# Back To The Classics

Perfecting The Emulation For Digital Eclipse's Atari Anthology

By Jeff Vavasour

Emulation has long been an interest for me. In fact, it was that interest that led me to become a part of Digital Eclipse in its infancy as a game development studio. Growing up as an avid player of an Atari 2600, Atari has long had a special place in my heart. Even in 1993, a year before joining Digital Eclipse, I was dreaming of the idea of getting an Atari 2600 emulator going on my PC. I had a prototype going, but it would be ten years before the dream would fully become a reality. In 1993, the challenge of trying to get an Atari 2600 emulator going on a 486 33MHz was immense. In 2003, our ultra-modest system requirement of a 200MHz Pentium was much more agreeable, and so *Atari: The 80 Classic Games in One* for Windows was born. In 2004, Atari invited us to follow up and expand on the concept with *Atari Anthology* for the Xbox and PlayStation 2—an endeavor that presented its own challenges.

Why would a modest console like the Atari 2600, with its 128 bytes of RAM and 1.2MHz CPU, pose even the slightest threat to a modern console? The answer is in the nature of emulation, and why we'd even contemplate emulation instead in the first place, instead of rewriting the games for the new console.

## HOW EMULATION WORKS

In general terms, emulation is the painstaking process of recreating the experience of playing a game on the original machine for which it was created. We consider the characteristics of every piece of hardware from the controller buttons to the CPU and figure how each adds to the feel of the game.

Technically, here's how it works:

Every computer ultimately runs machine code binaries. Programs are a series of instructions encoded numerically (e.g., in 8- or 16-bit values). The process performs a specific simple mathematical or logical step based on each instruction. People don't typically write programs that way. They write them symbolically in a programming language. Early on in arcade-style games, it might've been assembly language. Subsequent to that, it might've been a higher language like C.

When someone is asked to convert a game from one platform to another, there are a few approaches: rewrite, port, or emulate.

In a rewrite, you just write a new program that has the look-and-feel of the original. That effort is going to be as good as the new programmer's attention to detail might be, and there's a lot of quirky detail in an action game that can be difficult to reproduce precisely given how a game can act so differently in a different player's hands.

In a port, it's easiest to consider a game written in a high-level language like C (though that wasn't at all common in the first half of the '80s or earlier). As the person porting the game, you'd separate the program into two parts. There's the C code that represents the game logic itself, which you try to leave intact, and there's the platform-specific code (for example, a video driver might be considered part of the platform-specific code). Early computers, arcade games and home consoles had video chipsets that bore no resemblance at all to what we have now. So, you'd have to rip out that code and replace it with something that hopefully works the same way on the new platform.

## INACCURACIES

Inaccuracies come up in two areas here. First, the obvious thing is if the game timing, visuals, etc. derived in some way from the platform specific part of the code you ripped out, you've destroyed that part of the game. Likewise, the performance specs of the new hardware itself can alter game play.

For example, in the Williams game *Joust*, the player gets to update position every 1/60th of a second, but the enemies update only as many as they can in the CPU bandwidth that's left. If the processing time runs out for a frame of the game, the rest of the enemies are unable to make a new AI decision until the next frame. Thus, the game's AI slows down as the screen gets more crowded. The game has been balanced to compensate for this. If the exact same code were brought to hardware that was identical in every way, excepting only that the CPU was faster, the AI would be able to "think" faster on a crowded screen. The player would see more aggressive and skilled enemies compared to the original.

It really pains me when I read reviews that talk about how appalling it is that our emulation appeared to slow down somewhere, as, for example, one review commented of the smart bomb effect in the N64 version of *Defender* on Midway's *Greatest Arcade Hits*, released a few years back. The emulation slowed down because the original game slowed down, and emulation strives to reflect every nuance of the original game. There are often timing nuances and sometimes even original code bugs, which become part of a player's strategy in playing the arcade game. For a truly authentic experience, every one of these quirks needs to be reproduced.

A second source of inaccuracy is the C compiler itself and the fact that the new platform is likely to have a different set of instructions in its machine code vocabulary than the old one did. This means the same line of C is not going to be translated into precisely the same set of mathematical and logical steps. Maybe the new platform had different mathematical precision. You could get a butterfly effect where a small mathematical imprecision (either existing and compensated for in the original game, or introduced by the compiler in the new port) gets amplified over the course of a game until it has some significant consequence, perhaps to player firepower or the like.

