., the Atari 2600/5200/7800/400/800/ST), and region-specific manufacturers. It was common for an arcade hit like Pac-Man to receive ports for a dozen or more PCs and consoles—not to mention tabletop toys, watches, board games, and the like. The closest analogy we see to the platform opulence of yesteryear is in the mobile space, where several platform players compete (e.g., iOS, Android, Metro, Symbian, etc.).

Arcade ports to the vast array of PCs and consoles in the 1970s and 80s ran the gamut from near-perfect to abysmal, with cuts and concessions made according to the specifications of each platform. Porting was and is not a magical process whereby a programmer copies the game’s source from one machine and simply compiles it on another. At a time when computers were coded ‘close to the hardware’ in assembly language, the internal architectures of two competing platforms had a drastic impact on the visual results.

Assembly language, in itself, is not a codified grammar—it is more like a common base for a variety of dialects, each specific to a particular CPU family. In the same way that someone does not speak Romance language, but rather speaks French or Spanish, one also does not code assembly, but instead 6502 assembly or Z80 assembly. In other words, assembly language is tied to its underlying silicon. There will be similarities between ‘regional dialects,’ perhaps a few identical ‘words,’ but no clean one-to-one translation. Even identical cross-platform syntax can not account for the differences in, say, how a video processor draws to the screen, available sprite sizes, the range of available colors, or even the number and configuration of buttons on a controller. Source code notwithstanding, how does one translate the complex multi-button control scheme of Defender (1980) to the single-button joystick of the Atari VCS? These were the types of problems faced by any programmer tasked with an arcade port.

When arcade developers were slow to deliver ports, third parties picked up the slack. Especially in Europe, where the scope and variety of low-cost PCs overshadowed the burgeoning home console market, games inspired by or directly copied from arcade hits came fast and furious, from professional and amateur programmers alike. British magazine CRASH, who published one of the first references to ‘platform games’ (see chapter 4) in their ‘Living Guide to Spectrum Software,’ often ran reviews of Kong clones that made no qualms about lifting their design from the original. In CRASH’s blurb on Killer Kong, for instance, they called the game’s protagonist Mario, despite the lack of any similarity between Nintendo’s overall-clad carpenter and the shirtless stickman seen on screen.³⁷ Clearly, arcade fans were clamoring to play their favorite games at home, even if they got cut-rate or imitation products. PC-oriented magazines catered to the tastes of amateur
programmers, providing pages of type-it-yourself source code for clones of popular arcade titles. Despite its shortcomings, home play was an understandable salve to the arcade’s relentless quarter drain.

Imitators proliferated in the arcades as well, often unabashedly lifting a popular game’s screen layouts, gameplay, and characters. The most egregious cases were outright theft. Disreputable manufacturers slapped a new marquee on the arcade cabinet without bothering to alter the internal hardware, *et voila*, *Donkey Kong* became *Crazy Kong, Konkey Kong, Congorilla, Donkey King,* and a plethora of other tragic name variations. Reversing from defense to offense, Nintendo pursued these bootlegs and clones with litigious fervor in an effort to curb economic losses. Despite Nintendo lawyer Howard Lincoln’s efforts in litigating thirty-five copyright infringement cases and confiscating thousands of counterfeit circuit boards, Nintendo estimated they lost nearly $100 million to *Donkey Kong* copycats.

Meanwhile, Nintendo was keen to meet legitimate demand for *Donkey Kong* with an official console port. Home versions of popular arcade titles provided new revenue sources, prolonged a videogame’s life for years beyond its initial release, and helped publicize the company brand. Nintendo, guided by Yamauchi, had grand ambitions for worldwide success, so they licensed the arcade ape to nearly every viable (and sometimes non-viable) platform of the day, from Amstrad CPC to ZX Spectrum. The clear powerhouse among home console ports was Nintendo’s first licensee, Coleco. The ColecoVision’s pack-in version of *Donkey Kong* was lauded by press and consumers alike for its arcade fidelity, despite its decreased palette, altered sound and sprites, and missing ‘cement factory’ level. The game bolstered sales of Coleco’s fledgling console, challenging Atari’s multi-year reign of the home market. The ColecoVision was technologically superior to the aging VCS and the accuracy of Coleco’s arcade ports drove that point home visually and sonically. The ColecoVision soon became the arcade purist’s console of choice.

Coleco’s exclusive license with Nintendo lasted for six months, allowing them to sell half a million consoles. Once exclusivity lapsed, Coleco adapted their business plan, shifting roles from first-party to third-party development. They aimed to dip their toes in both ends of the profit pool, producing ports for hardware rivals Atari and Mattel while continuing production of the ColecoVision. Both competitors’ machines were ill-equipped for *Donkey Kong*’s specifications. The Atari VCS port delivered a version of the ape that looked more like a giant Gingerbread Man tossing chocolate-chipped cookies than Mario’s fearsome antagonist. The Mattel Intellivision port was less visually offensive, but still barebones: Kong remained a brown monochrome mess. The Atari 7800 version fared best, due to its more capable hardware, but Mario’s running and jumping sound effects were absolutely brittle compared to the original. In the end, the quality didn’t matter to Coleco. If they sold cartridges for Atari or Intellivision, they earned a cut; if consumers opted for the superior arcade facsimile, they would buy a ColecoVision. Win-win.
As the wide spectrum of quality attests, *Donkey Kong* was a challenging port for home systems. Like *Pac-Man* and many other Japanese cabinets, it ran on the Zilog Z80, clocked at a whopping 3MHz, nearly tripling the processing power of console contemporaries Intellivision or Atari VCS. The newer ColecoVision fared better partly due to a shared architecture, as it ran on a slightly faster version of the Z80. However, ports do not automatically benefit from similar CPUs or enhanced processing speeds. A host of hardware differences, from input mechanisms (e.g. joysticks vs. keypads) to available video RAM, have significant ‘behind-the-scenes’ consequences that impact the final visual display.

One such difference was *Donkey Kong’s* monitor. Arcade displays in the early 1980s were not off-the-shelf consumer televisions, though they were similar. Most cabinets of *Kong’s* era were stocked with screens from Electrohome (model G07) or Wells-Gardner (models K4600, K4900, and K7000), typically 19” or 25” color raster scans. Nintendo opted for a local supplier, so *Donkey Kong* and its kin were outfitted with the Sanyo 20EZ. Resolution- and frequency-wise, arcade monitors were identical to home sets. The key difference was the video signal. Most home televisions had, at best, a composite input. True to its name, the composite signal merged the monochrome (black and white) and color information into a single stream, significantly lowering the bandwidth necessary to convey the signal. However, bandwidth savings were offset by a loss in quality. TV cameras output color information in three separate streams: red, green, and blue. Ideally, a monitor would accept and reproduce these streams individually. Once combined, they could no longer be cleanly separated again. As a result, composite signals produced display errors like ‘dot crawl,’ where checked patterns of color artifacts noticeably creep along the border of two color bands. Arcade monitors were RGB (or Red Green Blue), and thus kept the individual color streams separate and intact. Compared to home televisions, arcade monitors looked crisp and colorful, with fewer image imperfections.

*Kong’s* display, to accommodate Jumpman’s vertical ascent, was also taller than it was wide. No custom hardware was necessary for such an orientation—horizontal displays were simply rotated on their side, while the arcade hardware handled the appropriate graphical flip. This ‘trick’ was more evident in cocktail arcade cabinets designed for seated play. Two or more players sat on opposing sides of the cabinet, looking down at the monitor. For each player’s turn, the screen would re-orient to their perspective. Typically the orientation was set in advance by an arcade operator or technician. For *Donkey Kong*, a jumper switch on the CPU could set upright or table orientation, along with the number of lives, score level, and number of coins per play.

The vertical 3:4 aspect ratio was common in arcade games, but a significant hurdle for console ports. Videogame consoles, unlike arcade cabinets and some PCs, did not include their own monitors. Fortunately, most game players had a convenient solution on-hand: their television. By 1980, over seventy-five million US households owned a television,
nearly 98% of the total population; half of these households owned more than one.\(^5\) Japanese households reported similar statistics.\(^6\) This provided great flexibility for display choices, as nearly every household had a screen suitable for videogames. Market saturation clearly benefited console manufacturers, as they did not have to factor a monitor into manufacturing costs. Of course, one tradeoff was the fixed orientation. Home televisions were designed for broadcast television, not arcade games, so they conformed to the industry-standard 4:3 aspect ratio. For Donkey Kong, this meant either compressing vertical platforms or eliminating them altogether. The ColecoVision port, for instance, had only five girders in Level 1 (not counting Pauline’s platform), in comparison to the arcade’s six.

Donkey Kong’s graphics were likewise tough to duplicate on a console, due both to the default size of its graphical tiles and the memory available to store them. The visual elements that comprised the screen were split between two tile types—sprite and character—that occupied distinct locations in memory. Though these were combined to form a seamless game mosaic, they were treated differently by processor and programmer alike. As we saw in chapter 1, sprites are typically the objects that move: Jumpman, barrels, bouncing jacks, Pauline and Kong himself. Donkey Kong had the memory capacity for 128 individual 16x16 sprites. Character tiles, on the other hand, are customarily used to build static ‘background’ elements: girders, ladders, letters, numbers, and the interface. Donkey Kong had memory for 256 character tiles, but they were smaller than sprites, only 8x8 pixel squares. Character tiles are also drawn onscreen before sprites, meaning that they have a lower priority than sprites. Elements drawn last (higher priority) appear above those drawn first (lower priority), explaining why the Jumpman sprite covers the ladder tile when he climbs and his shoes overlap portions of the girder when its height shifts. In short, sprites float freely on top of characters tiles, an allusion to their mythological etymology.

Donkey Kong’s large sprites facilitated the game’s cartoon design. A 16x16 pixel area granted four times the canvas for artistic expression than an 8x8 tile, so nearly every sprite onscreen was drawn to fill the larger area. Barrels, fireballs, and cement pies alike fall within those dimensions. Even Jumpman’s impressive range of animated movement—jumping, running, climbing, falling, dying—is blocked within individual 16x16 sprites. The sole exception is the hammer power-up. Once Jumpman collides with the floating tool, its sprite is appended above or beside him as he swings at barrels and fireaxes. Pauline is another exception—she is taller than Jumpman, so her head and shoulders require an additional sprite to accommodate her stature. Kong is the most obvious graphical outlier, filling six sprite slots—his head alone is larger than Jumpman’s entire body.

Sprites appear to have a clear functional advantage, so why not populate the entire screen with them and dispense with character tiles altogether? Character tiles’ conformity to a rigid grid conceals a hidden benefit: repetition. Since sprites are independent objects,
they are more computationally expensive. Each sprite in Donkey Kong must ‘carry’ its own unique location and (4-color) palette information alongside the flipped bits that define its shape. The additional memory overhead limits the number of sprites that can appear onscreen simultaneously, as well as how many can appear on a single scanline (sixteen in this case). Character tiles, meanwhile, may be iterated across the screen by reference. Displaying the platforms in the ‘rivets’ level does not require hand-positioning individual identical sprites one after another. Instead, the video processor can reference the index of a single character, the 8x8 blue girder, and repeat it along the horizontal grid x times. This technique saves considerable memory.

Nonetheless, Donkey Kong’s chunky sprites have a significant programming advantage for object movement. In sprite-based graphics systems, one tile is often not adequate to represent an entire object, e.g., a character’s body. Objects are thus built out of multiple sprites that move in unison. Moving a single-sprite object is straightforward—update the sprite’s x- and y-coordinates to the desired screen position. Multi-sprite objects are more complex; each constituent sprite’s coordinates must be updated simultaneously, so Kong’s head does not move disconcertingly without the rest of his body. Character animation compounds the complexity. Certain tiles in an object may need to update while others do not. In Jumpman’s case, each frame of his walk cycle occupies a single sprite, so the animation routine can rotate through single individual tiles. Animating Donkey Kong’s massive frame is trickier. At times, his body will rotate, requiring all of his tiles to change. However, when he faces the player and stomps his feet, portions of his body remain fixed, while others move.

In the grand scheme of game design, multi-sprite object movement is a minor hurdle, but it does come at some computational expense. In the next section, we will see how differences in sprite sizes can impose significant constraints on a port’s translation to a new platform. Even minor increases in computational overhead can decide the fate of individual characters, animation, or even entire levels.

The Ape in the Background

Nintendo’s first cartridge hardware was the NROM, labelled internally as either HVC-NROM-128 or HVC-NROM-256 depending on the PRG-ROM capacity. The initial spate of Famicom NROM games, such as Donkey Kong (1983), Mario Bros. (1983), Door Door (1984), and Nuts & Milk (1984), had rather spartan interiors. Inside the cartridge’s diminutive rectangular shell, the printed circuit board (PCB) had limited, if any, manufacturer identification, sometimes merely a utilitarian white stamp with the game’s ID followed by a number (e.g. ‘DK 223583’ for Donkey Kong). At least initially, the flat wafer had only two unlabeled integrated circuits (ICs) onboard—PRG-ROM and CHR-ROM—affixed with ‘glob-
top’ epoxy. This low-cost method of semiconductor production bonded and protected the IC and its connections with a coating of black resin.

The HVC-NROM-128 used in *Donkey Kong* had severely limited resources compared to an arcade game. Every bit of code, level data, and music had to fit within the 16KB PRG-ROM, while all the graphics, both background and sprites, had to squeeze into 8KB, barely enough space to fit the text of this chapter section in a plain .txt file. The initial constraints on cartridge hardware had repercussions on Nintendo’s early Famicom titles, considerable enough to require substantial cuts to their flagship arcade game. Although Nintendo matched and surpassed the high bar of the ColecoVision, the NROM proved inadequate to capture *Donkey Kong’s* arcade standard.

Level 1 of Famicom *Donkey Kong* is slightly truncated in comparison to the original (figure 2.4). The arcade monitor’s vertical resolution is 224x256 pixels, while the Famicom’s *visible* NTSC resolution, accounting for overscan, is the exact inverse. In resolution alone, the Famicom’s PPU has an advantage over the ColecoVision’s TMS9928A VDP. The latter could output 256x192 pixels, constraining its vertical proportions even more significantly than the Famicom. Coleco chose to delete the top girder and move Kong and Pauline to the right side of the screen, while Nintendo used the additional vertical resolution to rearrange its screen elements. The Famicom shares arcade *Kong’s Leve*l 1 girder count and configuration, but the console’s layout is subtly compressed. This detail is most evident in the reduced gaps between broken ladders and the increased clutter toward the top of the screen. Famicom *Kong* has its score, life, level, and bonus indicators squeezed obtrusively into the playfield. Spatially, the arrangement makes *more* sense for the game’s vertical narrative, since Donkey Kong carries Pauline directly off-screen, rather than into the scoreboard. However, due to memory limitations, the Famicom port eschews any animated sequences. Once Mario reaches the top girder, the sprites freeze, a fanfare plays, and the game cuts directly to the next level.
As expected, the Famicom’s girders stage is composed almost entirely of background tiles from pattern table #1 (or the ‘right’ pattern table). The surprise is precisely which elements are built from background tiles and which are built from sprites. In chapter 1, we learned that the Famicom PPU can only display eight sprites on a single scanline. After the eighth, the PPU will no longer fetch additional sprites to display. Programmers must devise methods to shuffle sprite priorities in OAM between frames so the tiles do not disappear completely. Visually, this results in the Famicom’s familiar flicker effect. If a player’s character is, say, built from multiple sprites (as is usually the case), portions of that character’s body might appear and reappear as it passes a particular horizontal vector shared by other objects. The largest element onscreen in Donkey Kong is—rightfully—the ape himself. He occupies an area of 48x32 pixels, a substantial twenty-four individual tiles in total. But keep in mind that this is a single frame of Kong’s movement. Small portions of
his body may remain static as he grabs and chucks barrels, but the majority of his tiles need to be updated. As a result, nearly seventy-five percent of pattern table #1’s available space is occupied by ape tiles. More importantly, Kong occupies a large portion of scanline space, meaning that any other elements on his horizontal vector would compete.

Cleverly, and unlike the arcade original, Donkey Kong is not built of sprites. Arcade Donkey Kong could handle up to sixteen sprites on a scanline. The Famicom makes up for this discrepancy by building Kong from background tiles, along with the barrel stock to his left, the score display, the blue oil can, the ladders, and the entire girder structure. Kong’s background status is a hardware necessity on the Famicom. Each of the thrown barrels is a 2x2 metatile of sprites. If Kong were also built of sprites (six horizontally), there would be situations where he and a barrel (two sprites horizontally) occupy the upper girder simultaneously, reaching the eight-sprite maximum. In situations where Mario, himself a 2x2 tile composite, reaches the top girder alongside Kong and any moving barrels, the lowest priority sprite(s) in OAM would disappear. This would be not only aesthetically unappealing, but would distract from Donkey Kong’s precision jumping demands.

The Famicom port is not completely flicker-free. There are instances where the game’s kinetic objects align horizontally for an instant, but they are rare. Players focused on the action likely never notice the brief flicker. In this respect, Donkey Kong’s vertical gameplay design is ideal for translation to the Famicom. Elements that travel horizontally, like barrels or fireballs, are sequestered into discrete ‘bands,’ i.e., the platforms and their current occupants. The arcade hardware permits double the sprites per scanline, so the game design does not have to be as stringent about the quantity of cascading objects. Subsequently, the original version feels a bit more frantic and fast-paced than the console port, since more sprites can crowd the available platform space.

Compare Donkey Kong to another of Nintendo’s early arcade ports, Balloon Fight. The game is similar to Joust, though the latter’s ostrich-riding knights are replaced with men suspended by balloons who flap their hands to gain altitude. Balloon Fight’s object is to pop your competitors’ balloons by colliding into them at a slightly higher elevation. Collide face-to-face and you will bump off one another; collide slightly lower and they will pop your balloons. Like Donkey Kong, Balloon Fight is vertically-oriented (the original arcade version also scrolled vertically, though the Famicom port does not), but the floating mechanic creates wider variations in sprite movement. Players and enemies can move along vertical, horizontal, and diagonal axes, space often shared with both static and dynamic environmental obstacles. Donkey Kong has more of a ‘gravitational pull,’ where most objects (besides Mario) are flowing toward the bottom of the screen. In Balloon Fight, the balloon men can flap their hands to actively fight gravity. Consequently, there is much less control over how many objects will appear on a single scanline simultaneously. Balloon Fight suffers from significant flicker, as sprites float about the screen erratically.
Sprite conflicts also help explain Pauline’s revised placement. In the arcade version, Pauline is captive atop the highest platform, a horizontal girder parallel to Kong’s line of sight. Here she cries ‘Help!’ waddling her legs back and forth in anticipation of Mario’s arrival. On the Famicom, Pauline is relocated to a new horizontal girder aligned with Kong’s torso. This placement puts Mario’s damsel awkwardly close to the ape, but it also clears space for the score indicators while keeping her out of the horizontal vector of barrel movement. From a narrative perspective, it also allows Mario to reach the uppermost platform without ‘reuniting’ with Pauline, since their girders are not attached. Arcade Kong snatches Pauline from her girder and carries her upward. The console version defers Mario’s victory and makes a more sensible transition to the following level. Here, a hardware-necessitated concession actually benefits the game’s compressed storyline.

Interestingly, the Famicom programmers included a simple in-game technique to differentiate sprites from background tiles. Pressing the Start button during play will pause the game, as expected, but also, after a few frames, disable all visible sprites. During its pause state, there is a write to the fourth bit of the PPU’s second control register ($2001) that instructs the PPU to mask (i.e., hide) all sprites. A range of early Nintendo titles exhibit this behavior, notably Mario Bros., Ice Climber, Balloon Fight, and Donkey Kong Jr. (where not only Kong is relegated to the background, but Mario as well).

Why would Nintendo enable such a function? One reasonable guess is to equalize home and arcade play. Donkey Kong, like most arcade games, has no pause button. Players are meant to play and move on so more quarters can drop. Omitting a pause function would be inconvenient for home play, since players often need to step away from a game in progress. But its inclusion creates an unfair strategic advantage; pausing permits a player to look ahead at the incoming obstacles and plan accordingly. A staccato pause/resume strategy would dampen Kong’s fast-paced, reflex-centered gameplay. Masking the Famicom’s active sprites prevents this advantage, but it is an all-or-nothing technique. Nonetheless, it is simpler to toggle a single bit to disable all sprites when Start is depressed versus re-positioning individual sprites offscreen. Vanishing sprites may not be visually elegant, but they are computationally frugal.

There is another equally reasonable explanation: vanished sprites provide a quick visual cue that the game is paused (and not locked up).

As explained above, there is an important difference in the way sprites and background tiles are placed on the screen. Each sprite has manipulable x- and y-coordinates that allow the programmer to place them with pixel precision. As Mario runs along the girders in Level 1, his sprite follows the graded single-pixel incline of each beam. For the Famicom’s PPU, the Cartesian coordinates of the television screen are mapped as if a traditional coordinate plane was flipped along the horizontal axis. The 0,0 origin starts at the upper left. A sprite’s x-coordinates increase as they proceed right, while y-coordinates
increase as they move down. As Mario runs right from the blue oil can on the first girder, the x-coordinates of the four sprite tiles comprising his body are incremented. When he reaches the first incline, his y-coordinate is decremented by a single pixel since he is moving closer to the screen’s upper edge. If you watch the animation carefully, you will notice that Mario’s sprites do not shift up immediately when his foot hits the incline. Instead, he moves several pixels ‘into’ the girder before the bump happens.

Gentle inclines have been a challenge to programmers since the advent of bitmapped display systems. The pixel is the smallest unit of gradation possible for the NES PPU to render, coupled with a display resolution that makes pixels visible to the naked eye (assuming a large enough television screen). How, then, do you make a character move convincingly along a sloped surface without appearing to visibly shift between pixels? One solution relies on the imprecision of CRT displays. As cathode rays sweep the phosphor-coated glass of the television screen, hundreds of colored triads illuminate and darken. Unlike the binary on/off flicker of a light switch, CRT phosphors have an afterglow that softens the hard edges of tile-based graphic displays. When phosphor glows overlap, individual pixels are less discernible. The transient blur does not eliminate the Famicom’s blockiness altogether, but it does smooth over subtle stair-stepped pixel grades. On a CRT monitor, Donkey Kong’s girders have a pleasing incline.

The incline comes at a cost: while Mario’s sprites can be manipulated pixel by pixel as he moves, the girders cannot. Like arcade Kong’s character tiles, Famicom background tiles do not have independent coordinates to manipulate; they are fixed to the 32x30 name table grid. This poses a challenge for drawing level 1’s graded girders, since it is beyond the hardware’s capabilities to use a single pattern table entry, shifted vertically by one pixel, to draw each segment. Inspecting Kong’s pattern table data reveals twenty-eight individual background tiles necessary to build the construction site (figure 2.5).
2.5 Famicom Donkey Kong’s background pattern table viewed in YY-CHR. Note the individual height variations necessary to draw the angled girders.

There are tiles drawn for every single pixel variation of the girders’ placement, along with those that require ladders. In other words, the first two tiles in a row may need a one-pixel sliver of the girder, the next a two-pixel sliver, the next a three-pixel segment, and so on until the girder’s length is complete. Considering NROM’s severe CHR-ROM restrictions, this technique is incredibly wasteful, but it is both true to the arcade’s look (and process, which draws its slopes the same way) and visually pleasing. In arcade Kong, the girder slopes are more drastic, with vertical pixel increments occurring every sixteen pixels. One beam may have up to thirteen individual segments. The Famicom port’s decreased vertical space could not support such steep inclines. Its middle girders are nine segments long, each segment measuring four tiles across (32 pixels). In either case, level 1 would not have the same appeal if its girders were flat parallels, nor would it offer the same risk/reward structure: taller ladders make Mario more susceptible to falling barrels but they provide a quicker route to the top. It is no surprise, based on memory constraints, that level 1 is the
only level to feature graded platforms.

**Girder Fonts and Missing Pies**

*Donkey Kong*’s programmers used another space-saving technique to construct the title screen. Arcade and Famicom alike used a single tile (the platform graphic from the rivets level) to spell out ‘DONKEY KONG’ in large letters. The letterforms have minor variations: the Famicom version has a wider ‘D’, one shifted tile in the ‘K’, a simpler ‘G’, and thinner letters on the bottom row (figure 2.6).

2.6 *The Famicom Donkey Kong title screen uses girder tiles as letterforms. (Emulator: Macifom 0.16)*

The letters are also better proportioned in the port. Both lines are center-justified and
‘KONG’ has symmetrical four tile columns of space on either side. The arcade version has an extra tile column on the left and two tiles for a trademark (’™’) crowding the right gutter, along with Donkey Kong’s sprite positioned menacingly below. The port drops the trademark, the upper hi-score display, and Kong himself—all due to disparities in vertical space and the need to list the range of selectable game modes for home play.

The use of background tiles as letterforms was not solely an artistic decision. In bitmapped graphics systems like the NES, there are no system-level character sets or prefab fonts. Every element seen onscreen is sourced from the pattern tables. Alphanumeric characters are drawn pixel by pixel and positioned onscreen like any other tile. This is why text in Famicom games is proportioned and positioned according to a strict grid, since each letter is typically an 8x8 segment of the name table. Both arcade and Famicom versions of Donkey Kong include a full alphanumeric set: twenty-six letters, digits from zero to nine, and a few special characters. It is evident that Nintendo borrowed directly from the arcade’s character set for the home version, since the alphanumerics are nearly pixel-for-pixel identical. Only a few are subtly modified (e.g., the arcade’s ‘W’ has angled strokes rather than vertical stems).

Lifting a typeface makes sense for an arcade-accurate port, but not for the limitations of the home platform. Many reviews mentioned various ports’ lack of the cement factory or elevator stages, but few mentioned the lack of the arcade’s high-score screen. During its attract cycle, Donkey Kong showed the title screen, the high-score table, then actual play. The order is significant. Imagine a skilled player who returns to the arcade on a regular basis to check up on her competition. She is not interested in a gameplay demo since she has played the game many times. She wants to see the leader board first. Kong ranks the five highest players with rank, score, and most importantly, a three-character name. The user-editable ‘NAME’ column justifies the ROM space devoted to the full (English) alphabet. It is a limited palette for competitive expression, but it serves the purpose of allowing a skilled player to leave her signature.

Arcade Donkey Kong’s character set, leader board, and upper display are residual evidence of its conversion heritage. The typeface, unsurprisingly, is identical to Radar Scope’s. The upper informational portion of the screen, save for its palette differences, also shares Radar Scope’s configuration, down to the character spacing between lines. Stranger still are the shared default values for the current 1UP (‘003700’) and HIGH SCORE (‘007650’). In fact, all five default leader board entries, as well as the leader board screen itself, are identical in both games. It is likely that Nintendo used a common character set across several arcades games in order to save time during development, a practice befitting their long corporate tradition of re-use. The few differences are primarily gameplay-related: Donkey Kong represents Jumpman’s lives graphically rather than numerically, changes the ‘P=00’ to ‘L=0’ to denote level changes, and adds the distinctive BONUS timer.
Famicom Donkey Kong, in contrast, adopts the full alphanumeric set without a real need to do so. The cartridge maintains the player’s current score and the overall high score, but both revert to defaults when the system powers down. In fact, the high score is wiped from RAM even on a soft reset. Beyond the score displayed along the upper portion of the screen during demo and game play, there is no high score leader board, no rankings, nor any meaningful score persistence. Unless a player left the machine running continuously (as arcades often did to maintain hi-scores), the scores reset. Players could not enter their initials, so the hi-score lacked any competitive attribution. If Nintendo had eliminated the tiles for letters that never appear onscreen, they could have liberated nine slots of pattern table memory. Alongside the thirty-odd tiles of unused space in pattern table 0, they may have been able to include some of the tiles necessary for the deleted interstitial animations. Time constraints surrounding the Famicom’s launch likely played a major role in Nintendo’s failure to maximize their 8KB of real estate.

Animation interstitials are arguably supplemental to Kong’s core gameplay, but the most significant cut to the Famicom port is the ‘pie factory’ level, nicknamed after the cement trays that resemble banana cream pies traveling along conveyor belts. As described above, Donkey Kong originally had four distinct playable screens. Once the full screen rotation kicked in, the cement factory slotted into level 2, following the construction site. For the port, Nintendo omitted level 2 due to space constraints: the additional graphics for cement pies, a center lattice, conveyer platforms and pulleys, along with the logic driving the stage would push the NROM beyond its limited PRG/CHR space. Likewise, in the arcade version, Kong himself is pulled along right and left by the shifting conveyor belts. Kong’s alternating motion serves as a quick visual cue for which direction the belts are moving since the tracks themselves do not animate (though the belts’ end caps do rotate). Since Kong is a sprite in the arcade version, updating his position is trivial. The same is not true for the Famicom Kong. Since he is comprised of background tiles, moving his body, if possible, would involve a sophisticated mid-screen scrolling effect not seen until later Famicom titles. Similarly, Kong passing in front of the ladders at the top of the platform would require a significant amount of programming work, as they would need to be built from sprites and set to a lower priority than the background tiles.

Despite these programming challenges, there is a code-level clue that level 2’s inclusion was at least considered. The zero page RAM address at $53 holds the current stage variable. It may contain three valid values—1, 3, and 4—to designate construction site, elevators, and rivets stages, respectively. Value 2 is conspicuously absent. With an emulator, one can manually insert ‘2’ into address $53 during gameplay, creating a glitched hybrid of game elements from all three stages: barrels fall, a duplicate Kong (with improper palette) is on the center of the top platform, and Mario can fall from a great height, as if the elevators are present. Clearly, the pie factory was cut during production, rather than prior.
In 2010, Nintendo officially re-released Famicom Donkey Kong pre-loaded on the PAL Wii as part of the 25th anniversary of Super Mario Bros. The updated ROM (i.e., game image) runs on the Wii’s Virtual Console via emulation. There are several minor visual differences from the NES/Famicom version, among them a revised title screen that reads ‘©1983-2010 NINTENDO’ instead of ‘©1981 NINTENDO CO., LTD.’ Note the subtle distinction: the Famicom original references the arcade game’s copyright date while the re-released version references the Famicom port’s copyright date. Again, Nintendo did not develop Donkey Kong in-house. Miyamoto designed under Yokoi’s supervision, but the programming and circuit production subcontracted to Ikegami Tsushinki Co., Ltd. This partnership was largely hidden until a lawsuit between the parties revealed that Ikegami sued Nintendo over the right to Donkey Kong’s source code. Though the outcome of the claim settlement was never released to the public, there is evidence of the partnership in the game’s character data: a small, turquoise eye-shape with the letters ‘ITC’ set in the center—Ikegami’s logo. The soured partnership explains the minor modification to the ROM’s title screen. Nintendo can no longer claim copyright for the arcade original, so they shifted their copyright date to the Famicom debut.

Legal subterfuge notwithstanding, most conspicuous in the Wii re-release is the return of the cement factory in its proper position at level 2. The update also reinstates a few interstitial animations—once Mario reaches the top of each platform in the first three levels, Donkey Kong grabs Pauline and carries her up the ladder. However, none of the opening cinematics or ‘How High Can You Get?’ screens are included. Similarly, Kong’s tiles remain unaffected by the shifting conveyor belts on his platform—though the programmers cleverly chose not to animate the end caps, opting for a diegetic mechanical solution to a technical limitation.

How were Nintendo’s retroactive additions possible, considering the original port’s lack of space? Industrious hackers managed to extract the Wii’s Donkey Kong ROM and examine its contents. The ROM no longer fits the NROM profile—rather, it was updated to the more spacious CNROM mapper, allowing for 32K of bank-switchable PRG and CHR. Disassemblies of the revised ROM reveal that much of the source code is identical to the 1983 release, with a few rather slapdash patches to shoehorn level 2 into the code. For instance, the level variable at address $53 now has a viable entry for level 2. Poking this value during other stages creates glitched behaviors consistent with the injected level (e.g., barrels begin to move like cement pies). Lost Levels forum members conjecture that Nintendo hacked their own ROM rather than digging up old source code.

The pie factory’s inclusion also makes the Famicom’s sprite limitations glaringly obvious. Since pies travel along horizontal conveyor belts, there is considerable flicker when multiple pies share the same scanline. Barring the other technical reasons Nintendo may have chosen to drop the cement factory, the rampant flickering likely helped ease the
decision of *which* level to omit. All things equal, it makes sense to drop the level that suffers the most from the limitations of hardware.

**Manic Compression**

Today, little of our media is measured in kilobytes. The annotated, plain text disassembly of *Donkey Kong I* consulted while writing this section occupies 353KB, nearly fifteen times the size of the NROM’s PRG-ROM and CHR-ROM combined. In such constrained conditions, every byte counts.

Compared to later Famicom/NES games like *Mega Man 2, The Legend of Zelda, Ninja Gaiden, or Metroid*, *Donkey Kong*’s gameplay is rudimentary. There are four non-scrolling play screens, a title screen, a small handful of sprites, limited character movement, and no persistent game states (i.e., you can not save your game or high score). But the aforementioned games benefited from advancements in cartridge or disk hardware that would grant bank-swappable memory for code and graphics, on-board batteries for game saves, enhanced sound, and other mapper-specific features. We know that Uemura and R&D2 had *Donkey Kong* in mind when they designed the Famicom, but its base specifications fell just shy of delivering the complete arcade experience. And despite excising an entire level, the high score table, and a number of animations, the programmers still relied on compression to minimize their use of ROM space.

The basic role of compression is to eliminate data redundancy. Fortunately, Famicom graphics are stored as symmetrical 8x8 or 8x16 pixel tiles. Owing to the limited set of pattern tiles available, many onscreen elements are repeated to form larger structures. To draw a platform, a designer might use only three tiles: two caps at either end with a long string of identical tiles in the center. These may be more complex or, in *Donkey Kong’s* case, even simpler—the bottom platform of the rivets level is a single 8x8 tile repeated horizontally across the screen.

How might a programmer create this bottom platform? Here’s one way:

1. Choose the first tile’s starting name table location (bottom left).
2. Pick the appropriate tile from the pattern table.
3. Draw the tile at that location.
4. Shift to the next position on the name table, eight pixels to the right.
5. Repeat steps 3 and 4 until the entire platform is drawn.

Step 5’s redundancy should be obvious. The programmer knows in advance what tiles will be used to draw the platform and how wide that platform will be. The data used to store the platform’s shape takes up precious ROM space. When only a single tile is involved, it is a
significance waste of both CPU cycles and memory to repeat the same steps again and again. Compression slims the redundancies.

One of the simplest compression schemes is called run-length encoding, or RLE. RLE is used in situations like the example above, when a run of elements repeats for a prescribed length. If our rivet tile’s pattern table index byte is, say, $21, and the platform is fifteen sprites long, using the five steps above we might represent the platform data in code as:

$21,$21,$21,$21,$21,$21,$21,$21,$21,$21,$21,$21,$21,$21,$21

We draw each individual tile fifteen times, using fifteen bytes of space. Using RLE, however, we could encode that run as follows:

$21,$0F,$00

The first byte is the hexadecimal reference to the rivet tile, the second byte designates the desired number of repetitions (hexadecimal $0F$ equals decimal 15), and the final byte terminates the sequence. The fifteen-byte data block compresses to three bytes—an eighty percent reduction.

In practice, RLE requires a few more details. For instance, there must be a related routine that understands how the compression scheme is formatted so objects may be properly uncompressed. Keep in mind that RLE was not an automated routine built into Famicom software, like inserting a disc into iTunes, clicking a button to ‘Import CD,’ and having a nicely packaged .mp3 come out on the other end. Coders had to ‘roll their own’ compression algorithms. But the extra overhead required to interpret the data was usually worth the effort relative to the memory saved, as Donkey Kong’s programmers proved.

We saw in chapter 1 that the PPU is synced carefully to the ebb and flow of the television’s electron beam. As the gun rakes its pattern of scanlines down the glass, it has two distinct periods of rest, the horizontal blank (HBLANK) and the vertical blank (VBLANK). HBLANK occurs after the completion of a scanline, when the gun must reset from one side of the screen to the other. The HBLANK window is minuscule, even in microprocessor terms—without tightly-timed code or additional hardware help, it is risky to issue new commands to the PPU during that time. (Updating the PPU while the scanline is actively being drawn is a Famicom development no-no, resulting in nasty onscreen glitches.) VBLANK has a wider window, since the electron gun must travel the full height of the screen. When the television enters this state, it triggers a non-maskable interrupt, or NMI. ‘Non-maskable’ means that this signal cannot be suppressed. In essence, whatever the CPU was doing at that moment is immediately interrupted and sent to the NMI handler. How the programmer chooses to use this time is largely up to their discretion, but the
VBLANK happens without fail (barring any hardware malfunctions).

Good programming practice on the Famicom dictates that developers use those fleeting slivers of time wisely. If the game’s state has changed during the previous frame (e.g., a player presses a button), graphic updates likely need to be made—sprites move a few pixels, name table tiles swap, and so on. And VBLANK is the only time (assuming the PPU is enabled) tiles may update without risking graphical glitches. The catch is that an entire screen of graphics—960 tiles total—can not update during a single VBLANK. The margin is too narrow and the CPU is too slow. To make the most of VBLANK, the programmer must make good use of time preceding the NMI, getting their proverbial house in order before the lights come back on. That means keeping a running list of what needs to be updated and where, then waiting for the NMI to trigger. As soon as the interrupt fires, the code is ready to run its updates, shifting or replacing tiles before the PPU resumes rendering.

In *Donkey Kong*, there is a background update list stored at memory locations $0330M-036F$. The first byte stores the current list size, which is reset to zero (#$00) at the end of each NMI. The bytes that follow describe a queue of background tiles that require updates during the next VBLANK. During normal gameplay, major and minor events take place, ranging from full level transitions to the bonus timer decrementing, that require name table updates. When these events trigger, a code subroutine updates the current list size and then appends the relevant background tile to the update list. However, due to the list’s limited memory scope (1 byte for the ‘header,’ 63 bytes of tile data), there is a hard upper limit to the number of updates per frame. Each time a new tile appends to the queue, a separate subroutine runs a check against the list limit. If the limit is exceeded, the subroutine rejects the append. The simple check manages the number of tiles, ensuring that all updates fit within VBLANK.

Tile updates may also use RLE compression when needed. The VRAM write routine called during NMI requires three bytes of information to draw the appropriate tile. The first two hold the 16-bit VRAM address where the tile (or first tile in a run) will be placed. A third byte contains three bit flags: orientation, RLE encoding, and run length. The orientation bit indicates whether tiles will run horizontally or vertically from their tile origin. This flexibility is catered to *Donkey Kong’s* design, since the levels contain both horizontal (platforms) and vertical (ladders) strips of tiles. The second bit indicates whether RLE is ‘active’ for the given tile update. The remaining six bits store the run length, whose maximum possible value is $\%111111$, or 63. Notice that the run length limit is identical to the update list limit. In practice, there is no background element in *Donkey Kong* that requires sixty-three repeated tiles, but the coincidence between limits is clearly there by design.

As the bit flag above indicates, RLE is optional. There are some instances where RLE is
detrimental to code compression. A single tile update, for instance, requires more information to encode the ‘compressed’ data than it does to use the data itself. Donkey Kong’s engine is impressively agile for a Famicom launch title. It uses vertical strips of RLE-encoded rivet tiles to draw the letters on the title screen, but keeps RLE inactive when it needs to update a single tile in the level indicator. It handles both cases with a single bit toggle.

Donkey Kong received one other notable amendment during its translation from arcade to console: the infamous kill screen. Due to a programming oversight in Donkey Kong’s code, arcade play cannot progress past level 22. The BONUS timer, which serves as the countdown clock for each level, is calculated by multiplying the current level number by ten then adding forty. Once the player reaches level 22, the calculated result exceeds the maximum value of a single byte, $FF$ (or 255). Rather than catching the byte overflow and adjusting accordingly, the value wraps around to zero and loads the BONUS with inadequate time to finish the stage. After a few seconds of play, Jumpman halts, spins, and dies. Donkey Kong’s kill screen earned widespread attention thanks to the 2007 film King of Kong, since the programming bug set a hard limit on high scores. Unlike many arcade games, competitors can not play Donkey Kong until they are too tired to continue or the score counter resets. Besting the world record requires careful strategy to maximize the score before the kill screen occurs.

Though arcade and Famicom Donkey Kong do not share the same source code, the latter remarkably ‘ports’ the kill screen. The difference is that the Famicom version delays its appearance for far longer. The game stores the current level in a single byte located at $54$ in zero page. However, the current level is not the same as the current stage; rather, the game requires a complete three-stage cycle (girders, elevators, rivets) to increment the level count by one. Internally, the level variable starts its count at 0, which is then incremented prior to display onscreen, starting Mario at level 1. The BONUS counter can only be one of four values—5000, 6000, 7000, or 8000—corresponding to current level values 0, 1, 2, and 3 respectively. To determine the BONUS counter (in reality, its first two digits, since the second two are always zero), decimal value 4 is subtracted from the current level variable. If the result is negative, a code branch occurs and the leading digits are fetched from data stored in a small lookup table. Any positive result (including zero) ignores the fetch and loads the BONUS with its maximum value. Ideally, any level beyond the third should yield a BONUS of 8000.

The subtraction check functions properly until the level variable reaches $84$ (or 132). Subtracting $04$ from this value yields $80$. Though it appears the result should pass the positive check, it does not, due to a quirk of binary math. A single byte can represent either unsigned values 0 to 255, or signed values $-127$ to 127. The latter choice is called two’s complement and relies on a special binary bit flag to determine the appropriate sign.
Two’s complement reserves the leftmost bit of a byte to indicate either positive (0) or negative (1) numbers. When Donkey Kong’s BONUS check yields $80, whose binary form is %1000000, the CPU may evaluate the number as either 128 or −128. Since the programmers did not include a check for the negative flag, the CPU evaluates a two’s complement value, causing the code to branch improperly and fetch the wrong leading digits for the BONUS timer (04). With only 400 time units allotted, Mario cannot reach the top of the stage. Consequently, level 133 is the kill screen.

It is likely the two’s complement edge case was not a pressing concern for Donkey Kong’s programmers. Few people have the skill or patience to endure the 300+ screens buffering the player from Mario’s binary-induced death spin. In fact, the programmers appear to have accommodated fewer than ten level iterations. The subroutine that fetches the background tile to display for the current level has no safeguard to cap its incrementing beyond numeral 9. Even if it did, the screen layout does not provide the adequate space to display double-digit values in the level bracket. And since the numeral tiles are stored at the beginning of the background pattern table, the code uses an index value to seek the appropriate digit to display on screen. Venturing beyond level 9 first causes the graphic index to cycle through the alphabet tiles, then the sequential array of background graphics. Far before they made it to the kill screen, the finest Donkey Kong players would have noticed peculiar tile fragments in the level display—a section of barrel or a bit of Kong’s body—indicating that they were in ‘forbidden’ territory.

**Beyond Jumpman**

The technical affordances of early Famicom cartridge hardware did not permit an arcade-perfect port of Donkey Kong. The 24KB NROM simply did not have the adequate memory. In fact, few home consoles were capable of porting Kong’s full stock of levels, graphics, and animations. But the NROM board served the technical needs of Nintendo’s opening salvo of games—the arcade conversions that would stabilize the Famicom’s launch, prove its hardware mettle, and attract the attention of third-party developers. Replicating and exceeding Coleco’s Donkey Kong port was a clear target for Nintendo’s console, but the results were mixed. The Famicom had better palettes, better resolution, and better sound than the ColecoVision, but Nintendo’s engineers still had to concede the cement factory level and the charming animated skits that stitched Donkey Kong’s love story together. They bested their console peers, but fell short of the ultimate goal.

*Donkey Kong’s* dual appearance in arcades and on the Family Computer bookended a rapid transformation. In two years, Nintendo evolved from eclectic Japanese toymakers to global players in the videogame industry. They did so by deftly navigating a series of challenging ports, not only in the technological sense of adapting *Donkey Kong’s* code from
its Z80 hardware to the Famicom's 6502, but also in the cultural sense—‘porting’ the scenario and character archetypes of Popeye to Kong's micro-narrative, or Japanese manga's four-panel visual style to a four-screen layout. Nintendo's engineers and designers consistently mined and repurposed their own work, leveraging otherwise outdated or inexpensive technology to architect new experiences. From the ashes of Radar Scope came Donkey Kong; from the outdated 6502 came a console that would shape and dominate the market for many years.

The Family Computer was designed with Donkey Kong in mind, despite the cuts necessitated by the limited scope of the cartridge ROM. But it was engineered with affordances that looked beyond Donkey Kong, toward new styles of play. None of Nintendo's three Famicom launch titles required scrolling, for instance, but Ricoh and R&D2 designed a PPU capable moving name tables easily. By all accounts, Nintendo was not looking to capitalize solely on their arcade successes; they were looking ahead, to the possibilities of future game experiences. There was more in store for Jumpman.

¹ Masaharu (trans. Tanner), “Part 6 - Making the Famicom a Reality.”
² The game’s name is often cited as Radarscope, though the arcade's title screen clearly displays ‘RADAR SCOPE,’ with a space. The error likely derives from the marquee title, where the space between the two words is slender (but legible).
⁴ The marketing and proposed cabinet design emphasize Nintendo's immersive aims. The flyer features the game screen superimposed with a first-person view of ‘your’ hands manning the ship's gun turret while enemies fly toward you. The ad copy reads, ‘YOU ARE HERE.’ (Fig. 1) Additionally, one version of the arcade cabinet is a partially-enclosed ‘cockpit’ type, meant for players to sit inside.
⁵ Ostermayer, The Radar Scope Pages.
⁶ See Kent, The Ultimate History of Video Games, 157.
⁷ Arakawa blamed part of Radar Scope's failure on the shipping time from Osaka to New York. Soon after, Nintendo of America relocated to a 60,000 square foot office in Seattle, reducing the ship time to one week. See Firestone, Nintendo: The Company and Its Founders 52-3.
⁸ “Zoo Keeper,” The Arcade Flyer Archive.
⁹ “Donkey Kong 3,” The Arcade Flyer Archive.
¹⁰ “After Donkey Kong’s success, King Features was less reticent. Nintendo later released Popeye as a Game & Watch handheld (1981), arcade title (1982), and Famicom launch game (1983).”
¹¹ “Gunpei Yokoi talks Donkey Kong in ‘Gunpei Yokoi’s House of Games’,” The End of Deep Layer.
¹² Note that Yokoi retcons Princess Peach, Mario’s damsel in distress from Super Mario Bros., into Pauline’s place.
¹³ Ibid.
¹⁵ Lacey, “Novelty Games - Cute is Crucial.”
¹⁶ Bloom, Video Invaders, 181.
¹⁷ Kohler, Power-up, 2 (author’s emphasis).
¹⁸ Napier, From Impressionism to Anime, 3.
¹⁹ Patten, Watching Anime, Reading Manga, 54.
"Patten, 55-7.
"An October 1977 bulletin for a C/FO meet-up features a fan-made illustration of intermingling American and Japanese cartoon characters. In the foreground, none other than Popeye grins and rests his hand on Astro Boy's shoulder. Patten, Watching Anime, Reading Manga, 58.
"See Patten, Watching Anime, Reading Manga, 22-43.
"Levi, Samurai from Outer Space, 7-8.
"Levi, 8.
"Kohler, Power-up, 11.
"Kohler, Power-up, 18, 24, 207, 11.
"Levi, 19.
"Patten, 58.
"Ironically, Atari is a Japanese word that founder Nolan Bushnell lifted from the game Go.
"For detailed accounts of the Nintendo / Universal litigation, see Sheff, 116-127 and Kent, 210-218.
[edited for clarity]
"Sheff, 48-9.
"Kent, 158.
"Gross, "Nintendo seeks to trademark ‘On like Donkey Kong’."
"Kohler, Power-up, 39.
"See “Popeye - You’re A Sap, Mr Jap (1942).” YouTube.
"Sheff, 19.
"See http://en.wikipedia.org/wiki/Donkey_Kong_(video_game)#Clones
"Oddly, Nintendo licensed the Donkey Kong to Japanese corporation Falcon Inc., granting them rights to produce an official clone: ‘In September 1981, Nintendo Co., Ltd. entered into a licensing agreement in Japan with Falcon, Inc., another Japanese corporation. This agreement authorized Falcon to produce a game called Crazy Kong which was to be identical or similar to Donkey Kong. Nintendo Co., Ltd. supplied stickers to Falcon to attach to the printed circuit boards used in the Crazy Kong game. The stickers were printed in the English language and indicated that the Crazy Kong circuit board was manufactured under a license from Nintendo Co., Ltd. Falcon paid a royalty of 10,000 yen to Nintendo Co., Ltd. for each game manufactured by Falcon. The license agreement expressly limited the right of Falcon to sell or use the Donkey Kong game under the name Crazy Kong to the territories of Japan. It prohibited Falcon from importing or exporting these machines or any similar machine and from having any third party produce or sell any such machine. It also prohibited Falcon from importing or exporting Donkey Kong or Crazy Kong circuit boards into the United States or anywhere outside of Japan. The license agreement was terminated on January 29, 1982.’ See United States District Court, “Nintendo of America, Inc. v. Elcon Industries, Inc.”
"Sheff, 116-17.
"Commodore missed the first-mover advantage on the Nintendo arcade license. CEO Jack Tramiel called a last-minute halt to the contract due to a competing agreement with Bally-Midway. See Bagnall, On the Edge, 218.
"Kent, 209-10.
"Atari eventually brought Donkey Kong’s development in-house, delivering ports for the Atari 400/800, as well as the XL and XE series. Atari engineer Landon Dyer was responsible for the adaptations.
"See Fromm, “Troubleshooting Flow Charts for Monitors.”
“Dot crawl.” Wikipedia.
“There are exceptions, most notably the Vectrex, whose embedded (vertical) monitor was a necessity; as its name suggests, it used vector- rather than raster-based graphics. Handheld or portable consoles would also qualify.
“See Katayama, 165: ‘By the late seventies, colour televisions had already reached a 95 per cent penetration rate in Japan...’
“However, name table tiles can be moved with scrolling.
“Parallel girders would also preclude Kong’s introductory show of strength, as he stomps the girders into their angled positions. This seems like a negligible detail, but consider its narratological and instructional purposes: for unfamiliar players, Kong is quickly established as the foe, and a formidable one to boot.
“On power-up, the cycle is abbreviated: the hi-score table displays first, then demo play. After that first iteration, the game locks into the loop described.
“Donkey Kong’s ancestry is visible outside the screen as well. The first few thousand converted units have Radar Scope’s distinctive red side panels instead of the blue that was later used. The red Kong cabinets are consequently much rarer and command higher prices from collectors.
“By my count, the following letters are unnecessary: F, H, K, Q, S, U, W, X and Z.
“This may back up my theory that it is more game logic than tile space that discounted the cement factory’s inclusion. Nintendo simply may not have had the available ROM space to include the logic driving an additional stage.
“Moving Kong along the conveyor was a challenge for any platform. The Atari 800 version and Ocean’s Commodore 64 port (1986) left Kong stationary. The DOS and Apple II ports scrolled Kong poorly, allowing his sprite to merge into the ladder graphics as he passed by. Atari’s 1983 Commodore 64 release did the most admirable job—unsurprising, considering the C64’s ability to handle larger sprites (24 x 21 pixels, 8 sprites onscreen simultaneously). See “Chapter 6 - Sprite Graphics” in Commodore Computer, Commodore 64 User’s Guide.
“Kemps, “Europe gets exclusive 'perfect version' of NES Donkey Kong in its Mario 25th Anniversary Wiis.”
“Game Developer Research Institute. “Company: Ikegami Tsushinki.” Also, from United States District Court., “Nintendo of America, Inc. v. Elcon Industries, Inc.”: ‘Nintendo Co., Ltd. expended over $100,000.00 in direct development of the game, and Nintendo Co., Ltd. hired Ikegami Tsushinki Co., Ltd. to provide mechanical programming assistance to fix the software created by Nintendo Co., Ltd. in the storage component of the game. The name "Ikegami Co. Lim." appears in the computer program for the Donkey Kong game. Individuals within the research and development department of Nintendo Co., Ltd., however, created the Donkey Kong concept and game.’
“ Ibid.
“All Famicom Donkey Kong source code examples refer to pditincho, “Donkey Kong Disassembly Revision 4.”
“Kulczycki, “Technical: What’s with the Kill Screen.”
“ For a lengthier discussion of the kill screen port, see Altice, “Porting the Kill Screen.”
“The same behavior occurs in Super Mario Bros. when the total store of lives exceeds nineteen. See chapter 4.
3: The Toaster

*Overall, if anybody can bring video games back, Nintendo, with its new fourth-generation game system, will be the one.*


*I only want Three Things for Christmas, this year. I’m getting older so I don’t want any toys. I want a ten to nine-teen inch color television, a jean acid washed, insulated jacket and last but not least The Nintendo Entertainment System.*


Today, the debut of a videogame console is a worldwide media event. Years prior to its appearance, the rumor mills churn over leaked details from microchip vendors, overseas assembly lines, and game developers tapped to receive advance development kits. Hardware prototypes are revealed during industry showcases like E3, Gamescom, or the Tokyo Game Show. Technical specs are scrutinized and compared. Preview builds of ‘next generation’ software are unveiled to showcase the superior capabilities of the new machine. Excitement and anticipation build until the console’s launch date, a term borrowed from the maiden voyages of nautical craft or space shuttles. Hyperbolic perhaps, but an apt comparison—these multi-million dollar experiments in cutting edge technology are jettisoned into the treacherous waters of free markets to either sink or survive.

