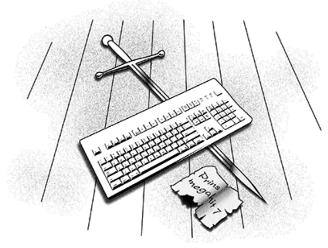
4 Prince of PoC; or, A 16-sector version of Prince of Persia for the Apple][.

by Peter Ferrie

Just in time for the 25th anniversary of Prince of Persia on the Apple][, I present to you the first ever two-sided 16-sector version!

The funny thing is that I never played it on the real Apple][, only on the PC. Even after I acquired an Apple][`.nib` version in 2009, I didn't play it. Of course, the reason for that was, I was still using ApplePC as my Apple][emulator, and it had a fatal memory-corruption bug that crashed the game. Finally in 2014, I made the switch to AppleWin. AppleWin had its own bugs, but nothing that I couldn't work around.

The retail version of the Apple][version of Prince of Persia came on two sides of a single disk. The sectors were stored in 18-sector format, and they were *full.* As a result, the 16-sector cracked versions all made use of an additional side to store those extra sectors. In 2013, about a year after the source code was recovered, Roland Gustafsson was interviewed and expressed the opinion that the three-side version "was silly and really not impressive." Taking this as a challenge, I decided to make a two-sided 16-sector version.



I started with the "rebuilt from source" version. The first thing that you will notice is that it looks different in one particular place. The reason is that whoever built it used the 3.5" settings but placed it in the 5.25" format. It means that it never asks to turn over the disk when you reach Level 3. It prompts to "insert" the disk instead, as though it is a single disk.

4.1 If you build it, they will come

So I decided to build it myself in an emulated Apple][. As no one seems to have ported Git to this platform, I went through a rather round-about ritual of converting and compiling the code.

First, I started AppleWin and formatted a DOS 3.3 disk. Onto this disk, I saved some binary files the same size as the source files, then exited AppleWin. Now that the disk was ready, I used a hex editor to change the file types to text, to avoid the need to carry the load address and size.

I converted the source code by changing all line endings from LF to CR, setting the high bit on every character and inserting them in my own tool. (I really need to port that tool to ProDOS.)

Starting AppleWin again, I used Copy][Plus to move the files from a DOS 3.3 disk to a ProDOS disk. Using the Merlin assemble, I loaded and assembled the source files, saving object files to disk. Now that the object files were ready, I copied them back to the DOS 3.3 disk with Copy][Plus and exited AppleWin.

Finally, I extracted the files with another of my own tools that needs a ProDOS port, inserted images at the appropriate locations in the track files, and used a hex editor to place those track files onto the disk image.

4.2 Try Try Again, and Again and Again

The first thing that I noticed is that it won't boot, as building the 5.25" version enabled the copy-protection, which began in the boot phase. I worked around that one by bypassing the failure check.

The second thing that I noticed is that—thanks to another layer of copy protection—you couldn't play beyond Level 2. The second-level copy protection relied on two variables, named `redherring` and `redherring2`. The `redherring` variable was set indirectly during the boot-time copy protection check. However, the `redherring2` was never set in the source code version. Presumably someone removed the code (but did not notice that the declaration remained in the header file) because it wasn't used in the 3.5" version, because that version was not copy-protected. Unfortunately, without that value in the 5.25" version, you couldn't start the later levels. It was set in the retail 5.25" version, however, and thus we also found out that the source code was only for the 3.5" version. I bypassed this problem by writing the proper value to the proper place manually.

The third thing I noticed was that the graphics become corrupted on Level 4. The reason was yet another layer of copy-protection, which was executed before starting Level 1, but the effect was delayed until after starting Level 4. Nasty. `:-)` The end sequence was affected similarly. If the copy-protection failed, then the graphics became corrupted and the game froze on Level 14 (the reunion scene). This was an interesting design decision. If the protection was bypassed in the wrong way—by skipping the check on Level 4, instead of fixing the variable that was being compared—then that second surprise awaited. I worked around that one in the correct way, by bypassing the failure check.

The fourth thing I noticed is that the graphics became corrupted and then game crashed into text mode when starting Level 7. The reason was the final layer of copy-protection, which was executed after completing Level 1, but the effect was delayed until the start of Level 7. Very nasty. `;-)` I worked around that one by bypassing the failure check.

Finally, I checked the rest of the "rebuilt from source" version. The most important thing (depending on your point of view) was that all of the hidden parts were missing—the hidden routines (see below) and the hidden message (which was the decryption key for the original code). I also found that track `$11` was completely missing from side B, so the side B '^' routine (see below) caused a hang. Some of the graphics data were truncated, too, when compared to the retail version which I acquired in the meantime. Even though I didn't notice any difference when I played it, I gave up on that idea, and just ripped the tracks from the 5.25" retail version instead.

4.3 Turn Disk Over

Another interesting thing is how the game detects which side of the disk is in the drive. The protected version uses a unique value in the prologue data for the two sides (`$A9` and `$AD`), and uses an API to specify which one to expect. Since a standard 16-sector disk also has a standard prologue, which is identical on both sides, that was no longer an option for me. Instead, I chose to find a free sector in a location that was common to both sides, and placed the special byte there. When the prologue API was used, I redirected my read routine so that the next read request would first seek to the free sector and read the byte. If they matched, then the proper side was inserted already. Otherwise, the routine would read the sector periodically until that became true.