So, the question is how do you make sure all these potential inaccuracies are kept in check? Well, the basic problem is that you're not running the game on the same hardware as the original game. That means you can't use the original machine code, and that means you can't rely on the original graphics or processor timings being the same. Emulation is the answer to this.

## THE BASICS

In its most basic approach, emulation is an on-the-fly translator. The analogy I'm fond of is this: In porting, it's like you took a foreign movie, gave the script to someone fluently bilingual, and got that person to rewrite the script in English. You'd rely on the translator's appreciation for the nuances of the original language, appreciation for the subtext, the message of the movie, etc. The quality of the product would be entirely a property of the translation effort, and regardless, what is important to one person is not what's important to another. Some double-entendres and the like just don't come across, and need to be replaced with something of equal value, or else ditched.

In emulation, you're watching the original foreign movie, but someone has given you a translating dictionary and all the books on the language's grammar and nuances. Looking up each word on the fly as it's spoken, and appreciating all the impact it has, and still being able to follow the movie in real time sounds impossible. It would be, unless you could think about 10 times to 100 times faster than the movie's pace.

This is what emulation does. It reads each byte of machine code from the original game's binary machine code (not the game's source code), looks up precisely how the original game's processor would've responded to it, how long it would've taken to respond, and what aspects of the hardware would've been affected (video, sound, coprocessors, etc.). It also updates a vast array of data, everything there is to know about the original system's state (what was in its memory, what state its hardware such as its timers were in, etc.), in response to these instructions. Just like the movie analogy, all this maintenance takes a lot of effort on the new platform's CPU, since it takes a lot of code in its native language to explain every nuance of the foreign language it's interpreting. So, likewise, a new platform's CPU needs to be 10 times to 30 times faster than the original hardware to process the original hardware's machine code.

The factor varies depending on just how alien the original hardware's capabilities were, and how many coprocessors it might've had. The "modest" Atari 2600 platform, in fact, is so alien to just about every console and computer platform that came after it, that the code necessary to explain it is extremely convoluted. So, emulating an Atari 2600 is much more challenging than, say, emulating Bally/Midway's *Rampage*, which was released nine years later, but more on that later.

## SOUND

When it comes to recreating the original game's sound, it's a similar process. Sound in an arcade machine or home console is usually built in one of two ways. There will either be some sort of custom sound hardware that's manipulated by changing values that the CPU can access: pitch, volume, tonal quality, etc.; or, the sound samples are built by a CPU and just played out the same way a .WAV file might be on your PC. The first method is typically called FM synthesis; the second is called DAC (digital-to-analogue converter) or PCM (pulse code modulated) sound. (There are minor variations on this. For example, instead of FM synthesis, early arcade games may have a trigger that makes custom hardware produce a small selection of very specific sounds, such as explosions or laser shots in different volumes or pitches. Or, in later games, they might be able to handle compressed audio, which might, for example, take the form of ADPCM instead of PCM.)

In any event, the sound controls accessible to the original game's CPU that are going to drive the sound hardware are either going to be manipulated by the main game's CPU, or one of its coprocessors. As our first step, we simulate the original game's CPU and all of its coprocessors. From there, we're going to see the control values it's trying to send to the sound hardware. Like processor emulation, we must then respond to these values and rebuild sounds according to the same rules the original hardware did. In most of our emulation efforts from the PSone on, we rebuilt the sounds as straight audio samples, like .WAV files. If the platform doing the emulating had the CPU bandwidth to spare, we'd build the sounds in real time, (hopefully) just an imperceptible fraction of a second ahead of when they were played back. (Building sounds in this fashion requires careful timing, and you need a CPU fast enough to make sure you have enough power to spare to maintain that timing. It's akin to trying to stay just far enough ahead of an avalanche so that you can get a nice shower of snowflakes down the back of your neck.)

In the case of FM synthesis (or other custom hardware), building the sound in this fashion also requires a precise rule set for how that sound hardware behaved and what its samples would've looked like. You don't just need to know how it responds to the control values the CPU is giving it, but also how it might be influenced by what it was doing before hand. In its own way, it is just as complicated (perhaps more so) than simulating a microprocessor.