In the early 1980s, there was no such mania. Videogame consoles rarely had publicized launch dates. Their manufacturers introduced them to department and toy stores like any other new product, typically testing the waters in select target markets before moving on to a nationwide—not worldwide—rollout. A single day global launch was both financially risky and a significant manufacturing challenge. This is still true today. While the ‘Big 3’ publishers—Sony, Nintendo, Microsoft—manage to launch their home consoles ‘worldwide,’ the adjective typically describes Japan, Western Europe, and the United States. India, Asia, Australia, South America, and other parts of the world receive new consoles on a staggered schedule, sometimes years after the advertised launch date. Portable systems and videogame software tend to follow traditional distribution models, launching in their native market first, then disseminating across the globe. The difference between staggered launches of the 1980s and the 2010s is timing. While American consumers may wait six months to receive the latest iteration of Nintendo’s DS handheld, they had to wait years for the Famicom to reach Western shores. In reality, there were few
waiting for it at all. Save for a handful of the most devoted videogame importers or games journalists, no one outside Japan knew what a Famicom was.

When the Famicom did arrive as the Nintendo Entertainment System, or NES, it looked radically different from its Japanese sibling. The colorful red and white plastic toy was replaced by a subdued monochromatic box, known affectionately as ‘the toaster’ due to its curious front-loading flap. It was no longer a family computer, but a legitimate ‘entertainment system,’ meant to sit inauspiciously among one’s VCR, stereo system, and cable box. The Famicom’s colorful cartridges—now stark grey ‘Game Paks’—no longer protruded from the top of the console, but slid discreetly into the front of the system. The console’s packaging floated the system and its peripherals against a gradient of deep blue melding into a star field (not unlike the backdrop of Radar Scope) and marketed its contents with forceful adjectives: ‘Control Deck,’ ‘Deluxe Set,’ ‘Action Set,’ ‘Power Set,’ ‘Super Set,’ and ‘Challenge Set.’ It was all serious, futuristic business—a months-long, calculated strategy to introduce Nintendo’s new console to the world.

**American Crash**

It took a remarkable mixture of talent, timing, and tenacity to bring the NES to life. By all industry accounts, Nintendo was heading toward a massive failure far beyond the scale of a few thousand unsold Radar Scope cabinets. In 1983, when the Famicom debuted in Japan, the US videogame market was tumbling toward disaster, threatening to wither away just as it had begun to blossom.

The so-called ‘videogame crash’ of 1983 is well-trodden in surveys of videogame history.¹ A complex series of economic, industrial, and cultural factors coalesced into the wholesale failure of the industry’s major players. But prior to the crash, the industry was booming. In 1981, arcade revenues exceeded $4 billion; Pac-Man alone accounted for $150 million of that total.² By 1983, home videogame sales, then dominated by Atari’s eighty percent market share, contributed an additional $3 billion, a more than sixfold increase from revenues in 1981.³

Atari’s explosive success became part of their downfall: with millions of consoles in American homes, the Atari VCS became the target platform for first-rate, second-rate, and cut-rate software alike. Beyond its in-house development teams, Atari had little control over the quality of software that reached its console. The flood of mediocre games surged so steadily that it became difficult for consumers to differentiate the good from the bad. When Atari bet big on licensed properties like Pac-Man and E.T., then shirked on the time allotted for programmers to produce quality code, the situation worsened.⁴ Retailers slashed prices on poor or overprinted games, creating a race to the bottom for software prices that made it difficult for quality games to stay profitable.
Supply-side troubles, again prompted by the videogame boom, led to unwieldy product lead times and increased competition among console manufacturers. Chip shortages compelled toy companies to hedge their inventories, forcing excess product into the retail channel. Warehouses and store shelves were flooded with videogame hardware. As more competitors followed the gold rush, advertising budgets skyrocketed. The growing personal computer market exacerbated these effects, as those manufacturers were now sourcing the same components for their systems and competing for consumer dollars. PC encroachment likewise spurred videogame manufacturers, hoping to capture part of the emerging market, to produce ill-considered hybrids like the Coleco Adam that failed to distinguish themselves as either capable computers or videogame systems.

Cultural differences between external management and free-wheeling Silicon Valley programmers were also coming to the fore. Development cycles were compressed to capitalize on the boom, while game designers were given little or no credit for their work. Videogame companies were acquired by large media and communications conglomerates. In a symbolic move, Nolan Bushnell, Atari’s founder and a figurehead for the early videogame industry, left two years after Warner acquired his company. As videogames became big business, the atmosphere of fun, creativity, and innovation gave way to profit margins, corporate infrastructure, and market demands.

Likewise, the public perception of videogames was in trouble. As children and teens spent increasingly more time in arcades or in front of screens, parents and legislators became concerned for their moral fortitude. Surely the barrage of twitchy, fast-paced graphics and violent gameplay was causing long-term harm. As Donovan recounts, even the US Surgeon General Dr. Everett Koop weighed in: ‘Everything is “zap the enemy,” there’s nothing constructive.’ Arcades began to face tighter restrictions, even outright bans.

The reckoning was brutal. What began as a general slow-down in demand careened over the brink into a vertiginous crash that all but wiped out the industry in North America. Time Warner/Atari sales of two billion dollars in 1982 dropped forty percent the following year and the division lost $539 million. The crisis worsened because companies had leveraged capital in anticipation of constantly escalating sales. By 1984 revenues had dropped to less than half of what they had been two years before.

Several promising US consoles and companies were caught in the ensuing death spin. The ColecoVision, despite its early success as the console for arcade ports, was swept into the dustbin. GCE/Milton Bradley’s innovative Vectrex console, featuring a built-in vector display, was abandoned before it had a chance to find an audience. Even Atari’s own 7800, a marked improvement over the VCS that rivaled the Famicom’s graphic capabilities,
drowned in the wake of the crash. Journalists speculated that the console videogame craze was giving way to a PC-centric future as computers reached price parity with videogame hardware. The New York Times reported that, ‘For $75 to $200, consumers can buy a basic computer that can play games - although without some of the advanced graphics available on video game machines - and can also serve some educational functions and figure the family finances. Computer makers are betting that consumers will sacrifice better game-playing for more serious pursuits.’⁸ The future for dedicated videogame consoles seemed dim in the face of cheaper, more capable, all-purpose machines.

**Advanced Video System**

With the US videogame industry in steep decline, Nintendo knew that they would not be able to export their Family Computer as-is. Toy retailers were fleeing at the slightest whiff of a new videogame console. They had learned a calamitous lesson with Atari that they were not keen to repeat. Nintendo’s success would require a radical re-thinking of the Famicom’s design, more consistent with the trends of the US PC, toy, and consumer electronics industries than the tastes of Japanese children. To do so, Nintendo turned to an American designer.

Nintendo’s first attempt at the American Famicom, dubbed the Advanced Video System (AVS), debuted at the January 1985 Winter Consumer Electronics Show (WCES) in Las Vegas.⁹ The AVS was an inspired sample of 1980s futurist industrial design. Nintendo of America designer Lance Barr was tasked to make the US version of the console ‘high-tech sleek yet accessible,’¹⁰ more computer than toy. Ironically, the resulting designs are more properly described as a ‘family computer’ than the Famicom ever turned out to be.
An early concept sketch of the AVS (figure 3.1) showed a sloped trapezoidal black unit, wider in the front than back, with controllers docked along its upper edge, separated from the remainder of the system by a narrow track of ridged black plastic. The controllers were stark black with square and rectangular silver buttons. A square plate with red arrows indicating the four cardinal directions replaced Nintendo’s patented cross-shaped switch. Like the Famicom, only the leftmost player one joypad included Select and Start buttons, though they were stacked vertically, aligned to the immediate right of the directional pad rather than the center of the joystick. The face of the system was primarily a keyboard whose keys were flush with the surface but had shallow concave depressions to accommodate typing. Below the keyboard were two rows of additional keys, similar to the alphanumerical keys, but arranged like a piano keyboard covering nearly two and one half octaves. The ‘sharps’ were labeled numerically from one to twelve, allowing them to pull double-duty as standard PC function keys (F1-F12). Similarly, in the space between the nested controllers, there were four silver keys, shaped as complementary parallelograms. A red Nintendo logo was emblazoned on the left corner of the system, a stark highlight on an otherwise monochromatic black console. The overall package resembled a misplaced Blade Runner prop—dark, sleek, and angular, but clearly inspired by early-1980s PC design.
This early concept sketch was never realized. The final AVS prototype shown at WCES carried over several of the design cues from Barr’s initial concept but in an ‘exploded’ view. The integrated system was scrapped in favor of a more modular design (figure 3.2). Three main components—control deck, keyboard, tape deck—were meant to stack together symmetrically and, Voltron-like, form the complete Video System. The concept sketch for this version of the AVS reveals a mildly enhanced palette of white, grey, and black, with accents in red—the same palette used for the final NES hardware. The keyboard is still the largest component, meant to supply the base for the tape and control decks. Since the keys are hidden in the concept drawing, it is unclear whether a keyboard was meant to slide out or simply unstack when needed. Based on the final prototype, the latter is likely true, though the stacking design was probably scrapped because the keys would be hard to type with if the keyboard component were flat. The combined width of the tape and control decks were symmetrical with the base. The tops of both components were white, split along their rear third by the same ridged black strip seen in the initial concept system. The grey base of each unit narrowed so its footprint was more slender at the bottom, echoing the trapezoidal form of the first design concept without sacrificing the components’ flat surfaces.

The text in the AVS drawing labels the top components as the ‘Tape storage
subsystem’ and the ‘Video game subsystem.’ These designators reflected Nintendo’s shifted focus in the American market—the videogame component was now a subsystem of a complete computational device, rather than the central unit. It was a computer that also played videogames, not the other way around.

The tape storage, common to PCs at the time, was meant to save and retrieve data as audio. Its lower edge had four conventional tape playback buttons—rewind, fast forward, stop, eject/pause—and a large ‘LOAD’ button (instead of ‘PLAY’) with a smaller red button labeled ‘SAVE’ (instead of ‘RECORD’) inset within. In the concept drawing, these buttons were on the tape subsystem’s face, but they were moved to a more conventional placement in the prototype. The tape deck was meant to be one part of a larger ecosystem of ambitious peripherals. The peculiar piano keys of the original concept design were replaced by a complete three and one half octave musical keyboard. A futuristic update of the Famicom light gun, also in grey and black, featured a pivot point that allowed the handle to fold into the gun’s body. A more conventional arcade-style joystick was also displayed, with the A and B buttons moved to the top of the stick and slender versions of Start and Select moved to the base. Barr also axed the nested controller design and planned to have them interface with the console via infrared. As they were no longer hardwired into the console, they had to be interchangeable. Both controllers, while still labeled 1 and 2, were provided Start and Select buttons, now much thinner and placed directly above the A and B buttons. The center of either controller face was now empty, save for the numeric labels and the Nintendo brand in red.

All of the major components were conspicuously free of visible ports or cords, thanks to both the infrared sensors and a clever bit of design. The tape door was opaque and swiveled flush with the edges of the unit when closed. The control deck was also perfectly flat on the top, with the faint outline of a rounded trapezoidal shape—the cartridge slot—on the left half of its ridged black strip (figure 3.3). This shape would not have accommodated the conventional grey NES cartridge, but it did mirror the cartridge slot of the Famicom, indicating that the final shape of the NES cartridge was not yet finalized at this point in the design process.
The modular prototype, with its host of peripherals, was the one shown at the 1985 WCES.¹¹ It strayed more from the integrated PC-esque form factor of the original design concept toward the industrial design of home stereo components, VCRs, or cable boxes. This was purposeful, as Barr later noted that, ‘The design was conceived as a wireless, modular system, designed to look more like a sleek stereo system rather than a electronic toy.’¹² The tape mechanism, keyboard, cartridge slot, and I/O ports were all hidden from view. Its design could integrate easily with the other components of a home entertainment system.

Still, the potential buyers at WCES were not impressed. The design was both too evocative of a PC and too complex for a toy, a worst-of-both-worlds scenario that failed to please either side. As Sheff explains, ‘No one cared about the remote control, and they hated the keyboard—a turnoff to kids, industry executive believed (parents were
irrelevant). The AVS had all the problems not only of the video-game business but of computers too. No one would touch it.”¹³

Barr simplified the design. The tape deck, keyboard, and joystick disappeared. The infrared was nixed for cost and reliability reasons, so cords returned. A number of new concept sketches appeared, each maintaining the simple palette, but adding a number of curves and rounded edges. One design looked like a record player split in half. Its curved reflective top surface resembled the lid used to keep dust and debris off of a turntable. The ridged black strip remained but was moved to the rear of the unit. Three small buttons dotted the front right of the machine.

Another control unit concept made it as far as the drafting stage. Blueprints of this control unit show yet another trapezoidal shape, this time on the upper half of the unit. Looking at its front profile, each edge sloped at a thirty degree angle then terminated to a slightly rounded base. The control deck was just over eight inches wide. Both controllers were housed lengthwise along either side of the body, so the unit was longer than it was wide. Enough space was carved into the deck to allow the controllers to nest flush inside the console with their coiled cords stowed underneath, like the ColecoVision and Intellivision. This solved two problems with the Famicom’s wired controllers: coiled cords did not take as much space, but stretched further, and the controllers tucked neatly into the deck rather than hanging off the sides. The controller designs also drifted back toward their Famicom heritage. Wires emerged from the sides, the plus pad returned, and a volume control was added. Select and Start were conspicuously absent on either controller.

The blueprint also noted that the deck's front panel controls would use either ‘blister' or ‘flat membrane' switches. These switches appeared on a number of consumer electronics during the 1980s, such as microwave ovens, television remote controls, and synthesizers—all items where constant debris buildup or heavy usage could damage conventional switches. Their low-profile design appeared more streamlined than standard mechanical switches. And since they were environmentally sealed, they appeared to be part of the ‘skin’ of the object. While perhaps more aesthetically elegant, they proved to be less reliable with longterm use.
Based on all extant documentation, blueprints, sketches, and prototypes of the Nintendo Entertainment System, it appeared as though the final console would have a ‘top-loading’ cartridge slot like the Famicom. An early promotional photograph sent to US journalists in 1985 (figure 3.4) showed the slimmer AVS style control deck, a wired AVS light gun, wired versions of the prototype controllers, the Robotic Operating Buddy, and a blank Duck Hunt cartridge that matched the dimensions of a Famicom cartridge. It was not until late in the design process that Barr would once again have to rethink the NES’s form to accommodate one last major revision.

**Zero Insertion Force**

In most cartridge-based consoles prior to the NES, videogames were loaded vertically, typically by slotting the cartridge into an exposed port that contained a card edge connector. Fairchild, Atari, Mattel, Magnavox, Coleco, and Sega had all used such loading mechanisms in their consoles. Nintendo did the same with the Famicom and looked to follow suit for the AVS. But convention had a cost—a cartridge protruding from the top of an appliance was unequivocally a videogame console and Nintendo was adamant about positioning their new video system as something more than a mere videogame. At the eleventh hour, Nintendo devised an alternative that literally hid the cartridge from view.
The revised front-loading design appeared in inventor Masayuki Yukawa’s US patent, ‘Front Loading Apparatus for a Memory Cartridge Utilized for a Data Processing Machine.’ Nintendo rightly described their cartridge mechanism as ‘novel,’ as it was unlike any design used in past consoles. The patent read:

The apparatus includes a tray for holding the cartridge and a frame to which the tray is pivotally mounted. The tray is mounted to permit rotation between a cartridge loading position and a cartridge operating position and is biased toward the loading position by a spring means. Also included is a low insertion force connector mounted to the frame for electrically interconnecting the cartridge circuit board to a second circuit board mounted in the data processing machine. Finally, a releasable locking mechanism for retaining the tray in the operating position is provided.¹⁶

Users flipped the hatch on the front of the NES, held the cartridge by its ridged grip, forced it horizontally into the console until they met resistance, then pushed it down into the hollow recess of the machine. As Barr explained, the additional space required to house the
pivoting tray necessitated a larger control deck (and ultimately larger game paks):

The biggest change was the orientation and size requirements to accommodate a new edge connector for inserting the games. The new edge connector was a “zero force” design that allowed the game to be inserted with low force, and then rotated down into the “contact” position. The case had to be designed around the movement of the game, and required the shape and size of the NES to grow from the earlier concepts. Many of the features remained, such as the two-tone color, left and right side cuts, and overall “boxy” look, but the proportions change significantly to accommodate the new edge connector.¹⁷

Nintendo’s patent asserted that their ‘zero insertion force’ (ZIF) design solved a number of problems: it protected the circuit board from ‘spurious radiation,’ it minimized ‘the abrasion of the connecting electrodes of the circuit board of the memory cartridge,’ and required less force for a small child to operate. It also kept the game paks out of sight.

However, Nintendo’s desire to solve a number of reasonable design and aesthetic problems produced a flawed device that spawned more problems than it solved. The ZIF mechanism was a clever loading solution, but the ‘spring means’ and pivoting tray introduced more moving parts, and with them greater potential for mechanical breakdown. Perhaps Nintendo underestimated the raw strength of non-Japanese children because there was certainly no lack of force for slamming carts into the NES. The delicate spring mechanisms either gave way over time or enabled too much contact between the cartridge PCB and NES motherboard. The 72-pin connector that coupled with the game pak’s exposed ROM would bend or corrode over time, leading to either the NES’s infamous blinking screens or a mess of garbled graphics.

An inherent design flaw of all edge connecting game cartridges exacerbated the issue.¹⁸ The exposed edge of the PCB was more susceptible to dirt, moisture, and all manner of unforeseen abrasions a child could invent. Nintendo aimed to preempt potential abuse by including coated vinyl dust covers to store games outside of the boxes (since the latter were often lost, destroyed, or discarded). These additions echoed the slip covers of VHS tapes, vinyl records, or Nintendo’s own Famicom disk sleeves. Though Famicom owners experienced their own errors due to dirty or corroded contacts, the top-loader design and swiveled plastic cover made this less prevalent. Tellingly, Famicom games were never sold with vinyl sleeves.

NES owners invented ad hoc solutions to ‘fix’ the machine as it gradually became less reliable: blowing into cartridges, sliding the cart in at strange angles, wiggling the cart inside the tray, or simply giving the console a few well-placed smacks like an old tube TV set. Ironically, every cart had a cautionary warning sticker affixed to its back that read, ‘DO NOT CLEAN WITH BENZENE, THINNER, ALCOHOL OR OTHER SUCH SOLVENTS.’ It was a
reasonable advisory for children, but alcohol-soaked cotton swabs was actually an effective solution for dirty contacts. Nintendo themselves patented and marketed cleaning kits as the malfunctioning games became more widespread. Even today, a vibrant aftermarket of 72-pin edge connector replacements has arisen to repair the inevitable effects of the ZIF.

In effect, Nintendo’s miscalculated engineering unwittingly became part of the folklore of the NES. Blowing along the exposed edge of the cart (which does more harm than good) has become part of the lingua franca of the Nintendo Entertainment System. But as we will see in the next section, it was not only the ZIF that caused consoles to go haywire—another NES-exclusive revision would further contribute to the millions of blinking screens.

10NES Racket

In 1986, the Famicom received a disk drive peripheral that afforded larger games, cheap re-writeable media, and the ability to save game progress without unwieldy passwords. The Family Computer Disk System (FDS) included numerous anti-piracy measures, both physical and digital, that users, hackers, and pirates summarily circumvented. As both piracy and in-cartridge hardware capabilities increased, Nintendo chose to drop support for the FDS in favor of cartridge mappers that soon equalled and surpassed the initial benefits of floppy disks.

A US patent for a disk-based peripheral indicates that Nintendo considered exporting the FDS to the United States. Presumably, the add-on would have connected to the NES via its expansion port, located on the bottom of the console beneath a snap-on plastic cover. The 48-pin port had lines for audio I/O, joystick reading, CPU interface, and other functions, so it would have been capable of interfacing with an FDS-like device, along with many other peripherals, much like the Famicom’s own expansion port. However, by the time the patent was awarded, the Family Computer Disk System was already on the decline. The US FDS never came to fruition and the expansion port remained unused for the duration of the console’s lifespan.

Nintendo’s regional piracy problems were not limited to disks. Though significantly more costly for pirates and hackers, cartridges could be dumped and copied like floppy disks. Manufacturers of bootleg cartridges thrived (and continue to thrive) in markets located outside Nintendo’s primary US-Europe-Japan triumvirate. Russia, for instance, never received an official Nintendo Famicom release. The Dendy, a Chinese-produced ‘Famiclone’ system distributed by Russian company Steepler, arose to meet market demand, sustaining a vibrant ecosystem of pirate hardware, software, television shows, magazines, and even an official mascot—the Dendy elephant—throughout the 1990s. The Dendy was an odd hybrid. While its internals were most consistent with a PAL NES system
(though not identical), its CPU clock rate was closer to the NTSC NES, while its exterior design and cartridge compatibility were consistent with the Famicom. Another strange hybrid, the Gradiente Phantom System, fulfilled similar market demands in Brazil.²³ The sleek console resembled an Atari 7800, had Sega Genesis-inspired controllers, a light gun strikingly similar to the Sega Master System Light Phaser, and played black 72-pin cartridges. Famicom and NES impostors such as these popped up across the globe.

Following the Pandora’s box of hardware and software bootlegs afforded by Famicom’s lack of copyright protection, Nintendo aimed to preempt all threats of piracy, importing, and unlicensed software with the NES design. Their solution was the Checking Integrated Circuit (CIC), or lockout chip, a custom integrated circuit borne inside every NES console and game pak. Thanks to its innovative marriage of hardware and software, Nintendo received two patents for their authentication systems in 1989.²⁵

The CIC was a 4-bit microcontroller with its own registers, instruction set, and ROM.²⁶ When players inserted game paks into the NES, each respective lockout chip executed a special handshake algorithm, known as 10NES. Both cart and console contained identical CICs, but they performed different roles based on how they were wired to one another. Among CIC’s sixteen-point pinout, there is one pin apiece devoted to data input, output, and a lock/key setting.²⁷ Console and cart chips could both send and receive data from one another when their I/O pins were wired to the other’s opposing pins, but the former chip served as the ‘lock’ while the latter functioned as the ‘key.’ The lock was wired to a capacitor whose output was used as a random seed to pick from among sixteen possible encryption streams. The lock relayed the stream selection to the key, then both had to output that stream while monitoring the input from one another to ensure a match. If a conflict occurred, the lock triggered a system reset, which also reset the CIC communication. Consequently, if a player inserted an unauthorized (i.e., keyless) cart, the reset cycle would repeat indefinitely.

With the CIC and 10NES, Nintendo implemented an early form of digital rights management (DRM), a hardware/software encryption mechanism meant to deter unauthorized access to their proprietary hardware.²⁸ Of course, under the auspices of consumer and copyright protection, they had also barred unauthorized developers from producing videogames for the NES. And thanks to the CIC’s potential for malfunction, Nintendo had engineered another means for the NES to break down.

In a study of the CIC’s influence on the establishment and evolution of DRM, O’Donnell argues that, ‘The 10NES chip in the NES...shifted user and consumer understandings of and expectations for videogames in ways that differ from music, movies, and other forms of emerging digital media technologies.’²⁹ He adds, ‘In nearly all respects, DRM was birthed by the videogame industry with very little media user and consumer resistance.’³⁰ While it is true that the CIC forcibly revamped the videogame industry’s licensing structure and
served as a model for future copyright protection mechanisms, it is questionable how much ‘consumer understandings’ were altered, especially by implicating them as passive participants in a copyright sea change. CICs were housed in cart and console interiors. Few customers had any idea that the lockout chips existed. The only artifact of their operation was in system failure. But to the end user, the cycling resets triggered by a failed CIC handshake was identical to those caused by a bent or corroded ZIF mechanism. There were no error messages or user prompts indicating whether the game, hardware, or user was at fault.

Beyond thwarting bootleggers (and legitimate users), the CIC ensured a competitive advantage by enforcing Nintendo’s restrictive licensing practices and locking software to its designated region (i.e., PAL game paks would only play on PAL consoles). As discussed earlier, Nintendo wished to avoid their industry predecessors’ fates in the US videogame crash, so they erected legal and hardware barriers to protect the integrity of their software catalog. Only sanctioned licensees could produce cartridges for the NES and said licensees were subject to harsh rules governing both the manufacture and content of their videogames (see section X below). Software developers were unaccustomed to such strict control after the Wild West free-for-all of the late 1970s and early 1980s. European developers in particular balked at the restrictions, in most cases refusing to cow to Nintendo’s demands, instead focusing their efforts on the vibrant PC market.\(^{31}\) US developers had no such luxury, as the NES quickly came to represent the videogame industry in toto. Nintendo’s early wager on the NES paid off; they filled the vacuum of the post-crash market and seized control mercilessly, reignining over vendors, retailers, and developers alike.

Those that played according to Nintendo’s rules were often handsomely rewarded. A captive consumer base of millions had obvious benefits for videogame developers. But those who opted for independence either gave up on NES development or sought means to circumvent 10NES. Atari subsidiary Tengen attracted the industry spotlight when they shifted abruptly from licensed to unlicensed developers, first defeating the lockout chip and then manufacturing their own NES cartridges outside of Nintendo’s walled garden.

In 1986, displeased by Nintendo’s licensing demands, Atari employees worked in earnest to reverse engineer the lockout chip. They first attempted to intercept and decode the communication between cart and console CICs, but that proved too difficult. They then decapped the chip, chemically peeling apart the silicon substrate to study its underlying physical structure. According to subsequent court accounts, in either effort Atari’s engineers failed to crack the code.\(^{32}\) In December 1987, Atari grudgingly agreed to Nintendo’s licensing arrangement and produced three sanctioned game paks: *Pac-Man*, *Gauntlet*, and *R.B.I. Baseball*. However, by early 1988, Atari’s lawyers resorted to social engineering to break Nintendo’s code:
Atari’s attorney applied to the Copyright Office for a reproduction of the 10NES program. The application stated that Atari was a defendant in an infringement action and needed a copy of the program for that litigation. Atari falsely alleged that it was a present defendant in a case in the Northern District of California. Atari assured the “Library of Congress that the requested copy [would] be used only in connection with the specified litigation.”

At the time, the threat of litigation was a lie, but Atari had the bravado to make it true. They sued Nintendo under antitrust law in December 1988, citing unfair monopolistic practices. Meanwhile, Atari’s engineers had successfully produced the Rabbit chip, a microprocessor that mimicked the 10NES key. Nintendo countersued in 1989, claiming breach of contract and patent and trademark infringement, petitioning the district court of appeals to halt further production of Tengen cartridges. Nintendo prevailed. The courts ruled that Nintendo designed an ‘original program’ and that 10NES contained ‘protectable expression’ subject to copyright law.

Other unlicensed cartridge developers devised inventive means to outwit 10NES without resorting to Atari’s subterfuge. Dan Lawton, a programmer for Color Dreams, managed to stun the CIC temporarily with a −5V surge right as the NES booted. Consequently, Color Dreams carts, housed in distinctive power blue cases, included a DC voltage pump mounted on the PCB to permit their games to run unhindered. American Game Cartridges (AGC), American Video Entertainment (AVE), Camerica, Bit Corporation, Active Enterprises, and Bunch Games (a low-price subsidiary of Color Dreams) employed similar mechanisms in their cartridges.

Australian company Home Entertainment Suppliers (HES) ported many of the aforementioned companies’ unlicensed titles for play on the Australian NES. Contrary to the common usage of the word, HES ‘ports’ did not involve software conversion from one hardware architecture to another. Instead, HES transplanted game ROMs into new cartridge enclosures, swapping the CIC stun for a bizarre ‘piggyback’ case design. In lieu of shocking the lockout into submission, players could attach a licensed NES game pak to the piggyback cart, using the former as a pass-through device that could deliver the necessary encrypted handshake. The kludgy workaround actually solved a new problem facing unlicensed developers. Once Nintendo got wind of the voltage spike technique, they released NES revisions that defeated the CIC stun, once again rendering unlicensed games unplayable. The piggyback cart’s Trojan horse solution obviated the need to defeat the CIC at all.

The debacles surrounding the ZIF and the CIC eventually led Nintendo to discard both when the Nintendo Entertainment System model NES-001 was revamped as the NES-101 in 1993 (figure 3.6).
3.6 The revised NES-101, known informally as the ‘top loader’ or ‘NES 2.’ The new design took cues from the NES’s successor Super Nintendo and fixed most of the mechanical flaws of the toaster design. (Source: Evan-Amos, Wikimedia Commons)

Nintendo shrunk the console considerably—thanks to a ‘new’ top-loading card edge connector—discarded the CIC (along with region lock), dropped the RCA composite out, and excised the expansion port. As Barr recounted:

The redesigned NES did not use the “zero force” connector, but instead relied on a direct insert connector. Form following function, the new connector placed the game 90 degrees to the main PCB and eliminated much of the bulk needed for the old electronics and connector. The redesign was made several years after the original, which was designed in 1984. The boxy look was out and I thought it was time for a more sleek and inviting look.

Interestingly, the NES 2 also brought the NES and Famicom into design parity. The new Family Computer (often called the AV Famicom) was identical to the NES-101, save for its flattened top (to accommodate the FDS RAM Adapter Cartridge), 60-pin edge connector, 15-pin expansion port, and higher-quality composite A/V output.

The NES/Famicom redesign was a compact, reliable machine, but it came too late in the 8-bit console’s lifespan. By 1993, the successor Super Famicom/Nintendo was already three years old in Japan and two years old in the United States.
Operating Buddy

The Nintendo Entertainment System Deluxe Set, the first US retail bundle, was the culmination of many months’ work of design refinement and marketing strategy. The sizable box included the NES control deck, RF switch, AC adapter, a host of instruction manuals and warranty cards, two controllers, the Zapper light gun, and the Robotic Operating Buddy (or R.O.B.). Two bundled games, Gyromite and Duck Hunt, served to showcase the Deluxe Set’s peripherals, but R.O.B. was the clear centerpiece.

The robotic toy was the gimmicky bait that finally got the retail industry to bite. A new videogame console was a non-starter post-1983, but a futuristic robotic playing buddy could distract consumers from the NES’s core purpose. When Nintendo introduced R.O.B. to the NES family, the toy industry relented, finally willing to take a chance on a risky Japanese product. Nintendo marketing worked the robot angle—their early promotional materials made R.O.B. the center of attention. NOA’s first in-store point-of-purchase displays were oversized R.O.B. replicas, whose illuminated torsos bore the full lineup of NES software. The first televised NES commercial featured R.O.B. emerging from an egg, marking ‘the birth of the incredible Nintendo Entertainment System.’ The picture of R.O.B. in a 1985 Macy’s ad for the NES dwarfed both the system and its games. The headline copy, ‘VIDEO ROBOTS,’ was set larger than any other text on the page, even the name of the console (figure 3.7).
At Nintendo of America’s New York launch party, ‘silver-plated R.O.B.s were strewn around the room as showpieces,’ with another oversized robot serving as the sculptural centerpiece.³⁹

For the Deluxe Set’s package design, the robot’s purple backlit head, glowing red LED, and vacant lensed eyes occupied the majority of the box. The requisite awkward tableau of a family crowded around the CRT, transfixed by *Duck Hunt*, saw R.O.B. staring idly back at the viewer. The text on the back promised ‘The first truly interactive home video game system,’ thanks to both the light gun and the futuristic robotic buddy:

No other video system features R.O.B. - the wireless video robot who plays games right along with you. Or the Zapper, the amazing light-sensing gun that puts sharp-shooting accuracy right in the palm of your hand. This extraordinary pair of video partners interacts with you and the screen, allowing you “hands on” video action.

How these peripherals trumped prior consoles’ ‘interactivity’ is unclear—nearly every console featured its host of unique and bizarre accessories—but Nintendo was correct about R.O.B.’s peculiar position in gameplay. The robot watched the television screen like a
human player. In fact, R.O.B. fulfilled the promise of the AVS’s proposed wireless peripherals in a unique, if roundabout manner. The robot was wireless, but not due to embedded infrared or any other proprietary wireless protocol. Instead, R.O.B. received its commands from the television via a sequence of carefully timed flashes that the robot detected with photosensitive lenses (i.e., its eyes). However, its only means to relay commands back to the console was through the same interface as the player—the controller. So R.O.B. truly did play alongside the user, albeit in a strange circuit connecting human, console, software, and television.

R.O.B.’s body was comprised of a plastic hexagonal base with a central post supporting its torso (with two attached arms) and head (figure 3.8). The robot’s movements were coordinated by a series of battery-powered motors and gears. Six notches along the ‘spine’ allowed R.O.B.’s torso to move up and down according to the onscreen commands. The central post could also rotate to align R.O.B. with five of the base’s six sides (R.O.B.’s could not twist its torso completely backwards). The same five sides were notched to accommodate accessories included with the game software. Attached to its torso were two articulating arms that could open or close. Supplemental hands were attached to the arm posts to allow R.O.B. to grip various accessories, such as gyroscopes or stacking blocks. No matter how the robot twisted, its head remained fixed, since it required a solid line of sight with the television. The user could also angle R.O.B.’s head up or down if the robot was not on the same level surface as the television. To ensure proper signal reception, supported games provided a calibration step to test R.O.B.’s vision. If the television picture was too bright, R.O.B. could malfunction, so a special filter was included to apply over its photosensors.
Gyromite was one of only two cartridges released for the NES that used the robot. Since the NES was sold in various hardware/software configurations after its initial launch, a standalone version of Gyromite came in an oversized box that included R.O.B.'s accessories. A special armature used to hold Gyromite's namesake gyroscopes clipped into the two accessory notches on R.O.B.'s right-hand side. The front and front-left notches supported another armature meant to hold an NES joypad plugged into the second controller port. Along this armature were two mechanical levers that terminated in small circular 'platforms,' one red and one blue. When weight (i.e., a gyroscope) was placed on the platform, the lever would mechanically depress either the A or B button on the second controller. The rear-left accessory notch held a motorized spinner. Using the included pincer hands, R.O.B. could grasp the gyroscopes and place them inside the spinner, where they would rotate rapidly, independent of their fixed post. R.O.B. could then drop the spinning gyroscopes on the circular platforms. There they could spin in place and keep the
buttons held down for a short time, until they spun down and toppled. If the player was agile enough, they could instruct R.O.B. to drop both gyroscopes in turn, allowing the A and B buttons to activate simultaneously.

*Gyromite’s* gameplay involves guiding a balding, lab coat-clad character named Professor Hector (Player 2 got Professor Vector) over a series of multi-tiered platforms blocked by blue and red cylindrical columns and a number of roaming birdlike foes called Smicks. The player has limited interaction with Hector—he can move horizontally across platforms, move vertically along ropes, and pick up turnips to drop and distract his adversaries. If Hector collects the bundles of dynamite scattered throughout the stage before the allotted time expires, the round completes and the next stage begins. However, the red and blue columns serve as gates blocking Hector’s progress. The only way to raise and lower gates is to instruct Hector to issue commands to R.O.B., his robot companion. In most NES games, pressing Start pauses the game. In *Gyromite*, Start triggers a ready state, indicated visually by the screen’s background changing from black to blue (what the manual calls ‘Robot transmission mode’). In the game’s code, this is done by performing a palette swap. The shared background color is changed from black (NES palette entry $0F$) to blue ($01$), a simple but effective means to indicate that the game’s state has changed. Hector then faces the player and cups his hands as if holding a joystick, a visual cue indicating that he is looking ‘outside’ the frame of the television screen. At this point, the physical robot is now drawn into the diegetic space of the videogame as an active participant. The player/Hector can then issue commands: Pressing up or down on the D-pad ratchets R.O.B.’s torso up or down two notches, left or right rotates the central post, and B or A respectively closes and opens R.O.B.’s hands. Using these simple commands in various combinations allows R.O.B. to interact with its accessories and, ultimately, the second controller. In order to lower a blue column, for instance, the player would have to enter a string of button presses to make R.O.B. do the following:

A) rotate to the proper position above a gyroscope
B) lower the hands into position near the gyroscope post
C) grasp the gyroscope
D) raise and rotate to the proper position above the motorized spinner
E) place the gyroscope into the spinner and allow it to fully oscillate
F) lift and rotate the gyroscope into position above the blue circular platform
G) lower the gyroscope onto the platform
H) release the gyroscope

This entire robotic retinue, still somewhat condensed, performed a single mechanical function: pressing the A button on the second controller.
More interesting than R.O.B.’s measured mechanical performance is the means by which R.O.B. receives its instructions. Whenever the player/Hector issues a command during ‘transmission mode,’ the screen will noticeably flicker between black and green. The flash is not meant for human eyes. R.O.B.’s optical lenses are synced to the refresh rhythm of the cathode ray television, so it perceives a sequence of optically-encoded ‘bits’ meant to trigger the appropriate movement.

Commands are encoded in two sequential ‘data packets.’ The first is a ‘ready’ signal, indicating that R.O.B. should anticipate further instructions. In Gyromite, the on and off bits of a message are represented by the color of the screen: off is black ($0F$) and on is green ($2A$). For each bit of the issued command, the screen is flooded with one color per frame. Using an emulator, one can slow down the flashes in order to read the sequence of colors that correspond to each command. (Stepping through the source frame by frame is the best method, since it’s possible to count multiple contiguous frames of the same color.) The ready packet is three frames of black, one frame of green, then a final black frame. In binary, substituting 0’s for black and 1’s for green, we can represent this message as $\%00010$. After the final black frame, an 8-bit ‘instruction’ packet is sent, indicating R.O.B.’s requested movement. Though there are 256 instruction possibilities in an 8-bit value, R.O.B. is only capable of performing a handful of commands, consistent with his range of mechanical motions. For example, the color sequence green, black, green, green, green, green, green, black ($\%10111110$) instructs R.O.B. to close his hands; $\%11101110$ opens them.

Gyromite’s programmers employed two simple but efficient techniques to display R.O.B.’s optical commands onscreen. First, painting the entire screen green or black uses the same type of palette swap that toggles between blue and black backgrounds in transmission mode. However, rather than swapping only the shared background color, all four colors of all eight palettes are filled with green and black on successive frames. In other words, all of the sprites and background tiles are still present on screen during the color flashes, but their constituent pixels are filled with a single color, effectively making them ‘disappear.’ This turns out to save a lot of processing time. An alternative strategy would be to clear the sprites from the screen and fill the frame with a single blank tile of the necessary color. However, this method would not be possible in a single frame—the programmers would have to disable PPU rendering, swap 960 name table tiles, then reenable the PPU. Additionally, the current screen’s sprites and locations would have to be stored prior to the flicker, then retrieved and restored once the sequence was complete. Either operation would be a massive waste of memory resources and, due to the increased rendering time, would make the flicker that much more noticeable to the player.

Second, the proper timing for the screen flashes is determined by using a hardware-level status flag unique to the PPU. R.O.B. is designed to respond to frame-accurate commands, synced to the television’s refresh rate. In other words, if the flashes are
lengthened to two or more frames, R.O.B. will not respond. In order to properly fire the palette updates that will flash the screen between black and green, the programmers had to sync them to individual frames. They did so by polling a unique status bit, illustrated in the following two short loops from *Gyromite’s* source:

```
$80C1: LDA $2002
$80C4: AND #$40
$80C6: BNE $80C1
$80C8: LDA $2002
$80CB: AND #$40
$80CD: BEQ $80C8
```

The first line loads the accumulator with the 8-bit value stored at address $2002, commonly called the PPUSTATUS register. PPUSTATUS contains bits of information about the PPU’s current state, including whether VBLANK is taking place or if the eight-sprite-per-scanline limit has overflowed. Once its current value is loaded, a logical AND is performed against $40, or $01000000 in binary. The AND operator is used to isolate, or ‘mask’ a particular bit in a given byte—in this case, the sixth (counting right to left, starting at zero). Bit six of the PPUSTATUS register indicates whether sprite 0 has been hit. Sprite 0 is a special, high-priority tile, named so because it occupies the first—or zero—position in sprite OAM. When an opaque pixel of a background tile overlaps an opaque pixel of sprite 0, the PPUSTATUS flag is raised. In fact, it is the only built-in collision detection that the PPU offers.

The sprite 0 hit is conventionally used for two purposes. The first, and more prevalent, is to time scanline-accurate raster effects (see chapter 4). In *Gyromite’s* main play mode, the screen is divided into two sections: the upper bar, which contains the player’s score and time limit, does not scroll; the lower play area, however, scrolls horizontally, most noticeably at the beginning of a round when the stage’s layout is revealed to the player. Notice that the upper status bar remains stationary while the play area scrolls independently underneath. The NES has no hardware capability to split scrolling points at an arbitrary scanline—such an effect must be properly timed and executed by the programmer. Sprite 0 marks the boundary between two sections and signals the CPU that the boundary point has been ‘hit.’ Once that boundary point is known, scrolling values can be manipulated mid-screen.

Disabling *Gyromite’s* background tiles in an emulator during gameplay reveals a lone pixel hovering near the top of the screen (figure 3.9). That graphical point is the opaque portion of sprite 0 used to trip the PPUSTATUS flag. As soon as the PPU encounters that pixel and notices that it overlaps an opaque pixel of the background tile beneath it, a sprite
0 hit is fired. During normal play, the pixel is hidden, since it is tucked behind background tiles.

Sprite 0’s secondary purpose, monitoring VBLANK status, is less common, but Gyromite uses it to time the screen flash. Returning to our code segment, we can see that the branch after the first AND loops recursively until sprite 0 is clear (i.e., bit six equals 0). Once cleared, another identical loop begins to iterate, but this time waits until sprite 0 is set (i.e., bit six equals 1) before it leaves the loop. Though less common than marking scrolling splits, sprite 0 could also be used to monitor the CPU’s VBLANK status. This use appears odd, since the PPU provides a separate flag for the current VLBANK status within $2002$. Bit

3.9 Gyromite with background tiles disabled. Sprite 0 is the single pixel visible near the top of the screen (Emulator: FCEUX 2.1.5)
seven in the PPUSTATUS register indicates whether the CPU is currently in vertical blank (1) or not (0), so this flag could presumably be used to check for the proper timing to flash the screen. The catch is that whenever the VBLANK status bit is read, it is subsequently cleared, regardless of whether VBLANK is still in progress. In other words, it can not reliably be used to signal the exact timing of VBLANK, merely whether that condition is 
\textit{taking place}. It is a one-time status check that is ‘exhausted’ once per frame. When sprite 0 is triggered, however, its status is maintained until the end of VBLANK. Once the pre-render phase of the next frame is reached, the sprite 0 status bit is reset until it triggers again. Checking the following two conditions in successive loops can reliably time the end of VBLANK:

1. Sprite 0 status bit is set
2. Sprite 0 status bit is cleared

Notice that \textit{Gyromite} checks these conditions \textit{in reverse}. This is because R.O.B.’s design does not require that the flashes take place precisely after VBLANK occurs, merely that they are timed to happen \textit{once per frame}. By cycling through the double loop sequence, the programmers ensured that sprite 0 has been both cleared (at the end of VBLANK) and then set again before the flash occurs. As long as sprite 0 remains on the screen and overlaps a background tile, the black/green message sequences will fire as intended. For the purposes of timing the color cycles, it does not matter \textit{where} sprite 0 sits, merely that it reliably signals successive frames. By setting a breakpoint at $80CD$ while stepping through the source, it is possible to ‘catch’ the transition between black and green flashes prior to the sprite 0 hit (figure 3.10). \textit{Gyromite’s} programmers effectively employed sprite 0 to both regularize communication with the robot \textit{and} split the scrolling and non-scrolling segments of the screen.
R.O.B. was an innovative feat of optical, electrical, and mechanical engineering, but its ponderous motorized movements limited the range of available gameplay. R.O.B. was too slow to control any videogame that might require fast reflexes or quick timing. It is telling that one mode of play in *Gyromite* involved raising and lowering gates while an automated Professor Hector literally *sleepwalked* through the stage. Players discovered that it was less time consuming to circumvent R.O.B.’s involvement and simply take control of the second controller themselves (or have a friend play R.O.B.’s part). The robot was an impressive showpiece that served Nintendo’s purposes, but it wasn’t particularly fun. As Tilden recalled, ‘That thing was definitely like watching grass grow...It was so slow, and to try and stand there and sales-pitch it in person and try to make it exciting; you had to have the eyes
lined up just right or it wouldn't receive the flashes. It was kind of a challenge.  

Nonetheless, the robot helped divert attention, at least temporarily, from the NES’s gaming core and focus consumers on a futuristic gadget. Once players discovered that the NES was an enjoyable console based on the merit of its games alone, R.O.B. was quickly forgotten. After its initial two games, the operating buddy was no longer allowed to play.

‘AFTER THE EYES FLASH…’

The Deluxe Set’s light gun peripheral fared better than R.O.B. throughout the NES’s lifespan. The Zapper was one of the few hardware add-ons that emerged relatively unscathed from the console’s metamorphosis from AVS to NES, but its legacy reached further into Nintendo’s past than the Famicom, Game & Watch, or even videogames in general. Nintendo (and specifically Yokoi) had experimented with light gun technology since the 1970s. Photo-electric shooting ranges, hosted in converted bowling alleys, were popular in Japan prior to the Space Invaders-fueled arcade explosion. Nintendo found success with their ‘Laser Clay Ranges,’ along with a number of gun-based toys for home play. The Famicom received its own light gun peripheral, fashioned after a classic Western six shooter, that attached to the console’s 15-pin expansion port. Since the light gun had software support in Japan, it was a logical choice to ‘port’ to the NES, though not without a few important translations.

Lance Barr’s prototype maintained its futuristic form, color palette, and ridged body in the revised Zapper, but subtracted the fold-up form factor and infrared capabilities in favor of a more economical and ergonomic design. However, the Zapper’s final sci-fi style was not simply meant to conform to the AVS’s future-forward design motif. Nintendo was rightfully concerned that Western parents would find a realistic revolver unacceptable for children to play with. A laser zapper would raise less ire than a six shooter. The NES Zapper fell in lockstep with a host of other gun-shaped toys from the mid-1980s, such as Lazer Tag, Photon, or the sundry Star Wars accessories stocking department store shelves. Nonetheless, Nintendo eventually had to make a further alteration to the Zapper in compliance with a US federal regulation requiring ‘toy, look-alike, and imitation firearms’ to either be constructed from translucent materials or have their exterior surface painted from a selection of bright pastel colors. Nintendo chose the latter. The revised Zapper had a blaze orange barrel and handle.

The Zapper was not as mechanically sophisticated as the Robotic Operating Buddy—its sole moving part was a red, spring-loaded trigger that sounded a resonant metallic clang when squeezed—but the photodiode housed in its barrel functioned similarly to the robot’s eye lenses. The light gun did not rely on a sequence of coded flashes, but it did require the same frame-accurate light/dark discrepancies that drove the robot’s choreography. And
unlike R.O.B. (or the Famicom Beam Gun), the Zapper connected to the NES’s standard joystick ports. In fact, two Zapper status bits are located in the same joypad registers ($4016$ and $4017$) used to poll for controller input. Bit 4 of $4016/7$ indicates whether the Zapper trigger is pulled (1) or released (0), while bit 3 records whether light is detected (0) or not (1).\textsuperscript{47}

Among the four light gun-compatible titles available at the NES’s launch,\textsuperscript{46} \textit{Wild Gunman} (1985) best illustrates both the hardware mechanics underlying the light gun’s function and Nintendo’s continued eagerness to repurpose early game concepts for new platforms. \textit{Wild Gunman} demonstrates two different modes of interaction with the Zapper based upon the selected game type. In Game A, a large outlaw moseys into frame from the left or right side of the screen. The backdrop is a nondescript southwestern landscape punctuated by a fiery red mesa beyond the horizon, two cacti, tufts of desert foliage, and a few strewn rocks and fossils. Once the outlaw reaches center screen, he pauses, yells ‘FIRE!!’, and his eyes flash like jewels (figure 3.11). A timer begins to count up and the player is meant to fire before the designated time limit is reached. The gameplay scenario mimics a Western duel, though with less fatal results. Pull too early and you trigger a ‘FOUL!’; pull too late and the screen blinks in shades of red, presumably indicating a survivable wound, since you are permitted to draw again. Precision gunplay is not necessary. Game A simply requires that the player pull the trigger while the Zapper barrel points at \textit{any} portion of the television screen. Shoot the cactus or the sky if you like—you will still register a successful hit.
The game’s name, Wild West dueling scenario, and flashing eyes were all callbacks to Nintendo’s electro-mechanical *Wild Gunman* light gun game from 1974. The Yokoi-designed game was a smart hybrid of toy hardware, photosensors, and film projection. As the International Arcade Museum describes, ‘The game has an image projection system that uses a 16mm film showing an outlaw gunslinger appearing in an alley and the player has to draw his or her gun and shoot that outlaw before the outlaw draws his gun and shoots.’