4.4 Size Does Matter

At a high level, the solution to the size problem is one of compression—technically, further compression, since some of the data are compressed already. However, I required a compression algorithm that packed well, was fast to decompress, and most importantly, small. The size limitation was significant. The game requires 128kb of memory, and uses almost all of it. I was fortunate enough to find a small (4096 bytes) region at `$d000` in main memory, in which to place my loader and the read buffer. This was the location of the original loader for the game. I simply replaced it with my own. I needed a read buffer within that region, because I had to load the compressed data somewhere before decompressing it into its final destination. I wanted the read buffer to be as large as possible, in order to reduce the number of read requests that I had to make. Shown in Figure 4, I managed to fit the loader code and data into under 1280 bytes: 752 bytes of code, 202 bytes for the sector table, the rest was dynamic data. That left me with 2816 bytes for the read buffer.

That space was so small that the write routine (for saving the game after you reach side B) would not fit in memory at the same time. To work around that problem, I separated the write routine, and loaded and executed it dynamically when a save request was made. It was discarded after it has done its job.

Back to the choice of compression.

I have written Apple][implementations for two well-known algorithms: LZ4 and aPLib. I did not want to write another one, so I was forced to choose between them. LZ4 was both fast and small (my implementation was only 152 bytes long), but it did not pack well enough. It had to be aPLib. aPLib packs well (about 20kb smaller than LZ4), is fast enough when factoring in the reduced number of sectors to read, and small (my implementation is only 228 bytes long, so less than one sector).

Some of the sectors are read only individually, some of them are read only as part of an entire track, and some of them are read using both methods, depending on the context. Once I determined how each of the sectors was loaded, I grouped them according to the size of the read, and then compressed the resulting block. I gave myself only two days total for the project, but it ended up taking me about two weeks. Most of that time was spent on finding an appropriate data structure.

I finally chose a variable length region set to describe the placement of the sectors within a track. This yielded a huge advantage for the sectors which were read only in track mode, when the packed size of the single region was too large for the read buffer. In that case, the file could be split into two smaller virtual regions, compressed separately to fit. The split point was determined by splitting into all 17 pairs (1 and 17, 2 and 16, 3 and 15 ...), compressing the pairs, then identifying the smallest pair. The smallest pair was chosen by the minimum number of sectors and then the minimum number of bytes. The assumption was that it costs more to decompress fewer bytes in more sectors, than to decompress more bytes in fewer sectors, even if the decompression was faster in the first case, because of the time to read and decode the additional sector. However, the flexibility of the region technique allowed the alternative case to be used without any changes to the code.

The support for the sector reads was flexible, too. Since the regions were defined only by their start and length, I could erase the individual addresses from the 18-sector requests. This allowed me to move sectors within a track, and to make the corresponding change in the 18-sector request packet. This was actually needed for track 4. For track 4, the region that began at sector `$0a` did not fit into 6 sectors even after compression. Fortunately, the region that began at sector 0 needed only 7 sectors, so the region at sector `$0a` could move to sector 9. This was enough to get it to fit. For track `$13`, the first two sectors were never accessed, so I could have moved sector 2 to sector 0, but there was no benefit to it.

Overall, my technique saved over 11 tracks on the first side, and over 16 tracks on the second side. Not enough for a single-side version, though.[7] `;-)`

4.5 And Now for Dessert: Easter Eggs!

While digging through the game code, I found several hidden routines. When playing side B, press '^' after completing a level to see an animation of Jordan waving, press a key at the end to view it again. In the byte bastards version, type RAMROD at the crack page for a hidden message.

Before booting, hold both Apple keys, then press one of the following to activate hidden modes.

DEL	Only on //GS, displays an oscilloscope.
!	Displays a message, and then a lo-res animation.
ENTER	Continually draws a fractal, press 'c' to change colors.
@	Displays a bouncing, spinning cube.
^	Pulses the drive head. Move joystick to change tone, sounds like a motorcycle.

Neighbors, is this not a tale of Shakespearean proportions and passions? A young prince, a mystery of code broken by underhanded blows in the dark, the poisoned daggers of copy-protection that even perpetrators forgot about—all laid bare by a contrived play of PoC! Is the Play the Thing, or is PoC the Thing, or are they the Thing together? You decide! –PML

[7]As a point of interest, I experimented with concatenating the entire data together, and including the sector offset in the table. That decreased the space quite significantly, but at a cost of increasing the size of the code, and making updating the data extremely difficult. That version saved over 13 tracks on the first side, and over 18 tracks on the second side. However, this was still not enough for a single-side version. In the end, it was not worth the effort, and it will not be released.

#	Side A	Side B
00		trk
01	trk	trk
02	sectors (00-0d)	trk
03	trk	trk
04	sectors (00-09, 0a-11)	sectors (00-05, 06-11)
05	trk	sectors (00-0b)
06	trk	trk
07	trk	trk
08	trk	trk
09	trk	trk
0a	trk	trk
0b	trk	sectors (00-05 / 06-11)
0c	sectors (00-05, 06-11)	sectors (00-0b / 0c-11)
0d	sectors (00-0b / 0c-11)	trk
0e	trk	trk
0f	trk	trk
10	trk	trk
11	trk	trk
12	trk	trk
13	sectors (02-11)	trk
14	sectors (04-11 / 00-03)	trk
15	trk	trk
16	trk	trk, sector 01
17	trk	sector 01
18	trk	trk
19	trk	trk
1a	trk	trk
1b	trk	sectors (00-08)
1c	trk, sectors (0d-11)	sectors (00-08 / 09-11)
1d	trk	sectors (00-08 / 09-11)
1e	trk	sectors (00-08 / 09-11)
1f	trk	sectors (00-08 / 09-11)
20	sectors (00-08, 09-11)	sectors (00-08 / 09-11)
21	sectors (00-08 / 09-11)	sectors (00-08 / 09-11)
22	sectors (02-11), trk	trk

Figure 4: Tracks and Sectors