In the case of the Atari 2600 and many of the Atari arcade games, the sounds were created using FM synthesis. So, we'd create special ROMs that would just write every possible unique command to the sound hardware, and capture the results that came out of the amplifier. We'd store these as looped samples (after carefully examining the original sound to determine where its loop point was). Then, in our game's emulator, we'd look for the commands that triggered these sounds and run the appropriate sample (with pitch, volume, etc. adjusted) into our console's sound mixer.

Actually, mixing these samples on the fly is something we've only recently been able to accomplish. For older platforms that didn't have the CPU power to do this in real-time and still have the spare bandwidth to emulate the game's main CPUs, we have to pre-process the sound as much as we can. As anyone viewing a streaming video on the Internet knows, you don't need to be able to stream the video in as fast as it plays back if you're willing to wait for it. So, in our "test bed" emulators, even if the computer is not fast enough, the emulator can take as long as it needs to build the sound, and we write the .WAV-like data to the hard drive, instead of playing it out to the speaker. We can isolate certain sounds in the game by patching the ROM in our test bed, so that other sounds can be shut down. This usually requires us delving into the original ROM somewhat. That requires reverse engineering skills and that's where things can get complicated. Fortunately, since we've already created a full simulator of the main CPU running the original machine code, we can see when it writes values to its sound controls, trace back to see what code did that, and then patch that code as we see fit. In that same vein, we find out where the triggers in the code are that trigger these sounds, and patch our emulators to play the pre-fabbed .WAV files when those triggers are seen.

Generally speaking, this is a fairly safe bending of the rules of emulation, because the code looking for triggers is "looking over the shoulder" of the emulator as it runs the original machine code. It does not affect how the emulator responds to the original code. Beyond that, the CPU tells the sound hardware what sounds to make, but it never checks to see what's coming out of the speakers.

The main hazard of this approach is if a game has a large variety of sounds it can make that are "dynamic", influenced in pitch, tempo, or volume, etc. by game events. Mercifully, most early arcade games do not seem to do this, for whatever reason. In the rare case where this does happen, we have to delve deeper into the original machine code, the triggers that change these properties, and how the sound responded to it. We usually have to do a lot of trial and error and hacking (typical of reverse engineering) to pull this off.

Thankfully, since about the N64 and Dreamcast days, home consoles have been fast enough that we don't need to use any of this pre-fab sound technique anymore and can just build the sounds on the fly, at least when it comes to the early '80s games. When it comes to the '90s games (games much newer than what appeared in *Atari Anthology*), another reason we might not build sound in real-time is if the sound hardware was so complex, that it is difficult to create a rule set that mimics it with sufficient accuracy. For that, we can't even pre-process the sounds in our "test bed" and we instead resort to extracting them out of the original arcade games by plugging a computer into the audio out of the machine and sampling it. We might use the same techniques of hacking the ROM as we would above to trigger sounds in isolation, if necessary, except instead of dumping emulated sound output to the hard drive, we record it from the hacked genuine arcade machine using our computer's "line in" jack.

## MAKING ATARI ANTHOLOGY

For most of the games in the *Atari Anthology* we don't have any source code and emulation based on the ROMs is the only possible approach to recreate these games. On the Atari 2600, precise timing down to the microsecond level is crucial for getting the games even close to perfection. The source code, if we had it, wouldn't help at all though an oscilloscope might.

By modern standards, the Atari 2600 is extremely unusual. The program code and the generation of video signal are completely intertwined. Given that we didn't want to mess around with electronics, our 2600 emulator was converted into a super 2600 debugger. This debugger allowed us to freeze the video beam in time and see exactly what the code was doing at that instant. Or, in fact, by positioning a crosshair anywhere on the window containing our simulated TV screen, another window would tell me exactly what code had been executing at the time that part of the screen was being drawn, and what the state of the simulated CPU was. If I had one piece of advice to give any prospective emulator author it would be this: The first thing you need to do is write a really good debugger to go with your emulator. It'll save you untold grief later. Getting an emulator to behave exactly like the original hardware is a matter of great tedious trial and error, and examination of detail. When things go wrong, it is invaluable to be able to stop your simulation, look at what state it's in, and be able to wind the "logs" back to find out exactly how it got into that state. This made it possible to hunt down subtle behaviors of the hardware. Run side-by-side with a real Atari 2600 we used a re-programmable cartridge (the limited-release *Cuttle Cart*, as it happens), we could also download test code into a real Atari 2600 and see what changes might make our simulation diverge from what a real Atari 2600 might do.