When the game begins, a pre-recorded narrator announces the instructions, repeated in truncated form onscreen: ‘AFTER THE EYES FLASH...SHOOT!’ The player must also wear a belt to holster the six shooter that is wired to the game’s cabinet. Inside the cabinet there are two projectors used to branch between film reels dependent upon whether the player
draws and fires in time or not. Like its NES successor, arcade Gunman relies solely on speed, not accuracy. Remarkably, the NES Wild Gunman is a port from an electro-mechanical to a digital platform.

Keen-eyed gunslingers will notice that, like Gyromite’s transmission mode, the screen flashes briefly when the player pulls the Zapper trigger. This flash contains the visual information necessary to communicate with the Zapper. Inside the gun barrel is a small photodiode, a light-sensitive transistor that converts light into electrical signals. Via the controller port, the converted signal feeds into the bit 3 flag described above. However, there is a sly bit of built-in signal translation that takes place to prevent false positives from spurious data—i.e., player cheating. In effect, the Zapper will not signal a hit unless its photodiode detects light from the television. Pointing the gun at a lightbulb, fluorescent overhead, or any other ambient light source not emanating from the cathode ray tube will result in a lost gunfight.

The implementation of Nintendo’s light gun error filter was novel enough to earn a US patent, awarded to engineer Satoru Okada. It turns out that successful onscreen targeting requires a concerted coordination between the Zapper’s sensor and the NES PPU:

When a pistol is leveled at a target on the screen of the television monitor and the trigger is pulled, a switch is closed to produce a trigger signal, on the basis of which black picture data is read from the program ROM and displayed on the television monitor. Immediately after the black picture was displayed, the white picture is displayed in the next frame in the position where the target was displayed just before and/or around the periphery thereof. The light from this white picture is detected by a phototransistor, whose detection signal, when extracted by a filter, is used as a detection signal from the target. Thus, there is no danger of mistaking the light from an illuminator for the light from the target.⁵⁰

Description of the filter is found further into the patent document:

The phototransistor serves to detect the light from a target on the television screen to convert it to an electric signal, the output from said phototransistor being amplified by the transistor, and only the detection signal which corresponds to the horizontal oscillation frequency in the television synchronizing signal is extracted by the filter formed of the capacitor and coil, the other frequency components being attenuated.⁵¹

In order for Game A to function properly, a number of checks must take place. First, the programmers poll the trigger bit flag to detect when the player has fired. Once detected, the screen is painted black for a single frame. On the next frame, a ‘white picture’ is displayed on the screen anywhere a valid target is located. In Game A, the target is the whole screen,
so the entire background is painted white. In the same frame, the light sensor bit is checked 
to ensure that the Zapper has detected a target. To ensure that the target is located 
onscreen (and not another light source), the photodiode’s capacitor/coil filter attenuates 
portions of the signal that do not correspond with the television’s HBLANK frequency. If the 
light source is the proper frequency, the bit flag is raised. The game logic then interprets a 
hit and proceeds accordingly.

This whole process is simplified by the fact that there is only one target: the screen. 
How does the targeting routine differ when there are multiple targets on screen? Instead of 
painting the entire screen white, each target zone is isolated and checked in turn. *Wild
Gunman* has an alternate gameplay mode that illustrates this process.

In Game C (‘Gang’), multiple outlaws emerge from the facade of a Western saloon. 
There are two windows on the upper floor; two below, and a door in the center. Game C is 
classic shooting gallery meets whack-a-mole, where the object is to fire at the enemies as 
they emerge from their respective portals, without expending too many bullets, and before 
the gang members have a chance to ‘fire back.’ Unlike Game A, both timing *and* accuracy 
count. Firing blindly at the screen does not ensure a hit (and wastes bullets). Instead, the 
player must aim at the intended entryway to register a successful kill. Behind the scenes, 
this requires a lot more work than the simple binary condition of Game A. To register hits 
for each potential enemy entry point, the screen is split into five rectangular blocks that 
correspond roughly to the section of the saloon where they will appear. I call these blocks 
the ‘target quintiles.’ The upper portion of the screen is split in half—one quintile for each 
window. The lower portion of the screen is divided into three conjoined quintiles 
corresponding to the left window, swinging doorway, and the right window. Unlike 
*Gyromite* or Game A, which use a palette swap to paint all sprite and background tiles a 
uniform black, green, or white, Game C must maintain the target quintiles separate from the 
saloon playfield seen onscreen. This is achieved by painting them to the PPU’s second name 
table.

Recall from chapter 1 that the NES is capable of *addressing* four name tables, but only 
has adequate VRAM to *store* two. In practice, this means that there are two name tables 
filled with a game’s necessary playfield tiles while the other two name tables *mirror* the 
first two. In early NES carts, a solder point on the PCB physically hardwired the mirroring 
for the game; in later carts, mirroring was mapper-controlled. The mirroring setting 
determines the VRAM location that each name table points to, thus organizing the 
orientation of the two name table mirrors. Vertical mirroring, for instance, tells the PPU 
that name table #2 will mirror the contents of name table #0, located at $2000; 
consequently, name table #3 will mirror the contents of name table #1, located at $2400. 
The key point is that mirrors act as *pointers* rather than *copies*. No data is transferred to fill 
additional locations. This is similar to a shortcut folder in Windows or OS X. One might
create a shortcut to their MP3 collection on their desktop, but this does not transfer the music collection along with the folder. Instead, that shortcut points to the location of the actual files, indicated by a small arrow in the corner of the folder.

Though name tables are located sequentially in the VRAM memory map, it is common to visualize name tables as a 2x2 matrix:

\[
\begin{bmatrix}
0 & 1 \\
2 & 3
\end{bmatrix}
\]

Emulators that include a name table viewer commonly arrange their name tables in this manner, as it provides a simple visual representation of how the name tables relate to one another. Vertical mirroring would look like the following:

\[
\begin{bmatrix}
A & B \\
A & B
\end{bmatrix}
\]

Name tables #0 and #2 are ‘stacked’ atop one another; likewise for #1 and #3.

Mirroring’s usefulness comes into play during scrolling. The combined contents of A and B are obviously larger than a single screen can display. We can think of the visible playfield as a screen-sized camera that ‘pans’ over the contents of two contiguous name tables. Feeding updates to the NES’s scroll register ($2500, or PPUSCROLL) pans the camera along the name tables according to a programmer-determined x- and y-offset. The perceived effect is a smooth scrolling motion. A general rule of thumb is that the mirroring setting is useful to produce the inverse scrolling direction, e.g., vertical mirroring is selected for games with horizontal scrolling and vice-versa. The 2x2 name table matrix arrangement should make this clear—panning the camera horizontally across A and B produces a seamless name table to name table transition, provided the background tiles are drawn accordingly. Panning vertically from A to A would produce a wrapping effect, where the top of the playfield would scroll off the top of the screen and emerge again from the bottom.

An obvious question is why scrolling matters at all in relation to *Wild Gunman*, since the game displays only static playfields. It turns out that *Gunman* employs scrolling in a rather unconventional manner; one nearly invisible to the player. The only visual hint is the quick screen flash. In figure 3.12 below, we can see three separate representations of *Wild Gunman* during gameplay: the current playfield (upper left), the pattern table contents (lower left), and the name table contents (right).
3.12 Game C of Wild Gunman in its fail state (i.e., player has been shot). FCUEX’s Name Table Viewer (right) displays the 2x2 matrix of name tables, mirrored vertically. (Emulator: FCEUX 2.1.5)

Though vertical mirroring is clearly indicated by the Name Table Viewer’s highlighted radio button, the setting should be obvious from the name tables’ contents. It should also be clear that name tables #1 and #3 contain the quintile targets, colored according to the current background palette entries (seen along the upper row of the PPU Viewer’s ‘Palettes’ table). The colors might seem odd, but the NES’s limited number of palettes means that the quintiles have to share with the game’s other background tiles. Of course when the quintiles are ‘backstage,’ their palettes do not matter.

We saw above how the Zapper requires a single frame of black before its target checking sequence begins. When a gang member emerges from the saloon door or windows and the player pulls the trigger, Gunman waits until the next VBLANK period to
execute a number of quick subroutines. One such routine, located at $C3AC, performs the following (excerpted) instructions:

```
$C3AD: LDA $0A
$C3AF: STA $2005
$C3B2: LDA $0B
$C3B4: STA $2005
```

Two variables from RAM, $0A and $0B, are loaded into the accumulator then stored, one after the other, in the PPUSCROLL register ($2500). If we set a watch on $0A during gameplay, we notice that it toggles between two values, #$00 (0) and #$FF (255), when a shot is fired ($0B remains #$00 throughout the game). In order to update the camera’s position (i.e., the visible playfield), two values must be fed to PPUSCROLL in sequence: an x-offset and a y-offset. These x- and y-coordinates describe the upper left corner of the camera’s viewfinder. RAM address $0A stores the scroll’s x-offset, so updating it to 255 during VBLANK ‘jump scrolls’ the camera to name table #1 in a single frame. In other words, the camera slides along the x-axis until its upper left coordinates align with the upper left region of the name table displaying the target quintiles. And since the scroll occurs in about 1/60 of a second, it appears instantaneous to the player.

Once the target quintiles are scrolled into place, the code runs a check of possible targets based upon where the player fired. Stepping through this sequence frame-by-frame in an emulator allows us to see the screen first cycle to black, then flash each quintile in turn. The targets are checked in the following sequence: (0) upper left, (1) upper right, (2) lower left, (3) lower right, (4) lower center. To save time, only the quintiles leading up to and including the intended target are cycled. In other words, if the player fires at the upper right saloon window, only target 0 and 1 are checked. If they fire at the lower right window, targets 0, 1, 2, and 3 are checked. Firing at the center door is the most time-intensive, since the game must check the entire target sequence. Each target quintile is checked by illuminating its region in white and leaving the others black (figure 3.13).
3.13 Wild Gunman paused during its 'target quintile' check. Name table #1 is visible onscreen. The player has fired at the saloon door, since that is the only target that triggers the corresponding 'saloon door quintile' to illuminate. Note its use of color entry #2 in palette #0, which also affects the saloon facade in name tables #0 and #2. (Emulator: FCEUX 2.1.5)

Again, to save time, simple palette cycling is used to paint the quintiles white. However, since each quintile is assigned a separate palette, there is a slight catch: one quintile must share a palette with one of the other four. In this case, quintiles 1 (upper right), 2 (lower left), and 4 (lower right) have dedicated background palettes (1, 2, and 3, respectively). Quintiles 0 (upper left) and 3 (lower center) share palette 0, but they receive separate color slots within that palette. Thus, when the target sequence check proceeds, all color slots in all four background palettes are switched to black except for the slot corresponding to the current target quintile. In figure 3.13 above, the quintile name table is offscreen, painted conveniently in white, black, salmon, and umber. If you check each quintile's color against the palettes shown in the PPU Viewer, it is clear how their colors are assigned. For instance,
though quintile 0 and 2 use different palettes (also 0 and 2), entry 1 in both palettes is salmon, so the entire left portion of the name table appears as a solid color block.

Several other light gun games use Wild Gunman’s ‘jump scroll’ technique. Both Duck Hunt and Hogan’s Alley, for instance, swap quickly to their spare name table to run target checks. However, Wild Gunman is one of the few Zapper games that uses background tiles for targets. In Duck Hunt, when the player fires, flying duck tiles are replaced with solid white sprites. Hogan’s Alley uses the same technique to replace its spinning shooting range targets with white sprites. However, Hogan’s sizable target characters are built with a combination of background and sprite tiles in order to avoid flicker. The smiling cop, for example, is built from background tiles, while the grimacing gangster from the box cover is built from sprites. Other characters, like the woman in pink, appear in both pattern tables, allowing her to appear as either a background or sprite metatile (though her white irises are always a sprite).

Gumshoe is a notable exception to the name table swap technique. Rather than a simple shooting gallery or targeting game, Gumshoe is a fixed-scroll platformer. And since the game is designed for the Zapper, the player does not have direct control over their avatar. Instead, in an odd reversal of videogame violence, the player must shoot the self-propelled character to make him jump, allowing him to vault floating terrain, avoid flying debris, and collect balloons. Certain dangerous missiles, like flying cans, can be shot directly. Since the game scrolls horizontally, Gumshoe has no ‘reserve’ name table to swap out during target check. Instead, when the current name table’s palette is cycled to black, white sprites replace the tiles of any valid targets onscreen, as in Duck Hunt and Hogan’s Alley.

Like many of Nintendo’s early light gun titles, Wild Gunman’s simplistic gameplay belies both the sophistication of its programming techniques and the split perceptual fields necessary for both player and Zapper to interact with the NES. From the PPU’s perspective, Wild Gunman is a scrolling game, however unconventional it may be. Instead of representing a seamless virtual world that extends beyond the screen’s borders, Gunman’s programmers used scrolling to quickly swap between the colorful saloon or desert playfields that the player perceives and the abstracted optical playfield that the light gun perceives. In all Zapper games, from launch title Duck Hunt to Color Dreams’ unlicensed Baby Boomer, the player and the light gun are targeting related but independent planes of visual information. And the NES’s PPU mediates, with frame-level accuracy, this curious perceptual divide.

The Zapper’s reliance on the specific oscillation of the CRT’s sweeping horizontal blank precludes its use on modern televisions. NESDev community member tepples, who has released the extensive Zapper test/demonstration program Zap Ruder, describes why this is so:
The Zapper and similar light guns have the photodiode connected to a 15.7 kHz resonator to detect whether light is actually coming from adjacent scanlines of a CRT's raster scan and not a light bulb. The trouble is that LCD TVs don't have that 15.7 kHz flicker. The only sort of Zapper game that works on an NES connected to a modern TV is one that uses the trigger ($4017.D4$ active high) and not the photodiode ($4017.D3$ active low).

So again a chasm of perception divides human sensors (eyes) and electro-mechanical sensors (photodiode). Besides the sharper, higher-resolution display of a modern LCD, we still see television the same way—whether with liquid crystal or electron gun, our eyes are consistently and sufficiently fooled to perceive believable moving images. But the Zapper's photodiode, trained to the rhythms of the CRT, is rendered blind.

**Black Box**

Early NES games had a uniform packaging style. Keeping in line with the outer space backdrop of the console's packaging, the first seventeen NES launch titles (and a subsequent thirteen) arrived in boxes printed front-to-back in black, punctuated with small glimmering stars. The upper half of the box front displayed a sample of the game's graphics, zoomed to a scale that showcased their flat pixellated topographies. The game's title bisected the box at a diagonal, set in an all-caps sans serif typeface. Below the title was Nintendo's red oval logo and 'Entertainment System' in muted yellow text. The bottom had two additional logos: the game's series type on the left and the Nintendo 'seal of quality' on the right. Nowhere on the box could one find the word videogame.

Nintendo was purposefully foregrounding the NES's graphics capabilities in direct response to the marketing tactics of earlier competitors. Atari's beautifully-painted box covers, for instance, were speculative leaps of fancy. VCS title *Air-Sea Battle (1977)* had a strikingly realistic montage of torpedo-launching submarines, battleships under fire from bombers and fighter jets, infantry manning heavy artillery, and a single sharpshooter in their midst. The graphical reality was far less dramatic. The chunky simulacra of military craft were impressive for the Atari VCS, but nothing like the promise of the artwork. The only link between game and cover was the striated blue sky that served as a backdrop to both. Nintendo could not risk such a disconnect during their tenuous launch. Gail Tilden, NOA's VP of Brand Management felt that, 'There was an over-promise in the games that had been introduced prior...The consumer might see some beautiful fantasy graphics on the front, or a photographic image of people playing tennis, and then it was really just some enhanced version of Pong.' The idea was to deliver exactly what the consumer expected.

The minimal color scheme, simplified pixel presentation, and seal of quality
emphasized Nintendo’s militant focus on a unambiguous, unified product line. The stream of mediocre unlicensed games that clogged the US market in 1983 would find no quarter on this new Entertainment System. Of course, the seal of quality was more marketing gimmick than actual guarantee. A more accurate moniker would have been ‘seal of control,’ the final symbolic stamp of Nintendo’s aggressive licensing structure. The seal did not fend off bad games, but it did guarantee that Nintendo had had their hand in sculpting the game’s content to conform to the corporate image.

The sole differentiator between launch titles was Nintendo’s own sub-classification system. Nintendo created seven series headings grouped by genre and peripheral: Action, Arcade, Light Gun, Programmable, Sports, Education, and Robot. As expected, the Action, Arcade, and Sports series were well stocked with Nintendo’s roster of hits. The Arcade series boxes (Donkey Kong, Donkey Kong Jr., Donkey Kong 3, Mario Bros., Popeye) were further distinguished by a metallic silver band beneath their title text, ‘The Original!’ printed in script above, and ‘Arcade Classics Series’ in bold blue text below (figure 3.14).

3.14 NES launch title Donkey Kong, part of the ‘Arcade Series,’ featured the black box design.

Prior to Super Mario Bros.’s success, the Arcade series was clearly Nintendo’s marquee. The
promise of arcade-quality ports, absent since the ColecoVision, was now a reality. The Programmable series (*Excitebike, Wrecking Crew, Mach Rider*) showcased games with built-in level editors, all of which were hampered by the cart’s lack of a save function. The Education series, represented solely by *Donkey Kong Jr. Math*, was clearly meant to placate parents’ fears about bringing videogames into the home.

Though they aimed to distance themselves from Atari in the name of honesty, Nintendo was not above cover embellishment. The graphics on black box games were still subtly deceptive. Most of the character graphics were given comic book motion lines to make the artwork more dynamic: the ball bounces from left to right in *Pinball*, the skier zooms into frame in *Slalom*, and barrels whoosh from Donkey Kong’s hand. Pixels were rotated at impossible angles, like the brick shrapnel flying from Mario’s hammer on the *Wrecking Crew* cover. Several sprites were given sub-pixel details to enhance their facial characteristics, like the angular grins of the *Volleyball* players, the supplexed wrestler’s articulated fingers on *Pro Wrestling*, and the x’ed eyes of the felled opponent in *Urban Champion*. One could argue that ‘The Original!’ claim of the Arcade series was a bit dubious considering *Donkey Kong* lost many of its trademark animations, much of its vertical height, and even an entire level (see chapter 2), but we can concede that these were the first versions of the arcade hits ported to consoles by Nintendo themselves.

Beyond the obvious artistic additions, the black box cover graphics ignore the NES hardware’s capabilities. Mario’s sprite on the *Donkey Kong* cover, inexplicably in blue and white attire, includes eight distinct colors: a peach flesh tone and its darker shadow; brown for hair, eyes, and mustache; two shades of blue for his shirt and boots; and, finally, white and two shades of gray for his overalls and hat. The in-game Mario has four: red for overalls and hat; a peach tone for his face, hands, and button; blue for his shirt and boots; and black as the fourth shared palette entry. His jump animation frame on the box is also incorrect. Mario’s left hand is raised during his jump, recasting Jumpman as the rising star of *Mario Bros.* and *Super Mario Bros*. The raised-fist jump is the action of bumping turtles and breaking bricks, not jumping barrels. The subtle allusion makes marketing sense, but neither it nor Mario’s new outfit are representative of what’s actually stored in CHR-ROM. Nearly every cover benefited from tasteful shading and color revisions. These supplements added depth and richness to the cover graphics, but they also exceeded the PPU’s palette and attribute limitations. Nintendo certainly scaled back the fantasy of earlier Atari fare, but the artwork was not a one-to-one representations of the product within.

Early black box titles also had a distinctive hang-tab on the back of the box. The tab could be punched out and flipped outward so the game would hang from the metal posts used in retail displays, like the blister packs used to encase toy action figures. The hang-tab boxes were quickly amended, encouraging retailers to shelve NES games like VHS movies or books.
There were also significant size differences between NES and Famicom boxes, due in part to the size of the cartridges within. As discussed in chapter 1, Famicom carts were designed to resemble audio cassettes. The Sony Walkman was wildly popular in Japan in the early 1980s and it made marketing sense to evoke a visual resemblance between cassettes and videogames. However, as cartridge hardware expanded to accommodate additional mapper ICs, batteries, and the like, Famicom cartridge shapes began to diversify. This happened as early as 1984, when the Family BASIC cartridge's extended WRAM and battery switch required a taller PCB and cartridge shell. Famicom cartridges likewise came in a vast array of colors, apparently left to the discretion of the licensee. The only consistency in Famicom design comparable to the black boxes were the early titles that used the 'pulse line' graphic on their cartridge shells. Fourteen Famicom cartridges manufactured between July 1983 and October 1984 featured the pulse line graphic—named after its resemblance to the output of a heart EKG—in lieu of game graphics or illustrations.⁵⁷

NES game paks had to be wider due to the additional pins along the PCB. But they were also taller, a design discrepancy necessitated by the NES's ZIF mechanism. The pak had to be long enough to slot easily into the console without losing either the player’s fingers or the game itself within the chassis. The increased length had the side benefit of accommodating further cartridge hardware. Later Koei titles like Gemfire (1992), for instance, that used the MMC5 mapper, a battery, and SRAM, needed the extra clearance to fit the PCB inside its plastic molding.⁵⁸ However, paks such as those were outliers that came late in the NES's life cycle. The majority of paks contained small PCBs that could have fit in much smaller cases.

Again, beyond the obvious mechanical reasons, Nintendo's industrial designers chose the cart’s dimensions based on a popular consumer electronics format: the VHS tape. If the NES console could integrate discreetly with similarly boxy home entertainment equipment, the games would need to do the same. Of course, extending the game pak even further to mimic a VHS tape would have made them both absurdly tall and far too large to fit in the console. Instead of adding more empty space to the game pak, Nintendo extended the box height. Consequently, all NES games had styrofoam risers nested at the bottom of the box, used to prop up the shorter cartridges. NES boxes measured approximately 12.5cm x 18cm x 2.25cm compared to the 10.5cm x 19cm x 2.5cm dimensions of a VHS sleeve. Though a touch shorter and thinner, NES boxes fit comfortably alongside a movie library or video rental display.

Several early NES titles shared a hidden bond with their Famicom brethren that was not obvious from their exterior appearance: they smuggled Famicom boards with Nintendo-manufactured adapters built in. In figure 3.15, two early NES PCBs are exposed side by side—Excitebike on the left and Gumshoe on the right.
The slimmer green PCB protruding from the top of the black connector is actually the rear of a 60-pin glob top Excitebike board. The black plastic housing and the PCB extending from its bottom is the NES-JOINT-01, a Famicom-to-NES converter. The NES-JOINT not only does the proper pin conversion to make the pak fit normally into the NES, it also adds the CIC lockout chip (seen to the immediate left of the resistor), effectively piggybacking copyright protection onto the Famicom board. If one dismantles the adapter from the cartridge and pulls out the Excitebike PCB, any Famicom board can be slotted in its place and played normally on the NES. Aftermarket Famicom-to-NES convertors, like the Honey Bee Family Adapter, were eventually sold as standalone products, but Nintendo added their own covert solution in order to speed the manufacture of carts for the US market.

Also note the posts supporting the PCB/adapter hybrid. The cartridge’s center post supports the Famicom PCB, while the black plastic bridge is slotted onto two posts at either side. Molded brackets hold the 72-pin board in place. The center post and the four additional posts at each corner were used to fasten the game pak shell halves together with flat head screws. Thus the earliest NES carts were easily disassembled with a standard screwdriver. However, once Nintendo discontinued the inclusion of the NES-JOINT, two of the posts were eliminated and the five slotted screws were swapped for three 3.8 mm security screws, requiring a special bit to remove. To compensate for the removal of the
upper two screws, the flat cartridge top was replaced by two interlocking plastic prongs used to secure the cart’s halves in place. The screw count and plastic surface along the pak’s upper edge is a quick visual cue to approximate the game’s production date (and has since become a variant sought by collectors).

Nintendo had several reasons to limit access to the cartridge’s interior. First, they could prevent children from opening and potentially damaging the delicate PCBs. Second, they could deter pranksters or thieves from swapping PCBs between cartridge shells, a potential problem in the videogame rental era. Third, they could bolster their nascent service industry. As the popularity of the NES skyrocketed in the States, Nintendo Service Centers popped up nationwide. Locking down access to the cartridge ensured that consumers would bring their service dollars to authorized technicians.

Worldwide, box styles varied by region. In Great Britain, Canada, and France, Nintendo largely used the extended box design of the United States. The enhanced pixellated cover artwork was also identical. Minor variations included a line of text (‘GAME OF’) to prefix the title, along with the relevant regional translation. The French cover for *Hogan’s Alley*, for instance, said, ‘GAME OF / JEU DE HOGAN’S ALLEY.’ The Seal of Quality was similarly duplicated and translated. In regions where Mattel distributed the NES, their corporate logo was affixed to the front cover near the Seal of Quality.

In Spain, the cover was altered more significantly. The pixellated artwork was replaced by the cartoon illustration style of the Famicom originals. The Nintendo logo, updated to the more modern red and white version, was moved to the lower left and subtitled with ‘VERSION ESPAÑOLA.’ The series icon was deleted and replaced with a translated text version. For example, *Balloon Fight*, now in a silver box, had the text ‘SERIE ACCION’ directly beneath the title (figure 3.16). The greatest international divergence in box design came from the ‘short box’ variants released in certain Asian and European markets. As seen above, the styrofoam risers were eliminated from the packaging so the boxes fit the actual height and width of the game paks.
As the NES entrenched in the US market, Nintendo strayed from its uniform marketing aesthetic. The final few titles to use the black box style were not actually black—Metroid and Kid Icarus, for instance, adopted the pixellated illustrations and overall design, but came in silver boxes. Ice Hockey was grouped into the Sports Series but had a solid blue box with the photograph of a hockey player in place of the usual pixels. Nintendo gradually relaxed the uniformity of their first-party titles, maintaining no consistent house style beyond the prominent Nintendo logo. The Series moniker was unceremoniously abandoned with the black boxes.

Some third-party developers adopted Nintendo’s packaging aesthetic to establish their own brand identity. Bandai, one of the first licensees to release games in the US, hewed closest to the black box aesthetic. Their boxes featured a black to gray gradient background, a Nintendo-style diagonal marquee, and a literal screenshot in place of Nintendo’s augmented pixels blocks. They also added a cartoon illustration of the game’s lead character overlaid on the screenshot and invented their own series categories. Chubby Cherub (1986), Ninja Kid (1986), and M.U.S.C.L.E. (1986), for example, were included in the short-lived ‘Character Action Series.’

Konami, as well as their subsidiary Ultra Games, adopted silver boxes with vibrant cartoon illustrations of in-game action dominating the cover. Early titles by Capcom, like Trojan and Mega Man, featured a gradient blue background with floating neon wireframe grids. Their later boxes shifted to a flat purple background. Other third-parties followed suit: Jaleco opted for white boxes, unlicensed manufacturer Tengen chose gold, and Tecmo
titles had a simple red band across the box bottom.

While packaging styles shifted to better differentiate their developers and distributors, the manufacture of NES games was kept under strict control. Nintendo was the sole gatekeeper to their console. Submitting to their licensing terms meant relinquishing all control over production, schedule, and quantity. Worse still, developers had to pay upfront for the privilege, as Donovan explains:

Licensees had to pay Nintendo to manufacture their game cartridges so even if the game sold badly Nintendo made a profit. Nintendo also took a cut of every NES game sold, dictated when the game could be released, told licensees how many games they could release every year, and got to decide whether a game was good enough to be released.⁶¹

The same company that had supplicated itself to US toy retailers now had unprecedented power over their own corporate image, from content to hardware. As Nintendo steamrolled the competition, the situation worsened for developers, now caught in a double bind. Producing a videogame hit meant tapping into the massive NES audience, but obtaining a license meant bending to Nintendo’s will. Nintendo further stifled their competitors’ efforts by demanding all games developed for the NES be exclusives. Atari, NEC, and Sega were hard-pressed to find any developers willing to risk the cash grab and port their NES titles to competing platforms. However, Nintendo did have their favored licensees. Top-shelf developer Konami, for instance, were able to circumvent restrictions by creating a subsidiary company, Ultra Games. When Konami reached their yearly ceiling for game releases, they could then transfer publishing credit to Ultra.

Nintendo received a US patent on the distinctive grey plastic shell (figure 3.17), solidifying an iconic look for the tens of millions of cartridges they manufactured.⁶² The rest of the world never saw the rich palette of carts unique to the Famicom. Beyond a few exceptions, such as the 3- and 5-screw variations explained above and the gold Legend of Zelda cartridge, Nintendo game paks were resolutely uniform.⁶³

NES game paks measured 12cm x 13.3cm x 1.6cm. The cartridge bottom was notched to accommodate the NES’s internal design, with the exposed 70-pin portion of the PCB jutting from the cartridge interior. A recessed area near the left edge of the cartridge (not seen in the patent illustration) fit the forefinger and thumb, a welcome aid for inserting and removing the cartridge from the console interior. Ridges were molded into both sides of this recessed area and extended vertically along the front of the cartridge, providing both a convenient gripping surface and a subtle aesthetic flourish echoing the striped ridges of the NES. A rectangular sticker was affixed to the front of the cart, featuring a reproduction of the game’s box artwork. The upper lip of the sticker, displaying the game’s title, folded around the top of the cartridge. This served two convenient purposes: the title was visible
both when the pak was inserted into the console and when it was stacked with other games (a common way to store games, since most boxes were discarded). To the bottom left of the sticker was a small embossed triangle indicating the proper way to insert the game pak into the console.

![Image of NES game pak design](image)

3.17 *Nintendo’s patent illustration for the iconic NES game pak design.*

Though Nintendo ultimately failed to curb the production of unlicensed games, they could enforce patent rights over their cartridge design. While unlicensed companies’ box designs could pass easily for a ‘real’ Nintendo game (minus the seal of quality),\(^6^4\) they had to alter their carts to avoid infringement. This resulted in a number of cartridge redesigns that creatively avoided Nintendo’s style while still enabling the cart to properly fit inside the console. Tengen produced a svelte black cartridge with symmetrical ridged notches built into the lower half of its prominent angled lip. While aesthetically distinctive, the lack of finger holds on either side made the cart difficult to remove from the NES. American developer Color Dreams, who later transformed to the Christian-centric software house Wisdom Tree, designed a curved gripping surface and attention-grabbing powder blue cases (later replaced with black). Camerica opted for metallic gold and silver carts, while AGCI simply chose an alternate shade of grey to complement its starkly minimal exterior. Today, the colorful banner of the unlicensed game pak is carried on by the fan community. Homebrew manufacturer and distributor Retrozone, for example, produces carts in a variety of translucent hues: green, blue, red, and clear.\(^6^5\)
Hitler’s Exploding Head

Nintendo’s control over NES cartridges extended beyond manufacture and distribution; they had the final say on content as well. Agreeing to be a licensee meant that Nintendo could amend any game-related materials, from commercials to graphics, they felt might be objectionable to their audience. Among Nintendo’s content guidelines were rules governing in-game portrayals of religious iconography, sex, nudity, racial stereotypes, controlled substances, profanity, violence, and politically incendiary content. According to former NOA Product Analyst/Specialist Phil Sandhop, these rules were clearly documented and uniform for all publishers—'Everyone had a copy of the policy and it was the same for all games'—including Nintendo. 'We maintained an even hand with our games as well as the third parties,' Sandhop recalls.

Indeed, Nintendo’s content guidelines were not reserved solely for licensees. Many of their own games were either modified for Western audiences or withheld from certain regions altogether. One conspicuous example is デビルワールド, or Devil World, released for Famicom in 1984. Designed by Shigeru Miyamoto and Takashi Tezuka prior to Super Mario Bros., the game was an amusing marriage of Pac-Man-style gameplay and Christian iconography. The player controlled a small, fire-breathing dragon named Tamagon, who was confined within a shifting maze administered by Devil, the winged blue demon perched at the top of the screen (figure 3.18).
Tamagon collected crosses, Bibles, and white dots while avoiding either being crushed by scrolling pillars or colliding into pink, one-eyed creatures. Despite its use of Christian iconography, *Devil World* was likely not a work of deliberate blasphemy—Tamagon and his foes were colorful cartoon sprites and the titular Devil wore red boots and underwear. Nonetheless, Nintendo chose not to release the game in the US based on its potential to offend conservative parents. Europe, however, received a PAL conversion with no alterations four years later.

Even Nintendo’s star franchises were subject to drastic revisions. Based on the monstrous success of *Super Mario Bros.* in Japan and abroad, it seemed sensible for Nintendo to quickly follow up with a sequel. *Super Mario Bros. 2* was a Family Computer Disk System exclusive that expanded the gameplay of its predecessor without significant graphical alterations. Instead, the design team ratcheted up the difficulty and introduced a
number of devious gameplay changes: Mario and Luigi had noticeable differences in running and jumping ability; power-up blocks could yield harmful poisonous mushrooms; timing jumps according to shifting winds was necessary to cross long gaps; and jumping the flagpole was specifically programmed into the engine, though it could lead to a warp zone that would send Mario to earlier levels, negating his progress. Game director Kensuke Tanabe worked directly against many of the conventions established in Super Mario Bros., creating a far more challenging game. As a result, the disk’s booklet and sleeve featured a gold ribbon that read, ‘For Super Players,’ meant to warn away those who had not mastered the first game.⁶⁸

Nintendo of America deemed the game unsuitable for American audiences due to both its difficulty and its close visual resemblance to its predecessor. Nintendo chose an alternate FDS game, Yume Kōjō: Doki Doki Panic (1987), to take Super Mario Bros. 2’s place. Doki Doki, also directed by Tanabe, had actually begun as the prototype for a possible Super Mario Bros. sequel, but both he and Miyamoto were dissatisfied with the early results. Super Mario Bros.’s horizontal gameplay was shifted to a vertical orientation and designed around co-operative platforming, but it did not prove to be much fun.⁶⁹ Tanabe shelved the prototype and focused on the Super Mario Bros. revision that eventually became the FDS Super Mario Bros. 2.

The abandoned prototype was eventually revived for a cross-promotional videogame designed to coincide with Fuji Television’s 1987 Yume Kōjō festival. The festival mascots starred as the four lead characters in the game, set in an Arabian-inspired dream world. Miyamoto and Tanabe worked in earnest to infuse Doki Doki with more Super Mario Bros.-inspired elements, like horizontal scrolling, warp zones, and hidden power-ups. As a result, Doki Doki’s gameplay evolved into a sensible substitute for the American sequel to Super Mario Bros. The FDS disk was converted to an MMC3 cartridge and received a graphical overhaul, replacing the Yume Kōjō mascots with Mario, Luigi, Princess, and Toad, along with other visual and audio tweaks meant to bring the game further in line with the Mario universe. However, the anachronistic Arabian setting and an all-new cast of enemy characters made the US Super Mario Bros. 2 something of a conspicuous outlier in the Mario catalog. In a final twist, the US version was eventually ‘ported’ back to Japan as an MMC3 Famicom cartridge titled Super Mario USA (1992).

Despite the level of polish applied to the Doki Doki / Super Mario Bros. 2 conversion, a notable content violation managed to fall through the cracks. The enemy character Birdo (mistakenly labelled ‘Ostro’ in the manual) had a cross-dressing predilection forbidden by Nintendo’s guidelines on sexuality. The manual read, ‘He thinks he is a girl and he spits eggs from his mouth. He’d rather be called “birdetta.”’⁷⁰ Though there is no evidence US children or parents were offended by the description, Nintendo softened Birdo’s gender ambiguity in later versions of the game.
As the Birdo example highlights—and despite Sandhop’s insistence otherwise—
Nintendo’s guideline enforcement frequently appeared inconsistent, even arbitrary. This
was especially true in the case of modifications made to Western-developed PC titles that
were ported to the NES. American developer Jon Van Caneghem’s first-person RPG *Might &
Magic* was originally released for the Apple II in 1986. Due to its popularity, it was
eventually ported to a range of PCs and consoles, including the Famicom, as マイト・マンド・マジック (1990), and the NES, as *Might & Magic: Secret of the Inner Sanctum* (1992).
The Famicom/NES ports received graphical upgrades, additional dialogue, and new
locations, puzzles, and enemies. However, in its transition from American PC game to
Famicom port and back again as NES cartridge, *Might & Magic* underwent a number
Nintendo-mandated content revisions. NES fan site Flying Omelette provides an extensive
list of differences between the Famicom and NES versions, including the erasure of small
halos above the angels’ heads on the Wheel of Luck plaque (though not their wings), the
deletion of a plaque adorned with a minotaur head (reminiscent of a Satanic goat’s head),
and all in-game instances of ‘devil’ revised to ‘incubus.’ Other objectionable content
apparently eluded Nintendo’s testers. Both the female Water Elemental and Medusa
sprites, for example, appear topless, with the latter fully exposed. *Might & Magic* is a
conspicuous example of double translation. Instead of porting the native English PC version
directly to the NES, it was refracted through both the Japanese language and Nintendo’s
content regulations before its US release.

The influential graphic adventure game *Maniac Mansion* famously received similar
treatment. Developed by LucasArts in California, *Maniac Mansion* originally debuted on the
Commodore 64 and Apple II in 1987. As LucasArts employee Douglas Crockford recounts,
the console port of *Maniac Mansion* was LucasArts’s first for the NES, so it was important
that they followed protocol in hopes of expanding their audience beyond the PC market. As
he says, ‘In the course of converting to Nintendo, we had to redesign all of the art in order
to conform to Nintendo’s screen geometry. We also made some changes to adapt the game
to a younger audience.’ Nintendo’s first round of amendments included changing several
unacceptable words and phrases, such as ‘pissed,’ ‘sucked out,’ ‘for a good time,’ and ‘NES
SCUMM.’ The last bit of text in particular perplexed Nintendo’s content screeners. What
was NES SCUMM? LucasArts reassured them that SCUMM was, in fact, short for Script
Creation Utility for Maniac Mansion—the engine used to craft *Maniac Mansion* (and many
subsequent LucasArts adventure games). Owing to their own misinterpretation, Nintendo
decided to nix the reference. They also vetoed the pixel nudity found on an in-game poster
and marble statue. Based on their careful scrutiny, Crockford was surprised that Nintendo
had passed over several acts of violence without mention, in particular a sequence where
the player could place a hamster in the microwave and make it explode. It turns out
Nintendo had inadvertently overlooked the scene. *Maniac Mansion*’s publisher Jaleco was required to purge the rodent microwaving option from a later PAL release of the game.

Like any emergent media, the rising popularity of videogames attracted the attention of franchises from film, television, and a host of other media and products that hoped to cash in. Thus the NES library received a glut of licensed properties, including cartoons (*The Little Mermaid, The Simpsons: Bart vs. The World, Tiny Toon Adventures*), toys (*Teenage Mutant Ninja Turtles, Micro Machines*), sports (*Major League Baseball, NFL, Michael Andretti’s World GP*), television game shows (*American Gladiators, Jeopardy!, Fun House*), movies (*Robocop, Total Recall, Mad Max, Darkman*), and even food (*M.C. Kids, Spot, Yo! Noid*). The Famicom saw the same influx of cross-media ventures, though understandably they were drawn from pop culture sources familiar to Japanese audiences. Developers adapted numerous anime and manga series into Famicom games, such as ドラゴンボール神龍の謎 (*Dragon Ball: Shen Long no Nazo*), アキラ (*Akira*), キャプテン翼 (*Captain Tsubasa*), 鉄腕アトム (*Tetsuwan Atom*), and クレヨンしんちゃん オラとポイポイ (*Crayon Shin-chan: Ora to Poi Poi*).

Although Japanese comics, cartoons, and animated films were gaining fans in the 1980s (see chapter 2), Famicom publishers did not expect American children (or their parents) to accept these unfamiliar or otherwise ‘weird’ foreign characters. When Japanese media properties did cross over, they were largely revamped for NES release. Kohler writes:

> Some major graphical alterations happen when the game in question is based on an anime series, and the game publisher would rather that Americans didn’t know that. So the famous manga character, a strange duck-like ghost, in Bandai’s Famicom game *Obake no Q-Taro* was changed to a curly-haired angel for the game’s US release as *Chubby Cherub*. The hero of *Gegege no Kitaro*, a Famicom game based on an anime about a strange little boy who hunts ghosts, was changed to a ninja and the game was released in the US as *Ninja Kid*, stripped of its anime roots.

Another notable example is 仮面の忍者 赤影, or *Kamen no Ninja: Akakage*, a 1988 Famicom game based on the anime series *Akakage*. The TV show, set in feudal Japan, starred the titular ninja, who wore a distinctive red mask shaped like a bird’s spread wings. In 1990, Capcom released the FDS game *Kamen no Ninja: Hanamaru*, itself a cartoonish adaption of the *Akakage* series starring a child-like version of the ninja in the red bird mask. Based on their source material, neither game was suitable for US release, so Capcom undertook a substantial localization. The result was *Yo! Noid*, an NES cartridge cum advertising game centered around the Noid, a Domino’s pizza mascot prevalent in US commercials during the 1980s.
Yo! Noid’s localization was an impressive, wholesale graphical update. Not only did Capcom replace Hanamaru with the Noid, but the natural landscapes— islands, arctic tundra, etc.— were redrawn to resemble urban locales reminiscent of New York. However, the underlying structure of the levels remained unchanged. An island level in Kamen became an odd pier-side cityscape in Noid, but the patterns of platforms and the distinctive vertical bobbing motion of the level were identical in both versions. In Kamen, Hanamaru’s primary attack was a mechanical bird perched on his shoulder that he would send out to dispatch foes. Capcom cleverly swapped the bird for a yo-yo, an object that could feasibly replicate the avian swooping attack. Similarly, the card game used to resolve boss confrontations was converted into a pizza eating contest.

Though not all developers could afford such drastic updates, Famicom games routinely had their cover artwork altered for worldwide release. The wide-eyed cartoon characters typical in Japanese anime and manga were swapped for more realistic characters inspired by Western comic book, sci-fi, and fantasy traditions. For instance, the cover of アルゴスの戦士 はちゃめちゃ大進撃 (Argos no Senshi: Hachamecha Daishingeki) portrays the cartoonish hero Senshi, who would not look out of place in an Astro Boy episode, wielding the Diskarmor, a shield/boomerang hybrid that closely resembles Captain America’s trademark weapon. The English localization changes Senshi’s name to the game title, Rygar, updates the character artwork to resemble a He-Man knock-off, and drops the Captain America characteristics of the Diskarmor. Beyond the artwork and name changes, little else is localized for the NES version.

Rockman is one of the most notorious examples of NES artwork localization. The lively anime-inspired Famicom box art closely matches the look of the in-game sprites. The US version, titled Mega Man, is an artistic train wreck of disjointed perspectives, improbable anatomy, and questionable characterization that bears little resemblance to the game advertised. ‘Box art Mega Man’ has since become part of NES lore, even resurfacing as a playable character in 2012 fighting game Street Fighter X Tekken. Despite the Mega Man series’s popularity worldwide, its inconsistent English translation actually contributed to the incoherence of its cast of characters. Rockman was originally designed as an homage to rock ‘n’ roll: Rock’s robot sister’s name was Roll and characters from later Rockman sequels had names like Rush, Forte, Bass, and Beat. A handful of names survived unscathed, but the main character’s name change created the unremarkable pairing of Mega Man and Roll, completely undercutting the Rock and Roll allusion. The translation irony came full circle when Yellow Devil, edited in accordance with religious guidelines, was renamed Rock Monster.

The exceptions to cultural alteration were generally seen in game scenarios that either had universal appeal for Eastern and Western audiences (e.g. Donkey Kong) or were already heavily influenced by Western culture. The FDS game 悪魔城ドラキュラ, or Devil’s
Castle Dracula (1986) came to the NES as the UNROM game pak Castlevania in 1987. Protagonist Simon Belmont is a whip-wielding vampire hunter whose ultimate target is Dracula, drawn from Bram Stoker’s famous novel. Along the way, Belmont encounters horrors plucked from Western literature, film, and mythology, including Medusa, Frankenstein, the Mummy, lagoon creatures, and the Grim Reaper. Considering Nintendo’s self-censorship of Devil World, it is surprising that Konami exported Castlevania with little alteration. Belmont’s profession obviously required significant violence against horrific creatures and the Dracula mythology is tied closely to Christian orthodoxy—crucifixes, resurrection, drinking blood, and the like. Crosses figured prominently in the game, both as set decoration and as one of Belmont’s auxiliary weapons. But beyond softening the title and altering the title screen graphics, Devil’s Castle Dracula received minimal localization. In fact, beyond the title screen, all in-game text was already in English. The most significant change was technological, since once again, a disk-based game had to be converted to a cartridge. Removing the disk’s save feature from Castlevania made it a far more difficult game.

Even in cases where Nintendo’s guidelines benefited cultural sensitivity, their application was inconsistent. Capcom’s unique ‘jump-less’ platformer Bionic Commando, featuring a protagonist with an extendable robotic arm that allowed him to swing across platforms and grapple objects, was first released for the Famicom as ヒットラーの復活トップシークレット (1988), or Hitler’s Resurrection: Top Secret. The game’s primary antagonists were soldiers of the Imperial Army, a neo-Nazi regime led by General Wiseman (ワイズマン). The Famicom cover art portrayed the hero Raddo (ラッド) swinging into frame while riddling an enemy soldier with bullets. Looming large in the horizon was an oversized Hitler who, true to the title, was resurrected for the final boss fight. In-game locations were littered with Nazi regalia: swastikas adorn flags, podiums, dossiers, and architectural facades. A cutscene featuring Wiseman shows him raising his left fist in the manner of the Nazi heil salute.

Nintendo rightfully decided that the game’s scenario and imagery would be distasteful outside Japan—or outright banned in Germany, where allusions to Naziism are strictly banned according to Strafgesetzbuch. Prior to US and European release, all swastika graphics were changed to eagles (still vaguely reminiscent of Third Reich propaganda), the Nazis were renamed to ‘Badds,’ Wiseman was questionably renamed as Generalissimo Killt, and Hitler received the pseudonym ‘Master D.’

Despite these revisions, there were numerous striking oversights. Bionic Commando’s instruction manual presents the story as follows, in poorly translated English:

I’ll talk about a person which I’ve met when I was young. In 198X we’ve found Nazz’s top secret material called Abatros, a plan which never was put to practice. Imperial
forces Generalissimo Killt had never seen this plan, and decided to materialize this plan. The federation decided to stop his attempt by sending our hero Super Joe, but lost contact with him. Our brave man (you the player) was sent to the empire with a special mission to rescue Super Joe, this story begins from here...

During the in-game prologue, the above text reads: ‘IN 198X WE’VE FOUND THE BADDS’ TOP SECRET MATERIAL CALLED ALBATROS WHICH WAS NEVER PUT INTO PRACTICE,’ while the original Japanese text translates as, ‘A classified document of the Nazis was found in 198X and we learned of the existence of the Albatross Plan which was never realized (by the Nazis).’ The resulting English transcription managed to create an unfortunate hybrid/portmanteau of Nazi and Badd.

The most egregious oversight is Hitler himself. Though his name changed, his in-game character and portrait sprites were unaltered. This is especially significant in the game’s final sequence. As Hitler attempts to flee in a helicopter, Raddo swings from above and launches a missile into the cockpit. As he yells ‘AH...!’, a four-frame animation shows Hitler’s head graphically disintegrating into chunks of blood and flesh (figure 3.19). Judging by the makeshift translations and roughly four month turnaround time between Famicom and US release, Capcom likely did the best they could to revise the game’s content. Much like Famicom Donkey Kong’s kill screen, few players were likely to see this gory sequence, since Bionic Commando was a notoriously difficult game. Substituting a non-violent finale would have meant either significantly overhauling the game’s pattern tables or excising the sequence from the source code altogether. In either case, it would have been a diversion of resources already dedicated to English localization.

3.19 The four-frame ‘exploding Hitler’ sequence seen at the conclusion of Bionic Commando. (Image: gambit.mit.edu)

In the years since Bionic Commando’s release, Hitler’s exploding head has metamorphosed from censorship oversight to NES cultural lore. The graphic animation
continues to propagate, both as Internet meme (such as the J-Pop-infused web animation ‘OMG Hitler’s Exploding on Bionic Commando!’) and in videogame adaptation (it was faithfully 'remastered' in the 2008 HD remake of the NES game, Bionic Commando: Rearmed).”

**Translations**

The Family Computer’s conversion from a Japanese to an international console was fraught with translation problems. Nintendo’s calculated effort to preempt their domestic piracy problems led to a machine burdened with a brilliant but flawed lockout chip. The market pressure to conform the console’s exterior to the expectations of a broken and reticent US videogame industry led to a novel but faulty spring-loaded cartridge mechanism. The desire to protect both consumers from objectionable content and their own console from the fate of its predecessors led Nintendo to implement draconian content regulations and licensing policies that frustrated developers and hampered Nintendo’s reach in the European market.

And yet like the Famicom, a console nearly derailed after launch by deteriorating buttons and degrading RAM, the Nintendo Entertainment System persevered and prevailed. Millions of Nintendo players absorbed the console’s errors and flaws into the cultural lexicon of videogame history. Players remember the Metal Gear guard’s exclamation ‘I FEEL ASLEEP’ or Error’s existential phrase not as unforgivable flaws, but as playful, ridiculous, and even mysterious expressions of a specific era of gaming. A popular t-shirt portraying an NES game pak with the words ‘Blow Me’ underneath encapsulate, if inelegantly, a shared cultural lore. ‘I AM ERROR’ is as much a personification of Nintendo’s 8-bit console as it is a mistranslated bit of dialogue from one of its games.

---

³ Williams, 3.
⁴ Bogost and Montfort, *Racing the Beam*, 76-9, 127.
⁵ Kline et al., 105.
⁶ Donovan, 95.
⁷ Kline et al., 105.
⁸ “Computer or Video Games.” *Nytimes.com.*
⁹ Kent, 286.
¹⁰ Sheff, *Game Over*, 160.
¹¹ The Advanced Video System prototype was also on display for several years at the Nintendo World
store in New York City. The tape deck, infrared controllers, light gun, and joystick were all viewable under glass, though the control deck was conspicuously absent.

"Margetts and Ward, “Lance Barr Interview.”

"Sheff, 162.

"Semrad, “New Nintendo system way ahead of the field.”

"The Fairchild Channel F used a card edge connector, but its cartridges loaded horizontally, similar to an 8-track cassette. The Intellivision loaded cartridges in the same manner.

"Yukawa, “Front Loading Apparatus for a Memory Cartridge Utilized for a Data Processing Machine.”

"Margetts and Ward.

"Edwards, “No More Blinkies: Replacing the NES’s 72-Pin Cartridge Connector.”

"Nakagawa, “Recordable Data Device Having Identification Symbols Formed Thereon and Cooperating Data Processing System Having Registering Symbols.”

“NES expansion port pinout,” NesDev Wiki

“While initial runs of the NES had an easily-accessible expansion port, later revisions covered the port with a plastic hatch that to be forcibly broken to access. With the introduction of the top-loading NES-101, Nintendo dropped the port altogether.

"Pichugan, “Steepler начал продавать Dendy” (Steepler starts sales of Dendy).”

"Also see Altice, “Dendy: The Unofficial Official Famicom of Russia.”

"Gifford “Aqui se faz aqui se paga: The NES in Brazil.”

"Nakagawa, “External memory having an authenticating processor and method of operating same,” and Nakagawa, “System for determining authenticity of an external memory used in an information processing apparatus.”

"Segher, “The weird and wonderful CIC.”

See “CIC lockout chip” and “CIC lockout chip pinout,” NesDev Wiki.

"DRM is any technology that inhibits uses of digital content that are not desired or intended by the content provider.” From “Digital rights management,” Wikipedia.

"O’Donnell, “Production Protection to Copy(right) Protection,” 54.

"Ibid., 61. Note that O’Donnell conflates (or at least is unclear about) the lockout chip’s program and its hardware. Throughout, he calls the CIC the ‘10NES chip.’ However, 10NES is the program that runs on the CIC, not the chip itself. The patent text he cites (55) refers to 10NES properly, but O’Donnell never makes the distinction in his article.

"Donovan, 174.

"United States Court of Appeals, “Atari Games Corp. v. Nintendo of America Inc.”

"Ibid.

"Atari/Tengen programmer Ed Logg says that while Atari’s lawyers did obtain the 10NES patents under false pretenses, the information was not used in the production of the Rabbit chip:

tsr: Yeah, Game Over painted Tengen as basically stealing the patents for the lockout chip.

EL: The trouble was it was already done before we saw it. We had already done the Rabbit chip long before we had seen it. So it's already done, and we see this and we're like "Oh shit". (laughs)

tsr: So you know for a fact the Rabbit was 100% original?

EL: Yeah. I walked into the lab and they were reverse engineering the chip, and I asked what they were doing and they said "Don't ask". (laughs) So I know the company was doing it, and I knew the people involved doing it.

tsr: Was this a major undertaking, the engineering?

EL: It was basically three people. And they were certainly looking at the chip, let's put it that way. I'm sure they did a lot more that I didn't see. Tweaking the signals, seeing what comes out, that kind of
stuff. And I was working on the FC at the time. We had reverse engineered the Famicom and I was already developing on it.

See Gifford, “Tetris... forever.”
"United States Court of Appeals, “Atari Games Corp. v. Nintendo of America Inc.”
"Valesh, “Nintendo, America!”
‘First Nintendo Commercial,’ YouTube.
"Nintendo of America, ‘VIDEO ROBOTS.’
"Cifaldi, “In Their Words: Remembering the Launch of the Nintendo Entertainment System.”
"Due to R.O.B.’s early demise and minuscule software support, ‘big box’ versions of Gyromite and Stack-Up (with included accessories) command high prices among collectors.
“My thanks to Tursi and x87bliss at AtariAge for decoding R.O.B.’s behavior through careful reverse engineering and experimentation. See godslabrat, ‘Any interest in NES ROB homebrews?’
“The precise color values are not critical, merely the contrast of light and dark. See ibid.
‘x87bliss confirmed this by changing the sprite 0 offset to another location. However, altering the timing from one frame per color to two frames per color caused R.O.B. to no longer receive messages. See cited thread above.
"Cifaldi, “In Their Words: Remembering the Launch of the Nintendo Entertainment System.”
“Another irony in light of Nintendo’s initial marketing was R.O.B.’s platform agnosticism. Since the robot has no direct interface with the NES, it is possible to issue commands via any console that can replicate the proper sequence of screen flashes. Once R.O.B.’s flicker sequences were decoded, for instance, AtariAge community member Pioneer4x4 wrote a simple program to control the robot with an Atari VCS. See Pioneer4x4, ‘I got my Atari to control my Nintendo R.O.B. Robot!’
‘Note that some early NES technical documents have the Zapper flags reversed (e.g., Chadwick, “Nintendo Entertainment System Documentation Version: 2.00”)
“The four light gun launch titles were Duck Hunt, Gumshoe, Hogan’s Alley, and Wild Gunman.
“Okada, “Video Target Control and Sensing Circuit for Photosensitive Gun.”
“Ibid.
“My use of quintile does not adhere to its precise definition, since the chunks are not uniform in area, but I simply mean to convey that the screen is divided into five distinct metatiles.
“Question about coding for NES zapper,” NesDev.
“Third-party company Activision struck a better balance of realism and embellishment on their covers. The arrangement and shape of the artwork typically echoed the graphics seen onscreen, but in a more colorful cartoon style.
"Cifaldi, “In Their Words: Remembering the Launch of the Nintendo Entertainment System.”
“Early black boxes also featured a black, circular Nintendo sticker to seal the box shut. Complete black box games with intact stickers and hang-tabs are now highly sought after by collectors.
"bootgod, “Family BASIC (Revision A1).”
“Pulse Line Cartridges.” Famicom World.
“bootgod, “Gemfire.”
“Feldman and Walters, “Konami Box Art.”
“Donovan, 168.
“Yukawa, “Cartridge for Game Machine.”
“Some other notable variations of the NES game pak: the yellow test carts used in Nintendo’s
authorized service centers, black carts used in Famicom game kiosks in hotels, and the holy grail of NES collectors—the gold Nintendo World Championship cart awarded to contest winners.

“Unlicensed distributors could not copy the Nintendo seal’s text, but they could brand their own. HES boxes, for example, had their own circular serrated seal, appropriately free of any quality claims: ‘H.E.S. | CARTRIDGE FOR | NINTENDO | Use with a Nintendo | cartridge. (Instruction | enclosed).’

See http://www.retrousb.com. Among Retrozone’s cartridge reproductions is the updated version of Donkey Kong with the ‘pie factory’ level reinstated (see Chapter 2).

McCullough, “Nintendo’s Era of Censorship.”


W., “Interview with Shigeru Miyamoto Volumes 1 and 2.”

Kohler, “The Secret History of Super Mario Bros. 2.”

Nintendo of America, Inc. Super Mario Bros. 2 Instruction Booklet, 27.

“Might and Magic: Oddities, Theories, and Unused Content,” Flying Omelette.

Crockford, “The Expurgation of Maniac Mansion for the Nintendo Entertainment System.”

For those who might miss the 80s food references, the final three games featured characters representing McDonald’s, 7-Up, and Domino’s Pizza, respectively.

Kohler, Power-up, 207.

See Fieldsted, “Category Archives: Cultural Anxiety.”

See “Strafgesetzbuch section 86a,” Wikipedia.


Ibid. In particular, see “Script: NES (Japanese)” and “Script: NES (English).”

See “YTMD - OMG, Hitler’s Exploding on Bionic Commando!” and “Bionic Commando Re-Armed: Exploding Hitler Head,” poeTV.
4: Platforming

I don’t think anyone at Nintendo ever thought we would be able to pack so much value into a cartridge with a ROM of that capacity, or that we would sell so many worldwide.

- Masayuki Uemura, *Iwata Asks, Volume 2: NES & Mario*

I’ll never forget the first time I jumped the flagpole...My sister was right there watching, and we couldn’t believe it. Well, I’ve done it many times now so BELIEVE IT!

- Scott Kessler, *Dman’s Game Domain*, 1999

Before you begin an adventure, you need to know as much about where you’re going and who you’ll encounter as you can.


In 2005, Roger Ebert rankled videogame fans when he claimed that their favored medium was ‘inherently inferior to film and literature.’ He wrote that videogames, ‘by their nature require player choices, which is the opposite of the strategy of serious film and literature, which requires authorial control.’ A medium centered around ‘interactivity’—a dodgy concept to start with—could never produce the laser-like narrative focus and expressive content of a film or novel, guided by the vision of a single author. Videogames could be ‘elegant, subtle, sophisticated, challenging and visually wonderful,’ but never Art.

Videogame fans took the criticism to heart and responded in kind. The ensuing debate about the medium’s artistic merits and the small crop of ‘artistic’ games trotted out as proof of such merits eventually elicited a minor concession on Ebert’s part, some five years later:

I was a fool for mentioning video games in the first place. I would never express an opinion on a movie I hadn’t seen. Yet I declared as an axiom that video games can never be Art. I still believe this, but I should never have said so. Some opinions are best kept to yourself.²

Young and emergent media always prompt such debates regarding identity, form, and purpose. An artistic medium must always be a ‘pure’ medium, presiding over a specific set of functions that no other medium can perform. Film must enable expression that poetry cannot, and vice-versa. Such ‘medium specificity’ debates date back at least to Gotthold
Lessing’s 18th-century work, *Laocoon*, which differentiated the practices of painting and poetry, and took on a more modernist cast in the writings of art critic Clement Greenberg. Videogame medium specificity is finding its own contemporary champions in the work of game designers like Rod Humble, whose essays and games attempt to distill the form to its barest necessities—in his view, rules.⁴

Ebert’s argument, despite its eloquence, is founded on a mixture of tenuous generalizations (of even film and literature) and admitted ignorance—he had played only two videogames in his life. Elevating any medium to the status of Art presumes that we have developed a stable consensus about what Art *is* and *is not*. Critics like to think they are the ultimate arbiters of such discussions, but the continual broadening of art’s definition since Marcel Duchamp submitted a signed, inverted urinal to an exhibition has taught us that contemporary Art is defined more by institutional validation than individual tastes.

And Ebert’s preference of the medium to which he has devoted his career is understandable, albeit reliant on a specific *auteur-centric* understanding of filmmaking that privileges a single voice above the unnamed multitudes necessary to make a film. Rhetorical prejudices notwithstanding, Ebert’s comments did prompt an interesting, if equally flawed, question: has there been a *Citizen Kane* of videogames? And if so, what is it?

Once again the question rests on tenuous premises. First, we must accept that any media is pure enough to extract a representative work from a diverse and motley crowd, ignoring the border cases that blur distinctions between videogames over here and not videogames over there. Second, we must grant that such cross-media comparisons are valid (or at least not patently absurd), in the same way that we might ask if there is a *Rites of Spring* of photography, a *Notre Dame* of comic books, or a *Mona Lisa* of television. And finally, we must reach some agreement about precisely how a single film shoulders the burden of artistic validity for an entire medium. Ebert would likely agree that *Citizen Kane* solidified much of the grammar of film through an artful wedding of storytelling and technical mastery. If we permit ourselves to don media specificity hats momentarily, we can say that editing, lighting, depth of field, special effects, and framing told the tale of Charles Foster Kane in a way that only cinema could.

It took over fifty years of experimentation and refinement for the film industry to produce a work like *Citizen Kane*. Videogames, in comparison, are a few decades old. As a coherent commercial industry, even younger. Their form and function are certain to evolve as decades and centuries pass, rendering our cutting edge first-person shooters into primitive, naive exercises analogous to an Edison short revolving around a man sneezing. Nonetheless, videogames have a remarkable range of purposes and uses, from pornography to advertisement.⁵ And if I were forced to write a short list of videogames worthy of the *Citizen Kane* association, *Super Mario Bros.* would certainly make the cut.
Nintendo’s breakout 1985 hit, one of the most popular and influential videogames of all time, made the company’s name synonymous with videogames. In the US, ‘playing Nintendo’ replaced ‘playing Atari’ as the linguistic metonym for playing any videogame. *Super Mario Bros.* also entrenched the platformer as the NES genre *par excellence*, spawning a chain of look- and play-alikes that continue into the present day, spanning disparate platforms and generations. Just as *Citizen Kane* plumbs the technical depths of cameras, lenses, lights, and celluloid film, *Super Mario Bros.* is the consummate demonstration of the Famicom’s strengths and limitations. Its tiled world of plumber-sized pipes and expansive blue skies are built around the platform’s peculiar hardware constraints. If *Donkey Kong* was the game the Famicom *was engineered for*, a minimum set of hardware specs necessary to feasibly reproduce the arcade hit, *Super Mario Bros.* was the game the Famicom *was made for*, pushing those specs to their technological and creative limits.

This chapter delves into the game’s source code and analyzes the peculiarities of the Famicom’s cart and console hardware that shaped the expressive qualities of the game. We are platforming in both the technical and generic senses. *Super Mario Bros.*’ novel combination of narrative, linear scrolling spaces punctuated by warps, and tiered levels structured around clear objectives marked a decisive break between arcade-style games and those designed for home consoles (though, ironically it was both the sequel to the arcade title *Mario Bros.* and eventually ported to arcades soon after its NES release). The technical terms introduced in chapter 1 are explored in greater depth, using *Super Mario Bros.*’s engine to explore practical implementations of scrolling, metatiles, data compression, attribute tables, palette swaps, sprite 0 hit, and the like. The chapter also explores the unique and sometimes unintended innovations in *Super Mario Bros.*, including player movement beyond ‘world’ boundaries, exploits, and glitches.

I aim to play both sides of the critical field: first, to acknowledge, as a fan, the joy of playing what is certainly one of videogame history’s finest examples to date; and second, to support that statement of uncritical appeal with defensible formal examples. *Super Mario Bros.*’s simplicity to modern eyes perhaps obscures the sophisticated complexity of its inner machinations. In twenty-odd thousand lines of assembly code, burnt to mask ROMs and housed in plastic cartridges, Nintendo inaugurated a new era of console videogames. And despite the tens of millions who have played it, few have taken the time to lay its code and cartridge bare to examine how it works.

**Platformers**

スーパーマリオブラザーズ, or *Super Mario Bros.* (hereafter *SMB*), is a side-scrolling platformer released for the Famicom on September 13, 1985. Although Miyamoto is often
given the sole credit for its creation, he worked alongside a small team of programmers, designers, and composers over a period of several months to bring the game to life. Takashi Tezuka, the game’s co-designer, had worked with Miyamoto on the ‘dot-eating’ maze game Devil World. Koji Kondo, who composed the game’s inspired soundtrack, was also a Devil World alum. Toshihiko Nakago, the game’s lead programmer, had worked on Famicom ports of several Nintendo arcade properties, including Donkey Kong and Donkey Kong Jr. The Miyamoto/Tezuka/Nakago/Kondo team would collaborate for many years on a number of Nintendo’s most famous Famicom titles. In fact, during SMB’s development, they were working simultaneously on The Legend of Zelda for the Famicom Disk System.

Genre firsts are notoriously difficult to pin down. SMB was not the first platformer, nor even the first from Nintendo. Elements of the genre existed prior to Donkey Kong, but that game’s popularity and polish solidified the archetype even before the genre had a stable name. The ‘platform’ (or alternately ‘platformer’) label describes the various obstacles that the player’s character must run across, vault, or climb as they make their way to a specified goal. In Donkey Kong, the platforms are girders that Mario must traverse to reach the top of the stage. Though Donkey Kong ostensibly took place at a construction site, the limitations of the arcade hardware dictated that the girders hung inexplicably in space, bound only to the ladders that connected them. In Super Mario Bros., the platforms are arrangements of various bricks and blocks that comprise the architecture of the world. Here, the fantasy setting precluded any need to explain why these blocks floated in mid-air. They were simply there for Mario, Luigi, and their foes to clamber across.

The original arcade Mario Bros. was one of the first Nintendo videogames to offer a theme and variation on Donkey Kong’s seminal mechanics—a shorthand gaming term loosely describing both the range of available character actions and their phenomenological feel. The object was no longer to reach the top quickly, but to defeat the myriad turtles, crabs, flies, and icicles spilling from enormous green sewer pipes. The critters emerged from the top of the screen, dropped down to platforms erected at various levels below, then crawled into pipes at either side of the screen to repeat the cycle. Mario (and brother Luigi) could eliminate their foes only by bumping the platform from below. This would flip the enemy and leave them helpless for a short amount of time. Mario then had to jump on the platform and touch the paralyzed foe to boot them from the screen. Mario’s range of motion was not significantly enhanced from Donkey Kong, but the transformation of jumping from a simple evasive tactic to an offensive strategy would have a huge impact on SMB’s gameplay. Mario Bros. also introduced a number of franchise mainstays: the subterranean setting, Mario’s job change from carpenter to plumber, green pipes, bricks, Mario’s sibling Luigi, green and red turtles, collectible coins, Mario’s skidding stops, the POW block, and even fireballs (though they chased Mario, rather than serving as an offensive tool).
SMB inherited the platformer conventions of Donkey Kong, Mario Bros., and other early Nintendo games and exploded them beyond the borders of the single screen. Again, SMB was not the first game to do so. On the arcade front, games like Defender (1981), Xevious (1982), and Moon Patrol (1982) all featured smooth scrolling worlds of both the horizontal and vertical varieties. In Moon Patrol, your agile lunar craft could even jump obstacles. Of course, these games were all firmly embedded in the sci-fi genre popular at the time so their looped or infinite scrolling often reflected the vastness of outer space. Rock-Ola’s little-known Jump Bug (1981), featuring a platforming red Beetle, and Namco’s Pac-Land (1984) were also early predecessors of SMB that combined scrolling with running and jumping.

The Atari VCS had its share of scrolling games, but few that could rightfully be called platformers. Pitfall! (1982) is the closest to form, featuring both a world that extended to the left and right of the visible screen and a memorable character, Pitfall Harry, who could leap barrels, swing on vines, and shuffle across alligator heads. However, instead of scrolling, Pitfall’s screens came one at a time, triggered by Harry crossing the screen’s border on either side. Mountain King (1983), while visually rudimentary, copped the familiar ‘levels-and-ladders’ gameplay style of Donkey Kong, while adding continuous four-way scrolling—an impressive programming feat for the VCS.
With so many able predecessors, why and how did SMB succeed so tremendously? What is it about the game that entertained millions of videogame players? There were no apparent technological innovations that would wow players seeking the bleeding edge. The Famicom was over two years old when SMB debuted. By 1985, superior competitor consoles were already emerging to challenge Nintendo’s machine. Even Nintendo’s programmers were facing the limits of their own hardware. The limited PRG-ROM and CHR-ROM of the stock NROM cartridge could only be stretched so far. Nintendo was already developing the Family Computer Disk System add-on to replace carts with disks and expand capacities beyond 40KB. SMB certainly had some brand familiarity with Mario, but he was hardly the international mascot he is now. Miyamoto originally envisioned Jumpman as the all-purpose ‘Mr. Video’ who could appear, like Mr. Game & Watch, in multiple games. But Mario had not yet had his standout debut. What, then, made Super Mario Bros. so popular?

Appeal is difficult to pin down, but I propose the following key traits: setting,
coherence, efficiency, physics, concealment, design, and sonics. The Mushroom Kingdom, SMB’s psychedelic setting, is a vivid, surrealist world composed of a strange melange of Eastern and Western influences. The striking blue skies alone were a marked departure from the black voids normally seen in videogames. Furthermore, Mario’s strange world made logical sense despite their fantasy setting—pipes took you underground or underwater, vines led to clouds, and large castles were as imposing inside as they appeared outside. Meanwhile, the means provided to the player to explore this world—Mario and Luigi—had a fantastic feel. They could walk and run, with realistic acceleration, affecting the distance and trajectory of their jumps. Hard stops were supplanted by realistic skidding turns. The all-or-nothing static arc of Mario’s jump in Donkey Kong and Mario Bros. was replaced with flexible ‘air control,’ the ability to coax the character’s movements after pressing the jump button. Thus players could weave Mario’s body through narrow passages, around blocks, and under enemies—all in mid-air. And these same skills were required to uncover the game’s secrets. The various pipes, vines, and blocks were enticements to explore, symbolized famously by the floating block stamped with a question mark. Most bricks were empty, but some contained hidden coins, 1-UPs, or power-ups. Mario shrank, grew, and gained fantastic powers like a modern-day fairy tale character, albeit one with overalls and a mustache. Most pipes could not be entered, while some led to underground passages or level-skipping warp zones. However, the Mushroom Kingdom’s arrangement was not arbitrary. Miyamoto’s team worked hard to lead the player through their novel world, teaching them skills via play, absent any in-game text or tutorial. SMB’s was a world meant for extended sessions and repeated play—a true console game. And finally, Kondo’s upbeat, approachable score drew the whole package together, and has arguably transcended the game to reach its own iconic status.

In combination, these traits cohere into one of those most beloved and influential videogames of all time. And each of these elements, in turn, was sculpted by the constraints of the Famicom. The level structure, the blocks, the colors, the feel of the jumps, the music, were all tied intimately to the limitations of the 2A03, the 2C02, the NROM, the Famicom controller, and the CRT television. The platform dictated the platformer. At every turn, Miyamoto’s team was faced with hard decisions on what to cut and what to include. SMB was the swan song of the modest NROM, the upper limit of what could be contained in 40KB of ROM. With that in mind, let’s look closely at the strictures that governed SMB’s design and the creative means Miyamoto’s team devised to push against those boundaries.

**Kinoko Kingdom**

Super Mario Bros.’s scenario is dead simple, albeit bizarre. A malevolent turtle king name Bowser has captured the Princess Toadstool and secreted her into a castle. Meanwhile the
peaceful residents of the Mushroom Kingdom (キノコ王国) have fallen under Bowser’s sorcery and have subsequently been transformed into the world’s ‘rocks, bricks, and horsetail plants.’ It is the Mario Brothers’ task to rescue the Princess from Bowser and restore peace to the Kingdom.

The damsel in distress trope has countless precedents in literature, film, and videogames, but the locale and residents of the Mushroom Kingdom are, like Donkey Kong, a unique amalgam of Eastern and Western influences. In fact, SMB is far weirder than its arcade forebear and wears its Japanese roots on its sleeve. Miyamoto steeped the game’s design in influences from Japanese manga, folklore, geography, and even food, meanwhile infusing the brew with anachronistic Western touches. Yet despite its hybrid origins, SMB required no localization for its in-game content. The minor bits of text and dialogue were already in English in the Famicom original. The real translation work went into the game’s supporting paratexts: the manual, box, and artwork.

This is particularly evident in the original Japanese names of Mario’s enemies. Bowser, for instance, is an English-exclusive name. In Japanese, he is called 大魔王クッパ, akin to ‘Great Demon King Koopa,’ a name that clearly would not have satisfied NOA’s strict guidelines on religious imagery (see chapter 3), but does allude to King Koopa’s resemblance to the demons of Japanese art history, both in ancient scrolls and mid-century anime. Miyamoto, who drew the original box art for the game, modeled Koopa’s design after the Ox King from the 1960 Toei animated feature Alakazam the Great (originally 西遊記). Thus the box art shows Koopa in a form far more bovine than reptilian, bearing little resemblance to his in-game sprite—a discrepancy later pointed out by Tezuka and amended by, ironically, former Toei artist Yoichi Kotabe.

Additionally, the クッパ (kuppa) portion of Bowser’s Japanese name contains a cultural reference that would be lost to Western audiences, since it is the Japanese name for a Korean soup dish. Miyamoto chose this name over two other possible Korean dishes: ユッケ (yukke) and ビビンバ (bibinba). Similarly, Princess Toadstool was originally Princess Peach (ピーチ姫) and the small brown Kuribo (クリボー), or ‘chestnut people’ were inspired by shiitake mushrooms. In English, they translated to the questionable Goomba, a term that might better describe Mario and Luigi than ambulatory fungi.

Several enemies are named after the Japanese onomatopoeia that describe their behavior: the ‘Pakkun Flowers’ (パックンフラワー) are named after the paku sound made when eating (also the original derivation of Pac-Man’s name); the fish are called ‘Puku-Puku’ (プクプク), a sound that denotes swelling up; and the flying turtles are called ‘Pata-Pata’ (パタパタ) to describe their flapping wings. Other names are culled directly from
Japanese folk tales: ‘Jugemu’ (ジュゲム), the cloud-bound turtle who rains spiny eggs on Mario in World 4-1, is borrowed from a Japanese rakugo (a form of theatrical monologue) wherein two parents cannot decide on the name of their child. When advised by a priest on several possible names, the father remains indecisive and thus chooses all of them. The comically long string of names begins with Jugemu. Likewise, the name of the eggs that Jugemu throws, ‘Paipo’ (パイポ), is borrowed from the same tale.

The Mushroom Kingdom’s cloud motif derived from the same source material as Alakazam the Great, namely Journey to the West, a 16th century Chinese novel. Journey’s primary protagonist, Sun Wukong (the Monkey King), possesses numerous magical powers, including the ability to jump across and ride upon clouds, transform into other living creatures or inanimate objects, and alter his size. Considering Miyamoto’s familiarity with Alakazam and the Monkey King tale, it is no surprise that Wukong’s fantastic feats found their way into Mario’s repertoire.

The unlikely flora, fauna, and even the name of the Mushroom Kingdom drew directly from the creators’ cultural landscapes. As Miyamoto explained:

Since the game’s set in a magical kingdom, I made the required power-up item a mushroom because you see people in folk tales wandering into forests and eating mushrooms all the time. That, in turn, led to us calling the in-game world the 'Mushroom Kingdom,’ and the rest of the basic plot setup sprung from there.

Tezuka contributed the concept of enormous sprouting vines, plucked from the Jack and the Beanstalk fairy tale, that led to landscapes of overgrown mushroom caps and traversable clouds. The enormous green pipes first seen in Mario Bros. were inspired by the ubiquitous ‘waste ground with pipes’ found in manga. Miyamoto knew such pipes would sensibly transport characters through space. The curved hills punctuating the backdrop of the Mushroom Kingdom were callbacks to the hills of rural Sonobe, where Miyamoto grew up. And thanks to the concurrent development of SMB and The Legend of Zelda, a few objects, like the spinning fire bars, were plucked from the latter for use in the former—another example of Nintendo adapting and repurposing its own cultural production.

Amidst this menagerie of noisy foes, ambling food, and fairy tale allusions, we once again find Mario and Luigi, a pair of Italian plumbers. Clad in caps, overalls, and mustaches, they make for unlikely foils to the psychedelic fantasy world of the Mushroom Kingdom. It is hard to imagine middle-aged plumbers starring in any contemporary big budget video game, save Nintendo’s own. They would likely suffer the judgment of online forums and internal focus testing. Not that Nintendo didn’t have the market in mind; according to Tezuka, the game was not designed from the outset to include the anachronistic heroes.
However, after he discovered continuing strong sales of the Famicom’s *Mario Bros.* port a year after its release, he and Miyamoto decided that the Mario Brothers should be the stars of their new game.²⁴

Clearly, as we saw with *Donkey Kong*, the exaggerated cartoon figures of manga and animation suited the limited pixel palettes of 8-bit platforms like the Family Computer. Realism was never the Famicom’s strong suit; Princess Toadstool’s mangled sprite is unfortunate proof. And offering a grab bag of media traditions, both Eastern and Western, certainly lent force to *SMB*’s universal appeal. It had a touch of familiarity that reached children and adults across the world, despite its surrealistic tenor. As Miyamoto said:

> When we make our games we try to make things that are not focused on one market or one particular culture or one particular people and where there is [sic] some difficulties in that I really think it does free us up in a different way to just make what we want and hope that universal appeal will branch across all cultures.²⁵

Tens of millions of copies of *Super Mario Bros.* later, it is clear that Miyamoto, Nakago, Kondo, and Tezuka accomplished their goal.

Of course, the look and profession of the characters or the narrative cohesion of the story are all mere set dressing for the game’s core conceits: running, jumping, stomping, swimming, bumping, bashing, skidding, kicking, and climbing. It made the most sense to import the characters’ physical abilities above all other attributes. Abstracted away from the Mushroom Kingdom, these activities defined the model for a genre that would sustain the Famicom, its hardware successors, and a duo of athletic plumbers, for decades. And at the heart of the platformer was a platform custom engineered to render colorful worlds of scrolling blocks.

**A World of Attributes**

How do you describe a world in forty kilobytes?

In chapter 1, I explained that a single name table of Famicom graphics measures thirty-two tiles wide by thirty tiles high, assuming each tile is 8x8 pixels. Storing screen data as a sequential list of tile IDs, each requiring one byte, would deplete 960 bytes of ROM space simply describing a *single screen*. Forty-one screens would max out ROM capacity with level data alone, leaving no room for graphics, sound, or even code to drive the game engine.

Obviously, *SMB* has far more than forty screens. In fact, it has eight worlds containing four areas each—a total of thirty-two individual levels—each of which comprises several full name tables stitched together along a horizontal plane. Within those levels, there are myriad terrain types, ranging from cracked stonework to smiling billows. Mario travels
overground, underground, among clouds, and underwater. He climbs vines, descends pipes, skirts dangerous whirlpools, vaults suspended girders, and navigates lava-filled mazes. So how is it possible to cram so much data into a lowly NROM?

The answer is twofold: metatiles and compression. As we have seen, a metatile is a grouping of two or more tiles treated as a single object. These can be moving sprites or static backgrounds: Mario is a metatile, as are Koopa Troopas, as are coin blocks, as are clouds, as are pipes. All Famicom games use metatiles of one kind or another. Limiting objects to an 8x8 pixel area would impose debilitating restrictions on graphical complexity.

Metatiles have clear computational advantages for data storage and manipulation. First, grouping multiple tiles into a single object simplifies tile updates. Moving a single Mario object is simpler than updating four individual tile coordinates. The individual tiles of a metatile do not congeal automatically, of course, but well-planned game engines are designed to genericize object handling. In other words, there are not individual subroutines that handle Mario’s movements, Bowser’s movements, Bullet Bill’s movements, Koopa’s movements, and so on. Such an approach would quickly bloat the source code beyond a manageable scale. Instead, a generic ‘player object handler’ can perform the backend calculations necessary to move Mario and Luigi metatiles, while the ‘enemy object handler’ can perform the necessary functions for their foes.

Second, breaking the game world into larger repeatable chunks and spooling them out in varied combinations permits larger and more diverse levels. With enough planning and variation, the player will not notice the underlying patterns. In fact, the repetition of a small handful of building blocks can contribute to the overall style and coherence of the game environment. SMB, for instance, employs simple palette swaps to great effect to introduce variation in otherwise identical bricks and tiles.

In a scrolling engine such as SMB’s, compression takes on additional importance. The Famicom does not have sufficient video RAM to store more than two name tables worth of graphics. Prior to the release of Excitebike, Balloon Fight, Spartan X, and similar games, few Famicom titles required more than two name tables of tiles; even fewer scrolled. This was partly due to the Famicom’s reliance on arcade ports, whose scenarios were largely confined to a single screen. Once scrolling opened up, programmers had to devise efficient methods to update name tables on the fly—offscreen—so the player could never detect discontinuities. With sufficiently lengthy levels, updating uncompressed name table tiles becomes both processor intensive and costly to memory. Metatile compression serves to alleviate both problems by grouping associated tiles into larger data structures (i.e., metatiles) that may be stored and unpacked efficiently. In other words, as the screen scrolls, the game engine decodes metatiles on the fly just beyond the perimeter of the screen. The world is delivered in chunks of data that fit within the range of available VRAM.

Metatile compression works similar to single-tile compression. With RLE, we might
use a three-byte string—$03, $21, $FF—to tell the game engine: take background tile $03, repeat it 15 times ($21), then terminate the command with $FF. An RLE metatile compression scheme could use the same string, where $03 would now denote metatile $03 rather than tile $03.²⁶ That byte could then be passed to a metatile-rendering subroutine that interprets a metatile as a string of four tiles. It could then reference metatile $03 in a lookup table that lists its individual tile bytes. This compression pattern can extend further ‘upward,’ packing metatiles into larger ‘metametatiles’ and beyond. Again, $03 might describe an object composed of four metatiles, each of which comprises four individual tiles. Nesting the structure of tiles in such a manner greatly decreases the amount of data necessary to build a multi-screen world. Data is packed into a Russian doll series of objects housing objects housing objects, strung into a chain of descriptive bytes.

The Famicom has an important PPU limitation that makes the metatile structure advantageous. In chapter 1, I introduced attribute tables, the 64-byte areas immediately following each of the PPU’s four name tables (two of which are mirrored). To recap, the background tiles used to build the name tables may choose from one of four palettes. Each palette contains four individual colors—though all four palettes must sacrifice one of their color slots to the dominant background color. The values in the pattern table bitplanes control which of the four possible colors within a palette the tile may have. However, the attribute tables control which of the four palettes those colors are defined by. The catch is that each background tile is not permitted to choose its own palette. Instead, the 960 tiles comprising a name table are divided into 32x32 pixel chunks—4x4 tile squares—and assigned a byte from the attribute table. A chunk is then further subdivided into four 2x2 tile squares. Each sub-chunk then receives two bits from the attribute byte, meaning that any 16x16 pixel section of background tiles is constrained to a single palette.

SMB’s entire world is carefully structured around attribute table limitations so no unsightly clashes crop up. The smallest background objects—bricks, terrain, ? blocks, coins—are all one 16x16 pixel metatile in size. The Mushroom Kingdom abounds with right angles, best exhibited by the massive stair-step structures that conclude each non-castle level. And these shapes were perfectly suited to the platformer genre. Various heights and widths of rectangular blocks make excellent obstacles for running, jumping, and climbing. Combined with the Famicom’s baked-in scrolling registers, there were near limitless possibilities for a genre that would subsequently explode following SMB’s debut.

**Engine Economy**

The game engine is a videogame’s code core. It functions like a software CPU, coordinating the processes that drive the game—preparing which graphics to render onscreen, handling physics, defining objects and their collisions, switching major game states. Today’s
middleware engines, like Epic’s Unreal or Crytek’s CryEngine, are designed to abstract these functions into modular APIs that can be adapted to myriad videogame genres, from first-person shooters to real-time strategy. The engines that power today’s games are immensely complex. Building a custom engine from scratch demands massive investments in time and resources. However, licensing a capable middleware solution speeds up the videogame development process. Reusing and adapting an existing engine, especially for a familiar game type, is often better than recoding the wheel.

Famicom game engines were far less flexible than today’s middleware engines. Space limitations meant that engines were catered to the needs of a single game or a series of similar games. Adapting the turn-based roleplaying engine of *Final Fantasy* to a platformer like *Mega Man* would be, at best, insane—but likely impossible. Nonetheless, game engines of the Famicom era still strove toward abstraction and modularity. Like the metatiles described above, efficient engines were designed to group objects and processes into like types in order to avoid redundancy. It is far better to design a physics routine that handles multiple objects than to apply the same basic rules individually to fifteen similar enemies. In short, the engine should not care if the object is a Koopa or a Goomba—they both fall into a pit the same way. *SMB*’s engine is designed with an elegance and sophistication that belies its modest ROM footprint. Nagako and team used a multi-layered, back-to-front approach that parses objects in surprising ways, contrary to how both player and processor perceive the final visual results.

*SMB*’s engine constructs each level according to three related processes, each handling its own range of objects. The first of these, what I will call ‘set decoration,’ is a semi-automatic algorithm that unrolls the level based on a set of initial conditions. The second is the area object data, primarily the platforms—bricks, pipes, power-up blocks—that Mario bashes, vaults, and bumps. The third is the enemy object data, governing the number and position of enemies blocking Mario’s path.

Keep in mind that each of these processes is largely handling background tiles (enemy sprites are a clear exception), meaning that any ‘layers’ we speak of are merely programmer-constructed abstractions. The Famicom’s PPU can only ever handle a single layer of background tiles. Sprites may be above and behind background tiles based upon their priority, but even this aspect of depth is virtual. Placing a sprite in front of another tile simply means that the PPU renders that object’s pixels rather than another’s. A television only has one plane of pixels; there are no pixels hidden behind other pixels.

Also keep in mind that the term object is not used in its conventional programming sense. Objects in modern object-oriented programming (OOP) languages like Objective-C or Java are data structures consisting of *properties* with associated *procedures* (i.e., methods) that can manipulate those properties. Individual objects are actually instances of a common *class*, an abstracted construct used to prototype all objects of a given type. To use
a common programming analogy, we could define a Car class that includes some basic properties, like wheels, color, and body type. Until we create a particular instance of that class, we are not describing any car in particular, merely an abstract model. Likewise, we might define a number of methods to query, update, or manipulate our class properties. For instance, a PaintJob method could change the color of our car, the FlatTire method could subtract a wheel, and so on. One of OOP’s key benefits is modularity. It is far easier to instance thousands of different cars for a traffic simulator, for instance, than to create new data constructs for each individual car.

6502 assembly language is not object-oriented. It contains no classes or pre-built data structures. Data is moved to and from registers where it can be added, subtracted, compared, or bitwise operated. Assembly source executes from top to bottom until an interrupt fires. Once the interrupt is handled, it returns to the sequence and continues. The code’s only ‘modularity’ derives from simple jump commands and branches that can detour the program counter to alternate memory locations. As such, I use the term object loosely to describe groups of tiles handled as a single entity, or, in rarer cases, a behavior or event that triggers at a specific location onscreen. With that in mind, let’s look at each of the engine’s processes in turn.

The set decoration is itself divided into three successive layers, what doppelganger’s Super Mario Bros. disassembly labels ‘background scenery,’ ‘foreground scenery,’ and ‘terrain.’ Background scenery is the dominant backdrop of each level, comprised of the weather, flora, and architecture of the Mushroom Kingdom. These are the hills, clouds, bushes, trees, and fences that give the Kingdom its distinctive character and break up the monotony of the saturated blue and black skies. There are sixteen individual 1x3 metatile (16x48 pixel) background blocks, or metametatiles used to construct three different forty-eight metatile-wide background patterns.

The first of these the player encounters is the hills, clouds, and bushes motif seen in 1-1, 4-1, 6-1, and 6-2 (Fig. X). Next is the clouds motif, used as the backdrop for any stage that appears to be suspended in midair: 1-3, 2-3, 3-3, 4-3, 5-3, 6-3, and 7-3. The final motif of clouds, trees, and fences appears in 2-1, 3-1, 3-2, 5-1, 5-2, 7-1, 8-1, 8-2, and 8-3. Any remaining levels have either solid black (underground, castles, nighttime cloud bonus level, coin rooms) or solid blue (daytime cloud bonus area, underwater) backgrounds with no background scenery.

Using either the Game Genie hardware peripheral or an emulator that supports cheats, it is possible to view uninterrupted patterns of background scenery. The ‘Moon Gravity’ Game Genie code allows Mario’s jump to reach extraordinary heights, even exceeding the tops of level-ending flagpoles. While such a stunt is possible unassisted, it is difficult to pull off and those who have are often disappointed to find that there is no hidden world or warp zone, but endlessly scrolling terrain. Even the Game Genie manuals
warns, 'If you’re playing to complete the game rather than just explore it, don’t jump over the flagpoles—or else you'll get "stuck" and have to reset.'

Getting ‘stuck’ is not a glitch or error—it is simply the engine performing its job as intended. The background scenery rendering is automatic in the sense that it does not rely on a predetermined level length. The engine simply unspools the specified forty-eight metatile pattern until a flagpole triggers a level’s conclusion. When the flag is skipped and the player proceeds right, the background scenery repeats happily until the timer winds down. Absent any enemies or obstacles, the flagpole jumping trick lays bare the underlying process and pattern. Figure 4.2 shows the unimpeded stretch of terrain beyond the flagpole. This 48-metatile rhythm repeats across an endless scroll.

4.2 The hills, clouds, and bushes motif, one of three forty-eight metatile-wide ‘background scenery’ patterns rendered by the Super Mario Bros. engine. If Mario jumps the flagpole, the same sequence will repeat indefinitely. The pattern above is used in levels 1-1, 4-1, 6-1, and 6-2. (Emulator: FCEUX 2.1.5, Photo editing: Nate Ayers)

Also note that each metametatile component of background scenery is ‘vertically exclusive.’ In other words, each hill, bush, cloud, fence, or tree never occupy the same vertical slice of the screen. In the screen above, you can see that a cloud never hovers directly above a hill or bush.

Foreground scenery is rarer and includes three repeated tile columns: the dark blue water that fills 2-2, 7-2, and a portion of 8-4; the lower water/lava columns used in castle pits or below bridges (e.g. 3-1); and the high brick walls seen in 8-3. Its appearance is largely driven by the area object list header, as we will see below.

The final set decoration, terrain, describes long stretches of horizontal tiles. As its name implies, the most common use of terrain is for the ground, but it also describes stretches of tiles along the ceiling and, in some cases, strips through the middle of the screen as well. (The term terrain may seem inappropriate to use for the ceiling tiles, but Mario is an agile plumber and often finds himself using the ceiling as ground too.) Terrain
rendering comes in sixteen combinations of top, middle, and bottom thickness, depending on the level’s needs, ranging from no terrain to a screen-filling solid. Terrain unspools at the specified height until a special ‘hole object’ overwrites it or an ‘area change object’ alters its combination.

Though the three parts of the set decoration are all tile members of the single name table plane, the game engine assigns each layer a different rendering priority, much like the PPU handles sprites in the OAM queue. The terrain sits on top, the foreground scenery sits below it, and the background scenery takes backseat. Again, there are no hidden foreground tiles lurking behind the terrain—those tiles with highest priority are simply the ones that get their pattern table pixels fetched. The routine that handles the set decoration starts with the background scenery first, then checks to make sure no foreground scenery needs to be rendered, and finally checks if there is any terrain to render. So, for instance, if there are no high walls or ground to be drawn, a rounded green hill can emerge. A quick visual check of level 8-3 confirms this process: the high wall obscures the lower clouds, trees, and fences, which in turn are obscured by Mario’s two metatile-high runway of cracked stone.

Perceptive players may notice that the terrain is the only portion of the set decoration that Mario and his enemies can collide with. The clouds, hills, trees, and walls are merely ornamental, like a beautiful tapestry in front of which sprites run and jump. In short, they are the background’s background. This is hard to conceptualize along a 2D plane, but contemporary artists and game designers have provided a glimpse of how the Mushroom Kingdom might look when tilted into perspective. Morgan O’Brien’s Super Mario Bros. 2.5-D (2006) is a PC game that adds depth to the familiar scenery and characters of Super Mario Bros. (figure 4.3)
Keyboard commands shift the camera’s perspective on the Mushroom Kingdom as if it were pressed into the shallow glass frame of an ant farm. Though its background priorities are different than SMB’s actual engine, Super Mario Bros. 2.5-D adopts the same layered perspective, allowing certain elements to advance and recede in depth. Traversable elements like blocks, pipes, and terrain gain volume as well, stretching along the z-axis. Sprites, clouds, and bushes, however, remain paper thin. Nintendo’s own Paper Mario (2001) series adopted a similar trope. The 2007 Wii sequel Super Paper Mario was ostensibly a 2D platformer in the vein of SMB. However, the player could rotate the perspective along a vertical axis, permitting them to see hidden routes, retrieve power-ups, and pass obstacles that appeared insurmountable in 2D.
If the set decoration comprises the 'background background,' what constitutes the 'foreground background'? There are two remaining layers: area objects and enemy objects. These layers constitute the 'real' substance of the Mushroom Kingdom: the stompable Buzzy Beetles, the unpredictable jumping springs, the bumpable, bashable bricks. And equally important, they also constitute what players would call the level design, the intentional arrangement of objects meant to teach, guide, and challenge the player. In short, the blue skies and verdant hills may provide 1-1 with a cheerful backdrop, but it is the specific sequence of blocks, pits, Koopas, and pipes that have made the level iconic.

A large percentage of SMB's PRG-ROM is dedicated to area object data, which encodes each specific level sequence into compact byte blocks. Level 1-1, for example, is stored as follows:

```
; level 1-1
L_GroundArea6:
    .db $50, $21
    .db $07, $81, $47, $24, $57, $00, $63, $01, $77, $01
    .db $c9, $71, $68, $f2, $e7, $73, $97, $fb, $06, $83
    .db $5c, $01, $d7, $22, $e7, $00, $03, $a7, $6c, $02
    .db $b3, $22, $e3, $01, $e7, $07, $47, $a0, $57, $06
    .db $a7, $01, $d3, $00, $d7, $01, $07, $81, $67, $20
    .db $93, $22, $03, $a3, $1c, $61, $17, $21, $6f, $33
    .db $c7, $63, $d8, $62, $e9, $61, $fa, $60, $4f, $b3
    .db $87, $63, $9c, $01, $b7, $63, $c8, $62, $d9, $61
    .db $ea, $60, $39, $f1, $87, $21, $a7, $01, $b7, $20
    .db $39, $f1, $5f, $38, $6d, $c1, $af, $26
    .db $fd
```

This 101-byte string describes the level’s sequential placement of objects from the first coin block to the final castle facade. The byte sequence must adhere to a strict format so that the engine properly funnels objects to their appropriate rendering routines. In general, the area object data describes metatetiles, larger chunks of metatiles that draw rows of bricks or an entire warp pipe. However, as we will see below, it can also handle a few single-metatile objects and a set of special objects that function as level attributes rather than groups of tiles.

Each of SMB's non-repeating levels, underground coin rooms, warp zones, transition screens, and bonus areas share the same format. Their area object data ranges in size from the 9-byte interstitial screen used to transition Mario to underground/underwater levels, to the massive 163-byte level 1-2. The subterranean level 1-2 contains one of the
densest arrangements of breakable blocks, rivaled only by the 161-byte level 4-2, also set underground. Unsurprisingly, these are the only two underground levels in the game.

The basic structure of area object data is simple: the first two bytes encode the header, the final byte (always $fd$) is a string terminator that signals the end of the data, and all the bytes in-between are grouped into pairs—the first byte is a metatile screen coordinate and the second byte chooses the object and its attributes. The complexity of the format lies in its compression. Nearly every byte in the block carries multiple flags, values, and controls within its bits.

The header defines the initial conditions of the level ahead. Two bits of the first byte, for instance, set the timer for 400, 300, or 200 ticks. Another two bits set Mario’s and Luigi’s initial vertical position. On most levels, they begin on the ground, but in levels following the interstitial animation that shows them entering a pipe, they drop from the top of the screen. Miyamoto thought it was important that the characters’ traversal of the world made logical sense—inasmuch as plumbers traveling through enormous green pipes could be logical:

I thought it was strange how Mario was already standing there underground when that level begins. Why is Mario, who just passed in front of a castle, standing underground? I couldn’t fit in a sequence showing him falling underground, so I decided to have him just plop down from the top of the screen, and—surprisingly—that was just fine.\textsuperscript{33}

The second byte controls values related to the set decoration, including the terrain height combinations, background and background scenery colors (e.g. nighttime with green hills), and what doppelganger calls the ‘area style,’ which can affect both the terrain tile type or its palette. Figure 4.4 illustrates how minor changes to the header byte can drastically alter the appearance of 1-1.
4.4 Hex editing the default level 1-1 header byte pair ($50,$21) to alternate values ($54,$F1) affects the background color (black), background scenery (trees and fences), terrain style (clouds), and terrain height (three-metatile ceiling). (Editor: Hex Fiend 2.1.2, Emulator: Macifom 0.16)

The real meat of the data lies in the chain of byte pairs sandwiched between the header and the terminating byte. Again, the first byte of each pair denotes the object’s position. Each half of the byte, commonly called a nybble, is assigned to the screen’s x- and y- coordinates. However, these positions are not the pixel-based Cartesian coordinates used to increment the scroll register or position sprites onscreen. Instead, SMB’s engine assigns a special grid of metatile rows and columns based on name table-sized ‘pages.’ Each page is divided into sixteen metatile-wide (16 pixels) columns, numbered $0$ to $F$, and
twelve metatile-wide rows, numbered $0$ to $B$ (see figure 4.5).

4.5 A single ‘page’ of level 1-1’s area object data. Column numbers are marked in grey along the top edge. Row numbers are marked in green along the left edge. The six byte-encoded objects are outlined red. The two ? blocks are outlined in lighter orange to show that they are drawn ‘on top’ of their brick row. Note that the author’s television composite output renders the full video field to the capture card. On the actual television, several pixels of the upper and lower edge are cropped in overscan. (Source: NES-101 capture)

Note that a full sixteen columns span the entire width of the page, while there are only twelve labeled rows. As we will see in ‘Exergue’ below, the upper four rows of tiles are reserved for the non-scrolling segment of the screen. While Mario and a few other sprites may occupy this segment of the screen, area objects are never placed above the invisible line dividing the status bar from the upper tip of the cloud. Row $B$ marks the threshold between the object area and terrain. Any coordinate bytes beyond row $B$ have a special function that do not require a vertical position.
To illustrate how the decoding process works, let's take a look at the first seven byte pairs after the header from the level 1-1 area data listed above:

<table>
<thead>
<tr>
<th>BYTE PAIR</th>
<th>COL,ROW OBJ BINARY</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>$07, $81</td>
<td>0,7</td>
<td>10000001 {P} [sm.] ? block (coin)</td>
</tr>
<tr>
<td>$47, $24</td>
<td>4,7</td>
<td>00100100 [lg.] row of bricks (length 4)</td>
</tr>
<tr>
<td>$57, $00</td>
<td>5,7</td>
<td>00000000 [sm.] ? block (power-up)</td>
</tr>
<tr>
<td>$63, $01</td>
<td>6,3</td>
<td>00000001 [sm.] ? block (coin)</td>
</tr>
<tr>
<td>$77, $01</td>
<td>7,7</td>
<td>00000001 [sm.] ? block (coin)</td>
</tr>
<tr>
<td>$C9, $71</td>
<td>C,9</td>
<td>01110001 [lg.] vertical pipe (height 2)</td>
</tr>
<tr>
<td>$68, $f2</td>
<td>6,8</td>
<td>11110010 {P} [lg.] vert. pipe (height 3)</td>
</tr>
</tbody>
</table>

I've expanded each byte pair into a tabular format to better clarify the data encoding. The first column is the original byte pair. Column two divides the upper and lower nybbles into a comma-delimited coordinate pair—column value first, then row. The third column expands the second byte into its binary value, which is necessary to parse the compressed data format. The final column is a more verbose description of the area object, notated in my custom shorthand.