As an example of where a good debugger really helps, for the 2600, not only did I have to emulate the chips that were in the machine, I also had to emulate the chips that weren't there! Some 2600 programs access illegal memory and depend on the results, which come back. A normal emulator might return zero. Not good enough; the games just crash. I had to answer the question "If the 2600 reads memory that doesn't exist, what does it see?" This kind of thing isn't covered in the specifications.

The 2600 is additionally challenging because it would build the video signal on the fly as the raster scanned down the TV screen. There was no video buffer, so the content of the registers never contained the whole picture. You had to look at what was in the registers whenever it changed and know exactly where the raster was supposed to be in order to know how the image was built. Essentially, there were several pages of emulation code that had to be processed at run time to deal with each pixel on the screen. It was such a drain on the CPU that, for a time, it looked like the PS2 wasn't going to be able to pull off the double-speed effect we wanted as a game variation.

That's the big irony about the 2600. What appears to be the simplest machine to emulate is actually one of the hardest and most CPU intensive. It only had 128 bytes of RAM and most games were 2048 bytes of ROM but every cycle counted.

## RECREATING THE EXPERIENCE

There are some specific things we did for *Atari Anthology* to make the games true to the originals. For example, the Atari 2600 console had 128 different unique colors. The circuits for generating those colors are hidden inside a custom chip. Rather than guess, I created a special ROM and downloaded it into my Atari 2600. It was programmed to cycle through all the possible colors. A bar code on the top of the screen identified which color was being selected. The result was then captured with a PC video card, and program read then scanned the captured video, deciphering the bar code on the top of the screen and noting the dominant color that was on the screen with it.

Another interesting problem was the vector games in *Atari Anthology*, such as *Asteroids*. These games did not draw their pictures the way a TV does, but worked more like an oscilloscope. The result was a picture composed of brilliant wire-frame ships, asteroids and saucers. Though a TV doesn't afford the brilliant luminous contrast or smooth lines of a true vector screen, we simulated the fringe effects of the vectors and anti-aliased the lines to bring the experience as close as possible on a normal TV. In fact, Atari's vector games use resolutions higher than a regular (NTSC) television—1024x1024 to be precise. On platforms that support it, we made use of the progressive scan modes to get an even better picture. *Atari Anthology* looks better on HDTV.

For the arcade games in *Atari Anthology*, we went beyond simply duplicating the image on the CRT and added some of the additional cabinet art you would see at the edges of the screen when playing. On many games this even includes the flashing buttons.

The European (PAL) versions are an additional challenge. All of the originals run at 60 frames per second, but TVs in Europe run at 50 frames a second. Six does not divide evenly into five so we have to come up with schemes to get the speed of the game correct.

## CONTROLS

Control is part of the experience, every bit as important as the game's visuals, timings, difficulty, levels, etc. Analog spinners, steering wheels, levers, etc. have all been specifically designed to make the control of the user's ability to control the game intuitive and precise. These are absolutely essential qualities in intense action games balanced to defeat the average player in 2-3 minutes.

Part of the point of a package like *Atari Anthology* is to bring back the fond memories of playing the game in the arcade (or at home). The emulator's video and sound are pretty significant cues to dredge up those memories, but the controls really make a difference. We can't duplicate the shape of the original controller, but if we get the response to the controls right, the player's muscle memory will kick it. It is one thing for the game to look like *Asteroids*—it is a whole other sensation for it to feel like *Asteroids*.

Just like there's no direct equivalent in the instruction set of one CPU to another, there's often no direct replacement for an original game's control and the new platform. Even in a joystick or button game, the travel of the control, position of the buttons, etc. can be an encumbrance. The more frustrating ones for us in trying to create genuine recreations; however, are often the paddle controls and the free-travelling spinners.

The trackball is one of the most innovative controls created. It's a spinner that can be spun on two axes—up/down and left/right, similar to a mouse. A mouse is an integral part of your PC. If you were told you weren't allowed to have a mouse anymore, and instead had to control your cursor with a joystick (or drive your car with a joystick, for that matter), you'd not be happy. It's a barrier for us, strictly as creators of software, that can be most frustrating. It would be absurd to suggest that the user is not an integral part of the equation. The most precise recreation of a game's logic and behavior is hobbled if the user can't get his/her intent across to the game as well as before. The game has been precisely balanced to respond to a decent player using a decently intuitive and precise control.

The best we've been able to offer in response to this, when a precise controller is not available, is to support as many controllers as we can. Beyond that, we provide as many options as possible (unfortunately, sometimes bordering on an overwhelming number of options) in order to tune other controllers to balance between precision and responsiveness. Something as simple as *Super Breakout* creates a problem of balance. You need to be able to move fast, but you need to be precise. Most thumb sticks don't give you enough travel to have a sufficiently wide range of speed with fine precision between those steps.

So, we often have options like "absolute" vs. "relative" mode, where you either get to choose whether your thumb stick will determine the speed of your onscreen paddle (more precise on position, but hard to reverse direction quickly or the like), or the position of your onscreen paddle (but, often, with 200+ positions on the screen, you are going to have trouble picking the exact position of your paddle with a thumb stick). The great thing about a paddle controller—and a trackball, too—is you could give it a good spin for sudden emergency moves, but still stop it dead in its tracks with a touch, and then fine-tune the position. We've come up with all sorts of schemes for attempting a balance between these two qualities on a thumb stick (or steering wheel, or whatever is available), but unless we can get a precise match for the original game's controller, it's always going to be an incomplete solution.

In some cases, modern controls are more complicated than originals (like the pressure sensitive buttons on the PS2 or Xbox). In other cases, we need to find a happy medium between what players expect from the original game and what they expect on the console. For example, we may be influenced by where the gas pedal should typically be located on the modern controller. After we find what we think is the best arrangement for the user, we add in UI elements so they can re-arrange the controls to suit their needs.

One particular problem, in that regard, is the common modern standard that the Start button also be a pause button. Sounds simple, right? Problem is, the Start button should also start the game. So what the Start button should do changes depending on whether a game has already started or not. How does the emulator know if a game has started or not? It doesn't. It just has a huge collection of information about the state of the original hardware and what's in its memory. It's up to us to divine some way of determining what state of the original CPU indicated a game in-progress. That had to be done "by hand" for each of the 85+ games in Atari Anthology. For example, sometimes lights might flash on the control panel of an arcade game when it's waiting for a new player. So, we'd change the behaviour of the console's Start button based on whether or not the emulator saw a recent request to flash the arcade cabinet's lights. Other times, it wasn't so easy. For another example, what if Player 1 presses Start while it's Player 2's turn? It probably shouldn't do anything, but how do you know it's Player 2's turn? Most times, that's just an obscure byte at some obscure location with in the game's RAM. Did I mention we didn't have any source code?

## ADDING EXTRAS

There's more than just game play, graphics, or sound to these games; there's culture.

Interviews with the people who created the games can put the games in a new light. Perhaps you'll learn about Easter eggs you weren't aware of in the game. These interviews may not be to everyone's taste, but it is similar to how some people are curious about what a band has to say about the songs they wrote or performed. For example, by listening to Nolan Bushnell, you'll find out about Atari's game behind the game: Atari contracted chip houses to develop a lot of different graphics chips as busywork to prevent competitors from developing their own consoles.

There is also an element of nostalgia to packages like the *Atari Anthology*. The emulators have the game covered, but with the extras we can attempt to recreate the experience of the entire product. It might sound silly, but if you buy a classic car, it adds something if you have the original manual. And we've found that people seem to appreciate it, for the most part, and it has become an expected thing.

Sometimes the publisher doesn't specifically ask for the content, but we add it anyway because we believe in creating a well-rounded package. Sometimes, they do ask for it, but it's usually left to us to consider the details. Typically, the selection and coordination of materials is entirely between us and the collectors who help us out. This even includes who we choose to interview, most of the time.

An innovation in *Atari Anthology* was the addition of the Challenge Modes. The Challenge Modes add a new dimension to game play by tinkering with the original game experience. Unlike remakes or the like, the game premise isn't changed. Rather, how you interact with the game changes.