The first byte pair above designates an object at metatile column $0$, row $7$. Using the gridded matrix seen in figure 4.5, it is straightforward to match object data with their respective onscreen objects. We see that the first question block lands directly in the spot we expected. Of course, without the reference image, we would not know which object occupies that position. To find the object, we must look at the second byte which, at this point, is an inscrutable series of zeros and ones.

Each object byte breaks down into the following scheme:

```
PssssMMMM
| |||||        “Modifier” bits contingent upon value of “selection” bits. |
| ||||+++------ Designates “small” obj. type or “large” obj. attributes |
| | |||| (i.e., obj. height/width or whether a pipe is usable). |
| | ||||
| | +++----- 0: If d4-d6 set to 0, d3-d0 select “small” objects |
| | with no height or width (e.g., ? blocks). |
| | 1-7: Use value to select “large object” type. |
| +-------- “Page select” flag |
```

To readers unaccustomed to this notation, I have grouped related bits by letter labels that designate their function. Each digit place is numbered from 0 to 7 in the little endian
format, meaning that digit 7 (d7) is on the far left and digit 0 (d0) is on the far right.

Notice in figure 4.5 that while the first object’s coordinate position is column $0$, row $7$, Mario is not currently on level 1-1’s opening screen. When the game begins on level 1-1, Mario is positioned near the left edge with an entire screen’s worth of ‘runway’ to the right. Yet there is no area object data until the first [?] block. This is thanks to SMB’s set decoration process that chugs away unimpeded until it is issued a command to alter its course. Notice that d7 of the first area byte is binary 1. This position is a flag that indicates a ‘page select.’ The flag notifies the engine that we are now moving forward to the next sequential page and thus should update all objects (and enemies) accordingly.

This clever bit of engine organization allowed SMB’s programmers to store object coordinates in a single byte, making them page-relative rather than area-relative. In other words, if the engine saw the entirety of level 1-1 as a single chunk, its column coordinates would extend far beyond the sixteen metatiles of a single screen. The horizontal coordinate would then exceed the capacity of a single nybble and require an additional byte to encode. Multiply that single byte addition by every y-coordinate of every object in the area data and SMB’s ROM size suddenly spirals beyond the limitations of the 32KB PRG-ROM. Such an expansion would have resulted in cuts to the number and diversity of levels. Dividing each level into pages compressed its data significantly without much additional overhead.

Likewise, limiting Mario’s movement to a single horizontal vector made paging that much simpler; pages would only ever need to increment in sequence.

Anytime d7 is set, a new page is turned (in the description, I denote this with the {P} label). Since our first object has its page select flag set, the engine knows to auto-generate an entire name table’s worth of set decoration before the first object is rendered.

The remaining binary object digits tell the engine which object to render. The ‘selection’ bits (d4-d6) act as controls: if all three digits are zero, the engine selects a ‘small’ object based on the value stored in the ‘modifier’ bits (d0-d3); if any digit is non-zero, the three bits store the selection value of a ‘large’ object and the ‘modifier’ bits store that object’s attributes. Small objects are those that require no height or width value; they always have a fixed size. This includes all types of [?] blocks (power-up, coins, hidden coin blocks), bricks (power-up, vine, star, coins, 1-up), hidden 1-up blocks, underwater exit pipes, empty blocks, and the jumping board. Large objects are those that may vary in height or width, including vertical pipes (usable or not), rows or columns of bricks, rows or columns of solid blocks, rows of coins, and what doppelganger labels the AreaStyleObject (the grass platform ledges, mushroom cap ledges, and the Bullet Bill cannon).

In binary, the opening area object’s second byte is %10000001. Following the page select bit (d7), the three selection bits (d4-6) read %000, signaling the engine to regard the four modifier bits as a small object. Reading right to left, the modifier nybble %0001 equals object 1. In the SMB, the various objects are stored by type in lookup tables located directly
after the area rendering subroutine. The value stored in the modifier digits acts as an offset within each individual lookup table to choose the appropriate object. The small objects table is listed below:

; small objects (rows $00-$0b or 00-11, d6-d4 all clear)
    .dw QuestionBlock ; power-up
    .dw QuestionBlock ; coin
    .dw QuestionBlock ; hidden, coin
    .dw Hidden1UpBlock ; hidden, 1-up
    .dw BrickWithItem ; brick, power-up
    .dw BrickWithItem ; brick, vine
    .dw BrickWithItem ; brick, star
    .dw BrickWithCoins ; brick, coins
    .dw BrickWithItem ; brick, 1-up
    .dw WaterPipe
    .dw EmptyBlock
    .dw Jumpspring

Starting our count from zero, we see that object 1 in the lookup table is the QuestionBlock (coin)—exactly as expected. Properly unpacked, two bytes of data tell us which object will be rendered, what size that object will be, where it will be positioned on the current page, and on which page it is located. Quite economical for sixteen bits.

Let’s look at the next byte pair in the area object list. The first byte, $47, indicates that the object is located at column $4, row $7. The second byte, $24, is equivalent to binary value %00100100. D7 is not set, so the object will render on the current page. This time, however, the selection bits have the value %010, indicating both the type of object (large) and its offset in the large object lookup table (2). Consulting the source, we see that large object 2 is RowOfBricks. And since the object type is large, the modifier bits will designate the Row’s attributes. Attributes vary according to the item type. The modifier bits for the RowOfBricks object sensibly designate the width of the row. If it were the ColumnOfBricks, they would designate height; likewise for the VerticalPipe. However, VerticalPipe has one additional attribute—if d3 is set, it is usable, meaning that Mario can enter the pipe to descend to a bonus coin room. The modifier digits for the current RowOfBricks have the value %0100, or 4. In other words, counting from zero, the row will be five bricks wide. And as expected, comparing the data from the byte pair to figure 4.5 above, there is a brick row starting at column $4, row $7 extending right until it ends at column $8.

Continuing through the remaining byte pairs should be self-explanatory. The next five object bytes expand to binary values %00000000, %00000001, %0000001, %01110001,
and %11110010. The first four objects have d7 values of 0, so they too will remain on the current page. The first three have 0s in their selection bits, so they will be small objects. According to the modifier bits, the first of these will be object 0 ([?] block with power-up) and the second two will be, once again, object 1 ([?] block with coin). However, notice the placement: the first and third [?] block are both on row $7$, positioned at column $5$ and $7$ respectively—directly on top of the previous five-brick row. Since there are no background layers, the engine simply renders the most recent object in the queue. In other words, if our area data list included three objects with coordinates $2$, $6$, listed in the following order—empty block, coin block, star block—only the star block would be rendered. Once again, this is an efficient rendering technique. Drawing the five-block row with two interspersed item blocks only takes six bytes with the current ‘layering’ method. Drawing five individual blocks one after another would require ten bytes.

Following the three small [?] blocks, there is an object with its selection bits set to %111. In the source lookup table, that value corresponds to the large object VerticalPipe. Since the pipe is vertical its modifier bits control both the height and whether the pipe is usable. The value %0001 corresponds to a height of two metatiles (including the pipe’s lip), while the 0 in d3 denotes a non-usable pipe.

Notice that, despite accounting for all six area objects in figure 4.5, there is a still a final byte pair remaining. However, its expanded binary value %11110010 shows the page select set in d7, so the resulting object (a 3-metatiles high vertical pipe) should render on the following page. Sure enough, if one plays beyond the screen pictured, the next object Mario encounters will be a slightly taller vertical pipe.

Working through a few level objects helps us appreciate both the complexity and efficiency of Nakago’s data compression. The paging metaphor breaks each level into manageable chunks that relativize the metatile coordinate system, while the individual bits of the object byte form an interlocked system of controls and modifiers that can choose from a large selection of construction elements, each with its own distinct attributes. However, there is still a final area object type left to explore.

As noted above, when an object’s row value exceeds $B$, it is classified as a special object. These objects are special because they (a) do not require a specific row position and (b) require additional ‘backend’ processing in the game engine based on their function. Further into level 1-1, for instance, we find the byte pair $9c$, $01$. The first byte indicates the object’s page position is column $9$, row $C$. Row $C$ is not a screen coordinate, but a group of special objects (with corresponding lookup table) including empty holes, rope pulleys, holes with water, high and low [?] block rows, and bridges. Expanding the second byte to %00000001 indicates that the object will be an empty hole two metatiles in width. A hole appears to be a rather banal special object, but it performs two special functions: it signals the terrain tiles to stop rendering for a specified length and it permits player and
enemy objects to plummet and die.

*SMB’s* trademark staircases are another special object case. They are selected with object row $F$. Like the empty hole, they require a starting column and length. Once those initial conditions are set, a subroutine renders the staircase starting at one metatile high and incrementing one step up until the specified width is reached. The staircase leading to the flagpole in 1-1, for example, has a modifier nybble value of %1000, indicating that it has a length of nine metatiles (as always, count from 0). However, the staircase subroutine only renders objects left to right. The pair of ‘pyramid’ staircases seen near the end of 1-1 are drawn with special staircase objects for their ascenders and individual solid block columns for their descendents.

Other special objects trigger an engine behavior rather than a specific segment of onscreen graphics. In row $D$, for instance, there are two object types that doppelganger labels AreaFrenzy and ScrollLock. The former handles levels such as 2-3, wherein enemy Cheep-Cheeps spawn continuously beneath Mario as he runs across suspended bridges, or level 5-3, wherein Bullet Bills fire at Mario from an unseen cannon beyond the screen’s right edge. The ScrollLock is more self-explanatory. Certain automated portions of the game, such as the cloud bonus levels or transitions to the warp zone selection, momentarily wrest scrolling control from the player, locking its movement to a fixed rate.

A final special object is notable due to *SMB’s* close affinity to *The Legend of Zelda*. The ‘loop command’ occurs in certain castle levels. In 4-4, for instance, Mario’s path is split into an upper and lower level. If the player chooses the lower path, the same sequence of screens will loop infinitely. Taking the upper path breaks the loop and permits Mario to move forward to another looping area, this time split into a three-tiered path. In 7-4, the spatial maze is more complex. The double- and triple-tiered paths require multiple branches (e.g. upper, bottom, middle, upper) to pass successfully. In code, the loop function involves checking Mario’s y-coordinate position and footing (he must be on solid ground) against a prescribed area coordinate. If either condition fails, the current page is reset and the appropriate area data is re-rendered. Notably, *SMB’s* spatial loops are the only circumstance where Mario can return to portions of the level that have scrolled offscreen—though, appropriately, he may only do so by continuing to the right.

*The Legend of Zelda* has a similar spatial maze. Link may enter a special screen within the Lost Woods from the eastern side. Surrounding trees outline four conjoined paths that branch to each of four cardinal directions. If Link heads east, he returns to the prior screen. However, moving north, west, or south returns Link to the same screen in an apparent infinite loop. Like *SMB’s* castles, only a certain pattern of path choices (north, west, south, west) will lead Link out of the Lost Woods and into the Graveyard. Both games’ mazes produce unmappable cartographies that can only be represented through movement vectors.
At this point, the engine has populated the set decoration and area objects, but the level is still devoid of Mario's enemies. Fortunately, the format used for enemy objects closely models the area object format. The complete byte block for level 1-1 reads as follows:

;level 1-1
E_GroundArea6:
.db $1e, $c2, $00, $6b, $06, $8b, $86, $63, $b7, $0f, $05
.db $03, $06, $23, $06, $4b, $b7, $bb, $00, $5b, $b7
.db $fb, $37, $3b, $b7, $0f, $0b, $1b, $37
.db $ff

Enemy object data occupies far less space than area objects since levels are more populated with bricks, blocks, and staircases than with Koopas and Goombas. Understandably, these data blocks also lack the two-byte header used to set the starting conditions of the level. Such a header would be redundant. However, the block does feature a terminating byte, $FF, to signal the data's conclusion.

The remaining bytes use the same position/object pairs from the level area data, with a few notable variations: First, enemy objects have valid page row values up to and including $0D. Row $0D extends beyond the screen’s visible lower edge, but the lower origin point is useful for enemies that emerge from the bottom of the screen (i.e., Cheep Cheeps). The remaining unused rows—$0E and $0F—are once again reserved for special objects. (Objects in row $0E are the sole exception to the byte pair rule; their function requires a third byte, explained below.)

Second, enemy object types are not sorted into large or small types. d0-d5 in the object byte designates the type from a single lookup list, so decoding objects based on selection bits is not necessary. Third, enemy object positions vary from area object positions. For the latter, for example, a position of $3, $6 denotes that the metatile’s position will be in the area described by the intersection of the specified row and column planes. For enemy objects, the row denotes the horizontal position upon which the object ‘stands.’ To illustrate these differences, we will once excerpt the opening portion of the byte block and expand it into tabular form:

<table>
<thead>
<tr>
<th>BYTES</th>
<th>COL,ROW</th>
<th>OBJ</th>
<th>BINARY DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1E, $c2</td>
<td>1,E</td>
<td>n/a</td>
<td>Area change (address offset: $C2)</td>
</tr>
<tr>
<td>$00</td>
<td>n/a</td>
<td>00000000</td>
<td>Area change ‘check’/page #</td>
</tr>
<tr>
<td>$6B, $06</td>
<td>6,B</td>
<td>00000110</td>
<td>Goomba</td>
</tr>
<tr>
<td>$8B, $86</td>
<td>8,B</td>
<td>10000110</td>
<td>{P} Goomba</td>
</tr>
</tbody>
</table>
The first byte triplet is a special case. Doppelganger labels enemy objects in row $0E$ as ‘area change objects,’ which function when Mario enters a pipe, climbs a vine, or falls into a pit in a bonus cloud section. Consequently, the second byte is an address offset used to designate which area Mario will end up in when such an event occurs. The lower nybble (d0-d3) of the third byte tells the engine which page of the designated area Mario should begin on when he arrives in the new area. The upper nybble (d4-d7) of the third byte, however, is used as a check for the area change object. If Mario’s current world does not match the world stored in the third byte, the area object change will not take effect. In the data above, the current world must be 0, so level 1-1 passes muster. When Mario enters the one usable pipe in the level, he descends to the underground coin room. In general, the world check permits the reuse of enemy object data in multiple levels, ensuring that the pipes and vines of a given level operate properly.

The following three byte pairs are more conventional. The first byte denotes page coordinates, while the second byte denotes the enemy type along with two additional condition flags. D7 of byte 2 continues to signal the page advance flag. D6, however, serves as the ‘secondary hard mode’ flag. If a player successfully completes SMB and chooses to begin again without resetting, enemy quantities and types are changed to make additional plays more difficult. This was an innovative option in 1985, providing expert players with new challenges. If the hard mode flag is not set, the enemy object is not rendered. The remaining six bits (d0-d5) are used to select the enemy type. In the table above, the first three objects are a single Goomba (%000110), another single Goomba, and a Goomba pair (%110111), each on separate pages. Note that the Goomba pair is treated as a single enemy object, even though its constituent Goomba objects behave independently.

Also note the position bytes for these three objects. The first two Goombas are positioned vertically on top of row $0B$, which coincides with the top of the terrain. In other words, the engine spawns them on the ground, like Mario. However, some objects have strange beginnings beyond the screen’s borders. When Mario encounters the first Goomba pair, they are pacing back and forth between two adjacent pipes like the solo Goombas from the two previous pages. From their onscreen position we might infer that they began on the ground as well. However, according to their coordinates in the table above, they spawn a few metatiles from the top of the screen. In other words, they originate in mid-air and drop into position between the two pipes, a process that is never seen by the player. The only way to check this behavior, besides parsing the code, is to replace the enemy object with a type that is unaffected by gravity. In figure 4.6 I have used a hex editor to replace the double Goomba object with a jumping board to ‘catch’ the object at its point of
As we saw in the discussions of R.O.B. and the Zapper in chapter 3, the ways that hardware ‘perceives’ the game onscreen is often incongruent with or even counterintuitive to player perception. The same applies to software—not only in relation to the player, but also to the hardware itself. The player, for instance, sees a single screen of graphics at a time. Beyond artistic tricks that simulate depth in tile-based graphics, the overall image is two-dimensional and flat. We already know that internally the Famicom differentiates between two types of tiles—background and sprite—that follow different positioning, palette, and priority rules. For the PPU, sensibly, the name table describes a single flat plane of background tiles. The *Super Mario Bros.* engine, however, differentiates further,
assigning software priorities to background tiles that do not exist in hardware. In other words, the PPU can only ever see a flat name table, while the engine can split that name table into multiple layers. As we have discovered, *Super Mario Bros.* relies on such perceptual distinctions to economize the data necessary to construct its varied levels.

**Vertical Mirror**

Videogame players returning to *Super Mario Bros.* nearly three decades after its debut may quickly notice that the game scrolls exclusively along a single vector—left to right. Once a portion of the screen has passed beyond the left border of the television screen, Mario can no longer return to that space. In other words, the player is unable to reverse the scroll. The left edge of the screen is an impassable barrier.

Subsequent Famicom games in the *Super Mario* series permitted movement along both axes. The American version of *Super Mario Bros. 2* included vertical and horizontal areas with bi-directional scrolling, albeit limited to one direction at a time. *Super Mario Bros. 3* upped the spatial ante: Mario (and Luigi) could run, jump, swim, and fly along any axis. The added diagonal trajectory supported a new gameplay mechanic—the raccoon suit—that allowed Mario to fly for short periods of time once he had gained sufficient speed on a stretch of unimpeded runway. However, both successors benefited from the TSROM/MMC3 mapper, which expanded the cartridge ROM significantly. In the case of *Super Mario Bros. 3*, the cartridge upgrade expanded the PRG capacity eightfold and the CHR capacity sixteenfold.

*SMB’s* gameplay style demands a modest amount of memory overhead both onscreen and beyond. Since Mario can break blocks, bump out their contents, and collect coins, those objects’ status must be maintained until Mario scrolls past them. Likewise, to maintain the illusion of a world that extends beyond the current screen, moving enemies or ricocheting shells should pass into and out of frame believably. But such actions must take place within feasible boundaries. If Mario could bash a hundred brick metatiles scattered throughout a level then backtrack to those screens, the engine would have to maintain a ledger of objects in RAM to ensure that broken blocks stayed broken. Likewise, a skittering shell cannot be tracked as it glides across an entire level. NROM cannot sacrifice the necessary memory space to keep track of objects for their entire lifespan.

The *SMB* engine has what I will call a ‘sliding ontological window’ wherein all objects are permitted to ‘exist.’ Once an object passes beyond that window, whether through their own volition or as a result of the screen scrolling past them, the engine ‘forgets’ them. They are summarily dropped from memory and no longer exist as active objects. The ontological window contains two full screens of decoded metatiles (32x13) that slide along in sync with the screen’s scrolling rate. As the player moves, the camera is positioned in the direct
center of the window, leaving an eight-metatile ‘buffer’ on either side of the current screen (figure 4.7).

4.7 Overlaying the active emulation window on top of FCEUX’s name table viewer simulates the engine’s ‘sliding window.’ The eight-metatile buffer to the left of the play screen’s border has already scrolled, but is still held for offscreen collisions. The eight metatiles to the right wrap around to the left side of the name table viewer and end at the abrupt edge of the green cloud. The cloud color error is visible because the metatile does not yet have its proper attribute bits assigned. (Emulator: FCEUX 2.1.5)

The buffers provision a plausible range of motion for enemies that travel offscreen. If a kicked Koopa shell or a shambling Goomba move out of frame but then collide with a pipe or enemy within the buffer, they will react accordingly. If the enemy wanders beyond the buffer, it disappears permanently.

SMB’s US instruction manual makes specific note of the peculiarities of the sliding window:

Because the screen moves from left to right, there are enemies off the edge of the screen that can’t be seen. You can’t kill enemies you can’t see by sending a [shell] off the screen after them. Why not? Maybe they jump over the enemy when Mario isn’t looking…! Strangely enough, however, if a kicked [shell] bumps into a [pipe] off the screen, it comes ricocheting back at Mario. If you hear the sound of a ricochet, jump right away so you’ll be ready when it comes flying back onto the screen.³⁸

Grasping the engine’s frame of perception makes it clear that it is not simply that the player
cannot see enemies—they do not yet exist. And as silly as it sounds for unseen enemies to jump shells, we have seen from the enemy object data examples that objects sometimes spawn in unexpected locations and would, in fact, hover above a hurtling shell. Similarly, there are instances when Mario can kick and follow a shell, which clears enemies in his path. If the shell moves beyond the screen’s right edge but stays within the sliding window, enemies may spawn between Mario and the ricocheting shell. In other words, the enemy object data has not fully initialized until after the shell has passed the object’s coordinates.

Literal edge cases such as these illustrate the weird perimeters of SMB’s ontological window. Along the window’s right border, the engine is busy preparing objects prior to their screen entrance. They do not pop into existence fully-formed, but must be assembled according to the rendering rules outlined in the previous section. As you will recall from chapter 3, the Famicom’s scrolling capabilities take advantage of the name tables’ mirrored layout. Since SMB is soldered for vertical mirroring, its name tables may be arranged in the customary 2x2 arrangement seen in figure 4.8 below.
4.8 The customary 2x2 arrangement of name tables exhibiting vertical mirroring. (Emulator: FCEUX 2.1.5)

Though name tables are listed sequentially in the PPU’s memory map, arranging them in the 2x2 matrix makes visual sense of the mirroring setting. The upper two name tables, their addresses starting at $2000$ and $2400$, are duplicated along the x-axis, as expected, since the lower two name tables, at addresses $2800$ and $2C00$, mirror their contents exactly. Vertical mirroring is suited for horizontal scrolling because the camera can track along two contiguous name tables solely by updating the scroll register’s x-coordinate.
The challenge for games that scroll left to right beyond two screens is wrapping around smoothly from the rightmost name table to the leftmost. SMB’s programmers’ solution is the sliding window. The eight-metatile buffer provides adequate space ahead of the current player screen to decode the set decoration, area objects, and enemy objects, render them onscreen, and have them ready for the player’s arrival (figure 4.9).

4.9 Layering Nintendulator’s PPU Debugger above the active emulation screen illustrates how an eight-metatile wide buffer metatiles is prepared in advance of Mario’s arrival. (Emulator: Nintendulator 0.975)

Each time Mario advances two metatiles onscreen, a two-metatile wide column of world is drawn at the right edge of the sliding window, while the same column is dropped from its left edge. The rendering engine acts like a more reliable version of the animator’s hand in the famous Daffy Duck short, “Duck Amuck.” It vigilantly tracks the character’s advance along the screen and draws in the scenery before he gets there, lest the entire scrolling facade falls to shambles.

Mirroring is also used in its conventional sense to reduce the pattern tiles necessary to draw objects. In doppleganger’s disassembly, for instance, we find the following table for power-up graphics:
PowerUpGfxTable:
    .db $76, $77, $78, $79 ; regular mushroom
    .db $d6, $d6, $d9, $d9 ; fire flower
    .db $8d, $8d, $e4, $e4 ; star
    .db $76, $77, $78, $79 ; 1-up mushroom

Each power-up metatile is composed of four 8x8 sprites drawn in the following order: upper left, upper right, bottom left, bottom right. Each byte value references the appropriate sprite tile in pattern table 0. The regular mushroom, for example, is stored in four sequential bytes between $76 and $79, visible in FCEUX’s PPU Viewer. An economical use of palette swaps and sprite mirroring reduces the number of constituent pattern table tiles by half. Notice that both mushrooms use the same sprites (with different colors) while the fire flower’s and star’s left and right halves are simply mirror images across the metatiles’ y-axes. Coins, Podoboos, Piranha Plants, Bloopers, Spiny eggs, empty power-up blocks, drawbridge axes, stomped Goombas, fireworks, empty shells, jumping springs, the Mushroom Retainer, and even portions of Mario’s body are composed of mirrored tiles. Top to bottom, the Mushroom Kingdom is full of subtle symmetries.

Since the sprite pattern tables only had room for 256 individual tiles apiece, using tile mirrors to free up additional slots could mean provide memory for additional objects. As Nakago and Tezuka recount, the designers strove to save space whenever possible:

Nakago: Even with mushrooms and flowers, we’d be looking to limit the bytes we used, so we’d draw half of the object then flip it around to display it.

Iwata: That’s why these objects are all symmetrical.

Nakago: That’s true of the stars too. They’re symmetrical. There was the advantage that you could get an object that was double the size using only half the bytes.

Iwata: So all of these things were ways of limiting the number of bytes you were using.

Tezuka: That’s right. We came up with all kinds of objects, all the time trying to limit the bytes we were using.

Decisions such as these shaped the look of the Mushroom Kingdom. Technological constraints were structuring the world and its objects with an underlying symmetry, from the macro—large repeated patterns of metatiles that shaped the rhythm of hills, fences, and clouds—to the micro—flowers and stars constructed from mirrored halves.

Due to SMB’s limited CHR-ROM space, the programmers had to devise numerous ways
to repurpose graphics for multiple uses. Mirroring is one technique. Palette swapping is another. The palette swap is a mainstay of videogame design. Identical tiles with different colors are an economical means to generate ‘new’ characters without relinquishing additional memory space. SMB has its share of palette swapped tiles: blue water/red lava, red/green Koopas, power-up/1-up mushrooms, and even Mario/Luigi. Simple color changes could squeeze a little extra mileage out a limited set of tiles.

Where SMB’s design really shines (literally and figuratively) is in its clever use of palette swaps to generate special effects. For instance, the question mark blocks, status bar coin counter, and the various collectible coins strewn about the landscape all shimmer synchronously thanks to a shared palette. Color entry one ($3F0D) of the fourth background palette quickly cycles between three color values to produce a convincing metallic glimmer (figure 4.10).

4.10 The block’s fill color is cycled between three values to produce a glimmer effect. (Emulator: Nintendulator 0.975, Photo Editing: Nate Ayers)

Increments of $10 are subtracted from the base color $27, a warm orange hue, to produce a brownish orange ($10), then a dark umber ($07). Increments of $10 are then added back to the dark umber to reverse the sequence until the value returns back to the lighter base. In short, the cycling of one palette’s color slot produces animation without altering the underlying metatile in any way.

Both the Starman and Fire Flower power-ups use a related but slightly more sophisticated technique to produce their distinctive animations (figure 4.11). Rather than cycling a single color slot, both the star and the flower’s upper half cycle rapidly through the four available sprite palettes. At high speed, both items appear to flash. And appropriately, when Mario or Luigi grab the star, their metatiles are subject to the same technique.
4.11 The four-frame palette cycle used to animate the fire flower. (Emulators: Nintendulator 0.975, FCEUX 2.1.5, Photo Editing: Nate Ayers)

In effect, neither power-up has its ‘own’ palette since it uses them all in equal measure. Furthermore, due to their limited number, each sprite palette is catered to a ‘high priority’ object, while the remaining sprites are assigned colors based on those constraints. The sprite palettes in World 1-1 (figure 4.12) make this point clear: Palette 0 and 2 are devoted to Mario’s normal and fireball states, palette 1 colors the Koopas, and palette 3 paints the Goombas.

Ancillary sprites typically pick and choose one or two colors from a palette: flower stems and the final flag use Koopa green, shattered bricks use brown and black from the Goomba palette, and fireballs sensibly share the fiery Mario palette.

One of SMB’s more notorious palette swap tricks was best summarized in a NeoGAF thread from January 2008 called ‘Videogame facts that blow your mind,’ wherein member Ezduo commented: ‘The bushes in Super Mario Bros. were just recolored clouds.’

Numerous subsequent commenters, true to the thread’s title, called the revelation ‘mind-blowing.’ Indeed, casual players may have never noticed this creative touch, since the cloud metatiles of the background scenery are partially obscured by the terrain. The subtle crop sells the color change: background palette 2 (pictured above), which gives the clouds their
billyow blue and white colors, swaps to palette 0, shared by hills and pipes, to produce the mind-blowing bushes.

**Exergue**

In chapter 3, we discussed a special tile in Object Attribute Memory used for timing and mid-screen scrolling techniques. Sprite 0 is the first entry (i.e., position 0) in OAM and provides the Famicom’s sole hardware collision detection. When a non-transparent pixel of a background tile overlaps a non-transparent pixel of sprite 0, a special bit flag is raised. In *Gyromite* it was used both to sync the color-coded screen flashes that cause R.O.B. to move and to mark the split point between non-scrolling and scrolling portions of the screen.

Famicom programmers commonly used sprite 0 to split the screen at a designated scanline, a handy tool when one wants to provide the player with status information—lives, bullets, health, items, etc.—without that information perpetually scrolling offscreen. *Super Mario Bros.* is no exception. At the top of the screen, there are four columns of information, distributed across two rows. Reading left to right: the current player name (Mario or Luigi) with their score underneath; a small coin icon, a multiplication symbol, and a value between 00 and 99; the current world and level (e.g. 3-1); and the remaining time.

Mid-screen splits are necessary because the Famicom has only one background layer. In modern videogame systems like the Nintendo DS, multiple planes of graphics can overlap to form a single flat background image, greatly simplifying visual tricks that give the illusion of depth in a 2D graphical space. Parallax scrolling, for instance, simulates the variable rates of movement of objects in a horizon based upon their distance from the viewer. For a passenger peering through the window of a moving vehicle, the guard rail speeds by in a blur, the trees beyond the guard rail move slower, and the hazy mountain peaks in the distance move slower still. In the scrolling background of a DS game, programmers can devote a plane to each of these background elements and scroll them at independent speeds. The net effect for the player is a convincing range of perspectival distances. The Famicom’s PPU provides no such affordance. Any variation in background scrolling requires precision timing in code.

The sprite 0 hit permitted *SMB*'s designers to forgo scrolling until after the status region was drawn. Although *SMB*'s status bar appear to hover atop the scrolling Mushroom Kingdom, both bar and kingdom exist on the same background plane.

Sprite 0 is rarely conspicuous despite its position of honor in OAM. When the PPU renders objects, sprites with a lower position in OAM have higher screen priority. When objects overlap, the PPU must use a number of rules to decide which object’s pixels will be drawn and thus appear ‘on top.’ In other words, if all sixty-four sprites were stacked
onscreen in an 8x8 pixel pile, sprite 0 would always appear to be above the rest. However, sprite 0 does not have to sit atop background tiles in order to fire properly. Like any other sprite, sprite 0’s priority bit can be set to be either in front of (0) or behind (1) background tiles. The most sensible setting is the latter, assuming the game’s programmers do not want the player to see a stray floating pixel. To ensure sprite 0’s discretion, it is typically a single pixel (Tecmo Bowl), a solid rectangular tile (Excitebike), or a slender line (Ninja Gaiden).

In SMB, sprite 0 has a novel shape: it mimics the bottom sliver of the coin tile that records Mario’s current stash. Its upper-left x- and y- coordinates are 88,24—directly behind the status bar coin whose shape it partially imitates (figure 4.13).

4.13 Disabling Super Mario Bros.’s background tiles in an emulator reveals the hidden sprite 0 near the top of the screen, shaped like the lower portion of a coin. (Emulator: FCEUX 2.1.5)

Although sprite 0 is the first entry in Super Mario Bros.’s OAM, it is the last sprite tile in pattern table 0 (i.e., tile $FF$ at address $0FF0$). In fact, it never matters where sprite 0 is located in pattern table memory. In a fixed 8KB bank of CHR-ROM, pattern tables tiles are
static, but they can be shuffled into and out of OAM and name tables as needed. However, sprite 0 must remain in a fixed position both in OAM and in its designated name table. To do otherwise would either demote its status as sprite 0 or cause the scroll split to go haywire.

*SMB’s* sprite 0 placement coupled with the game engine’s porous upper screen boundary reveals an interesting quirk of the PPU’s rendering process. Sprite 0 has two potentially contradictory priority attributes: its OAM position means it appears *above* all other sprites, while its OAM priority bit can place it *behind* background tiles. So what happens when a sprite with the inverse attributes—lower OAM priority, but higher background priority—passes in front of the coin that hides sprite 0? In most cases, the overlap never happens. Most sprites are not permitted to breach the invisible border demarcated by sprite 0, even those that are airborne, like cloud-riding Lakitu. Mario is the obvious exception. He is permitted to scale tall brick structures and often jump beyond the top of the screen, as is the case when you attempt to hit the top of the flagpole at the end of a stage. In certain levels, Mario’s ability to cross the screen’s upper threshold allows him free reign to run across the status bar tiles. As platformers became more sophisticated, programmers would have the status bar discreetly slide away when the character’s head reached its boundary or treat the status bar as a hard upper limit. But in 1985, and within the limitations of NROM, it was permissible to allow Mario’s sprite to simply run *in front of* the status bar, allowing the player to progress forward without text obscuring their path. Mario’s ability to break through this diegetic wall, obfuscating data meant for the player’s eyes, added to the game’s mystique. Players felt as if they were violating the rules by running along the upper border, despite this being a necessary technique to discover *SMB’s* hidden warp zones.

Mario’s spatial transgression provides a simple method to reveal sprite 0’s hiding place without needing to disabling background tiles in an emulator. In World 1-2, it is fairly straightforward to position Mario above the upper brick border and nudge his leg into position over the coin icon (figure 4.14), causing the PPU to render an odd ‘triple overlay’ effect.⁴²
4.14 Breeching the upper bounds of World 1-2 exposes a PPU priority conflict, seen here in front of Mario’s left leg. (Emulator: Nestopia 1.4.1).

While the majority of the coin background tile is positioned properly behind Mario’s sprite, the lower portion of the coin rests in front of his leg. But notice that the prioritized tile is the background coin, not sprite 0, since the former is colored orange rather than blue (figure 4.15).

4.15 Super Mario Bros.’s isolated sprite 0 tile painted with the Level 1-2 palette (NES CHR Editor 1.1.1).

The layer confusion between leg, coin, and sprite 0 demonstrates several competing priorities at play: sprite 0 has priority over all other sprites, but its individual sprite priority attribute places it below any background tile; Mario’s constituent sprites have priority above background tiles, but below sprite 0. When the PPU evaluates the scanline with overlapping tiles, it must ‘decide’ which pixels to display on top. We would expect Mario to trump all other tiles, since he has priority over the background and the background has priority over sprite 0. But sprite 0’s OAM position creates a strange compromise: the shape of sprite 0 sits atop Mario’s leg, but the pixels are painted like the background tile. So sprite 0 is paradoxically both above Mario’s sprite and behind the background tile. To the trained eye, that slight crescent of pixels marks the invisible line
beyond which Mario’s world is permitted to scroll.

Though this visual check is trivial to reproduce on real hardware, it is worthwhile to use an emulator to understand how the sprite 0 split appears in relation to the game’s name table arrangement. Once again, FCEUX’s Name Table Viewer arranges VRAM’s four nametables into the familiar, vertically-mirrored 2x2 matrix seen in previous sections (figure 4.16).

![Name Table Viewer](image)

4.16. A tool-assisted view of Super Mario Bros.’s four name tables mid-scroll. Note that the status bar information is present only on the left name tables. (Emulator: FCEUX 2.1.5)
The intersecting lines drawn across the viewer are FCEUX’s scroll lines, representing the current X/Y coordinate position stored in the PPU’s scroll register ($2005). As Mario moves across the screen, the scroll lines track the scroll coordinate updates. The vertical line indicates the leftmost position visible on screen, i.e., the point of no return past which Mario can no longer backtrack.

Notice that the small dialogue box underneath the bottom left name table reads ‘Display on scanline: 32.’ If we leave this setting at the default scanline value 0, the scroll lines will never move past their 0,0 origin point. Based on what we know about the sprite 0 split, this makes sense. As far as scanlines 0 through 31 are concerned, the name tables do not scroll. This should be clear from the upper left position of the status bar seen above. Though the scroll lines indicate that the onscreen graphics are now drawn predominantly from name table 1, the status bar is always drawn from name table 0. For the first thirty-two scanlines, the PPU is directed to fetch the status bar tiles from table 0, then, based on the scroll register’s position, grab the tiles necessary to construct the remainder of the screen. FCEUX provides us with a nice visual representation of what the PPU ‘sees’ at any given scanline.

Though many games used sprite 0 splits to halt the scroll for static status bars, both before and after Super Mario Bros., timed raster effects were not always necessary. Depending on the quantity of information necessary to convey, it could be simpler to use sprites to compose the heart icons, bullet counts, and enemy boss health bars. As always, the PPU’s sprite limitations applied: sixty-four total onscreen at a time and eight on a single horizontal scanline. These constraints set limits on the shape and density of information displayed in sprite-based status displays.

For example, Contra’s horizontal status bar is minimal compared to SMB’s. Contra uses the upper region of the screen solely to track player one and player two’s current life count, marked with blue and red ribbon sprites, respectively. If both players manage to gain a few extra lives during play (or input the Konami code), the onscreen display will max out at four ribbons each—unsurprisingly, the horizontal sprite maximum. During the game’s attract loop (or if one player loses all of their lives during 2-player play), the status bar reads ‘GAME OVER.’ Contra’s restrained status bar is possible because the programmers included an independent interstitial screen displaying the players’ total lives count, score, and current stage name, eliminating the need to keep that information visible onscreen. Nonetheless, Contra’s raised platforms, vertically-scrolling level (‘Waterfall’), and main protagonists’ prodigious jumping talents all pose problems for the status bar. Enemy, player, and projectile sprites frequently pass through the ribbons’ horizontal plane, creating noticeable flicker.

In fact, Contra’s gameplay, especially with two simultaneous players, generates a vast volume of sprites—especially bullets—onscreen, all vying for priority. Konami’s
programmers designed the game engine to constantly cycle the OAM positions of all onscreen sprites. If one watches the sprite slots in the Nintendulator emulator during gameplay, the sprites appear to cascade in waves across OAM in a hypnotic rhythm, adding and subtracting sprites from the flow as they move into and out of frame. The phenomenon is difficult to convey in screenshots (see figure 4.17), but its elegance is a testament to Contra’s design. The programmers conceded a constant flow of sprite flicker in exchange for an overwhelming amount of onscreen action. And no sprite elements were given undo precedence—the player was just as likely to briefly disappear as a bit of Spreader shrapnel. The end result was a pleasing screen density despite the near-constant flicker.

4.17 Two frames from Contra display the game engine’s OAM cycling at work. The ‘Sprites’ field at the bottom shows all sixty-four slots of OAM memory and their contents. Here we can see the blue status bar ribbons, the spherical Spreader bullets, the sprites comprising player one’s body and gun, and two halves of an explosion (also note the 8x16 sprite profile). OAM positions are continuously cycled, creating a hypnotic ‘cascade’ effect when viewed in realtime. (Emulator: Nintendulator 0.975)

Mega Man shares Contra’s sprite-based status bar approach, but adopts an inverse orientation. Depending on the player’s game progress, up to three status elements will be
onscreen simultaneously: Mega Man’s life bar, equipped weapon energy, and score. Since the life and weapon energy bar are several sprites long, they are displayed vertically. This arrangement prevents the horizontal flicker problem present in Contra, but it both limits the number of total sprites that can appear within a wide swath of vertical screen real estate and permanently subtracts from the pool of sixty-four simultaneous sprites. In figure 4.18, we can see that the life bar is seven tiles high, permanently limiting the horizontal sprite density of nearly one quarter of the screen’s vertical resolution. When the energy bar appears beside the life bar, sprite density is even further constrained.

![An excerpted frame from Mega Man with author annotations. The seven stacked sprites of the life bar are outlined in red, while the horizontal planes they constrain are shown in yellow. A separate yellow line bounds the score region. (Emulator: Macifom 0.16)](image)

The tile row directly above the life bar, used to display the player’s score, suffers far worse. The score requires seven tiles, so any sprite object with a width greater than two tiles that passes through its plane will trigger flicker. This too is visible in figure 4.18: Mega Man’s torso and bent knee exceed the sprite limit so the two digits that are supposed to follow the numeral 5 are only partially visible. In subsequent frames, Mega Man’s body will flicker out of view as his sprites trade priority with the score.

Mega Man’s engine uses an OAM cycling scheme similar to Contra’s. When sprites are active onscreen, their OAM positions are continuously swapped. However, unlike Contra’s cascade effect, Mega Man’s sprites are shifted around OAM in related chunks. For instance, during a section of the Fireman stage where columns of fire rise from the floor (see figure 4.19 below), the sprites are arranged in OAM as follows (while Mega Man is stationary and the game is paused): sprite 0 (positioned offscreen); seven score display tiles; seven life bar tiles; seven weapon energy tiles; ten Mega Man tiles; sixteen tiles used to draw fire column one; and sixteen tiles used to draw fire column two. When OAM is cycled in the next frame,
the arrangement changes to the following: sprite 0; fire column two; fire column one; Mega Man; score display; life bar; and weapon energy. In other words, the sprites are cycled as metatiles and shuffled according to a scheme that assures no screen element will disappear for multiple consecutive frames. In this configuration, the score display metatile always has priority above the life and energy bars, but in-game the arrangement does not matter—their fixed positions ensure that their respective horizontal planes never interest.

The OAM cycling routine creates odd circumstances where sprites will disappear even when the scanline limit has not been exceeded. If, for example, Mega Man fires three bullets while the fire columns are raised, portions of the right column will disappear (figure 4.19). This is clearly not a scanline overflow, but an overflow of available sprites. The fire columns are given lowest priority when the onscreen sprite count exceeds sixty-four. Consequently, the game engine shuffles the three trailing sprites of column two’s metatile out of the OAM queue on odd frames to make room for Mega Man’s Power Buster bullets, which for obvious hit detection purposes must always remain onscreen.
4.19 When Mega Man fires three bullets in this sprite-heavy section of the Fire Man stage, a three-tile chunk of the rightmost fire column metatile must be momentarily removed from OAM. Bullets must always remain in OAM once fired in order to properly register collisions with enemies. (Emulator: Nintendulator 0.975)
Like *Contra*, *Mega Man’s* mixture of vertical and horizontal level design creates numerous occasions where sprite limitations are violated. Figure 4.20 shows a common scene—*Mega Man* uses a ladder to exit the top of the screen. As he passes the score display, a tile-sized portion of his helmet disappears.

![Image](image_url)

*4.20 As Mega Man ascends through the horizontal plane of the score display in the Cut Man stage, one of his constituent sprites is not rendered due to exceeding the eight sprite per scanline limit. (Emulator: Macifom 0.16)*

Fortunately, the bulk of status information the player requires for gameplay (e.g., number of lives, available weapons) is either tucked away in a sub-screen or communicated through diegetic cues. For instance, *Mega Man*’s current equipped weapon does not require a status bar prompt—instead, his suit palette swaps to a new color to indicate the change. If additional status elements were moved into the normal gameplay screen, there would be barely an sprites left over to display enemies or items.

It is a testament to the games’ quality that *Contra* and *Mega Man* became so popular in spite of their obvious technical shortcomings. In both games, there are few screens that do not suffer from significant flicker. An overabundance of onscreen objects, subject to CPU-intensive collision checks and sprite prioritization, frequently overwhelm both game engines and cause noticeable slowdown. Ironically, *Mega Man*’s shortcomings have become part of the long-running franchise’s trademark style. So much so that in 2008’s Famicom-inspired ‘retroboot’ of the series, *Mega Man 9*, simulated sprite flicker and engine slowdown were selectable features via the game’s ‘Legacy Mode.”

Though both *Contra* and *Mega Man* benefited from more advanced mappers than *SMB*, the formers’ choice of sprite-based toolbars were not affordances of better cartridge hardware. Both the sprite 0/background tile and sprite-based approach had advantages and drawbacks. For *SMB*, employing sprite 0 meant sacrificing a large chunk of the upper
playfield to static status information that, in some cases, could be covered by the player’s sprite. But the use of background tiles eliminated any potential sprite flicker. For *Contra* and *Mega Man*, the gains in screen real estate required a sacrifice of sprite fidelity—a certain threshold of flicker was allowable as long as it did not significantly inhibit gameplay. It also meant adapting the game engines to account for constant shuffling of sprite OAM. Decisions like the size, orientation, and information density of status elements at first appear to be dictated more by the videogame designer’s whims, but at the platform level, they are decisions hewn closely to hardware and software constraints, often tying into the structure and design of the game engine itself.

**Camera Lakitu**

In videogames journalism, ‘camera control’ is associated with the advent of 3D polygonal gaming. Expanding the flat, tile-based worlds of 2D videogames into an extra dimension created new problems for player perspectives. If a programmer must represent a player’s avatar within a space that adheres to perspectival rules, they must consider both how that avatar is positioned and from what vantage point that avatar is seen. If, for instance, the camera is positioned low to the ground, angled up, and tight on the avatar, that character may occupy the majority of the screen, obscuring terrain, obstacles, and enemies. If the camera is placed high in the sky, directly above the avatar’s head, the player now has a bird’s eye view of their surroundings—ideal for gameplay that demands a wide scope of the playfield (e.g. sports or real-time strategy) but poor for precise jumping or melee combat.

As videogame designers grappled with this problem during the first generation of 3D consoles and PCs, they experimented with a variety of solutions. In 3D platformers like *Super Mario 64*, camera control was both automated and player-controlled. In most cases, Mario’s relationship to the terrain or other objects dictated the camera position. Sliding down an icy path, for instance, would trigger a high angled view, while dialogue segments would swing the camera behind Mario’s shoulder so the player could see with whom he was conversing. In egregious cases of camera misbehavior, the player’s perspective might be trapped behind a block of in-game architecture, completely obscuring their view of Mario.

*Super Mario 64’s* designers gave the player limited camera control, mapped to the Nintendo 64 controller’s C buttons. At the time, the 3D camera and its sometimes unpredictable behavior were so novel they had to be explained by a diegetic device. In-game, the Lakitu Bros., the Spinies-chucking Koopas who menaced Mario from the clouds in *Super Mario Bros.*, track Mario’s movements with cameras slung from fishing poles. The camera is thus anthropomorphized and mobilized, creating at least a plausible explanation...
for why it might sometimes have ‘a mind of its own.’ This device is introduced early in the game when one of the Lakitu Bros. descends from the sky and tells the player about their journalistic enterprise: ‘Mario has just arrived on the scene, and we’ll be filming the action live as he enters the castle and pursues the missing Power Stars’ (see figure 4.21). We never see more than one brother talking with Mario, presumably because the other has to film the conversation.

4.21 In Super Mario 64, the camera is anthropomorphized as the Lakitu Bros. in an attempt to plausibly describe its function (and malfunction).

Three dimensions added new challenges to camera positioning, but this was a not a novel phenomenon, even in the 2D-era. Viewpoint is a concern for any video game, from *Pong* to *Gears of War*, and often establishes not only the player’s position in the game world, but the genre as well (e.g. *first-person* shooters). Of course, camera position was less complex in single-screen games common to the arcade era. Game designers rarely had to shift perspective from screen to screen or grant the player camera control.

The introduction of 2D scrolling made camera position more complex, since new data was streaming into the player’s view at variable rates. The camera had to be close enough for the player to recognize and interact with objects onscreen, but also distant enough to permit the player to react to new objects as they entered the frame. Scrolling direction
played a crucial part in this decision. If the player moved left to right, it made sense to allow a larger buffer on the right side than the left, so the player might better anticipate and react. Likewise for vertical scrolling space shooters: the player’s spaceship was placed along the bottom of the screen as play began. Multi-axis scrolling increased the complexity further. If the player could move in any of four cardinal directions, where should the player’s avatar be placed on the screen and how distant should the camera be from that position?

Although SMB scrolls along a single, non-reversible horizontal trajectory, its engine handles scrolling in a remarkable way. At the opening of each outdoors level, Mario is positioned at X/Y pixel coordinates 40,192. (Note that this and subsequent coordinates describe the upper left pixel of the upper left sprite in Mario’s metatile, a point located a few pixels to the left of his cap.) In other words, Mario begins his adventure five tiles forward from the screen’s left boundary. His position provides the player their first visual cue on how to proceed: Mario is nearly backed against a border (allowing a buffer for overscan) with an expansive world opening up to the right. The player is encouraged to move toward open space. This is especially evident in World 1-1, where Mario faces an entire screen’s worth of unobstructed runway, set against a backdrop of open blue skies.

Once the player moves, the scrolling almost immediately kicks in and Mario’s position appears to lock close to center screen. If the player maintains a steady running pace, the world moves in lockstep, only halting when the player chooses to slow their progress—perhaps to vault a platform, bump a block, or squat into a green pipe. The camera is fluid and responsive, always permitting the player ample time to adjust to enemies or obstacles as they appear. But Mario’s slight offset of center subtly suggests that the player should continue their path to the right.

To achieve the camera’s flexible pace, the game engine employs two distinct scrolling thresholds. The first boundary (A) is 80 pixels from the left edge of the screen. As soon as the leftmost pixel of Mario’s metatile passes that invisible line, scrolling commences. The first screen of World 1-1 is ideal for checking the thresholds, since we can scoot Mario’s body slowly past the first five cracked floor blocks (80 pixels = 5 metatiles) until the scroll kicks in. The second boundary (B) is 112 pixels (7 metatiles) from the left edge of the screen. The 32 pixel region between boundary A and B serves as a ‘momentum field’ (y) wherein Mario accelerates toward his maximum sprinting speed and gradually ‘locks’ into his rightmost position (figure 4.22).
The two scrolling thresholds designated in Super Mario Bros.’s game engine. Boundary A occurs at player x-position $50$ (80px), while boundary B occurs at player x-position $70$ (112px). Region y is a ‘momentum field.’ (Source: NES-101 capture)

If the player walks forward tentatively, Mario will remain in this field until the player either stops to reverse direction or accelerates to either a running (holding right) or sprinting (holding right + B) speed. Boundary B functions as a soft limit: if Mario accelerates to a high enough speed, his x-coordinates can sometimes drift several pixels right of the boundary.

This may seem like bog-standard scrolling behavior for a platformer, but consider the implications of the subtle interactions taking place between player input, sprite positioning, and scroll movement. Between x-coordinates 0 and 80, the player has direct control over Mario’s metatile, as we would expect. When we press the D-pad and buttons, we sensibly intuit that we are directly moving the character onscreen. However, once the Mario passes boundary A, we begin to control both Mario’s movement and the scroll amount simultaneously. If Mario continues to accelerate, his x-position updates slightly faster than the scroll, allowing him to lock into position near boundary B. However, at this point, if we continue to move Mario forward without halting his momentum, his metatile x-
coordinates no longer update. Our control has now been transferred almost fully to directing the scroll.

The sole exception is jumping, since pressing A still grants us control of Mario’s y-coordinates. Of course, stopping or turning left immediately wrests Mario’s horizontal position back into our hands, but from the game engine’s perspective, our input orchestrates a series of fluid hand-offs between sprite and scroll control. This process is wholly opaque to us and, frankly, counterintuitive to our common phenomenologies of ‘interactivity.’ We think Super Mario Bros. plays well because we have total control over Mario’s movements—his jumps, ducks, and frictional slides feel responsive and fluid. But this is a perceptual illusion. Sometimes we do move the man, but other times we move the world. And sometimes we move both.

**Minus World**

The Mushroom Kingdom is full of secrets—some intended, some not. As early as World 1-1, Super Mario Bros.’s designers left clues that there was more to the world than what appeared on the surface. The short runway prior to Mario’s first pit jump, for instance, contained a hidden 1-up block that the player could reveal if they mistimed their jump. It was a secret reward for miscalculation. And once that first invisible block was apparent, the whole Kingdom took on a mysterious air. What else was hidden within its scores of blocks?

Players picked apart Super Mario Bros.’s intricacies with fervor. In Japan, the complete guide to SMB was a surprise bestseller, taking the top paperback sales spot in 1985. Such guides provide players with exhaustive maps of the game’s hidden power-ups, warp zones, and bonus coin levels. They outline scoring strategies, shortcuts through stages, and tricks to stockpile lives. A Koopa descending the staircase at the end of 3-1, for example, can be stomped in such a way that the player can hop endlessly on its shell as it ricochets against the wall, multiplying point values until it starts granting extra lives. But the trick has an underlying programmatic consequence—overly greedy players who amass too many lives find that their next death results in a game over. And the results of that action first tipped off players to a world of secrets that might be beyond the purview of tips and tricks books.

The first indication that this behavior is unintended appears during the black interstitial screen that displays the player’s current world and their accumulated lives. Once Mario has more than nine lives, the counter begins to display strange characters. In figure 4.23, Mario has $39 (decimal 58) lives, represented in-game by the ‘crown’ graphic and a blue triangle.
4.23 Accumulating more than nine lives in Super Mario Bros. causes unintended CHR-ROM tiles to seep into the lives counter. (Emulator: Macifom 0.16)

The crown hieroglyph is an intentional artifact used to indicate the tens place. Mario team expected some players to amass more than nine lives, but they did not build in any checks to prevent the graphical errors seen beyond nineteen. Like the Level indicator in Famicom Donkey Kong discussed in chapter 2, the lives display uses an offset to fetch the appropriate tile for the ones position, based on the current number of lives. 0 through 9 display in sequence, as expected. Once the player reaches ten lives, a crown marks the tens place and the 0-9 cycle starts again. Once twenty lives are earned, an ‘A’ curiously appears in the ones position.

In the background pattern table, 0-9 occupy the first ten tiles slots, then A-Z, followed
by a number of solid colored tiles, symbols, and then the game’s remaining background elements. Once the numbers have cycled from 1-9, then work through 0-9 a second time, the display routine begins fetching tiles beyond the numbers. The blue triangle above represents 58 lives ($39$ is decimal 57, but $00$ lives is 1 onscreen, so we add one to the actual value). The graphic is actually the slope of a hillside and appears blue because it is using the numeral tile’s attribute assignment.

But why does death trigger a game over when the player accumulates too many lives from the shell hopping trick? The answer is identical to the programming flaw that triggers the kill screen in Famicom Donkey Kong. The memory location that stores Mario’s lives, located in RAM at $075A$, is a single byte long. Once that byte reaches values exceeding $7F$ (or 127), the sign bit becomes 1. Without a proper check, the game interprets values $80$ (128) and higher as negative lives. When that occurs, Mario is considered to be on his last life, so the next death triggers game over."

The 3-1 trick illustrates the careful balance between programmer and player expectation. The Mario team concluded that most players would not exceed 127 lives and therefore it was not worth the bytes to build in proper checks to create an upper bound for 1-up accumulation. When players discovered the means to push the system’s boundaries, both beneficial (lots of lives) and non-beneficial (potential death) results surfaced. Miyamoto and team knew about the 3-1 trick prior to the game’s release, but they decided to leave it in because it was fun. And they likely knew that it sustained its own set of checks and balances, rewarding players who wanted a reasonable amount of extra plays, but penalizing those who went overboard.

SMB also had glitches that the team never predicted. The so-called ‘minus world’ is the most famous of these. Reaching the minus world requires some skillful maneuvers that are difficult to reproduce, lending to its mythological allure. Near the end of world 1-2, where Mario enters the pipe to return to the surface, it is possible to jump in such a way that Mario’s body can glide through the brick boundary separating the pipe from the warp zone area. Once Mario ejects into the warp zone—and if the player is careful not to trigger the scroll lock—Mario can enter the first pipe before the standard ‘Welcome to Warp Zone!’ text appears onscreen. When the interstitial screen appears, the current level is listed as ‘World -1.’

The level that follows is less mysterious than its name, since it is merely a duplicate of World 7-2 that loops endlessly. Mario can proceed left to right as normal, but once he reaches the exit pipe, he is cruelly deposited at the level start with no recourse but to wile his time and lives away.
Nintendo officially corroborated the existence of the minus world in issue 3 of their popular PR mouthpiece *Nintendo Power*, accurately deeming it ‘an endless water world from which no one has ever escaped’ (figure 4.24). The bare bones explanation gave players little indication of exactly how Mario was able to pass through walls, but the photographic proof in print settled the question of whether it was possible. The minus world fell in line with the numerous other secrets hidden throughout the Mushroom Kingdom, fueling rumors that the designers had tucked away secret levels for only the most skilled players to find. But the minus world is not designed—there is no area object code set aside for the looping water level (beyond that intended for 7-2), nor any other bonus levels beyond the conventional thirty-two. There simply was not enough leftover ROM space to tuck away extraneous secret levels. Instead, the minus world is based on a serendipitous coupling of engine and collision detection shortcomings.

As we have seen, beyond the limited capability of sprite 0, the Famicom has no hardware provisions for checking collisions between tiles. Collision detection must be handled exclusively in software. However, checking every point around a given metatile against all valid collision points of all other onscreen objects is far too computationally costly for the 6502 to handle in a single frame. Instead, programmers create bounding
boxes around objects to check and compare only a handful of collision boundaries. In *SMB*, Mario has two points, one for each foot, that check whether he collides with the floor. The drawback is that boundary boxes are more porous than solid boundary lines. If collision points on two objects happen to pass by one another without a proper collision, character objects can pass through otherwise ‘solid objects’ like bricks or pipes.

*Super Mario Bros.*, like many other games of its era, creates an ejection routine to compensate for holes in boundary checks. If, for instance, the horizontal position of Mario’s right foot is determined to be inside a block while he is moving right, the engine will eject Mario in the opposite direction, forcibly pushing his metatile out of the wall. This works for most common cases, but it is not a failsafe method. Part of the minus world technique involves gaming the collision ejection routine. In the lower left photo in figure 4.24, you can see that Mario is facing left while standing at the edge of the pipe. If the player directs Mario to jump in a crouched position while facing left, then lands ‘on’ the brick wall at an exact metatile boundary, pressing left can cause collision detection to eject Mario to the right, passing through the wall tiles until he is dropped into the warp zone area. Doing so finds Mario in the warp zone earlier than expected, since he did not run into the area from overhead and properly trigger the scroll lock.

Mario’s faulty collision shortcut is the first key to triggering the minus world. When Mario arrives early, the engine has improperly loaded the warp zone data for world 5-1 (which is normally found in 4-2). However, that warp zone contains only one pipe. Since the warp zone in 1-2 has three pipes, the far left and right pipes are set to warp to world 36-1. The reason why erroneous data is loaded in the warp zone area is too complex to discuss here, but where the pipe sends Mario is a result of the world number variable being set by an arbitrary data byte. As doppelganger explains, the world level is improperly set to $01:

When the value 36/$24 is used as the world number, this code uses the number $24 as the offset, and gets the number $33, which tells it to look exactly $33 bytes past the start of where the area address offsets, which happens to have the byte value $01. The problem is that the look-up table it uses to find the addresses to its level and enemy data is only 36, or $24 bytes long, and $33 bytes past the starting byte is actually outside of the area offset lookup table, which puts it in the enemy data address table. The value $01, which is equivalent to world 2-2/7-2, is actually retrieved from data it’s not supposed to access!

In short, when Mario ducks down a pipe that rightfully should not be there, the engine seeks the current level to load in a data area beyond its intended lookup boundary.

*SMB*’s use of metatile-encoding to store its level layouts combined with 6502 assembly’s byte-based mnemonics means that level data can be loaded from arbitrary
locations in PRG-ROM. Consider the following short code snippet:

```
LDA #$00
STA $0775
```

As discussed in chapter 1, each line of assembly has two parts: the instruction and a value/memory location. The code above loads the accumulator with the value $00, then stores that value at memory location $0775. The alphabetical characters mean nothing to a CPU. Prior to execution, the above code must be assembled—i.e., translated—into a raw byte stream that the 2A03 can understand. Since each three-letter mnemonic has a single byte equivalent, when the above code snippet is assembled, it reads:

```
A9 00
8D 75 07
```

Again, the text mnemonics are simply for the programmer’s benefit; they make the code more legible. The CPU simply reads bytes. And whether the CPU interprets a byte as instruction or data depends on the context. In the given context, the code above is meant to load a value and store it somewhere. However, if the program executed a subroutine that jumped to the code above and read the five bytes as music data, for instance, the assembled bytes would not perform the same load and store instructions. Instead, A9 to 07 would be treated as five byte values meant to be relayed to the music engine.

This agnosticism toward data and instructions is precisely why SMB’s engine can ‘generate’ level layouts beyond the thirty-two that the Mario team designed. When the subroutine that fetches metatiles for a given level is manipulated to point toward an arbitrary location in PRG-ROM, either by accident or through player intervention, the engine can read instruction bytes as area object data. When glitches occur, that flexibility is also its curse.

In the FDS version of SMB, for example, entering a minus world pipe does not lead to the same endless water level found in the cartridge version. The results are far more bizarre. Mario begins above ground in a mushroom cap level completely submerged in water. Floating Princess sprites and gray Bowsers dot the landscape, which is colored in the green and pink hues of an underwater stage. And unlike the cart version, World −1 ends with a modified staircase and a flagpole rather than an endless loop. Players may actually proceed further into the minus world, to −2 and −3. The conclusion of the latter leads Mario to an empty Bowser bridge and a congratulation message, absent the Princess. The game then resets to the title screen and permits level selection, as if the player had completed the game.

The discrepancy between minus worlds lies in the code layout. Data on cartridge and
disk are arranged according to different specifications based on the hardware. PRG- and CHR-ROM on cartridges are stored in separate ICs while disks store their data serially, along with the various headers, gaps, and start bits the Disk Player requires to read that data. As a result, the offset fed to the warp pipe in 1-2 will land on a different data byte, pointing to an alternate byte block for the engine to interpret as area and enemy object data. In short, when traversing the minus world, media matters.

**The End of NROM**

*Super Mario Bros.* exhausted the limits of the stock Famicom cartridge. Miyamoto and team packed its 40KB to the brim with only a few bytes to spare. Even they knew they had pushed the cart as far as it could go, as they were simultaneously working on *The Legend of Zelda*, one of the first titles for Nintendo’s upcoming Family Computer Disk System. And though both games shared the same team, they did not share the same gameplay style. *Super Mario Bros.* had relentless forward momentum, pushing the player along a single plane from left to right, lest the timer expire. Once the player pressed Start, there was nothing but blue skies ahead.

*The Legend of Zelda* was more ambiguous. Its opening screen placed the player in the center of the screen with four possible paths to choose: north, east, west, and into the hollow mouth of a cave. There were no cues to suggest which path was correct. And as the player passed beyond that opening screen, they became aware of an expansive and mysterious world awaiting exploration. Forests, lakes, sands, and mountains hid entrances to labyrinthine dungeons filled with dangerous creatures. Hyrule offered the same mystery and enchantment as the Mushroom Kingdom, but each world’s secrets were revealed through radically different means.

And all of this was afforded by the disk, which extended the depth and density of videogame worlds beyond what contemporary carts could offer. *The Legend of Zelda* could not have been made on an NROM.

---

¹ Ebert, “Why did the chicken cross the genders?”
² Ebert, “Okay, kids, play on my lawn.”
³ Humble, “Game Rules as Art.”
⁴ The 2012 Smithsonian exhibition on the art of videogames is granting videogame fans the institutional validation they seem to desire.
⁵ See Bogost, *How to Do Things with Video Games*.
⁶ Miyamoto recalls that the team included about ‘7-8’ members. See W., “Interview with Shigeru Miyamoto Volumes 1 and 2.”
⁷ In most cases, I will opt for the ‘platformer’ name in an effort to distinguish the genre from a general computational platform.
In Mario Bros., the turtles are called ‘Shellcreepers,’ while in future Mario games they are called ‘Koopa Troopas.’

“Platform game,” Wikipedia.

See the ‘Manuals’ section of MatoTree, “Legends of Localization: Super Mario Bros.”

See ‘Names’ in MatoTree, “Legends of Localization: Super Mario Bros.”

See Papp, Anime and its Roots in Early Japanese Monster Art.

Nintendo of Europe GmbH, “Iwata Asks: Volume 8 - Flipnote Studio - An Animation Class.”

Ibid.

W., et al., “Mario in Japan.”

W., et al., “Super Mario Bros. - From Japanese to English.”

See ‘Names’ in MatoTree, “Legends of Localization: Super Mario Bros.”


The mushroom-induced size change was originally attributed to Alice in Wonderland (Sheff, Game Over, 51; Kohler, Power-Up, 58), but Miyamoto has recently said that there was no overt influence from the Lewis Carroll tale. See Nintendo of America Inc., “Iwata Asks - New Super Mario Bros: Volume 1,” Section 4.

Gifford, “Super Mario Bros. 25th: Miyamoto Reveals All.”


Chaplin and Ruby, Smartbomb, 66-9. The same childhood landscape would inspire the caves of Hyrule in The Legend of Zelda.


Snider, “Q&A: 'Mario' creator Shigeru Miyamoto.”

RLE is just one of many schemes suitable for metatile compression. There is no universal ‘best fit’ for all game engines. Compression schemes are chosen based on the needs of each particular videogame. If necessary, programmers may employ multiple compression techniques in a game.

‘Area objects’ and ‘enemy objects’ are labels adopted by the author and coined by doppelganger, not official source comments from Nagako, Miyamoto, or the rest of the SMB team.

See ‘Object-oriented programming,’ Wikipedia.

Unless otherwise noted, terminology and source code excerpts from Super Mario Bros. are drawn from doppelganger, “SMBDIS.ASM - A Comprehensive Super Mario Bros. Disassembly.” Specific code segments are cited with the line numbers from the disassembly. I also adopt many of doppelganger’s naming conventions for objects, subroutines, variables, etc. to maintain consistency for readers.

Galoob, Game Genie Programming Manual and Codebook, 141.


Certain levels share identical area object data: 1-4/6-4, 1-3/5-3, 2-2/7-2, 2-4/5-4, and 2-3/7-3. However, their enemy object placement varies.


See ‘Nibble,’ Wikipedia.

doppelganger, “SMB disassembly.” Though the tables and illustrations are mine, I am indebted to doppelganger’s correspondence. He explained in exhaustive detail how the byte encoding worked.

doppelganger, “SMBDIS.ASM,” line 3472-3529

The perception of left to right or right to left depends on the frame of reference. One could say that the background tiles appear to shift right to left, but that is technologically incorrect. Background tiles cannot move. Instead, the PPU’s scroll register is incremented to update a shifting camera that pans over the game’s name tables. To simplify matters, we can say that from the perspective of
Mario, the player, and the camera, the game scrolls left to right.


"Michael, “Videogame facts that blow your mind (SuperMarioBros. SHOCKING SECRET INSIDE p #70).” Note that the forum post no longer hosts the original cloud/bush image.

"My thanks to Damian Yerrick for explaining the technical details (and providing the name) for this PPU rendering quirk.

"Note that future iterations of Mega Man dropped score tracking.

"Webster, “Mega Man 9 to feature intentional, optional glitches.”

"All images on a television screen are technically two-dimensional, but 2D in this sense means that character movement is restricted to the x- and y-axes. 3D gameplay introduced depth along the z-axis. In recent years, spatial hybrids have appeared, for instance, introducing multiple 2D planes along the z-axis (permitting objects to jump ‘into’ the screen) or restricting 3D object movement to 2D coordinates.

"See Chapter 1, ‘Sprite Nouns and Adjectives’ for more on sprite registration.


"Nintendo Power’s inaugural issue outlines the 3-1 trick in the ‘Counselor’s Corner’ section. They tell players, ‘You may want to stop building lives at around 100. If you get too greedy, the program has a built-in “Game Over.”’ Nintendo of America, Inc., ‘Counselor’s Corner - Super Mario Bros.’


"Also see Kaluszka, “How the Super Mario Bros. extra lives system works (I think).”


"doppelganger, “the minus world explained v2.0.” doppelganger’s document, at over 4000 words, is the most detailed description of the minus world behavior, based on his thorough disassembly of the SMB source.
5: Tool-Assisted

SIM-U-LATE vt: to pretend, feign.
EM-U-LATE vt: to equal.
- Hewlett-Packard Journal (October 1980)

Emulators herald the end of the era of the proprietary video game console because they render such dedicated gaming boxes technically superfluous. Emulation programs improve, PC hardware technology advances relentlessly — and the notion that games must be played on the console hardware for which they were developed is becoming as antiquated as an old Atari game system.
- Howard Wen, Salon.com (1999)

Since the early 1990s, a special subculture of play, called speedruns, has pushed the limits of videogame skill, performance, and technical mastery. The aim of the speedrun is to play a game as quickly as possible, by any means possible, short of cheating, passwords, or other ‘non-diegetic’ exploits. Games that might take an average player tens of hours are reduced to an hour or less, often at the highest possible difficulty. Notoriously challenging NES games like Contra or Ninja Gaiden are completed in mere minutes.¹

In 2003, speedrunner ‘Morimoto’ posted a virtuosic eleven-minute run of Super Mario Bros. 3,² executing a flawless demonstration of Mario’s athletic skills. And he did so with style—during the airship portions of the game, where the scroll speed is fixed, he danced around the screen effortlessly, weaving through cannon fire, shifting platforms, and spinning wrenches. Of course, the run was too perfect to be true—Morimoto used the Famtasia NES emulator to seamlessly splice small segments of gameplay into a near-perfect performance.³

The discussion surrounding the ‘falsity’ of Morimoto’s run publicized the long-running philosophical rift within the speedrun community: the purists on one hand and those assembling tool-assisted speedruns (or TAS) on the other. Though the ‘tool-assisted’ moniker, like ‘speedrun,’ originated among DOOM’s high-level players, it quickly flourished in console emulation communities, describing any run that leveraged the features of an emulator to complete games in the fastest possible time.⁴ With newfound software assistance, the TAS opened up exciting possibilities for speed improvement. One of the first websites to host such performances, TASVideos, describes the practice succinctly in their

1
site header: 'Tool-assisted game movies. When human skills are just not enough.'

Compare their motto to that of the long-standing ‘unassisted’ speedrun community site, Speed Demos Archive (SDA): ‘Playing through games quickly, skillfully, and legitimately.’ SDA does not allow the use of emulators, primarily due to their ability to slow games to individual frames, manipulate input piecemeal, then replay the performance segments at normal speeds. Another factor is technological consistency, since ‘most emulators and virtualization programs have minor inaccuracies in timing and slowdown that prevent accurate comparisons between runs.’ Unassisted speedruns are akin to rigorous athletic or musical performances, structured around exhaustive practice, minor tweaks to form and execution, and subtle aesthetic flourishes. Competitors will undergo intensive practice regimes, playing games hours a day for months or years to work up a record speedrun. Improving the world-record run for popular contested titles is like Olympic competition in the 100m dash—a game of tenths of seconds. The fastest Super Mario Bros. run has improved by a single second since April 10, 2007. These acute thresholds of performance require standards to ensure fairness for all competitors.

The TAS, in contrast, is less about the perfection of physical performance and reflex than it is about entertainment, propelled by clever hacking, programming, and meticulous technological research. Games are run and re-run between thousands of manual save states until a perfect sequence of key presses is achieved. Code and hardware are scrutinized to reveal any exploits, glitches, or programmer errata that might improve the assembled run. Often physically impossible key presses (e.g., up and down simultaneously) or ‘luck manipulation’—understanding the underlying algorithms that manifest as ‘random’ events, like beneficial item drops—produce speed gains achievable solely beyond the realm of human skill. Performance times are no longer measured in human scale but in microprocessor scale, down to the individual rendering frame. TASVideos encourages movie-makers to, ‘Probe the game. Try, observe, and learn how it calculates things, and use the data to your advantage. Remember, it is only a computer program, and computer programs are predictable.’ At this level, there is an unparalleled intimacy of play between human and machine. Together, tool and flesh choreograph an elegant mastery of code.

TASVideos acknowledges the accuracy concerns underlying SDA’s prohibition of emulators by restricting its movies to an approved list of emulators. Presently, the site condones variations of FCEU, an emulator that strikes a balance between accuracy, community popularity, and useful features. A TAS is about the mastery and manipulation of a game’s rules, not the exploitation of an emulator or a deliberate alteration of the game’s source. Traditional and tool-assisted runs are similar in this regard: glitches are acceptable since they exist within the game’s designed rule set (whether explicitly intended by the game’s programmers or not), but cheat codes are generally verboten. An equivalent board game analogy for Monopoly would be using a consensual house rule like stacking fine
money in a central pot (OK) vs. photocopying your own bills to slip into your personal cash supply (not OK). Ethical regulations like these date back to the original Doom speed demos, which allowed external tools like slow motion or segmented recording, but not the in-game ‘godmode.’ In practice, it is a rather fine-grained distinction. According to SDA:

Using glitches is simply trying to use whatever is within the rules of the game to your advantage. When you use a cheat device or outside alteration, then you’re breaking the game’s rules. As for cheat codes and debug codes, they differ from glitches in being intentionally programmed, so they are naturally outside the rules of the game as defined by the designers.

Compare this to TAS’s more flexible policy:

If the key sequence is mentioned in the manual as a normal means of playing, it is (usually) allowed. Additionally, continues used in arcade games bought through the use of coins is considered similar to a cheat code, as it provides advantage to the player, and goes against the typical concept of a TAS. These rules are not strict, but are motivated by the same concept as the guideline that says you should play on the hardest difficulty.

The guideline referenced above emphasizes entertainment, an important distinction between traditional and tool-assisted speedruns. The TAS is meant to be compelling to watch. Even if a game is popular or challenging, a rote execution of skill might be boring for the viewer. Though speed is the preeminent concern, it may be sacrificed in exchange for entertainment. The governing guideline is to keep it interesting. And ‘guideline’ is the accurate word, since all of TASVideo’s suggestions for producing entertaining movies are cordoned to their own section, separate from the more stringent rules. For SDA, speed is king and that overarching mandate is stated clearly in their rules: ‘We only publish the fastest runs submitted to us. Players are expected to use every method at their disposal, including glitches, to minimize time; side issues such as entertainment are secondary.’

The speedrun has now gained widespread recognition among gaming communities, so much so that speedruns are often built into videogames as supplementary challenges. This runs the gamut from multi-million blockbusters like Grand Theft Auto IV, which includes an achievement for completing all story missions under thirty hours, to independent titles like Braid, which includes an achievement called ‘Speed Run’ for completing the game in less than 45 minutes. There were precedents for speedy play before ‘speedrun’ was coined—Metroid’s best ending was only unlocked by completing the game under an hour—but it is now established as a legitimate means of play external to any in-game reward. A side benefit has been a renewed interest in legacy games and a concomitant rise in the accuracy and features of emulators for obsolete systems.
But before we delve further into contemporary Famicom/NES emulation, we need to rewind several decades to the birth of emulation and track its influence on the present-day definition of the term.

**The Conversion Problem**

In the 1950s, IBM was in a transitional period. Rapid developments in tube and transistor technologies were antiquating their electro-mechanical punched-card machines, then widely used for military, business, and scientific applications. But obsolescence was born from within: IBM was cannibalizing the ‘old-fashioned’ data processing industry they’d helped create with a new, modern successor, the computer industry.

IBM’s growth in the 1940s and 50s was overwhelming. Their initial line of vacuum tube computers were so popular and technology was moving so fast that they were continually churning out hardware revisions to keep pace with both market demand and engineering breakthroughs. Unrelenting innovation was great for new buyers, but it left existing customers in the lurch. Early computers were custom-configured to both their price segments (e.g., high-, mid-, low-cost) and their industry-specific applications (e.g., military, business, scientific), with no system of standardization in place to transition between old and new machines. As IBM stretched out to wider markets, product diversity became unwieldy, while existing customers found it difficult to upgrade to better computers absent any compatibility solution. Imagine spending hundreds of thousands of dollars on a custom-configured computer, adapting your business to its unique idiosyncrasies and resource demands—including new staff to program, run, and maintain it—then repeating that process from scratch when a newer model came along. What would compel you to upgrade? Or worse, compel you not to switch to a competitor’s product? IBM knew this was a serious challenge to their long-term growth.³² They had to cater to the conflicting demands of established customers on one hand and rapid innovation on the other.

By the early 1960s, IBM had reorganized internally to reflect its diverging interests: its punched-card and early vacuum tube machines were cordoned into the General Products Division (GPD), while its more recent (and future) computers fell within the Data Systems Division (DSD). Though there was frequent cross-pollination of individual engineers, each group was responsible for its own profit quotas. Naturally, rivalries formed around competing ideologies: the GPD reflected IBM’s heritage, along with the bulk of its profits, while the DSD represented its uncertain future—a future meant to obsolete and replace the products of the GPD. As new or revised products were introduced independently in each division, internal schisms increased, with no clear solution to the diversification problem.
In late 1961, IBM established a task force dubbed SPREAD (Systems Programming, Research, Engineering, And Development) to stem inter-divisional conflict and propose a new, ambitious goal for their corporate future. Members of both divisions were tapped to outline a plan for a company-wide, unified line of computers. The market was shifting and IBM had to adapt or otherwise lose their competitive edge. Adaptation meant a more flexible, unified computer architecture:

With each processor a member of a graded, compatible line, processor capabilities could not cater to a particular application class. A looming question was whether the observed differences between business and scientific applications truly demanded differing processor instruction sets for cost-effective performance. Contemporary products...while still manifesting either a business or scientific emphasis, were tending to blur some of the distinctions evident in earlier products. Customers increasingly seemed inclined to serve both an accounting office and an engineering department with a single computer facility.

SPREAD’s work was prescient. They tackled not only the pressing economic realities of IBM’s long-term growth, but a number of fundamental computational problems that we now take for granted, from the use of a stack architecture to the bit length necessary to encode characters for a full alphanumeric set (they settled wisely on eight bits versus six). In the end, SPREAD drafted a proposal that outlined their recommendations for IBM’s NPL (New Product Line) alongside its potential strengths and weaknesses.

The resulting product family, the System/360, was introduced in 1964. True to the foresight of the SPREAD group, a large part of the NPL’s eventual success (and long-term legacy) was program compatibility across all System/360 models. Of course, compatibility was only useful for customers once they had bought into the new product family. SPREAD’s report advised about a serious complication, the so-called ‘conversion problem’—a need to translate programs from old machines to the newer, now radically different, architecture. Conversion was not only time-consuming, costly, and error-prone, but it had to be done without grinding their customers’ businesses to a halt. Customers that now relied on computers for their day-to-day data processing could not cease operations while a new system was put in place. Likewise, it was not reasonable to expect customers to jettison the sizable investments they had already made in programs and programming costs.

The conversion problem was not new. Its specter had haunted IBM’s engineers for several years prior to System/360’s introduction, with no apparent solution. A number of tactics were proposed. One avenue was semi-automatic conversion, wherein software would perform an instruction-by-instruction translation from the old architecture to the new, which would then be pruned and edited by a human programmer. This turned out to be optimistic vaporware—no one had actually built such a program. And beyond the
challenges of devising and implementing it, the number of personnel hours necessary to provide support and documentation were quickly deemed impractical.

Simulation was another avenue. The process was essentially mimetic—the simulator machine stored a program that mirrored both the functions and components of the simulated machine:

The first processor’s memory contained not only the simulator program but also areas used to represent registers and memory of the other. The area representing the other processor’s memory was initially loaded with the application program. Then one of the simulator’s subroutines would fetch an instruction from simulated memory, analyze the instruction, load simulated registers, and then branch to another subroutine designed to simulate execution of the given instructions.¹⁶

Though simulation was a proven conversion technique, it was unreasonably slow. Simulation was not a one-to-one translation, but one-to-many. The simulation program not only allotted resources to imitate the target’s memory area, but also the simulation program itself. Executing a single instruction on the simulated machine resulted in multiple instructions running on the simulator (the ‘fetch, analyze, load, branch’ steps quoted above). Any multiplication of necessary instructions meant a subsequent increase in execution time. Simulating computers with similar architectures could mitigate instruction proliferation, but even in the best case scenario, simulated programs ran forty times slower.¹⁷ In the early 1960s, the net effect of simulation’s performance hit was measured in minutes rather than microseconds.

In summer 1963, engineers Stuart G. Tucker and Larry M. Moss devised a clever solution to the conversion problem. A late addition to the NPL spec granted its hardware some excess space in the control store, a high-speed area of read-only memory dedicated to running microcode. Microcode is different from the source code we typically associate with programming, typically the domain of high-level languages like C++, Perl, Lua, or Java. The term microcode was given its prefix due to its close relationship to the microprocessor, at a scale of intimacy beyond even low-level languages like assembly. The control store was devised in the early 1950s as a means to implement circuit-level control of the CPU—logic gates, voltage fluctuations, etc.—in a more flexible and programmable memory store. Writing routines in the control store was called microprogramming—similar to programming, but at a more fundamental level.¹⁸ In contemporary lingo, the control store is akin to firmware. In other words, an iPhone developer coding an app in Objective-C will (or should) never touch the firmware, as it alters the fundamental functions of the CPU itself. Such tasks are best left to Apple’s hardware engineers.¹⁹

Tucker and Moss used the NPL’s expanded control store to implement several dozen microprogrammed instructions customized for conversion. Their microcode, running in
conjunction with a simulator and a few specialized circuits, decreased conversion speeds dramatically. So dramatically, in fact, that their emulation was practically indistinguishable from a program running on the original machine, save for the fact that it often ran faster on the new host. The engineering team’s ‘firmware upgrade’ effectively updated a legacy program’s performance to levels beyond its native hardware. The novelty of reversing the conversion problem was not lost on Tucker and Moss:

[They] deemed the new combination of software, microcode, and hardware sufficiently different from a conventional simulator to merit a new name. [Moss] suggested emulator, a word whose root goes beyond the notion of imitate to embrace “equal” or “excel.” An emulator came to consist of two entities, a software part and a processor part (the latter being microprograms and special circuits) called the compatibility feature.²⁰

Tucker published his and Moss’s work in the 1965 article, ‘Emulation of Large Systems.’²¹ There he outlined the previous attempts at conversion, from simulation and translation (i.e., automatic conversion) to reprogramming, what we would now call porting. And each, in turn, was dismissed for its inefficiencies, though emulation turned out to be the best solution to the conversion problem primarily because it cherry-picked the best bits of prior failures. Simulation, for instance, was used as a hardware ‘diagnostic’ step, highlighting any trouble spots that might hamper performance. Those wrinkles were ironed out manually, with specialized microcode or hard-wired ‘transistor logic.’ In other words, there were elements of automated machine analysis and human revision working in unison at all levels, from circuits to the control store to programmed software.

Four points are important to emphasize in the early history of emulation: first, that the original conception of an emulator was a hybrid solution, involving both software and hardware, that harnessed the advantages of previous unsuccessful strategies; second, that emulation differed from simulation by exceeding the simulated source; third, that emulation was provided by IBM for its own hardware, not its competitors; and fourth, that emulation was not devised in the PC era. Emulation developed concurrently with the computer industry, in the era of ‘large systems,’ and was in fact essential for its continued growth.

Machines That Do Not (Yet) Exist

Today, in the context of videogames, we think of emulators as software-based solutions. Nestopia or Nintendulator do not require custom-soldered transistors or special graphics cards to emulate NES games. They run alongside other applications, like a word processor or web browser, but load ROM images rather than text or HTML files. Likewise, emulators
are not constrained to a particular operating system or even PCs. NES emulators run on nearly every operating system since the mid-1990s, from MS-DOS to Linux, and most consoles, from Sony’s PSP to Nintendo’s own Virtual Console on the Wii.

Despite this popular perception of emulators, the hardware/software emulators of the past have not disappeared, nor are they relegated to arcane embedded systems or vintage mainframes. ‘Hybrid’ emulators are still prevalent in both professional development and videogame enthusiast communities. But before we discuss the persistence of ‘true’ emulators, we should ask how the notion of software-only emulators came to be.

We need not look far—Tucker was already hinting at the possibility of pure microcode emulation only a few paragraphs after he coined the term. He wrote, ‘Although all the functions could be handled with only microprogramming, a significant speed advantage is gained by adding some transistor logic.’ Conceptually, the idea for software-only emulation was there; IBM’s engineers were primarily obstructed by logjams in processing speed. Until CPUs became faster, dedicated transistors would prove more capable.

Moore’s Law took care of the problem. In 1969, computer scientist Robert Rosin published an article on emulation that divested the term’s definition of any reliance on supplementary hardware:

[W]e use the term “emulator” to describe a complete set of microprograms which, when embedded in a control store, define a machine. We shall call a machine which is realized by an emulator a “virtual machine” and the machine which supports microprograms a “host machine.”

Rosin’s footnote to this quote specifically mentions Tucker’s earlier paper but points out that the latter’s definition is ‘reflected in the products of the IBM Corporation,’ while ‘several other manufacturers appear to use the term as defined in the present paper.’ Rosin’s conceptual divergence had two key implications: First, it reframed emulation as a theoretical and practical concern outside the scope of a single corporation. Rosin was an academic rather than an IBM employee, so his research was not geared specifically to the success of a product line. Second, it abstracted the native platform from its roots in circuits and registers to a machine defined in code. Of course emulation in the late 1960s was still in the realm of microcode rather than high-level languages, but the conceptual break was already taking place.

As ‘virtual machines,’ emulators could take on all the connotations of the term: idealized, practical, imitative, ‘close enough.’ Even immaterial. A virtual machine was an abstracted machine; hardware could be represented in code prior to its physical instantiation. In practical terms, a programmer could configure, test, and modify a
computer that had not yet been built. In 1978, Marsland and Demco published an article on the PDP-11’s (a microcomputer popular for educational, scientific, and business use) potential as a ‘universal’ emulator that could support multiple architectures. Besides the obvious advantage of running programs from several legacy mainframes on a single host, they listed a number of other appropriate cases for the use of emulation. Among them:

(1) The configuration of the target machine is too small for software development. During such development the emulator could provide extra assistance in the form of better debugging aids, larger virtual machine, and access to the host’s peripherals.

(2) The target machine does not (yet) exist.

These two use cases are particularly pertinent to the development of videogame hardware, especially in the Famicom era.

When new consoles are in development, they begin as a set of specifications, usually built to suit a specific software profile. In Nintendo’s case, they aimed to build a console capable of playing Donkey Kong (see chapter 1). As the realities of cost, speed, and industrial design came to the fore, concessions had to be made: the Z80 core was swapped for a 6502, controllers were connected directly to the board, the keyboard peripheral was dropped, and so on. Meanwhile, software had to be developed while the hardware design was still in progress. Otherwise, no games would have been available for console launch. Early photos of the Famicom prototype show bare metal chassis, one each for the CPU and PPU, slotted with large circuit boards (figure 5.1).

5.1 The hardware prototype of the Family Computer, with the CPU module on the left (a) and PPU module on the right (b). (Source: Nikkei Electronics)
When Teiser and his fellow Atari executives visited Nintendo, they did not even see the skeletal frame pictured above, since Nintendo ‘had only just received their 1st pass silicon (with some bugs) and were not able to show us a fully assembled and working prototype.’²⁴ Instead, early versions of Donkey Kong Jr. and Popeye were demoed ‘on their TTL emulator.’ In other words, Famicom emulation preceded the actual Famicom.

Consistent Marsland and Demco’s first point, the memory constraints of consoles necessitated software development on separate systems. Today, most consumer PC software is developed on the same hardware that runs it. Mac developers use Macs to develop software that will run on other Macs. The same applies for Windows or Linux developers.²⁵ Console development requires a different strategy. Typically, development studios license ‘dev kits,’ specialized hardware whose architecture, in concept, mimics the Tucker-style emulator. Consoles of the 1980s had no operating system, firmware, or BIOS.²⁶ Videogames were not developed on the consoles that played them. The Famicom’s processor and RAM were insufficient to support a full suite of development tools.

Early Famicom games, for example, were developed by Nintendo on the NEC PC-8001,²⁷ a popular, Z80-based Japanese PC launched in 1979. Later in the Famicom’s life-cycle, separate systems were used to create a game’s individual assets. Photos from a 1989 Japanese educational text show Nintendo employees working on level layouts, graphics, and code for Super Mario Bros.³³ The graphics designers used Fujitsu FM R-50s, a DOS-based business PC released in 1987, featuring an 8MHz processor, a colorful 640x400 CRT display, and a hard drive. The programmers were relegated to more antiquated fare: the HP 64000 Logic Development System, first introduced in 1980, three years prior to the Famicom’s debut.²⁹ This unique mainframe setup allowed up to six developers to work on a shared network without the constraints of conventional time-sharing systems—each work terminal had its own processor and memory. The HP also included an ‘emulator pod’ with a socket to host interchangeable target processors, including the 6502. Programmers could write, debug, and even download their code to physical ROM—the lower right of each terminal had a ‘PROM personality interface unit,’ a friendly marketing name for a built-in burner.³⁰ They could then socket the ROM into a cartridge and test their code on a Famicom.

Development-based emulators like the HP 64000 were not simply replicating, but also augmenting their target devices. Not equal, but better—a relationship that persists into the modern era of console emulation.

‘The World’s Most Versatile Personal Computer’

Computers became smaller and more affordable in the 1970s and 1980s, thanks in part to low-cost microprocessors like the 6502 and the Z80, innovations in storage media, and
ever-shrinking transistor sizes. The large systems of the 1950s and 1960s, devoted to
government, business, scientific, and academic work, were transforming into the personal
computer, a multi-purpose machine meant for home use. PCs, now affordable for single
individuals or families, were finding their way to millions of homes. In this climate of
unprecedented growth, emulation reached the masses.

Though early PCs were far less bulky and costly than the mainframes of the 1960s—and
subsequently less specifically targeted to corporate or research needs—the conversion
problem did not magically vanish. In fact, the whole history of modern computing, from the
System/360 to the iPad, is so enmeshed with emulation that treating them as concurrent
but separate tracks, with the latter practice cast as an ancillary addendum to their shared
history, makes little sense. The history of digital computing is the history of emulation. The
conversion problem was present at the computer’s birth and persists to the present day.
Though emulation most often functions as a silent, imperceptible partner, meant to fade
into the background while its processes seamlessly stitch together the old and the new, it
surfaces acutely during transitional shifts between dominant platforms (e.g., DOS to
Windows) or entrenched software packages (e.g., the corporate world’s heavy investment
in Microsoft Office). The early history of personal computing is rife with the legacy of
competing platforms, with each transition of market dominance accompanied by new
approaches to the ubiquitous conversion problem.

For the majority of today’s computer users, the operating system (OS) is their closest
interaction with hardware. The OS abstracts low-level functions—file systems, input/
output, memory management—into a user-friendly interface, like windowed views, the
desktop, folders, and icons. Early operating systems had few of these modern
accoutrements, as they were typically text-only, but like their contemporary ancestors they
did provide a consistent user interface across multiple machines. They also provided
interoperability between diverse microprocessors and peripherals, which were not
designed for easy plug-and-play. Custom disk operating systems (or DOS), for instance,
were necessary to make external storage hardware talk to the CPU. As more manufacturers
introduced inoperable hardware targeting a specific chipset, the need for more flexible and
generic control software arose.

Gary Kildall first programmed CP/M (originally ‘Control Program/Monitor’ but later
‘Control Program for Micros’) for the Intel 8080 microprocessor (on a System/360 no less)
as a means to interface with the floppy disk drive, a recent innovation designed for fast,
compact data storage and retrieval in large systems.²¹ Kildall realized that a floppy’s
random-access storage was more efficient than linear cassette or paper tape systems, so he
developed his DOS to usher the peripheral into the microcomputer fold. Computer
hobbyists quickly embraced CP/M after its 1976 introduction, so much so that Kildall and
his wife formed a software company around its continued development.²² As its popularity
grew and Kildall was asked to adapt it to other machines, he decided to rewrite the source as a generic control system that could be custom-fit to multiple microprocessors—what he called the BIOS, or ‘Basic Input/Output System.’ Developing for the 8080 turned out to have a happy side effect: since the ubiquitous Z80 could execute 8080 code, CP/M was suddenly compatible with a broad range of microcomputers. Thanks to its flexibility, low cost, and ease of use, CP/M soon dominated the microprocessor market—and helped spur its growth. CP/M appeared on the Altair 8800, IMSAI 8080, Sinclair ZX, Osborne 1, and MSX, among other computers. Thanks in part to the floppy disk and its capable DOS, programmers were suddenly able to develop and distribute software targeting a single operating system rather than a single machine, a move toward abstraction that helped kickstart the commercial PC software industry. As a result, popular and influential programs like WordStar, dBase, AutoCAD, and Turbo Pascal all debuted on the CP/M operating system.³³

In 1977, Apple introduced the Apple II personal computer. A coordinated effort between Steve Wozniak’s engineering skill and Steve Jobs’s design and marketing prowess, the Apple II was one of the first machines advertised outside of computer hobbyist circles. Its rounded beige case, color graphics capabilities, inexpensive floppy drives, and ample expansion bays appealed to both mainstream consumers and computer enthusiasts alike.³⁴ However, Apple faced a significant roadblock to widespread success: its 6502 core posed a new conversion problem for a business market newly entrenched in the Z80-compatible CP/M DOS. Echoing the sentiments faced by IBM’s mainframe customers in the 1960s, business clients were reticent to abandon their software investments in favor of a sleeker, but less useful, machine. If there were a simple means to tap into the wealth of existing software, it could ease the leap from CP/M micros to the Apple II.

Ironically, a hardware emulation solution arrived from Apple’s future competitor, Microsoft. Their Z80 SoftCard, released in 1980, was an expansion board with a built-in Zilog CPU, bundled with licensed copies of CP/M 2.2, Microsoft BASIC-80, GBASIC, and a handful of interoperability utilities.³⁵ Though the term emulation was never used, an advertisement for SoftCard (figure 5.2) touted its ability to, ‘Turn your Apple into the world’s most versatile personal computer.’ In fact, you would now have ‘two computers. A Z-80 and a 6502.’³⁶
An advertisement for Microsoft’s SoftCard. (Source: apple2info.net)

The underlying technical implementation was not the marketing focus. Users cared more about the resulting compatibility than exact terminology. But this was precisely the type of
emulation outlined in Tucker’s original paper, a marriage of hardware and software, reflected in the product’s name.

By the early 1980s, IBM had cast its lot into the personal computer race with their first line of PCs: the 5150, XT, and AT. Microsoft built their MS-DOS operating system—either influenced by or infringing on CP/M, depending on the source—specifically for IBM. But again, CP/M’s software library loomed large, potentially halting the transition of the substantial business microcomputer market to new PCs. This time, however, Microsoft did not provide their own emulation solution.

In 1984, Sydex released the 22Nice CP/M emulator for MS-DOS, expanding the ‘universality’ of IBM’s PCs and, ironically, tolling the death knell for CP/M. 22Nice was meant to make the transition between the native OS and emulation seamless, so that ‘the user is unaware that a program is CP/M- or DOS-based.’ Like the SoftCard’s on-board Z80, 22Nice could leverage a hardware component called the NEC V20 plug-in. However, the addition was optional. 22Nice could emulate CP/M solely in software, a first for OS emulation. According to Sydex’s documentation, the hardware supplement had a familiar benefit: ‘much greater speed.’

With the close visual and functional similarities between MS-DOS and CP/M and the seamless software experience provided by Sydex, the transition to IBM PCs was relatively painless. Businesses could transparently run WordStar alongside Lotus 1-2-3. As a result, IBM’s platform wrested control from Apple and Commodore (in the US) and positioned Microsoft as the dominant software and OS developer. IBM’s preeminence soon fused the shorthand ‘PC’ to a distinct brand identity, transforming the generic name for all home computers into a term describing those manufactured by a specific group of vendors.

The evolution of software emulation mirrored the cycles of the PC market. As new competitors emerged, so did the conversion problem. How could a hardware manufacturer transition a customer base entrenched in a competitor’s OS ecosystem without invalidating those customers’ software investments? PCs were still costly and novel enough to make customers wary to upgrade without a clear transition solution. This was especially true in corporate environments, where an upgrade could mean thousands of machines, millions of dollars, lost productivity, and supplemental training.

A company’s answer to the conversion problem could make or break a new line of hardware. When Commodore announced their Amiga A1000 at a gala event in New York City on July 23, 1985, it looked to be the most advanced personal computer to date. It was a multimedia powerhouse featuring four-channel stereo sound and video modes supporting up to 4096 on-screen colors. Its beefy 7.16 MHz Motorola 68000 CPU and innovative multitasking OS made it formidable business computer as well. On paper, the Amiga’s specs bested its competitors, but raw power could not guarantee market success. Commodore had to carefully manage Amiga’s brand to appeal to multiple demographics. Though their
marketing led with the visual dazzle of the Amiga's graphic capabilities, they followed up with business-oriented applications. A 1986 advertisement featuring an impressive full-page 'digital painting' of King Tut read:

But there’s more to the Amiga than whiz bang graphics. It’s also a top-of-the-line PC ready to solve problems, business and personal. The incredible power that brings Tut back to life can also make gigantic spreadsheets sit up and beg, can make eloquent the wordiest of word-processing tasks, and thanks to the Amiga’s unique multi-tasking capabilities, it can do word processing and spreadsheet analysis and graphics and more simultaneously. ³⁹

The Amiga was lauded by creative professionals but its ultimate success depended on legacy support for IBM PC software. New York Times columnist Peter Lewis remarked a week after the unveiling that Amiga faced a ‘tough challenge in cracking the conservative business market,’ and that its most attractive feature was ‘an optional “I.B.M. emulator” that will allow the Amiga to run most off-the-shelf I.B.M. software, including Lotus 1-2-3, dBase III and Wordstar.’⁴⁰ By November, columnist Sandberg-Diment was still anticipating the emulator that even Commodore could not seem to pin with a production date, writing, ‘Most buyers quite rightly do not list such qualities as superior sound and colorful graphics as must-haves in a business machine.’

The promised emulator was called Amiga Transformer, developed by Simile Research. Though the software was teased at the July event, it was not released until the following year. Commodore bundled Transformer with the Amiga 1020 5.25” floppy drive, hoping to provide an easy conversion solution for IBM customers. The Transformer was an utter disappointment. Its performance was sluggish despite the Amiga’s powerful processor being several orders faster than earlier IBM PCs. As predicted, absent support for popular MS-DOS software, the Amiga struggled to find a foothold in the business market and was relegated to the same niche appeal that plagued Apple and Atari, who could not shake their association with less ‘serious’ pursuits like videogames. Commodore scrambled to shore up Transformer’s failure with a speedier hardware-based emulation solution, the A1060 Sidecar. Released in 1986—but still too late—the chassis housed a full PC XT, which made the peripheral bulky, expensive, and consequently unappealing to consumers.

In the interim between Transformer’s announcement and release, Commodore’s competitor Atari reactively released PC Ditto for their half-priced Atari ST. Technically, this made PC Ditto the first IBM PC emulator to market, but its performance was similarly lackluster, running PC-compatible software slower than their native hardware. Reviewers were cautionary about recommending the emulator, suggesting Atari users opt for hardware-based solutions or otherwise indulge in PC emulation on ‘a casual basis.’⁴¹ Again, without PC compatibility, IBM’s competitors faced an impenetrable business market.
In the home videogame market, the conversion problem took a different cast. In the early 1980s, most manufacturers had only introduced a single console. Or, more precisely, only a single cartridge-based console. Prior to the VCS, for example, Atari had spun Pong into countless hardware variations. Cartridges helped stave off hardware proliferation. It was cheaper to mass produce individual ROMs in plastic cases than to manufacture the games and their hardware in a single package. The transition from single-purpose to cartridge-based consoles, meant to ease both manufacturing costs and consumer investment, developed into a conversion problem known by another name: backwards compatibility.

As new competitors to the VCS arose, the inevitable hardware refresh loomed. As technology marched ever onward, the VCS’s processing limitations were more and more apparent. But its popularity meant consumers were hesitant to move on to a new platform when they had a sizable library on the older machine. As more manufacturers entered the fray, a familiar question surfaced: how will Shiny New Console support the back catalog of Old Antiquated Console?

For many console generational shifts, the answer has been: they won’t. The problem is often dismissed by simply ignoring backwards compatibility. Nintendo in particular has taken that path numerous times. The Super Famicom could not play Famicom games; Nintendo 64 could not play Super Famicom games; GameCube could not play N64 games. Prior to the Wii, Nintendo’s compatibility exceptions were either in the portable domain (e.g. the original DS played GBA software) or in odd lateral hardware support, such as the Super Game Boy, a peripheral that not only supported Game Boy games on the Super Nintendo, but added limited colorization, sound, and two-player functionality.

Part of the reason for lack of backward compatibility was the early consoles’ limited processing capabilities. Though the Super Famicom’s 5A22 microprocessor was capable of up to double the 2A03 CPU’s speed, it still could not fully emulate the NES. The Super Game Boy, which played Game Boy carts on the Super Nintendo, managed its emulation via piggybacked hardware—the full Game Boy CPU was housed in the Super Game Boy cartridge adapter. And Nintendo was not the first to adopt this strategy for videogame consoles. The 1983 version of the VCS successor, the Atari 5200, was revised for compatibility with the CX-55 VCS Cartridge Adapter, an unwieldily bit of kit that allowed 2600 games to play on the newer console. The CX-55 was a reactionary move on Atari’s part, in response to competitors Coleco and Mattel releasing their own 2600-compatible hardware, the Expansion Module No. 1 and the Intellivision System Changer. In a 1983 Mattel catalog, Atari owners were reassured that they could ‘finally upgrade to Intellivision, without leaving all their Atari 2600 cartridges behind.’ Providing backwards compatibility
for one’s rivals was a litigiously risky move, one rarely repeated in console videogame history.\footnote{43}

The Sega Genesis supported compatibility for Master System games with the Power Base Converter. Unlike Atari or Nintendo, Sega benefited from the foresight to build in the Master System’s Z80 and sound processor into the Genesis itself. Similarities in the two consoles’ VDPs allowed the newer system’s processor to stand-in for the elder system’s graphics processor. As a result, the Power Base served primarily as a pass-through device to rectify the cartridge mismatch between Master System games and the Genesis cartridge slot—the heavy lifting was done onboard.