The idea for the Challenge Modes came from a couple sources. When we released *Atari Arcade Hits* for the PC back in 1999, there was an interest in updating the graphics a bit. This included adding some more detail to the sprites in *Centipede*, more color to *Battlezone*, etc. As an experiment, one of the optional modes in *Asteroids* was "Trippy Mode." Unlike the other visual enhancements, Trippy Mode affected game play. It left a psychedelic, blurring decaying trail as everything moved. It made for an interesting game and created a unique challenge. It was quite entertaining to us, but didn't fit in with the "authentic play, new look" presentation Hasbro and we were going for in *Atari Arcade Hits*. It was left in, but buried in obscurity in one of the game options menus.

A second unlikely source of inspiration for the Challenge Modes was a pointless thing I'd do for fun every time I upgraded my computer. I wanted to get a real sense for just how much faster my new computer was, so I'd take one of my emulators and remove the code that made one second of simulated CPU time take one second of real time. Without that code, the emulator still did everything properly, but time sped up. It got to the point on my 3.2GHz system, the time from the start of a game of *Robotron* and Game Over three lives later was about 1/3 of a second. However, way back on a 166MHz, the games might be insanely fast but still almost playable. This was the inspiration for the Double Speed Challenge Mode in *Atari Anthology*.

Rounding out the five Challenge Modes were three other challenges: Time Warp mode was a variation on Double Speed where the game speeds up and slows down periodically. In Time Challenge, you had to score as many points as you can in a fixed amount of time. In Hot Seat mode, you would play several games at the same time, switching games—no matter what is going on—every 10 seconds.

In Atari Anthology, there was an interest in creating unlockables to enhance the replay value of the package and encourage people to explore the collection fully, rather than sticking with the games they knew. However, we didn't want to lock out any games that might turn out to be someone's favorite. The Challenge Modes seemed a good solution. They added replay value, but all the original games were still there and accessible from the start.

## LEGAL ISSUES

How do these packages come about? Being a developer, we don't obtain the rights to these games on our own. We talk to the publishers who already own the rights. The publishers we are most likely to end up working with are the ones with the more concentrated catalog of games and who were a dominant presence in some form, both then and now. This minimizes the legal issues, though they still exist.

For example, when spanning 20 years of the industry or more, records may have changed corporate hands several times through buy-outs and the like. There may have also been special contracts for contributors to a game's development. Back then the industry was far from standardized. It can be a very time consuming process for the publisher to verify they even have clear ownership of a game, and that the rights weren't shared, or lapsed or the like.

Multiple owners can make it nearly impossible to get some classics republished. My favorite Atari arcade game of all time is indisputably *Star Wars*, but that would require the consent of not only Atari, but also LucasFilm. This also happens with games that might have been licensed by Atari for North America but are owned by, say, a Japanese company. Namco in fact, owned a number of arcade games with the Atari logo on them.

Though this problem didn't occur with *Atari Anthology*, there have been cases where corporate logos have changed hands. For legal purposes, we have to modify the original games and erase the old logos! The emulator is perfect, but not legally perfect. Similar to the start button, we figure out very carefully how to remove the offending logos without altering the original ROMs and that can be tricky.

The time it takes to research that alone can be prohibitive to getting new games in a compilation. This is part of the reason the first round of games on a new console might be a collection of those released on the previous console (e.g. the PS2 *Midway Arcade Treasures* more-or-less pooled all the PSone titles). In the time it takes to research new titles legally and technically, there isn't time to get those games in the package between the day it's green lit and the day it's to ship. Successful sales, of course, breed follow-ups, and give publishers the time and motivation to expand the catalogue of "legally clear" titles. Meanwhile, we continue to add more games to our library, bringing them to a technical level with the accuracy that consumers have come to expect.

## STRIVING FOR PERFECTION

Emulation brings a long way toward recreating the original game experience. As each successive generation of console comes out, we feel we're edging these games a little closer to getting the experience *just* right. For example, sounds can be emulated more precisely, more games are added to the compilation, new material can be added, better resolution can be achieved, etc. It's a never-ending challenge, but one we're very fond of, because we simply love these games. Doing these compilations is just our humble way of preserving the legacy, and trying to pay these games back for all the enjoyment they've brought us. And, given the continued enthusiasm each release has been met with, and the ever-expanding catalogue of "classic" games as the industry marches forward, it's something we're excited to keep doing for a long time to come.