Many of these examples straddled the line between Tucker-style emulation and so-called ‘clone’ systems. Often the host console did not shoulder any of the emulation weight, but acted as the I/O and video processor for a parasitic platform. The Intellivision certainly could not bear the processing burden of emulating the VCS, so its System Changer included a 6507 replica built from ‘off-the-shelf’ components, at least according to Mattel’s counterclaim to Atari’s threat of litigation. Today, consoles are powerful enough to emulate their ancestry in software alone, though they still leverage hardware solutions for certain cases. Nintendo’s Virtual Console on the Wii offers an assortment of NES, Super Nintendo, and N64 titles as well as offerings from its once bitter rivals, Sega and NEC. GameCube support, however, is handled similarly to the Power Base. Initial versions of the PlayStation 3 offered hardware-level support for the PlayStation 1 and 2 library, but later ousted the latter in favor of software emulation—a decision advertised as cost-related but certainly motivated by the bevy of HD ‘remasters’ of PS2 hits that soon followed. Microsoft’s Xbox 360 is the only current generation console to completely opt for software emulation. Microsoft rolled out this support incrementally, picking and choosing backwards compatibility based on the most popular and/or demanded Xbox titles, eventually supporting several hundred games. Their Xbox Live Arcade (XBLA) service has also seen a diverse set of emulated ports spanning console and arcade generations from the Atari VCS to the Sega Dreamcast. Most notably, the aborted Game Room service, a downloadable simulacra of a personalized arcade (including virtual quarters), emulated a host of VCS, Intellivision, and arcade titles. Though contemporary emulation offerings appear to mimic the cross-compatibility tactic of the Expansion Module or System Changer, note that all of these services emulate competitors who are no longer active in hardware manufacturing. In other words, Sega and Atari are fair game for virtual consoles, but you will not find a Nintendo property on a Sony console, or vice-versa.

**The Birth of .NES**

The aim of console emulation is simple: to allow users to play game software from a given
platform with the closest approximation to the original experience as possible. In all cases, accuracy is a key constraint, but never perfect. Emulation is not solely a matter of replicating the target console’s CPU, but also any additional co-processors, I/O devices, lower level instruction sets, and so on. In the Famicom’s case, that means the CPU, PPU, APU, controllers, light gun, and any number of peripherals (e.g. the Family Computer Disk System, Galoob’s Game Genie, etc.). Each of these core and ancillary components are necessary for complete and accurate emulation.

For the needs of most players, low-overhead emulators that play most popular games with reasonable accuracy are acceptable. Small glitches or color inconsistencies are acceptable for (or go unnoticed by) most players, so long as the overall look and feel of the original are intact. For the TAS community, there are more stringent requirements that most players would never notice, like cycle-accurate CPU timing and frame-level control. But higher accuracy comes with a concomitant increase in processor demands, especially if the emulation is purely software-based. Emulation is a constant balancing act between usable tools and allegiance to the source hardware.

Until the 1990s, emulating a console on a personal computer was not a viable option. Most PCs simply did not have the necessary processing power. Those that did were prohibitively expensive. As a computational device, the NES had only one mandate: execute the game code. Every bit of RAM, every byte of storage, every spare clock cycle was rallied for the sole purpose of getting the game on the screen. PCs, in contrast, were multi-modal Renaissance machines meant to run games alongside spreadsheets, documents, email, and web browsers. All processes were obliged to share from a single pool of resources. This discrepancy in purpose meant that consumer PCs were typically a generation or two behind the curve in console emulation. Only the most expensive, cutting-edge machines stand a chance.

Equally troublesome for console emulators was the lack of adequate technical documentation. Developing licensed videogames for Nintendo, Sega, Atari, or Sony required sanctioned development kits. These kits were both costly and laden with legal expectations. American NES developers, for instance, had to abide by Nintendo’s strict censorship code (see chapter 3). Japanese, European, and American developers alike had to submit to Nintendo’s manufacturing and production policies. Any official technical documentation was legally protected, as Nintendo held copyrights over the form and function of their console. Circumventing the licensing track required outright theft or tedious reverse engineering. Even if unlicensed developers could obtain the official Famicom documentation, they still faced a steep language barrier. While some developers enlisted the aid of translators, others opted to trudge through the guts of the NES to figure it out themselves.

The latter tactic was the strategy of most emulation developers. The NES’s operation
had to be sussed out through experimentation. The early NES emulation scene benefited from a community spirit, sharing their meticulous study, homespun documentation, and source code with one another. But many of these coders, being young or otherwise shrouded by online anonymity, were brash, competitive, or outright malicious. Competing emulators were mocked online; ethical debates arose between freeware, shareware, and commercial software advocates; prominent coders left the scene in anger and frustration; source code was copied and even stolen from unprotected hard drives. Emulation development appears less contentious now, as robust NES documentation is freely available online, but in the nascent years of NES emulator development, there was a fragile balance struck between enmity and community.

The first Famicom emulators appeared as early as 1996.⁴⁴ Developer Nobuaki Andou programmed the first publicly-released software, a Japanese shareware program called PasoFamicom (or PasoFami).⁴⁵ Though it is still in active development (and now emulates a number of other consoles), PasoFami failed to gain the widespread popularity of subsequent emulators due to its language barrier, cost, and its complex file structure for game images. PasoFami required a special 'split format' for Famicom games, consisting of four separate files: .PRM (header file), .PRG (data), .CHR (data), and .NAM (game title). Though unwieldy for users, the separate data components did more accurately mirror the physical hardware present in the cartridge and separated supplemental header information from the game’s contents.

PasoFami also required a registration fee of 3000 yen (roughly $30). The unregistered version ran for one minute, then halted the game and flashed a pop-up window asking for registration. In response to PasoFami’s linguistic and monetary barriers, hackers produced patches and cracks to translate the GUI into English and circumvent the ‘nag message’. Threads on the comp.emulators.misc discussion board frequently labeled it ‘crippleware,’ due to its prohibitive timer lockdown. One commenter justified his own and others’ software piracy based on the inconveniences of cost and translation:

There’s a difference between shareware and crippleware, and PasoFami is definitely the latter. And, besides, it’s in Japanese. We can hardly be expected to evaluate a crippled Japanese program in America. THAT’S why you see the cracks, THAT’S why you see the translations. Shareware is software you evaluate before you buy. The demos of PasoFami don’t provide NEARLY enough to evaluate the product. Also, by pirating in the US market, we are not affecting the Japanese market, which is what the emulator was aimed at. So who are we hurting by our rampant piracy of PasoFami in the US? Certainly not Noubbaki [sic], as we’d never have stuck with it at ALL if it weren’t for the cracks and translations. It cost him no sales, and was a form of free advertising.⁴⁶

The worldwide ‘advertising’ was effective—English-language sites hosting PasoFami
commonly included the crack download alongside the retail binary. In retaliation to the rampant piracy, Andou began setting code traps in the source of his official releases, meant to spring when the emulator was patched or otherwise tampered with. He likewise focused exclusively on the Japanese emulation market, requesting that PasoFami be removed from all sites besides his own. But by the time Andou had rejected non-Japanese ‘evaluators,’ several new emulators were appearing, eventually pushing PasoFami to the periphery of worldwide attention.

By fall of 1996, there were at least six new NES emulators either in development or publicly available: Marat Fayzullin’s iNES, Alex Krasivsky’s LandyNES, Mr. Snazz’s VeNES, Y0SHi and Riff’s qNES, TaNdRuM’s dNESe, and Paul Robson’s NESA. iNES, despite its $35 registration fee, emerged as the early leader due to a number of factors. For one, it had first-mover advantage. In the early 1990s, Krasivsky (aka Landy) was working on a Nintendo emulator for MS-DOS that he planned to call interNES, or iNES for short. Early on, Fayzullin stepped in to assist in the development, then, when Landy ‘lost interest in the project,’ he carried on with iNES on his own. In the documentation accompanying an early version of interNES, Fayzullin cited his collaborator’s contributions several times, writing, ‘The original code was written by Alex Krasivsky from Moscow. I added missing CPU commands, wrote screen drivers, and did some thorough hacking to make the emulator run about 85% of games.’ Landy soon shifted his attention to the development of LandyNES, which never came to fruition.

Fayzullin’s interNES arrived at an advantageous time. The emulation community was looking for a simpler alternative to PasoFami. iNES proved to be stable, functional, and well-documented. Fayzullin was one of the first developers to document and share technical details that he, Landy, Robson, Y0SHi, and others in the emulation community had discovered through testing and experimentation with the NES. Fayzullin’s compiled ‘Nintendo Entertainment System Architecture’ is a fascinating snapshot of the state of knowledge about the console in late 1996. In particular, mapper and sound emulation were uncharted territory. Fayzullin provided scant documentation for the four most common mapper types and noted, ‘There are several other mappers, some of them very sophisticated. iNES partially supports them, but as this support either doesn’t work correctly, or the mappers are uncommon (such as 100-in-1 cartridge mapper), I don’t cover them here.’ The ‘Sound’ section simply read, ‘To be written’—fitting, since early Windows builds of iNES had poor sound emulation.

iNES is still in active development, but its popularity has waned in comparison to fuller-featured and less costly (i.e., free) competitors. However, its legacy is still alive in the .NES file extension. Apart from being an emulator, iNES was also a standard format for encoding the data ripped from NES cartridges. Early on, as NES emulators became more popular, the desire to stock one’s hard drive with a full catalog of NES games intensified.
However, stripping a NES cartridge’s data was not as simple as ripping a CD or transferring photos from a flash drive; there was no cartridge slot in the computer that one could plug a game into for quick transfer. ‘Dumping’ a cart’s contents, as it was called, required a hardware cartridge copier to transfer the data. (Early on, the iNES homepage provided the schematics for such a device.) The resulting binary dump was called a ROM image, or simply ROM—‘Read Only Memory’—the portion of a cart’s memory that could be read from but not written to. Though this shorthand was something of a misnomer, since a cart could also contain RAM or the image could be derived from a Famicom disk, the name stuck. ROMs were and are the stock-in-trade of the emulation community.

Of course, raw binary images were not enough. Once dumped, the data then had to be formatted into a file that the emulator could understand. Since NES carts contained a variety of augmentative hardware, emulators could not rely on a one-size-fits-all configuration. Had Nintendo stuck solely to the NROM board, a flat ROM image would have been sufficient. Since that was not the case, Fayzullin devised a straightforward sixteen-byte string to append to the images. This iNES header conveyed a number of crucial hardware descriptions of the dumped cart, such as the mapper number and the cart’s mirroring. Once concatenated, the header and the image were known as .NES files, ready for processing by the iNES emulator.

Though .NES was developed in conjunction with the for-pay iNES, it was by no means proprietary to its host emulator. Other developers built in iNES support in order to accommodate the influx of .NES files circulating online in favor of developing another competing format. Paul Robson, developer of the open-source DOS emulator NESA (and later TNES), explained that he chose the format because:

Most of the ROMs were already in that format and it was documented properly. It was the only sensible choice because of the different mappers – you couldn’t just have a binary dump of the ROMs, you had to have some form of system for saying how it was wired up. There’s umpteen “mappers” for the NES.⁵¹

The first version of NESA released September 1996 (figure 5.3).
Progress in the NES emulation scene moved extraordinarily fast. Developers had to quickly glom to a sensible standard or otherwise see their emulator fizzle into obscurity. Fayzullin had first-mover advantage, coupled with a simple, open format for describing ROMs. Fayzullin’s gracious attitude toward sharing technical information (and reciprocal sharing back to him) meant that the entire emulation community benefited from the iNES format. The snowball effect intensified as subsequent emulators adopted .NES because previous emulators had done so.

The .NES file improved on the PasoFami split format in several ways. It was a single, unified file with a commonsense extension. Users did not have to understand the underlying cartridge architecture that informed the .PRG and .CHR extensions or why they required a header to function properly in an emulator. It made sense to the average computer user that an NES emulator would run .NES files. With the introduction of the iNES format, it became much easier for NES emulation to spread. The hunt for ‘NES ROMZ’ took off and a grassroots network of ROM-hosting sites popped up online. Especially in non-Japanese countries, where PasoFami was near-inscrutable, users shifted their attention to iNES. As its popularity grew, so did the need to support .NES files.
A new conversion problem arose between competing emulator formats. iNES did not support PasoFami’s split images, so users had to develop utilities to translate .PRM to .NES. Initially, Fayzullin posted command line instructions for assembling .NES files manually:

1. Create a 16-byte header:
   "N""E""S"$1A$xx$01$01$00$00$00$00$00$00$00$00$00$00$00$00$00$00$00$00$00$00$00$00$00$00$00$00$00$
   this byte is either $01 for 16kB games or $02 for 32kB games
   and call it, let us say, mario.hdr

2. Do
   cat mario.hdr mario.prg mario.chr > mario.nes
   You have the .NES file now.\(^52\)

Ambitious users could also strip the raw .PRG and .CHR segments from their PasoFami files, use a hex editor to append the appropriate iNES header, and assemble a workable .NES ROM. By the late 1990s, all-in-one utilities like Matt Conte’s cajoNES (‘the only NES ROM tool with balls’) automated the process, permitting conversion from and to PasoFami format, as well as another emerging Japanese format, .FAM.\(^53\)

Enabling the conversion of PasoFami and other incompatible formats to iNES was not simply a means of besting competing emulators. Many of the games available in .PRM, .FAM, or .DKA (Famicom Disk format) were Japanese-exclusive Famicom games. For the majority of Western NES players, emulation was their first avenue to experience a substantial portion of the Nintendo catalog. Importing games posed a series of challenges: differing cart shapes and sizes required either the purchase of a Famicom or 72-pin/60-pin converter; language barriers would cordon off a significant number of text-heavy titles, like Japan’s popular adventure and RPG games; and the time and cost involved in shipping titles across the Pacific would dissuade all but the most persistent or wealthy Famicom fans. Emulation mitigated all of these challenges. Cartridge pinouts were not a problem for ROM images and language barriers would soon be surmounted through ROM hacking efforts. Shipping cost and distance were eliminated by online distribution. The sole physical barrier was getting the Famicom cart dumped and packaged, but this only had to be done once. Of course, this did not solve the problem of Western gamers getting their hands on the carts in order to dump them, thus the importance of leveraging the work that had already been done by the Japanese emulation community. A treasure trove of Famicom games lay locked behind their emulator formats. Conversion tools like cajoNES were the key.

The critical portion of the iNES format was the header, the first sixteen bytes of
an .NES file that precisely described the contents and arrangement of the data to follow. Bytes 0-3 are always the same: $4E, $45, $53, and $1A. These hexadecimal values are the ASCII equivalents of the letters ‘NES’ followed by the ASCII SUB control character, meant to denote an EOF (end-of-file).” Ironically, even as the .NES format lives on in modern operating systems, it still bears a permanent trace of its MS-DOS heritage. Byte 4 describes the size of the cartridge’s PRG-ROM split into 16K blocks, while byte 5 describes the size of CHR-ROM split into 8K chunks. Bytes 6 and 7 are flags whose individual bits denote a number of cartridge features, including the mirroring type, the presence of a special ‘trainer,’⁵⁵ and whether battery-backed SRAM is present. Most importantly, the last four significant bits of bytes 6 and 7 contain, respectively, the lower and upper nibble of the mapper number.

Mapper numbers are a classification system Fayzullin implemented to answer the need for a standardized system to organize and collate the manifold in-cart circuitry used in licensed, unlicensed, pirate, or otherwise bootlegged carts. Emulation aims to support any Famicom/NES/Dendy/clone/pirate cartridges available worldwide, not just those sanctioned by Nintendo. Since a single byte was reserved for the mapper number, there are only 256 available mapper slots. In the 1990s, this was thought to be sufficient to cover all possible mappers, but the continuing global life of Nintendo’s 8-bit console, especially in the bootleg markets of Asia, South America, and Eastern Europe, has led to the continued proliferation of unique mapper hardware.

The original iNES mapper specification organized the numbers by similar board types and function. Fayzullin’s numbering did not follow mapper development chronologically nor adhere to a master list provided by Nintendo (since there was none). Thus, for example, MMC3 (mapper #4) and MMC5 (#5) appear sequentially before MMC2 (#9) and MMC4 (#10). Likewise, variations within single board profiles, e.g., various flavors of CxROM or MMC3, are grouped into a single number. The underlying logic behind the numbering choices appears to be based on abundance—the most common PCB types were assigned the lowest numbers. According to the NesCartDB, iNES mappers #0-4 (NROM, MMC1, CNROM, UNROM, MMC3) comprise nearly 80% of the most commonly used PCBs. Other choices were more functional: Mapper 0 makes sense for NROM, since it is technically not a mapper at all, so the number designates the absence of any mapper. As additional mappers are developed or discovered, assignment is now based on number availability.

In the early iNES specification, header bytes 7 through 15 were ignored—or more accurately reserved for future implementation. Ignoring byte 7 meant that initially only four bits were allotted for mapper numbers. Cutting the mapper byte in half results in an exponential loss of available mapper slots, from 256 down to 16. Sixteen was adequate for the lion’s share of ROMs with the added future security of implementing the latter four bits
as mapper support expanded. Consequently, proper headers were supposed to pad this remaining space with 0s. However, since emulators at the time ignored this portion of the header, the space ended up as a dumping ground for self-promotional graffiti. One of the more infamous cases was the nine-byte string $44,$69,$73,$6B,$44,$75,$64,$65,$21, whose ASCII translation read ‘DISKDUDE!’ This header signature was a residual artifact of using the NES Image utility, a ROM format converter released in 1996-97 by Australian programmer John Pappas, aka DiskDude (figure 5.4).

5.4 John Pappas, aka DiskDude’s, NES Image v3.40 running in Boxer on OSX 10.7. (Source: Author).

NES Image was one of the first conversion utilities available at a time when many PasoFami users were switching to iNES. Dumping new ROMs or sourcing .NES versions was either resource intensive or time-consuming, so users made the obvious choice of using NES Image to translate their existing ROM stockpiles. This sent thousands of DiskDude ROMs into the wild.

As emulators evolved and began using the reserved iNES bytes, DiskDude’s self-aggrandizing gesture began to gain infamy, though not in the manner he probably hoped for: DiskDude ROMs were breaking emulator support. With unexpected data in the reserved header bytes, the ‘dirty ROMs’ failed to load properly due to a mapper mismatch.
This prompted a wave of new ROM utilities designed to scrub dirty headers. Disk Dude ROMs continually frustrated emulation developers working toward a standardized iNES format and more accurate emulation. If a user’s ROMs failed to work in a given emulator, they were more likely to blame the emulator than the ROM, especially if it had worked in the past. Supporting legacy ROM formats became a new conversion problem within the emulation community. Should emulator authors support checksums, header fixes, or other workarounds to accommodate dirty ROMs, or simply jettison support in favor of accuracy and standardization?

To his credit, Pappas’ utility was not the only header corruptor. The ‘aster’ signature ($61,$73,$74,$65,$72), named after a utility of the same name, appears frequently, along with ‘DisNi,’ ‘NI 1.3,’ ‘NI 2.1,’ (both from NES Image), ‘MJR,’ and other ASCII remnants. By version 3.34, NES Image no longer injected its author’s name, a welcome modification that unfortunately, even in 1997, arrived too late. In the utility’s documentation, Pappas wrote, with mock incredulity:

[Version 3.34] removed any extra "junk" (?) in the .NES format header. In previous versions, the version number of NES Image, or the word "DiskDude!" was present: I have seen many Nintendo ROM Images with this information within the header… great to see people actually used this utility!

His enthusiasm will be long-lived. Despite the emulation community’s efforts to whitewash dirty ROMs, they continue to be an infinite-headed hydra. The rampant circulation of ROMs in the 1990s ensured that DiskDude multiplied beyond any manageable scale. If just one ROM hosting site used NES Image, their corrupted ROMs could have reached potential hundreds of downloaders, each of whom could trade, copy, or host those files for other users, who could pass them along in turn, and so on down the chain. Even today, torrent packs of every imaginable NES and Famicom ROM are littered with DiskDude and his header kin (figure 5.5).
5.5 A screen capture from the author's custom PHP script that strips header information from an NES ROM collection shows the variety of junk data (far right column) still present in contemporary ROM collections.

Admittedly, the persistence of junk bytes are a boon to digital researchers. They create an embedded archive of a file's genealogy, an ASCII trace of its circulation within a community of programmers and programs. Without such evidence, these minor histories would be lost.

The Severed Hand

Bloodlust Software, founded by Icer Addis (aka Sardu) and Ethan Petty, posted the first public release of a new Nintendo Entertainment System emulator on April 3, 1997. News of NESticle v0.2—a freeware emulator—quickly circulated online. Its unique portmanteau,
anatomical icon, and laissez-faire documentation initially did not inspire much confidence in its quality. Even Bloodlust’s bundled README.TXT called the emulator ‘essentially the product of 2 weeks of boredom and a smattering of effort.’ But the juvenile overtones belied the emulator’s speedy interior and easy-to-use feature set. NESticle could run handily on common 486 or Pentium PCs. Bloodlust had leveraged a number of breakthroughs within the NES development community to build a better emulator. Sardu based his CPU code on Neil Bradley’s m6502 emulation core, gleaned technical information from Fayzullin’s NES.DOC, based the sound emulation on Y0SHi’s NESTECH.DOC, and supported the established iNES header format (with promises to later support PasoFami files).

NESticle v0.2 was spartan compared to today’s NES emulators, but it had a visual charm that helped differentiate it from its contemporaries. The GUI was colorful and easy to use, wrapping its chunky windows in vibrant shades of blue accented with grey, green, and white (a look strangely similar to Windows XP’s default ‘Luna’ theme, which debuted four years later). Consistent with Bloodlust’s image, the mouse icon was literally skinned with a severed left hand extending its pointer finger, terminating in a bloody stump (figure 5.6). In subsequent versions, the stump dripped animated blood.
Taste notwithstanding, NESticle featured several useful utilities beyond playing NES ROMs. Via the ‘View’ pulldown menu, users could access visual representations of the NES’s behind-the-scenes technical operations, including the pattern tables, name tables, palettes, and waveform output. Viewing the pattern table, one could see the ROM’s entire graphics set, arranged into contiguous background and sprite sections. Clicking either section allowed users to cycle between available palettes in order to better identify specific tiles. More remarkable was the palette view, which allowed users to isolate and edit palette entries in real time. Want to make Donkey Kong pink instead of brown? Locate his associated background palette and adjust the three RGB sliders. The ‘Messages’ view outlined NESticle’s various processes as well as a few details about the currently loaded .NES file. Selecting ‘Get ROM info’ would display the number of ROM and CHR (or VROM, as NESticle labelled them) banks, mirroring, and the mapper number.

NESticle also implemented simple systems for both saving one’s game and taking snapshots. Keying F5 or F7 would respectively save and load the current emulation state, independent of the game’s in-game saving mechanism (or lack thereof).
effectively took a ‘vertical slice’ of the NES’s memory at a given frame. This included the contents of RAM, status register flags, stack contents, program counter location, patterns, name tables, attributes, mirroring setting, and so on.⁶⁰ The resulting file was saved as the ROM’s name with an .STA extension (e.g. BalloonFight.sta) in the NESticle directory. Loading a save state reversed the process, injecting the emulator with a set of parameters to reinstate the virtual NES (and its accompanying ROM) to its previous state. Similarly, F9 captured a .PCX (a DOS image standard later superseded by .GIF, JPG and .PNG) snapshot of the current screen and saved it alongside the ROM file.

Both features had instant appeal. Born-digital screenshots obviated the need to set up a camera to capture a game’s graphics or record a high score. Players accustomed to the limited or non-existent save systems implemented in NES games were suddenly able to save and re-load as they pleased. Difficult games could now be broken into smaller segments—levels, screens, or a handful of steps, if a player so desired. It was now possible for many more players to tackle nigh impossible games like *Ghost & Goblins* or *Dragon’s Lair*. And more significantly, since save states were stored separately from the ROM file, they could be shared and swapped online and on floppy disk, provided the receiving player had the matching ROM on their computer. Popular emulation sites like Zophar’s Domain served as repositories for downloadable save states, especially those stacked with a host of in-game power-ups or positioned at the conclusion of a final boss battle, allowing the user to easily view the game’s ending. NESticle’s feature bred a new form of assisted play—either cheating or player advantage, depending on your perspective—that would later evolve into the tool-assisted speedrun. Save states were the first step along that path.

Although many save states were captured legitimately by skilled or patient players, they could also be opened and modified with a hex editor. Players with enough time and patience, coupled with some elementary knowledge of the NES hardware and the ROM image’s memory layout, could isolate variables that determined the player’s in-game attributes. This was particularly useful for maxing out lives and collectibles or equipping RPG avatars with enough gold, equipment, and experience to breeze through battles. Silver X’s guide to state hacking *The Legend of Zelda*’s item inventory highlights the mix of technical knowledge, experimentation, and blind luck that was involved:

The way this works is that there aren’t individual bytes representing each item. All of them are represented by 01, with the exception of the 2nd potion, White and Magic swords, the Silver Arrow, the Red Candle, Triforce, and the Red Ring...Another thing to note is that items you can have multiple numbers of, such as bombs, keys, and Rupees can be represented by numbers up to their maximum limit of 255, which is FF...A few instances of weird shit can happen if you change a byte to something out of it’s range, like changing the sword byte to 04 or the Potion to 05. Try messing around with some crap, you might get some cool results.⁶²
Cottage communities grew up around the creation, hacking, and distribution of NESticle .STA files. For the most popular titles, hackers took it one step further and programmed game-specific editors. These custom programs allowed players to edit NESticle’s save states in a more user-friendly environment than the bare numeric fields of a hex editor.

Other utilities allowed users to manipulate ROM images directly. One could edit sprite and background tiles, in-game attributes (via variables in RAM), and even entire level layouts. The ZELDIT utility, for example, could edit a handful of The Legend of Zelda’s overworld and underworld areas (figure 5.7). The dungeon editor was notable for its visual presentation, as it revealed the tetromino-esque layout of the dungeons in memory.
A small square indicated your current editor position, wherein you could cycle through a number of pre-fab screen arrangements, altering door types or floor tiles. Below the individual screen display there were two sliders that controlled the type of monsters or objects that would appear in the room. Players with no prior technical knowledge of the NES could learn visually how memory constraints dictated the in-game arrangement of tiles, items, and foes.

User modification was a new phenomenon for console gamers. Few had the resources
or know-how to dump, edit, and burn custom EPROMs. A handful of NES games like 
*Excitebike* and *Wrecking Crew* featured built-in level editors, but they were a rarity. PC 
gamers, however, were more accustomed to mods. The ‘-type-it-yourself’ source code 
available in early PC gaming magazines encouraged gamers to be both players and 
creators. Don’t like the look of an enemy craft or the starting number of lives? Redraw a tile 
or tweak a variable. The culture of PC user modification extended beyond magazines into 
software development. *id* Software, for instance, built a rabid following around the *Doom* 
and *Quake* franchises based on their encouragement of user-built levels, graphics re-
skinning, and other modifications. Even unsanctioned game modifications were easier, 
since transferring data from a floppy to one’s hard drive was far less complex than cracking 
a cartridge ROM.

Emulation and .NES files eliminated the hardware barrier from the modification 
equation. Suddenly players had the ability not only to play NES games, but to edit, 
reconfigure, and remix them. ROM hacks, as they are called, introduced strange cross-
fertilizations of game worlds, a digital fan fiction that combined characters and scenarios 
from different games: Link’s sprite appeared in *Super Mario Bros.*, Mario made cameos in 
*Metroid* (and every other conceivable game), and *Galaga*’s armada of spaceships were 
replaced by beer cans and bottles. Many hacks were less innocuous—overtly racist, violent, 
and homophobic versions of popular games cropped up in equal measure. With few 
barriers to expressive creation, the best and worst traits of online culture were on display.

*Super Mario Bros.*’s popularity made it an early and frequent target for ROM hacking. 
Its NROM foundation also made it simple to edit graphics. The lack of bankswitching made 
it trivial to pick out and edit individual tile patterns in CHR-ROM. Most alterations were 
visual tweaks, such as writing one’s name in the clouds, changing colors, or ‘upgrading’ the 
sprite and background tiles to resemble those from later games in the series. Frequently, 
Mario was costumed in a variety of humorous or off-color guises: a wheelchair, Nazi 
regalia, an afro, an assortment of drug paraphernalia, nude, and even donning a Ku Klux 
Klan hood. Intentionally or not, ROM hacks were intermingled with clean dumps of *Super 
Mario Bros.*, so unsuspecting downloaders might be surprised to find a ninja in Mario’s 
place when they launched their .NES file. Eventually, as these variations spread online, it 
was a challenge to find a *Super Mario Bros.* ROM that hadn’t been altered in some way.

Other more serious efforts were made to design new worlds for players to explore. 
Mario’s original batch of thirty-two screens was re-imagined and re-arranged to the point 
that they resembled a sequel more than a remix, like homegrown versions of the Japanese 
*Super Mario Bros. 2*, meant to offer new challenges to experienced players who had 
mastered Mario’s first adventure. Fan-remixed versions of *The Legend of Zelda* scrambled 
the dungeons and power-ups, much like Nintendo’s own Second Quest. And all of these 
hacks were executed within the original ROM’s constraints. Without altering the source
code, any hack was still governed by the game mapper’s parameters and the specifics of the
game’s engine. No ROM hack would allow the Super Mario Bros. engine to scroll left, for
instance. This was a code-level limitation, beyond the bounds of simple modification.

However, contemporary hacks now straddle the border between modification and
programming. IKA’s Rockman No Constancy (2007) is a hack of Rockman 2 that changes
level layouts, increases the difficulty, modifies weapon and boss behaviors, alters the
graphics, and inserts a new soundtrack. GameMakr24’s Zelda Challenge: Outlands gives The
Legend of Zelda a similar treatment—every aspect of the original is changed, from the
storyline to the NPC dialogue. DahrkDaiz’s Mario Adventure is a hack of Super Mario Bros. 3
that delves into the game’s engine, adding save features, new enemy types (not only
graphic changes, but new behaviors), random weather effects, and more. This so-called
‘assembly hacking’ requires in-depth technical knowledge of the NES hardware coupled
with the advanced capabilities of modern emulators. Assembly hacking is tool-assisted in
the same sense that speedruns are.

Another important hacking innovation was the translation patch. As mentioned
earlier, mainstream NES emulation granted players access to games that were otherwise
unavailable, short of importing Japanese or European consoles and their region-specific
games. ROM availability was a definite improvement to accessibility, but language barriers
still hampered a bulk of Famicom titles. This was especially true for text-heavy adventure
games, an immensely popular genre in Japan, but poorly-represented on the NES. ROM
alteration combined with simple distribution of files online led to fan translations of
releases that never made it stateside, like the original Famicom versions of Final Fantasy II
and III. (Players in the U.S. only knew Final Fantasy II and III as Super Nintendo games,
since the series was renumbered for American release.)

Again, the limitations of a game’s mapper and engine came to bear on translation
efforts. In many cases, Japanese text was denser than the English equivalent. A single kanji
character might require one or more words in translation. This limitation led to
condensations of dialogue or menu items that ranged from artful to nonsensical. It was the
‘I AM ERROR’ problem all over again—though this time as a result of limited memory
rather than a language barrier. Also, altering in-game text was often more complicated than
editing character sprites. Although characters are stored as tiles, like any other graphic,
text-laden RPGs like Final Fantasy II used compression to pack dialogue into fewer bytes.
Simply locating the text in memory was a challenge, much less decrypting and unravelling
the techniques used to store the text. Many translation groups developed their own
compression tricks to sneak more English text into a game’s limited ROM space. The
documentation accompanying Demiforce’s 1998 translation of Final Fantasy II details one
such process:
Another technique is one of DTE (Dual Tile Encoding). DTE method was cracked on 1/16/98, meshed by the combined talents of Landy, Alex W. Jackson, and Dark Force. What it means is we took advantage of a coding technique Square uses in its Japanese for the “chon chon” marks. Since there are two tiles used in chon chon characters, we took the subordinate tile and placed it after the dominant character, allowing us to display two characters with just one byte call in the ROM. When first implemented, we had about 1000 extra bytes to work with (!!!), so ever since then there’s been a lot more leniency concerning detail and story length.

Homebrew translation efforts such as these, which could take months or years to complete, expanded the diversity of accessible Famicom software and exposed non-Japanese audiences to games that were previously unplayable. They also foregrounded the mediating effects of constrained hardware on the game’s expressive content. In other words, there was an additional layer of translation necessary to adapt the Japanese text to English. Without adequate ROM space, numerous concessions would be required to adapt a game’s narrative content.

The explosion of user-friendly emulators like NESticle helped catalyze the cultural cross-fertilization between Asians, European, and American nations presaged in the decade prior by arcade games, science fiction, comics, and animation (see chapter 2). Non-Japanese gamers now had a less mediated version of Japanese culture, since the Famicom-exclusive games now available as .NES files were never meant for audiences outside of Japan. Dumping ROMs directly from the Japanese source circumvented Nintendo’s corporate control over the content of their games. Even Miyamoto’s Devil World, the Nintendo figurehead’s only game that never made its way to the US, could be played by fans for the first time.

Beyond the obvious Famicom exclusives, players began to notice discrepancies in games that were released in all regions. Famicom Disk System games had richer sound and disk-based save systems. Familiar titles like Bionic Commando had altered graphics due to religious or political whitewashing (see chapter 3). Discrepancies in cartridge hardware often hamstrung US releases of Famicom titles, like Contra, whose Japanese version featured richer animated backgrounds and interstitial cutscenes. A slew of arcade titles came in NES, VS., and PlayChoice-10 flavors. As a result, a standardized ROM labeling system developed to classify titles by their origin of release. Utilities like GoodNES (part of the GoodTools suite) audited a user’s ROM catalog against a verified database and labeled them according to region, variation, and quality.

ROM packs downloaded online often contain a text guide to the trail of symbols and letters that follow a file name. Psych0phobiA’s ‘GoodCodes.txt,’ for instance, covers the GoodTools hieroglyphs used for Genesis, SNES, Game Boy, and NES ROMs. Country codes are typically single capital letters in parentheses: (J) for Japan, (U) for USA, (E) for Europe...
and so on. Bracketed codes indicate variations such as a hack [h], bad dump [b], pirate [p], translation [T], or; ideally, a verified good dump [!] Why do bad dumps even exist? Data flaws that occurred during image transfer or corruptions that occurred during zipping or uploading might create a non-working file. If a ROM was particularly rare, it might be the only dump available until a new one could be made—semi-working ROMS were better than no ROM at all. Since these files propagated alongside good dumps, utilities like GoodNES made it simple to scan one’s library and suss out the offending ROMs. If a user chose not to discard the ROM (or did not understand the [b] designator), its further dissemination would at least be flagged for future downloaders.

Emulation introduced several orders of translation to the console fold. First, there were the obvious linguistic translations—Japanese to English, French, Spanish, and so on. Emulators eliminated the need for corporate intervention to make translations possible. Second, the translations inherent to the persistent conversion problem continued to mediate the experience of the NES as a platform. The NES was now more than an NES, a series of tools and augmentations meant to facilitate new forms of expression and play. And finally, more subtle, 'lower-level' orders of translation were taking place in the standardization and adaptation of competing ROM formats and labeling systems. PasoFami’s split format had to be converted to iNES or .FAM and vice-versa. Headers had to be cleaned to adapt to stricter, more advanced emulators. ROMs had to be checked and classified against a master list to ensure their pedigree and sort the chaff of hacks and corrupt images from the wheat of unmodified ROMs. And all along, the specter of conversion loomed over the question of emulation accuracy—how far could emulators be pushed toward precisely mimicking the behavior of the original console.

For instance, despite its range of influential features, NESticle had its share of drawbacks. Due to the sparse documentation of the NES’s APU at the time and the CPU resources necessary to properly emulate it, NESticle’s audio implementation was severely underdeveloped. Bloodlust’s README.txt recommended simply turning the sound off ‘if it gets on your nerves.’ NESticle’s palettes were not as jarring as the soundtrack, but they were wildly inaccurate compared to the NES’s actual composite output. Mid-screen raster effects, used for status bar splits or palette swaps, were not properly understood, so games that relied on carefully-timed VRAM updates (e.g., Super Mario Bros. 3) either did not work or exhibited noticeable glitches. Bloodlust attributed this to the shortcomings of the NES:

There are still some bugs with games that utilize split screens. In my opinion, the NES method of split screen-ing is utterly horrible. Lots of games rely on CPU speed to tell when to split the screen, others use a dumb hitflag, and others use IRQs. Games that switch pattern tables halfway through the frame were a bitch. Adjusting the HBlank/Vblank length under Settings/NesTiming may fix some split screen quirks.
Whether Sardu’s assessment of the PPU was accurate or not, today’s emulator authors withhold such judgments of engineering prowess in favor of better accuracy. NESticle also offered limited mapper compatibility: only mappers 0 through 4 were supported in v0.2. This covered a significant chunk of the NES and Famicom library, but excluded a number of popular titles like Castlevania III (iNES 005/MMC5), Battletoads (iNES 007/AOROM), Cobra Triangle (iNES 007/ANROM), Mike Tyson’s Punch-Out!! (iNES 009/MMC2), and the entire Wizards & Warriors series (iNES 007/AxROM).

However, development progress on NESticle moved rapidly. Within two months of its April release, NESticle introduced features such as online play (for Windows users), recording and saving audio output, and the ability to record and playback gameplay movies. This latter feature was the logical extension of the .STA format. NESticle movie files (.NSM) recorded the current state as an .STA, then any subsequent player inputs until the user selected ‘Movie/Stop.’ The resulting file was the combined record of a machine’s initial state and a player’s frame-by-frame performance. The .NSM was also modestly-sized—approximately 432K/hour according to NESticle’s documentation. And since an .NSM was simply a state file with appended input data, the resulting file had built-in backwards compatibility. As Sardu noted, ‘renaming zelda.nsm to zelda.sta will allow you to load the movie as a normal state file and play from the movie's beginning point.’

Once Bloodlust introduced the .NSM format, sites featuring archives of NESticle movies proliferated. For the first time, NES players could witness game performances remotely. Prior to emulation, that type of telepresence was only available via video, in the form of VHS direct-feed captures of gameplay usually sold by game magazines to demonstrate tactics and tricks for a handful of games. But large-scale VHS production and distribution was expensive and limited to companies that could afford it. A player-produced digital distribution system for video was impossible. In the mid-1990s, hosting or viewing video online was rare. The infrastructure of necessary bandwidth, capable PCs, and consumer-grade digitization software was not yet in place. Even downloading video was resource-intensive when hard drives were measured in tens of megabytes and modems shuttled data in kilobits per second. However, describing a movie as a sequence of key presses circumvented the limitations of bandwidth and storage. As long as the viewer had the proper emulator, ROM, and .NSM file, they could replay a lengthy gameplay movie in full resolution on their computer screen.

By the time NESticle was introduced in 1997, the Nintendo Entertainment System was several years out of its prime. Nintendo was already two console generations ahead of the Famicom, having released the Nintendo 64 the previous year. Former hardware ally turned competitor Sony now challenged Nintendo’s long console hegemony with the CD-based PlayStation. But the thriving aftermarket of used NES games and the continual
rebirth of its valued franchises kept the legacy console fresh in the minds of videogame fans. Many of the children who had grown up playing the NES, now teens and twentysomethings, were becoming developers and programmers in their own right. In large part, these were the demographics responsible for emulator development.

The elder 8-bit console and its vintage 6502 core hit the sweet spot of manageable complexity, popular appeal, and PC storage/network limitations. ROMs were small enough to transfer via modem or floppy and store locally in a handful of kilobytes. This was not feasible with then-current platforms like the Sony PlayStation; its 3D capabilities were too computationally costly for mainstream PCs and disc-based media was not yet economical to duplicate or store. The NES stood at a unique turning point in the processing capabilities of mass-market PCs and the emerging distribution channels of the World Wide Web. The NES was popular enough to attract enthusiast programming interest; its limited CPU, video, and audio processors were manageable for software emulation; and its compact ROMs were ideal for storage and transmission. The rapid success of NESticle and the torrent of emulators that followed catalyzed mainstream interest in console emulation and allowed the NES platform to ‘live on’ beyond its limited hardware life cycle.

Text Movies and Choreographic Play

Though NESticle was not the first NES emulator, it introduced or popularized a number of now de facto augmentations like save states and movie recording. Above and beyond those early innovations, contemporary NES emulators add near-IDE levels of code manipulation. FCEUX, one of TASVideos’ sanctioned emulators for tool-assisted speedruns, features a built-in hex editor, background graphics viewer, inline assembler, debugger, PPU viewer, and conditional breakpoints. These software emulators not only play games, but strip them bare to the code. The user can watch the game engine access and write registers during gameplay. It is like going to a movie theater and watching the film run from Final Cut on an seat-mounted laptop—and you can tweak the speed, angles, and edits to your liking.

The editing metaphor extends to the TAS, as the outcome of the performance is commonly called a movie. But movie is a misnomer for both the process and result. NESticle .NSMs were save states with twice-per-frame key presses appended. The underlying content was not visual, but a chain of bytes meant to trigger response from an emulator loaded with the appropriate ROM. The .NSM was more akin to a macro or script than what we would commonly call a video. Likewise, tool-assisted runs are not spliced sections of normal gameplay, but meticulously-assembled frames of player input, closer to animation and choreography than filmmaking. In a practice called ‘re-recording,’ players drop the game’s playback speed significantly and cycle through small segments of
gameplay—bookended by save states—altering variables until the run is perfected. Once that segment is satisfactorily complete, a new save state is created and the process continues.

In FCEUX, a movie is a specially formatted plain ASCII text file saved with the extension .FM2. The .FM2’s ASCII format is advantageous since it is human-readable. Once one is familiar with .FM2’s syntax, it is easy to decode the system of key presses and edit them like a document or spreadsheet. Like the ROMs they enact, movie files contain a short header followed by a data stream. The header is a chain of key-value pairs used to describe the emulator version, the type of input device used, the frame length of the movie, any comments, and so on. The header for Nick Mong’s (aka mmbossman) TAS of Arumana no Kiseki, for example, reads as follows:

```
version 3
emuVersion 20100
rerecordCount 13078
palFlag 0
romFilename Arumana no Kiseki
romChecksum base64:hjO8JpyaAlI9rmzVvydCA==
guid F91714F1-7905-8548-6B40-C4600750A645
fourscore 0
port0 1
port1 1
port2 0
comment author mmbossman
```

The checksums, ids, and version numbers are meant to ensure the TAS’s exact replication on a user’s system. Without the properly calibrated setup, a movie will not playback accurately.

The data section following the header is called the input log. Each frame of the movie is given its own line, bookended by pipe (‘|’) characters. Following the first pipe is the input port value (e.g. ‘0’ for joypad 1), another pipe, then eight ‘bits’ representing the possible joystick presses for that frame. The following shows an excerpt of four frames from Mong’s TAS:

```
| 0 | R . U .... | ........ | |
| 0 | R . U .. B | ........ | |
| 0 | R . U .... | ........ | |
| 0 | R . U ... A | ......... | |
```

Line/frame 1 represents the D-pad’s right (R) and up (U) keys pressed simultaneously. Frame 2’s input continues to hold right and up while also pressing B. The final two frames’ inputs release B then press A. Keep in mind that the preceding text block describes a mere
1/15 of a second. The full TAS input log of an NES speedrun, which is typically under twenty minutes in real-time playback, can run tens of thousands of lines long.

A completed movie reads as a frame-by-frame list of inputs in sequential order, an editable input recipe for perfect play. Conceivably, with superhuman skill and memorization, a player could replicate this list on real hardware. In practice such play is impossible, since speedruns often rely on mutually exclusive key presses (e.g. left and right simultaneously) in a single frame, a feat not only humanly but *mechanically* implausible. Mashing left and right on the D-pad requires at least two fingers and a fair bit of force—a task much too convoluted to execute in 1/60 of a second, much less in a series of similarly deft key presses. These strange inputs, in turn, trigger conditions that the game’s programmers did not account for. When object interaction in a game engine expects to execute branch statements based on binary inputs—if the player presses up, move up, otherwise, move down—‘overloading’ the system with unforeseen input can create unexpected behaviors. A sprite’s coordinates, for instance, might suddenly update by several pixels more than intended, bypassing the collision detection check that was meant to prevent the sprite from passing through a solid wall. Holding up and down simultaneously while riding elevators in *Zelda II*, for instance, allows the player to warp vertically through dungeon levels.⁷⁴ Input glitches like these are the holy grail of TAS assemblers, allowing them to short-circuit large segments of gameplay in the name of faster completion.

What motivates this type of play? Can we still label it ‘play’ as such? And what do the limits of play teach us about the limits of platforms? James Newman provides a number of answers to the first question in his discussion of ‘superplay.’⁷⁵ First, there are the goals common to videogames, non-digital games, sports, and many types of unstructured play: creative expression, competition, socializing, self-improvement, community-building, and public recognition. Another process at work, and a proclivity of digital media like videogames, is the mastery and manipulation of an underlying procedural system—‘playing’ and ‘gaming’ in the sense of bending or breaking rules to gain an advantage. Newman explains that ‘this system of rules and boundaries is not fixed but rather is permeable and in a state of flux as it is interrogated, operated on and played with. Moreover, the system may behave in an unpredictable manner unintended by the game designers due to imperfections in the code or unanticipated emergent contingencies.’⁷⁶ In this interplay, Newman concludes, ‘both player as performer and game system should be considered agents in the process of gameplay.’

Rule-breaking is certainly enticing to TAS assemblers. There is a transgressive aspect of playing a game ‘wrong,’ while in the process, playing it better than any other human could. The perceived rigidity of computational platforms—slaves to the stark microcosm of algorithms and binary digits—loosens as we witness the odd spectacle of a violated game
space. Algorithms run awry. Binary becomes and instead of either/or. The game engine behaves erratically when fed unexpected inputs. This is why the TAS movie, in its text form, is eventually recorded as a movie proper that can be posted on sites like YouTube and shared with others. The TAS community’s emphasis on entertainment value insists that a run must be seen. And seeing provides insight into the underlying algorithmic processes at work in the game engine.

TAS text movies archive encounters between source code and input code. Over time, as a platform is better understood, one might reasonably assume that the human player’s role will eventually disappear, that a computer might learn to play the emulator and ROM with limited external assistance. TAS assemblers are already building external tools to automate some aspects of the TAS-building process. Adelikat’s TAS of Gradius, for instance, uses a ‘macro editor patch’ for FCEU that allows him to program complex input strings, resulting in the ship flying in a number of predefined patterns:

To create this run, I used a "macro-editor." With this editor, I can create a series of button presses and put them together into a single command. This allows me to push one button to initiate very complex movement patterns. Writing my name "adelikat" for instance was a large programmed macro...I simply had to find a frame to start it that didn’t result in the ship being destroyed."

As the TAS evolves, tools are encroaching into play itself.

N. Katherine Hayles coined the term ‘dynamic heterarchy’ to describe a ‘multi-tiered system in which feedback and feedforward loops tie the system together through continuing interactions,’ whose tiers ‘continuously inform and mutually determine each other.’ She develops the term to describe our encounters with various electronic texts, but reminds us that these heterarchies can be biological as well, like the complex ‘system’ formed between mother and fetus. The adjective ‘tool-assisted’ points to a dynamic heterarchy between player and emulator that operates in the execution of a speedrun. The player initializes the system through input, making an onscreen character run, jump, or shoot. The system them procedurally reacts—an adversary appears, the character’s body makes contact with an object, a timer ticks down. Based upon this feedback, the player then adjusts their subsequent input—or, more crucially, revises prior inputs by resetting the system to a known state in order to either perfect their input or evaluate an alternate system response. In real time (e.g. the traditional speedrun), this trade of actions and reactions is linear and unpredictable. In other words, a player cannot retroactively adjust their input stream beyond the typical looped structure of play, death, and re-play, while the range of system reactions are too numerous or complex to react to with normal human reflexes.

Leveraging the assistance of the emulation tool, the execution of the underlying
computational processes becomes legible. A player witnesses the machinations that take place in a single frame or cycle. At this fine-grained level of perception, fissures in the system come clear—the sparse boundaries of objects permit elisions of solids and bodies, a system expecting only single inputs per frame re-route to unexpected junctures when this expectation is overloaded, the bare clockwork of computation becomes predictable, even manipulable. Tool-assisted speedrunners have even coined a clever oxymoron—‘luck manipulation’—to describe their ability to coax a game’s pseudo-random processes toward predictability, like avoiding enemy encounters in *Dragon Warrior* or triggering favorable item drops in *Mega Man 3.⁷⁹*

Even experienced video game players are left bewildered by a number of the most extreme tool-assisted runs. The content of a glitched performance borders on illegibility as fragments of displaced tiles rearrange into 8-bit cubist collages, bizarre visual references to what was once a playable game.⁸⁰ Other TASs are comical in their brevity. The run for Genesis title *King’s Bounty* completes the lengthy turn-based strategy game in under ten seconds—and most of that time is spent loading the screen. In a 2009 feature on ‘insanely thrilling’ tool-assisted speedruns, games journalist John Teti describes a *Rockman* run that uses bugs and exploits ‘to such a degree that the action is distorted and barely recognizable, almost like an avant-garde film.’⁸¹ Teti is alluding to the systematic breakdown of spatial and temporal boundaries, violating viewers’ expectations of continuity, coherent objects, and even legible behavior of the underlying procedural system. In everyday computer use, the glitch or bug are phenomena to be avoided. The TAS elevates and foregrounds error as a means to mastery.

What happens at the limits of the tool-assisted speedrun, when all possible avenues for gains in speed and glitch manipulation are exhausted? One possibility is the emergence of a TAS that is meant to be visually interesting or aesthetically pleasing for its own sake, regardless of whether speed is sacrificed in the process. It exposes both the limitations of an underlying computational system and the considered performance of its player. It hovers at the fringe of many disciplines, including dance, animation, video, and sport. I call this special subform of TAS *choreographic play.*

Adelikat’s *Gradius* TAS is a prime example of the form. Since *Gradius* is a side-scrolling shoot-em-up, it seems an odd choice for a speedrun, as most of its gameplay speed is fixed. Scrolling progresses at a constant rate regardless of player action. Barring death or prescribed action sequences, the player is meant to arrive at the end of a stage on schedule. Only the boss fights that occur at the end of each stage grant any leeway in the final run time. Adelikat’s description under the ‘Overall Aim’ heading in his author’s notes emphasizes both speed and beauty:

The goal is to take a game with this kind of speed potential and push it to the limit.
This allows me to create a run where the on screen animation becomes a visual pattern resulting in the less & less importance on the game itself [sic]. The run transcends the usual gaming logic and becomes a sort of abstract art. The run becomes something unique and emphasizes the potential for TASes to truly be considered art.³²

I am less concerned with the artistic merits of the TAS than the careful marriage of speed, form, and motion. Using the spaceship to paint geometric patterns or the author’s name emphasizes the subjective element of the TAS while still leveraging the automated assistance of non-human tools. The underlying aesthetic dimension of play—a trait we appreciate in sport, musical improvisation, and videogames alike—points to a new direction in emulation-assisted play that is not focused squarely on raw speed or entertainment.

Baxter’s mesmerizing TAS of *Arkanoid* had similar aims.³³ He chose to skip the time-saving Laser power-up, which can obliterate blocks by shooting them with the paddle, in favor of the 3-ball power-up, which releases two additional balls on the screen. The result is a hypnotic high-speed juggling act that is as compelling sonically as it is visually. The geometric arrangements of kaleidoscopic bricks emanate erratic melodies as they struggle to keep pace of the ricocheting balls. Similarly, Baxter and AngerFist’s collaborative multi-screen TAS of *Mega Man 3, 4, 5, and 6*—played simultaneously by the same sequence of inputs—coalesces into a manic cacophony of sound and image, more John Cage than Keiji Inafune.³⁴ In all cases, the weird creative artifact that is a TAS exposes the strange limitations of medium specificity that exist at the borderlines of all media. A completed TAS is called a movie, but it is not. Instead it is, all at once, a text file, a performance, a dance, an animation, a procedural event, sport, entertainment, a social act, an ethics, an archive, and yet still a videogame.

The vitriol initially surrounding Morimoto’s *Super Mario Bros. 3* TAS implied that the human element was somehow slipping away, that automatons would eventually surpass the efforts of flesh and blood. An artificial intelligence system could conceivably find the best routes, manipulate luck in beneficial ways, and minimize the key presses necessary to complete a game. Certainly some of the scripting tools now available to automate play lean toward that scenario. But we should be wary to rely solely on the end result of the TAS process. The final ‘movie’ is a human impossibility, but using it as an example of eliminating human intervention ignores hundreds of laborious hours invested into perfecting a run. Moreover, as long as games are differentiated at the engine level, there can be no reliable means to algorithmically generate speedruns for a wide swath of games. The engine driving *Ninja Gaiden* is fundamentally different than the one driving *The Legend of Zelda*. Their object handling, collision detection, sound routines, bankswitching, and compression algorithms make it impossible to write a generic TAS-generator that could compete with
the most meticulous human assemblers. And considering *The Legend of Zelda* exists on different physical formats (e.g., NES game pak and FDS disk), there is a potential challenge simply to automate play across multiple *versions* of the same game.

Finally, as videogames increase in complexity the number of possible inputs and outcomes rises exponentially. This fact is not limited to generational leaps in hardware. Witness the increased complexity within the platform genre during the NES’s heyday. The strict one-way scrolling and limited power-up system of *Super Mario Bros.* was superseded by the non-linear stage selection and myriad weapon choices of the *Mega Man* series. The relatively constrained ruleset of millennia-old board games like *Go* continue to stymy AI researchers.\(^5\) How long would it take to computationally master the complexity of *Kirby’s Adventure*?\(^6\)

One final facet of player agency is rule-making, and for a community devoted to bending software to its algorithmic limits, there are a surprising number of sacrosanct rules for movie-making, from using sanctioned emulators to running checksums on ROM images to verify their pedigree. Part of this impulse is based in competitive fairness, standardizing rules to provide a level playing field. But there is also a peculiar threshold of coherence at work, an agreement about what a platform is and is not and how far players can stretch that definition.

Among the NES homebrew community, there is the same shared respect for the platform’s constraints. Every few months, a community member will propose a revised hardware specification for the NES that eliminates per-scanline sprite limitations, palette constraints, and so on. The typical reply is: ‘If the NES PPU isn’t enough for you, just don’t use the NES, get over whatever problem you have with other platforms and use those.’\(^7\) At a certain point, an ‘improved’ NES starts to look like a Super Nintendo. In other words, the platform is defined as much by what it can not do as what it can.

When we try to pin down exactly what an NES is, we see the strange liminality at the edges of a platform. One might argue that the NES is the grey and black ‘toaster’ released in 1985, a specific arrangement of silicon, aluminum, and plastic. But this discounts the elder Family Computer, the Nintendo-licensed Sharp Twin Famicom, Mattel’s European NES, and the bizarre array of add-ons and accessories that appeared, including plastic robots, modems, inflatable motorcycles, and karaoke microphones. One might argue that these are mere supplements to a consistent core, but that would discount Nintendo’s multiple board revisions, the PAL modifications necessary for play on European televisions, and memory-mapped controllers—the special hardware inside game cartridges that augmented the NES’s limitations and allowed Nintendo to stay competitive as more capable consoles arrived. If the NES is defined as something that plays NES games, then we have suddenly expanded the field beyond Nintendo’s grasp, to unauthorized clones systems like the Russian Dendy, ‘Nintendo on a Chip’ boards built into simulacra of Atari 2600 controllers,
software emulators, and weird hybrids like the Powerpak, which allows you to play downloaded ROMs on a ‘real’ NES. Even then, we have not yet scratched the surface of the numerous hacks and modifications that allow the NES to function as an audio/video synthesizer, to host emulators of other systems, or to live in an actual toaster.

A platform is a negotiation between static and dynamic forms—on the one hand, a finalized configuration of hardware and software must be settled upon as a basis for creating digital objects; on the other hand, that configuration is susceptible to translation, transposition, and metamorphosis through emulation, hacking, and modification. Tool-assisted speedruns would not be possible without tools that augment the original platform—variable speed play, frame counters, save states—but they still rely upon the platform’s distinct errors and quirks. If, for example, the sprite or CPU limitations of the NES were corrected, the underlying game code would not run properly and the identity of the NES as a platform would be compromised. In turn, the quirks facilitate creative performances and new modes of play. The TAS elevates the glitch to a viable play mechanism, transforming the intended coherence of a game world into a strange dance of objects freed from temporal and spatial logic. A platform is thus more convention than console, more abstraction than assembled product. And how strange that a study so rooted in material objects should lose grasp of those objects around their borders.

The phenomenon of tool-assisted play is so recent that it is difficult to predict its impact on the future of videogames. The NES was the right console with the right kind of games at the right time. There is a reason we do not see communities of players excited about Atari VCS or ColecoVision speedruns. In some respects, the VCS was more difficult to emulate than the NES, due to its acute fidelity to the sweep of the television’s electron gun. More importantly, earlier platforms did not support the sophisticated types of games that make for entertaining movies. Their lineage was still too closely tied to the arcades, where score and longevity ruled. Watching an endless loop of Asteroids does not have the same visual flair as a sequence-breaking run of Metroid. We are now at the event horizon of tool-assisted performance in complex 3D worlds—it takes time for desktop PCs to pace the hardware of modern consoles, so the TAS is necessarily a few generations behind. TASs of early 3D-capable consoles like the PlayStation and Nintendo 64 are revealing fascinating new violations of space and time afforded by complex worlds of polygonal depth.

---

¹ See Andersson, “Contra_1017 [Contra in 10:17]” and Carstensen, “NinjaGaiden_1232 [Ninja Gaiden in 12:32].”
² “[HD] TAS: NES Super Mario Bros 3 (JPN) in 11:03.95 by Morimoto.” YouTube.
³ Morimoto, “NES Super Mario Bros. 3 (JPN) in 11:03.95.”
⁴ Lowood, “High-performance play: The making of machinima.”
⁵ See the ‘Rules’ section of Speed Demos Archive, http://speeddemosarchive.com/lang/rules_en.html
⁶ Gardikis, “Mario1_458 [Super Mario Bros. in 4:58].”
Nach. “Guidelines.”


See the ‘Movies Rules’ section of TAVideos, http://tasvideos.org/MovieRules.html

‘Rules,’ Speed Demos Archive.

See Pugh et al., IBM’s 360 and Early 370 Systems, 111-35.

Ibid., 127.


With one exception: the smallest processor model, announced a few months after the initial System/360 line.

Pugh et al., 159.

Ibid.

See Pugh et al., 131 or ‘Microcode,’ Wikipedia.

From ‘Microcode,’ Wikipedia: ‘A processor’s microprograms operate on a more primitive, totally different and much more hardware-oriented architecture than the assembly instructions visible to normal programmers. In coordination with the hardware, the microcode implements the programmer-visible architecture. The underlying hardware need not have a fixed relationship to the visible architecture. This makes it possible to implement a given instruction set architecture on a wide variety of underlying hardware micro-architectures.’

Pugh et al., 161.

Tucker, “Emulation of Large Systems.”

Ibid., 754.


The contemporary exception is mobile development. Smartphones can be either too constrained for development (earlier WAP systems) or vendors will not allow code to compile on their device (e.g. Apple).

Mattel’s Intellivision is a notable exception: it had a BIOS and a built-in character set—rarities in the early console era.

Masaharu (trans. Tanner), “Part 8 – Synonymous With the Domestic Game Console.”

Covell, “The Stars of Famicom Games.”


Ibid., 4.


Ibid., 239.

Ibid. Also see “CP/M.” Wikipedia.

Laing, Digital Retro, 20-23.

Touvell, “Microsoft Softcard.”

Ibid.


Sydex, p. 21

Stengel, “Old Computer Ads [Amiga].”

Lewis, “Commodore Introduces New Amiga.”

Owen, “PC Ditto.”


Today, differentiation is more important than generic compatibility. Microsoft and Sony invest
millions on exclusives—games programmed for their console solely—in order to draw consumers away from rivals.

The official Pasofami site's hit counter has been active since 8/20/1996. Marat Fayzullin also notes on the active iNES site that his emulator was released in 1996, after Pasofami was already available. Kun, “Funny Fantasy,” also provides download links to versions of PasoFami up to 2.7a, with a ROM pack dated from June 27, 1996. See http://www.ee.ust.hk/~algoma/music.html

Some sources will also refer to it as PASOWING, though this is a ‘translation error’ related to the DOS filenames being set in caps. The Windows binary was labeled PasoWinG, as it required the [WinG .dll] file to run properly.

Cochems, “Marcel de Kogel.”

Nielson, “The Nintendo Entertainment System (NES) FAQ v.1.12.”

Fayzullin, “INES.”

Fayzullin, “iNES.doc (v 0.6).”

Fayzullin, “Nintendo Entertainment System Architecture (version 1.4).”

Altice, “Interview: Paul Robson, programmer of the NESA emulator.”


Conte, “cajoNES version 0.99.”


Trainers were a 512-byte code block originally used to hack games into a particular mapper. Trainers are deprecated and largely ignored by current emulators.

See Dane, “Help with Nester.”

This remains an open question. Since emulators are largely designed by individuals and distributed for free, it is up to the developer’s discretion.

Another important trace is the ‘Date Modified’ field common to Windows Explorer or OS X’s Finder. ROMs freely available online are often unmodified versions of images that have been in circulation since their initial dump. When I downloaded the well-seeded ‘NESRen’ torrent in March 2012, for instance, it contained hundreds of .nes files with ‘Date Modified’ entries predating 2000.

Bloodlust Software, “NESticle Version 0.41 [README.TXT].”

goroh, “Nesticle(ver 0.42) stafie information.”

And still does. See http://www.zophar.net/savestates/nesticle.html

Sliver X, “Hacking The Legend of Zelda NESticle save states.”

According to the NesCartDB, Super Mario Bros. is the most commonly dumped ROM in all three regions (North America, Europe/Asia, and Japan). See bootgod, “Top 10 dumped games.”

Demiforce, “Final Fantasy 2j English Patch [139readme.txt].”

Bloodlust Software, “NESticle Version 0.41 [README.TXT].”


Bloodlust Software, “NESticle Version 0.41 [README.TXT].”

TASVideos, “Emulator Resources.”

“FCEUX,” Zophar’s Domain.

See adelikat, “How To make a Tool Assisted Speedrun.”

adelikat, “FM2 Movie file format.”

See Mong (aka mmbossman), “Submission #2486: mmbossman's FDS Arumana no Kiseki in 08:26.67.”

A literal analog to this setup process is the calibration necessary in magnetic tape recording. Audio engineers who record analog must, for instance, carefully align, clean, and bias their tape machines in order for the recording to play back properly.

Kraft (aka Gigafrost), “Submission #690: Gigafrost's NES Zelda 2: The Adventure of Link ‘glitched’
in 06:16.93.”


“Ibid., 124.

“adelikat, “Submission #1324: adelikat’s NES Gradius in 10:52.35.”

“Hayles, Electronic Literature, 45.

“TASVideos, “Luck Manipulation.”

“For example, see T.M. (aka knbnitkr), “Submission #2836: knbnitkr's GB Makai Toushi SaGa in 01:47.17.”

“Teti, “As Fast as Impossible: 10 Insanely Thrilling Tool-Assisted Speedruns.”

“adelikat, “Submission #1324: adelikat’s NES Gradius in 10:52.35.”


“W., Lennart (aka Baxter) and AngerFist, “NES Mega Man 3, 4, 5 & 6 (USA) in 39:06.92 by Baxter & AngerFist.”


“See the ‘Pure AI TAS?’ thread on tasvideos.org: <http://tasvideos.org/forum/viewtopic.php?t=10725>

““How do I create a enhanced VDP/PPU?” (Hamtar0126). NesDev.

“Montfort and Bogost, Racing the Beam.
Appendix A: Famicom/NES Bibliographic Descriptions

Platform studies owes much to the discipline of bibliography, which provides systematic descriptions of manuscripts, books, codices, pamphlets, rolls, and other printed matter. Bibliography is itself split into manifold sub-disciplines, each with its own corpus of study, technical language, and process. ¹ Readers are likely most familiar with enumerative (or reference) bibliography, the list of works, print and otherwise, found at the end of this and other books. In its utilitarian form, the enumerative bibliography is meant to provide readers with enough information to locate the sources mentioned in the text.

Other branches of bibliography analyze printed matter for evidence of their production. Codicology, for instance, narrows its study to the codex as a material form, distinct from papyrus rolls (papyrology) or various legal documents (diplomatics). Similarly, analytical bibliography studies printed books as ‘products of a particular manufacturing or technical process.’² In other words, the content of the book is not the focus. Rather, the bibliographer examines the work as a specific material artifact comprised of ink, paper, binding, printer's marks, and so on. Two books with identical authors, chapters, and words may in fact exist as wholly distinct physical objects. The related discipline of descriptive bibliography widens the analytic net to describe all features of a particular book, whether they are related to the print process or not. Stray marks, bookplates, page counts, and missing leaves all fall within the descriptive purview.

To claim that videogame bibliography demands a closer allegiance to the practices listed above assumes that a unified practice called ‘videogame bibliography’ even exists. At their best, videogame citations adhere to the barest enumerative models. Even in those texts that most seriously grapple with electronic artifacts as objects that exhibit physical properties worthy of description, such as Kirschenbaum’s Mechanisms or Montfort and Bogost’s Racing the Beam, videogames are still afforded scant bibliographic information. Their treatments within the text are generally better—Kirschenbaum, for instance, devotes an entire chapter to a ‘forensic investigation’ of the Apple II game Mystery House—but the comprehensive work done in that chapter is supported by the following bibliographic entry:


The entry clearly uses the standard enumerative listing as a model, but in a study so intimately concerned with the material substrate of disks and drives, what does this tell us about the object itself? What is the disk size and file format? What do the multiple dates
denote? Are they the release dates of the original disk or the disk image available for download? Kirschenbaum mentions the APPLEWIN emulator as a means to play Apple II disk images, but was that the emulator he used to investigate *Mystery House*? And should we differentiate between emulated and non-emulated forms?

One problem is videogames’ relative youthfulness as a medium. Commercial videogames have only been with us since the 1970s. Generations of consoles and games still litter yard sales, attics, pawn shops, and thrift stores. Our familiarity with and access to videogames is still taken for granted, since many of us are old enough to recall first-hand experience with the entire history of videogames—a claim that cannot be made by scholars of other media. There is an implied assumption that we all know what a *Super Mario Bros.* cartridge looks like, so why bother with thorough descriptions.

Consider the following citations for *Super Mario Bros.* culled from a number of scholarly texts on videogames. In Newman, all references to *Super Mario Bros.* are listed with title alone—no dates, authors, or bibliographic citations. In contrast, traditional printed texts, films, and even websites merit standard enumerative listings. The game fares the same in Kline—no date, attribution, or citation. In Wolf, we see a slight improvement, complete with media-specific distinctions to indicate that both a videogame and a film share the same title: ‘*Super Mario Bros.* (game, 1985; film, 1995).’ In Montfort and Bogost, we have the following: ‘Nintendo. *Super Mario Bros.* Nintendo Entertainment System. Designed by Shigeru Miyamoto. 1985.’ And in Bogost’s *How to Do Things with Videogames*, a terser version: ‘*Super Mario Bros.* Nintendo Entertainment System. Developed and published by Nintendo, 1985.’ In both cases, we see the emergence of a console, acknowledgement of a developer/publisher, and a US release date. Bateman is a notable outlier, tipping his hat to materiality, despite a factual inaccuracy in the citation: ‘Nintendo EAD (1983). *Super Mario Bros.* [Cartridge], Nintendo Entertainment System, Kyoto, Nintendo Co., Ltd.’

Compare these citations to bibliographic descriptions found among fan communities. BootGod’s NES Cart Database lists fourteen individual entries that include ‘Super Mario Bros.’ in the title. These range from the original Famicom release to the German RevA multi-cart *Super Mario Bros./Tetris/Nintendo World Cup*. Each entry include copious technical details, part listings, scans of carts and boards, manufacturer information, revisions and variations, and so on. The NintendoAge game search, which pays closer attention to variations in cart labels, box art, and related miscellany, lists more than two dozen entries for *Super Mario Bros.* The rabbit hole runs deeper when we consider arcade versions, ports, tabletop toys, ROM hacks, and the hundreds of extant pirate cartridges featuring *Super Mario Bros.* that circulate worldwide.

Are these variations important? Consider an analog from the print world:
Would this citation be sufficient for a scholarly article? Am I citing the original Russian? If not, who were the translators? Is the French translated inline or in footnotes? Is the text based on the original manuscript, the printed version, a later revision, etc.? I use this example as an extreme case, but the above citation style is the shorthand we see used time and again for videogames.

Worse yet is the conflation of emulated and physical videogame artifacts. Chapter 5 has hopefully divested readers of the notion that ROMs are identical to cartridges. Emulation serves admirable aims in preservation, archiving, scholarship, convenience, and accessibility, but it is never perfect. In *The Medium of Video Games*, Wolf provides a prescient warning for scholars relying on emulation:

> For researchers trying to track down hard-to-find games, emulators can sometimes give a good idea what certain early games were like. However, not all emulators give exact renditions of the games they are emulating; graphics may not appear at their original ratios, and the experience of watching a computer screen is often quite different from that of a television screen, or better still, a period television of the sort on which the games would have been played. Emulators can be of use in video game research, but users should beware of the differences and get firsthand experience whenever possible.¹²

Even the lowly 6502, an ancient slab of silicon compared to today's microprocessors, is an almost infinitely complex bit of hardware, subject to manufacturing bugs, temperature fluctuations, corrosion, and programmer exploitation. And this is only one component of a complex console, a machine that continues to yield surprises unknown to programmers and engineers of the Famicom era. And while ROMs may reproduce identical visual results when played alongside physical cartridges, we know that they often carry along unseen textual artifacts of their history, circulation, and distribution. None of these aspects should be ignored when we take bibliographic account of our digital objects.

Montfort, following the work of Clara Fernandez-Vara, proposes that emulators count as ‘editions’ (or ‘printings’) of a computer.¹³ This approach leans in the right direction, but there is an assumed hierarchical and chronological organization that belies the reality of emulated systems. As he says, ‘The first edition would be the original piece of hardware,’ but the ‘original’ is often not a given. In many cases, for both hardware and software, emulation precedes the final form. Even in the Famicom era, game software often ran on an emulation terminal before it was burnt to an EPROM for hardware testing. In such cases, the physical object is the edition of a prior digital form.

Though I have listed specific works and authors above, the point is not to pick on this
or that scholar. The point is to raise the bar for the minimum acceptable quality of bibliographic information. Quality descriptive bibliography is hard work. It is, on its own, a platform studies problem that warrants careful attention to the types of descriptions available for any given computational object. In other words, rich bibliographic records necessarily require a baseline technical understanding of the objects they describe. If I do not understand the form and function of the Famicom’s various mappers, for instance, I may not comprehend their importance to a game’s descriptive listing. Likewise, the bibliographic description that suits a Famicom cartridge will not necessarily suit a ColecoVision cartridge, TurboGrafx-16 HuCard, Xbox Live Arcade download, or PlayStation 3 Blu-Ray disk. There is not a generic enumerative style that will apply to all videogames across all platforms.

This presents a potential problem for videogame texts with a wide scope. As a Famicom scholar, one may possess the proper terminology to describe that platform’s media, but lack the platform-specific knowledge to cite a PlayStation 2 game. The listing for the latter will suffer as a result. This book will inevitably face the same problem. Chapter 2 alone lists dozens of arcade games, each with its own peculiar hardware. Granting each its due description poses a sizable research challenge. One solution is to build up a body of platform-specific descriptions that others may use as a model for their own research, in the same way that codices or contracts have their own unique languages and methodologies. Once a workable model forms, it may disseminate out to the community for adoption and, more importantly, refinement. But such shared knowledge will take time and work.

*I Am Error* aims to be an example of good form, both in theory and practice. Outlined below are three models for bibliographic records of Famicom/NES artifacts, both physical and digital. The first is a proposed enumerative form suitable for scholarly texts. This is the notation I adopt for all Famicom/NES cartridges and disks cited in the book. The second model describes ROMs meant for play in a compatible emulator. Since ROMs are a different digital object than the hardware they describe, they deserve a separate entry. Again, I have adopted this model throughout the book, making sure to notate when a game is played on real hardware or on an emulator, especially when screenshots are provided. The final example is a lengthy description of a single object (a *Duck Hunt* NES game pak), similar to how an analytic/descriptive bibliographer might collate an early manuscript. While such descriptions are not practical for the purposes of endnotes or an enumerative list at the back of a book, they are essential for the future record of scholars, librarians, archivists, and the like.

The proposed listings are models, not canon. Especially in the final example, I borrow extensively from the bibliographic tradition, even using standard collations for printed matter included with the game. Over time, these conventions may prove inadequate. But until videogame bibliography develops its own voice, it must speak with borrowed tongues.
Listing 1: Enumerative type for citing Famicom, NES, and compatible cartridges and disks.

General Format:

*Title.* Platform (media), TV format [Region]. Catalog ID (Form, Revision). PCB Class [Mapper | ROM1 size/type | ROM2 size/type | … | Lockout model | Mirroring ]. Developer {Credits}: Publisher, Release date.

Videogame authorship poses problems for citation, especially games hailing from the early years of the medium. The standard enumerative model that lists author first is often stymied by the absence of a single author (since games are usually designed by teams of tens or hundreds) or any credits at all. Listing the game title first is the preferred model. Games are commonly known by their title rather than by their designers, so this aids readers consulting citations. It also groups similar titles across divergent platforms. Comparing the myriad ports of *Donkey Kong* is far simpler when they are listed sequentially.

The choice of how and where to list the game’s creative team is up to the discretion of the bibliographer. One choice is to treat the videogame as an ‘edited volume,’ using the creative lead/director as the ‘editor.’ This does not solve the auteur-centric problem inherited from cinema, wherein the director takes top billing above all other collaborators, but it does serve the practical purpose of narrowing down dozens or hundreds of contributors to a short list. I opt to list known contributors in braces next to the developer credit. This may be omitted for editorial purposes or if the contributors are unknown. There are no perfect solutions to videogame attribution, only those that suit the text at hand.

The title should be as accurate as possible. Japanese titles in particular pose problems for Famicom citations (in English) since the labels often display Japanese titles, English titles, or a mixture of both. When possible, list the most prominent title first, along with the alternate title in parentheses. When the title is a translation not seen on the actual cart, indicate it as such.

Next list the platform and media type. Famicom games come as carts or disks and should be labeled as such. Pin sizes further differentiate Famicom (or pirate) carts from NES game paks. Pin sizes may be truncated to ‘72-p. cart’ or ‘60-p. cart’ if preferred. The TV format and region follows, indicating the cart’s proper playback mechanism and country of origin.

Catalog IDs designate Nintendo’s internal cataloguing scheme, though many third-party licensees adopted similar nomenclature. Within parentheses, one can list further details about the cartridge form factor (e.g. 3- or 5-screw, prototype, test cart) and any...
relevant revision information (e.g., bug fixes or scrubbed content were often shipped as separate revisions).

The PCB class and subsequent cartridge hardware descriptions can vary in length according to the complexity of the mapper and the underlying ICs. When applicable, the mapper should be listed first, followed by ROM types and capacities, lockout chip (or circumvention hardware), batteries, SRAM, and sound chips. The last element in this field will designate the cart’s mirroring setting (e.g. ‘V’ for vertical, ‘H’ for horizontal, ‘MC’ for mapper-controlled).

The final line lists the development house, any known contributors, publisher, and release date. Many FC/NES games do not have known release dates. Sometimes we know the specific day, other times merely the year. Use the most accurate date possible.

Family Computer Disk System games require a slightly different format. The media type is double-sided QuickDisk, which may be truncated to ‘d.s. QD’ if preferred. Unlike cartridges, FDS disks only have a single capacity, so it is not technically necessary to list the number each time. However, for the sake of consistency, I have chosen to do so in the examples below. Also note the number of disks when known and, if applicable, the disk side(s) the game occupies.

**Examples:**


Listing 2: Enumerative type for citing Famicom, NES, and compatible ROMs/patches/save states used in emulation.

General Format:
Original cartridge/disk title [Type]. Author. “Filename and extension.” File size. Mapper format: Mapper number. [File header in byte format]. Date modified. Emulator: <Download source>

Though originally derived from a dump of a physical cartridge, a ROM intended for play on an emulator is a fundamentally different object than its progenitor. ICs that are separate in a physical cart are combined into a single binary data stream and appended with a header to describe its contents. The underlying data may be identical, but its form and configuration are significantly altered. As such, the ROM file demands a separate enumerative format.

The original cartridge or disk title is still listed first, as above. A small bracketed designator indicates the format (ROM, IPS patch, save state, etc.), followed by the name of the file(s), including extension(s). Of course, since files may be renamed at the user’s discretion, such titles may vary considerably. However, ROM files’ names and contents have proven surprisingly resilient, contrary to the apparent ‘ephemerality’ of digital artifacts. Likewise, homebrewed utilities for naming ROMs according to prevailing community standards have ironed out many of the inconsistencies that arise from user intervention, file transfer, hacking, and so on. Certain ROM hacks, demos, or homebrew programs may have a known author. In such cases, the author may be listed prior to the file name.

Next list the file size followed by the mapper format and mapper number. In combination, these details can indicate whether a given ROM has been altered from its initial mapper configuration in order, for example, to add additional capabilities, increase ROM space, play on a specific emulator, and so on. The mapper number indicates a specified cartridge hardware configuration and emulator compatibility. Not all emulators support all mappers. In most cases, the mapper format will be iNES or iNES 2.0, since those are the reigning standards. However, alternatives did and do exist, so they should be specified.

All valid .nes files will include a 16-byte header. If the format is a ROM, the header should be listed in full. For iNES headers, grouping bytes into four groups of four is recommended for better readability. Bytes do not need to be labeled with hexadecimal notation (e.g. ‘$’) unless it aids readability. It is also acceptable, based on the bibliographer’s preference, to list the ASCII equivalents to the byte-encoding. For non-iNES formats, a comparable header description should be included.

Next list the ROM’s last date modified field (not the user’s last access, as used in
website citations) and, if applicable, the emulator(s) used for analysis. For patches, hacks, or save states, the release date may be available and would be preferable to the modified date (though they are typically identical). Since emulators vary widely in accuracy, the emulator listing provides the reader with information about how the author viewed the particular file. If Nesticle is listed rather than Nintendulator, for instance, the reader will know that the file's raster effects, sound, or palettes may have been emulated improperly.

Finally, if known, the file’s download source may also be listed in angled brackets. Standard URL format is desirable.

Examples:


**Listing 3: Descriptive bibliography.**

**General Format:**
The descriptive bibliography expands on the enumerative styles listed above by describing a specific object rather than a generic type. As such, it should be as exhaustive as possible, describing not only the cartridge itself, but also any accompanying boxes, manuals, inserts, packaging materials, marks, and so on. Again, the content of the videogames or instruction manual is not important—the physical object is the focus.
Example:


**Cartridge Hardware Profile:**
- PCB Class: NES-NROM-128-01
- PRG-ROM: 16 KB
- CHR-ROM: 8 KB
- Mirroring: Vertical
- Battery Backup: No
- WRAM: 0 KB
- VRAM: 0 KB
- CIC Version: 3193A
- TV Format: NTSC
- Region: US

**Cartridge / PCB Description:**
The cartridge is grey molded plastic, measuring 133 x 118 x 16 mm. Held vertically, the bottom two sides are notched 7 mm, leaving the bottom 25 mm of the cartridge narrower to accommodate the NES console’s internal design. There is an 18 x 26 mm notch 18 mm from the left edge of the cartridge (mirrored on the verso) that serves as a finger hold for inserting and removing the cartridge from the console. Ridges are molded into both sides of this notch and extend vertically along the front of the cartridge from the notch’s right and left edges.

A 97 x 55 mm sticker is affixed to the front of the cart, featuring a reproduction of the game’s box artwork. The upper 6 mm of the sticker is folded around the top of the cartridge and features the game’s title (visible when the cart is inserted in the console). To the bottom left of the sticker is an embossed 10 mm triangle indicating the cart’s proper console insertion direction.

The back of the cartridge has a 7 mm bezel around the left, right, and top edges. Near the bottom center is a 31 x 79 mm sticker featuring care instructions for the cart. Below this sticker is an embossed stamp featuring the text: Nintendo® | PAT.PEND. MADE IN JAPAN. The molded plastic case is formed from two halves, which are bound on the back by five screws arranged in an ‘x’ pattern.

The upper left and right sides of the cart have six raised ridges, presumably for gripping purposes. The bottom of the cart has a hollow well approx. 13 mm deep, allowing room for the exposed edge of the PCB.
The interior of the cartridge is largely empty, save for a small PCB held in place by molded plastic ridges. The PCB measures approx. 40 x 100 mm, which a slight notch on either edge, mirroring the overall cartridge design. The board’s recto features three prominent integrated circuits: the CIC 'lockout' chip (used to detect and prevent unlicensed or pirated carts from playing in the console), the CHR ROM (typically used to store the individual graphic sprites), and the PRG ROM (used to store the program data).

Box:
The cartridge, cartridge sleeve, instruction booklet, and Styrofoam block are housed in a cardboard box (178 x 125 x 22 mm). Oriented vertically, the box opens at the top with three protruding 'tabs' cut to fold inward so the box closes. The largest, center tab (which extends from the back of the box) is 39 x 125 mm with rounded corners on its protruding edge. It is folded 16 mm from its top edge. A small black sticker (20 mm diam.) with ‘Nintendo®’ printed in white is affixed on the cardboard’s fold, though residue on the box front indicates it was originally used to seal the box. Two smaller tabs (23 x 23 mm) extend from the box’s sides, with their facing edge rounded to accommodate the downward fold of the center tab. The box’s exterior is all-over printed with a black background containing small stars.

The box front features a stylized detail of the in-game graphics, followed by the following text: ‘DUCK HUNT | [‘Nintendo ENTERTAINMENT SYSTEM’ logo]’. To the bottom left is an illustration of the series logo, depicting the light gun accessory and descriptive text (LIGHT GUN | SERIES). At the bottom right is the Nintendo seal of approval, featuring the following text: THIS SEAL IS | YOUR ASSURANCE THAT | [Nintendo logo] | HAS APPROVED AND | GUARANTEED THE | QUALITY OF THIS | PRODUCT.

The box verso lists the title, summary text of the game’s features, four screen photographs of in-game footage (with CRT screen bezel), suggested number of players, the Nintendo logo, seal of approval, genre logo, barcode, and the following legal text:

NINTENDO IS A REGISTERED TRADEMARK OF NINTENDO OF AMERICA INC. | TRADEMARK OF NINTENDO OF AMERICA INC. © 1985 NINTENDO OF AMERICA INC. The cardboard at top center is perforated by a 30 x 40 mm un-punched ‘hang tab.’

The box’s left edge is printed with the game title and Nintendo logo. The right edge is printed with the game title and warranty information: WARRANTY | 90 day limited warranty on game paks | (complete warranty information inside box). The box’s center tab is printed with the game title, publisher information (NINTENDO OF AMERICA INC. | P.O. BOX 957, REDMOND, WA 98052 U.S.A), and the text, ‘Made in Japan’.

A rectangular block of white Styrofoam (39 x 117 x 19 mm) rests in the bottom of the box. Since the box is taller than the cartridge, the Styrofoam is used to seat the cartridge properly at the top of the box.
**Sleeve:**
The cartridge is housed in a black vinyl sleeve (135 x 123 x 18 mm) used to prevent dust from accumulating on the cartridge’s exposed PCB contacts. A trapezoidal section of the vinyl is cut away, approximately 50 mm from the top and 77 mm from the left side (although left or right is relative to the original orientation of the sleeve, which is unknown). This cutaway allows the cartridge to be removed easily from the dust sleeve. The text ‘Nintendo®’ and a simple geometric border are stamped in red ink at a slight angle horizontally on either side of the sleeve.

**Booklet:**
Ff. 8, glossy paper, 103 x 133 mm, ‘pamphlet-style’ single staple binding. The instruction booklet contains a description of the Nintendo seal of approval, game objectives and instructions, an illustrated diagram for connecting the NES to a TV and its peripherals, FCC regulations, a Memo space for notes, and a description of the game warranty.

**Additional Notes:**
Duck Hunt was developed internally at Nintendo, but no specific production credits are listed on the cartridge, in the instruction booklet, during gameplay, or on the box. Extant documentation indicates that several first-party launch titles for the NES were developed by Nintendo Research and Development 1 (R&D1), Nintendo’s oldest development team, headed by Gunpei Yokoi. Wikipedia cites sources designating Takehiro Izushi as the game supervisor, Yokoi as the producer, and Kōji Kondō and Hirokazu Tanaka as the game’s composers.

---

² Greetham, 7.
⁴ Newman, *Videogames*.
⁵ Kline et al., *Digital Play*, 177.
⁷ Montfort and Bogost, *Racing the Beam*, 165.
⁸ Bogost, *How to Do Things with Videogames*, 177.
¹² Wolf, 186.
¹³ Montfort, “Emulation as Game Facsimile (or Computer Edition?)”
¹⁴ My thanks to Neal Wyatt for her helpful suggestion on the model structure.
Appendix B: Glossary

10NES: The lock-and-key ‘handshake’ software that runs on the Checking Integrated Circuit (CIC).

2A03: Shorthand for RP2A03G, the IC package that contains both the Famicom’s modified 6502 CPU and custom APU.

2A07: Shorthand for RP2A07G, the PAL version of the 2A03, which features a modified memory divider ratio and adjusted PCM audio playback rates.

6502: Shorthand for the MOS Technology 6502 8-bit microprocessor. The Famicom’s 2A03 package featured a Ricoh-manufactured revision of this popular chip lacking decimal mode. Many popular computing devices used the 6502, including the Apple II, Commodore 64, and Atari 800XL.

Accumulator: A special 8-bit register of the 6502 used to store data for the purposes of arithmetic and/or logical operations.

Address: The location of a data word in memory.

Advanced Video System (AVS): The initial prototype of the Nintendo Entertainment System, designed by Lance Barr and shown at the 1985 Winter Consumer Electronic Show. Its modular design included a control deck, cassette tape storage, QWERTY keyboard, infrared controllers, joystick, light gun, and a musical keyboard.

Assembly Language: A low-level programming language corresponding closely to its host CPU architecture. Assembly language is a general syntax, not a specific implementation.

Attribute Table: A 64-byte array located at the end of the Famicom’s four name tables that designates the palette entry used for each 16x16 pixel (2x2 tile) area of the background.

Audio Processing Unit (APU): The Famicom’s programmable sound generator, which has five dedicated channels: one triangle wave, one noise, two pulse-width, and one delta modulation.

Bank Switching: A technique used to extend the usable memory addressable by a microprocessor. In the Famicom, mappers could swap banks of ROM into the CPU’s address space when commands were passed to a specific hardware register. The bank switching technique permitted Famicom carts to exceed the initial memory limitations of the NROM board.

Binary: A base 2 number system that represents all numbers as combinations of either 0 or 1. For instance, decimal 9 is 1001 in binary. Binary values in the book are prefixed with ‘%.’
Bit: Shorthand for ‘binary digit,’ the smallest fundamental unit of information that a computer can understand, abstracting the two physical states of any bistable element, e.g. the ‘on’ or ‘off’ state of a semiconductor gate.

Bit Flag: A single bit used to indicate the occurrence of a specific ‘event,’ e.g. a carry overflow resulting from binary addition.

Black Box Games: The first game paks released for the US Nintendo Entertainment System that shared a uniform packaging style, most prominently the black star field background and exaggerated pixel graphics. Some collectors extend the designation to include a few exceptions, such as the silver box releases Kid Icarus and Metroid.

Byte: An eight-bit unit of data, typically represented as either eight binary digits (%01011101) or two hexadecimal digits ($5D)$.

Card edge connector: The ‘top loading’ style of cartridge connector that was the industry standard prior to the NES’s ‘front loading’ zero insertion force connector.

Cathode Ray Tube (CRT): A highly pressurized tube containing a barium-coated cathode that, when heated, emits negatively-charged electrons. A narrow gun focuses electrons into a beam. The wider side of the tube opposite the cathode is coated with a luminescent phosphor material. When the gun sweeps across the screen in a fixed pattern, its beam strikes the coating, causing the phosphor’s electrons to emit visible light. CRT is also a shorthand term used to describe a television or monitor that contains such a tube.

Character Internal RAM (CIRAM): The 2KB portion of VRAM that stores the Famicom’s name and attribute tables.

Character ROM/RAM (CHR-ROM/CHR-RAM): The cartridge IC that contains the pattern data used to draw sprites and background tiles onscreen. CHR-RAM is fed this data during program execution.

Checking Integrated Circuit (CIC): A hardware microcontroller included in all Nintendo Entertainment System consoles and game paks. When a pak is inserted, both lockout chips execute the 10NES ‘handshake’ software to ensure that the pak is valid, i.e., manufactured by Nintendo under their licensing terms.

Dendy: The unauthorized Russian Famiclone system originally marketed and sold by Steepler.

Digital/Directional Pad (D-Pad): The patented cross-shaped directional input device first devised by Nintendo engineer Gunpei Yokoi for the portable Game & Watch LCD systems. The joystick alternative became the de facto standard for game controllers after the NES’s release, continuing until the advent of 3D gaming. Also commonly called the ‘cross pad’ or ‘plus pad.’
**Emulator:** Originally a hybrid hardware/software solution meant to mimic (and often augment) a target platform, specifically for the purposes of supporting legacy software. In contemporary videogame parlance, an emulator is software designed to play games from older consoles (or PCs) on a more modern machine.

**Erasable Programmable Read-Only Memory (EPROM):** A ROM whose contents can be erased (typically by exposing it to sustained ultraviolet light) and rewritten.

**Famiclone:** A hardware clone of the Nintendo Famicom, typically associated with the unauthorized sale and distribution of pirate software. Since Nintendo’s NES hardware patents have lapsed, Famiclones are no longer illegal to manufacture. Thus third-party NES-compatible consoles are sold in videogame stores or used as a hardware baseline for affordable computing in developing nations.

**Family Computer Disk System (FDS):** The disk drive add-on to the Family Computer originally intended to make up for the shortcomings of cartridge mask ROMs. The disks initially had higher capacities, permitted game saves, and were cheaper to manufacture. Due to piracy and the introduction of cartridge mappers, Nintendo eventually abandoned the peripheral.

**Family Computer:** Nintendo’s first cartridge-based videogame console released in Japan in July 1983. It is significantly smaller than its international counterpart, the NES, and is distinguished by its white and red color scheme, top-loading cartridge slot, and hardwired controllers.

**Game Pak:** The marketing term devised by Nintendo of America’s Gail Tilden to describe NES cartridges.

**Glitch abuse:** A practice in the speedrun community that exploits programming errors, bugs, or other in-game errata to grant the player a competitive edge.

**Glob Top:** A low-cost method of semiconductor production that bonds and protects the IC and its connections with a thick coating of black epoxy resin.

**Hexadecimal:** A base 16 numbering system commonly used in assembly language programming. Values 0-9 are numbered normally, but 10-15 use the characters A-F. Each digit of a hexadecimal value represents four bits. Hexadecimal addresses and values in the book are prefixed with the ‘$’ character. In other sources, a trailing ‘h’ may also be used (e.g., $3C0 = 3C0h)

**Hex Editor:** Software that represents a game’s raw binary data file as hexadecimal-encoded bytes, specifically for the purposes of editing, hacking, or analysis.

**Homebrew:** A term adopted from the home beer brewing community to describe videogames and software tools created by amateur or non-professional programmers.
**Horizontal Blank (HBLANK):** In Famicom programming, the HBLANK is the interval of time between when the electron reaches the edge of the screen and when it resets to the opposite edge to resume scanline rendering.

**interNES (iNES):** An early shareware NES emulator, released in 1995, as well as the community-adopted standard header appended to NES ROMs to indicate mapper number, PRG-ROM banks, mirroring, etc.

**Kill Screen:** An impassable final screen of otherwise ‘infinite’ arcade-style games, usually resulting from programmer oversight, limitations of 8-bit architectures, and prolonged expert play.

**Large Scale Integration (LSI):** An integrated circuit containing roughly between one thousand and tens of thousands of logic gates/transistors.

**Light Gun:** A videogame peripheral that employs a light-sensing circuit housed inside a gun barrel to simulate target shooting on a television monitor.

**Launch Title(s):** The software available on the same day that a new videogame platform is released.

**Localization:** The process of modifying a game’s content for release in a foreign market. In the simplest cases, localization strictly involves translation (e.g. Japanese to English menus, text, dialogue, etc.). In sophisticated examples, cultural allusions that might be misunderstood (or found offensive) are either updated to references relevant to the target audience or excised completely.

**Lockout Chip:** see Checking Integrated Circuit (CIC).

**Mapper:** Additional cartridge hardware that permits the Famicom to perform tasks that were not possible with the ‘base’ hardware, such as bank switching or scanline timing.

**Mask ROM:** A cost-efficient form of read-only memory named after the ‘masking’ technique used during fabrication.

**Memory Management Controller (MMC):** The official name for Nintendo’s ASIC mappers.

**Memory Map:** A tabular representation of a microprocessor’s addressable memory and each segment’s associated function and/or contents.

**Metatile:** A graphical and computational object composed of multiple sprites (or background tiles), typically used to build characters larger than a platform’s default sprite size or to locate and compress larger ‘chunks’ of a game world.

**Mirroring:** Duplicating a memory area across multiple addresses in a memory map.
**Name Table:** A 960-byte region in PPU memory used to store pattern tile indices designating the arrangement of the 32x30 tile background. The PPU has addresses for four name tables, but sufficient memory to store two. Consequently, two name tables are mirrored.

**NES-001:** The original model of the NES, characterized by its boxy shape and subdued color scheme. The NES-001 included the ZIF cartridge loader, both composite and RF outputs, expansion port, and CIC lockout chip. It is frequently referred to as the ‘front-loader’ or ‘toaster’ model.

**NES-101:** The 1993 redesign of the NES, removing the NES-001’s ZIF connector, expansion port, composite output, and front-loading cartridge slot in favor of the more common top-loading style. Commonly called the ‘top-loader’.

**Nintendo Entertainment System (NES):** The ‘localized’ version of the Family Computer released by Nintendo outside of Japan in 1985. Commonly pronounced ‘Ness’ or ’N-E-S’.

**Non-Maskable Interrupt (NMI):** An interrupt handler generated by the 2A03 that signals the start of the VBLANK period.

**Nybble:** Four bits, or half of a byte.

**Object Attribute Memory (OAM):** A 256-byte segment of (independent) PPU memory that stores attributes for the Famicom’s sixty-four available onscreen sprites.

**OAM Cycling:** A technical term describing onscreen sprite flicker. When sprites exceed the eight-per-scanline limit, programmers cycle the contents of OAM to prevent a given object from disappearing completely.

**Overscan:** A variable image area around the four edges a television or monitor that may not reliably be seen by the viewer. Famicom graphics in the overscan area may be cropped, depending on the display device.

**Pattern Table:** The first 8KB of the PPU’s VRAM. Each pattern table is 4K and contains either 256 background or sprite tiles.

**Printed Circuit Board (PCB):** The material substrate used to support and connect electronic components. Its conductive pathways are typically etched from laminated copper sheets.

**Picture Processing Unit (PPU):** The common name for the Famicom’s custom Ricoh RP2C02G-0 graphics processor. The PPU handles all aspects of rendering the Famicom’s video signal.

**Pixel:** Shorthand for ‘picture element,’ the smallest graphical unit of a pattern tile.
**Platformer:** A videogame genre that involves a character running across, jumping, climbing on, or otherwise surmounting multi-tiered obstacles or terrain, i.e., platforms. Typically, platformer gameplay takes place in scrolling, two-dimensional space.

**Program Counter (PC):** A 16-bit register that contains the address of the next instruction to be executed by the CPU.

**Program ROM/RAM (PRG-ROM/PRG-RAM):** The cartridge ROM directly addressed by the CPU. It contains the program's source code and data.

**Random Access Memory (RAM):** A form of computer data storage that may be both read from and written to.

**Read-Only Memory (ROM):** A form of computer data storage that may only be read from.

**Register:** A CPU memory storage location. In the 6502, registers are eight bits wide.

**ROM:** The colloquial term for ROM images dumped from cartridges for play on emulators. NES ROMs typically include the contents of CHR- and PRG-ROM and an appended header.

**ROM Hack:** A videogame ROM that has been altered from its original commercial release. Often these are simple graphic replacements or enhancements, but they can also include significant revisions to level designs, enemy behaviors, physics, in-game items, narrative, dialogue, and so on.

**Robot Operating Buddy (R.O.B.):** An optically-controlled robot peripheral included both in the original NES Deluxe Set and as a standalone accessory. Although heavily emphasized during the NES's initial promotion and marketing, only two software titles supported it. A Famicom version was also released.

**Run Length Encoding (RLE):** A simple form of compression commonly used in the 8-bit era to eliminate the redundancy of multiple repeated tiles. Instead of listing each tile in sequence, a tile's reference is provided, followed by its 'run length' (i.e., how many times it is repeated) and a final terminating byte.

**Scanline:** A single line, or row, of the raster scanning pattern traced by the CRT's electron gun.

**Scrolling:** The simulation of movement through a virtual space that is larger than that contained in a single television frame.

**Speedrun:** A competitive practice of gameplay that aims to complete a game as quickly as possible, without the assistance of cheat, hacks, or computer tools. In some cases, emulators may be used, but they are not required. A special variation called segmented speedruns permits stitching together multiple runs to form a single, master speedrun. Traditional speedruns are performed in one continuous session.
**Sprite:** A pattern table object that may be moved independently from other objects. Sprite also designates a group of related tiles—e.g., the Mario sprite—though technically such examples are metasprites. The Famicom permits two sprites sizes: 8x8 or 8x16 pixels.

**Sprite 0:** The first entry (position 0) in sprite OAM.

**Tile:** An 8x8 or 8x16 pixel area of graphics data.

**Tool-Assisted Speedrun (TAS):** A speedrun that is performed on and assisted by an emulator and its associated enhancements (i.e., tools), such as save states, re-recording, slow motion, macros, etc.

**Vertical Blank (VBLANK):** After the electron gun sweeps the final scanline of a television field, it must turn off and reset to the upper corner of the screen. VBLANK describes either the distance it must travel or the period of time it takes to do so, depending on the context. In Famicom programming, VBLANK is the ‘safest’ period to make PPU updates.

**Video RAM (VRAM):** In Famicom parlance, VRAM describes the memory allotted to both the PPU’s name tables and palettes (CIRAM), as well as the CHR-ROM/RAM on the cartridge. In some documents, CIRAM and VRAM are synonymous.

**Waveform:** A visual representation of sound’s variation in air pressure over time. Waveforms are commonly named according to their approximate geometric shape, e.g. square wave, triangle wave, sawtooth, etc.

**Word:** A two-byte unit of data. This is the common length of an address used in Famicom/NESt programming (e.g. $2001). (Note that word length varies according to the processor architecture.)


**Zapper** (See Light Gun).

**Zero Insertion Force (ZIF) connector:** Nintendo’s patented, VCR-style cartridge loading mechanism used in the NES. Though novel, it proved to be more susceptible to corrosion, debris, and wear after longterm use. Nintendo eventually released an updated console, the NES-101, that returned to the more traditional card edge connector.

**Zero Page:** The memory addresses located at the beginning of a CPU’s memory map, beginning with a leading zero. In 8-bit architectures, addressing zero page memory takes fewer processor cycles, so it is used to store variables that require frequent access.
Sources

**Famicom / NES Cartridges, Disks, and ROMs**


Donkey Kong [ROM]. “Donkey Kong.nes” (25KB). iNES: 0. [4E45531A 01010000 00000000 00000000]. 19 Mar. 2000. FCEUX v2.1.5, Macifom v0.16, NESticle v0.2, Nestopia v1.4.1, Nintendulator v0.975.

Donkey Kong [ROM]. "dkoe.nes" (66KB). iNES: 3. [4E45531A 02043000 00000000 00000000]. 29 Nov. 2010. FCEUX v2.1.5, Macifom v0.16, Nestopia v1.4.1. <http://xkeeper.net/private/pacman/> [Note: ROM originally appeared as a pre-installed download for the red Super Mario Bros. 25th Anniversary Wii, exclusive to Europe.]


Super Mario Bros. [ROM]. "Super Mario Bros..nes" (41KB). iNES: 0. [4E45531A 02010100 00000000 00000000]. 10 Dec. 1999, FCEUX v2.1.5, Macifom v0.16, NESA v0.17, Nestopia v1.4.1, Nintendulator v0.975, RockNES v5.08.


**Wild Gunman [ROM].** “Wild Gunman.nes” (25KB). iNES: 0. [4E45531A 01010100 00000000 00000000]. 17 June 2000. FCEUX v2.1.5, Macifom v0.16.


Print


Gamespite. GameSpite Quarterly 5 (Fall 2010). Print-on-demand.


Online


