



Pinball Fantasies on the MSX turbo R

How the 1994 PC game by Digital Illusions was brought to an MSX turbo R with a GFX9000 graphics card and a MoonSound sound card: all four tables, the intro and the menu, in a 4 MB cartridge.

SOURCE

PC DOS, x86 real mode, 60
Hz Mode X, Sound Blaster,
ProTracker music

TARGET CPU

R800 in the MSX turbo R
(Z80-compatible, about 5.8×
a standard MSX Z80 on code
running from RAM)

VIDEO

GFX9000: Yamaha V9990,
64-colour bitmap, hardware
scroll, blitter

SOUND

MoonSound: Yamaha OPL4
with its own sample RAM

THE RULE

The PC game is the reference, always

The whole port rests on one decision: never guess what the original does. Every piece of the MSX version is compared, frame by frame or byte by byte, against the PC game itself.

To make that possible, the first tool built was a headless PC. It runs the real TABLE1.PRG in a CPU emulator library (Unicorn), with small stand-ins for DOS, the mouse, the sound driver and the VGA card. It can play the game with scripted key

presses and record any frame, any variable, any call. The game's released source code served as documentation; the binaries stayed the reference, because they are what players actually played.

STEP 1

The ball physics, byte-exact

The heart of a pinball game is the ball, and the original's ball routine is pure arithmetic: positions, velocities, collision masks, a sine table. It was ported to Z80 by hand and run in lockstep with the PC version on the same inputs.

Result: 656 out of 656 calls produced identical bytes. Small details mattered here. For example, the sine table holds 2,560 16-bit words, and an early version that mirrored 2,560 bytes read zero for every angle past the halfway point. Lockstep found that in one run.

STEP 2

The game logic, translated by a program

Everything else on a table (rules, ramps, missions, scoring, the dot-matrix marquee scripts, the sound calls) is tens of thousands of x86 instructions. Porting that by hand would take years and would never match exactly. Instead, the port uses a **static translator**: a build-time tool that reads the x86 machine code and writes equivalent Z80 assembly. It is not a PC emulator running on the MSX; the MSX runs native Z80 code.

- **Finding the code.** A disassembler walks every reachable instruction. Indirect jumps (jump tables, code pointers stored in variables) are collected from real PC runs plus known code patterns.
- **Coverage.** A fuzzer plays the PC game with random inputs and checks that every instruction it reaches was translated: 38,200 PC frames, zero untranslated instructions.
- **Memory model.** The PC game's data segment sits at the same addresses in Z80

memory, so addresses in the code need no rewriting.

- **Verification.** The translated game and the PC game run side by side; after 6,000 frames of play their state is identical. When they disagree, a tool reports the first x86 instruction where they diverged.

Some hot routines were then replaced by hand-written Z80 versions ("natives") where the translated code was too slow: the physics core, the ramps, the marquee's font renderers (regenerated as Z80 compiled sprites), the palette loads and the plunger. Each native was checked against the original in the same lockstep.

STEP 3

Getting the graphics out of the PC game

The graphics were never taken from screenshots. Each piece comes either from the data files themselves or from running the PC game's own drawing code in the headless PC and reading back what it drew.

- **Table artwork.** Each `TABLEn.PRG` carries its table as four embedded IFF PBM pictures (Deluxe Paint's format), 320×144 pixels each, stacked into one 320×576 playfield. A small decoder reads them straight out of the program file, keeping the original colour indices.
- **The palette as the game sets it.** The colours come from the VGA palette as it stands after the game's start-up code has run, not from the picture files, because the game rewrites parts of it.
- **Moving parts drawn by the PC itself.** For the flippers, the game's own flipper routine was run once per animation frame and the changed pixels were read back from VGA memory. The same was done for the plunger spring (33 positions), the ball (its own draw routine, called on a blank screen), and the score panel. The masks that hide the ball under ramps are the ones the game builds at start-up, copied as they are.
- **Intro and menu pictures.** 17 IFF pictures (both the planar ILBM and the chunky PBM kinds) sit inside `INTRO.PRG`. A scanner finds them by their headers and a decoder unpacks them (ByteRun1 compression) to PNG.

STEP 4

Converting the graphics

The PC uses 256 colours, 6 bits per component. The table mode on the V9990 has 64 palette entries at 5 bits per component, so every colour had to be refitted.

- **A 64-colour palette per table.** Colours are grouped with a weighted k-means fit in the Lab colour space, where distances follow what the eye sees, then rounded to 5 bits. The fit covers the table with every lamp off *and* with every lamp on, so both states look right. A second pass gives a slot to rare colours the fit served badly, such as a lamp's dimmed state. Slot 0 stays black, and a few slots are kept for the ball, flippers and panel.
- **Lamps changed technique.** On the PC, a lamp is a set of palette entries: lighting it means rewriting those colours, which costs nothing per pixel. With only 64 entries, the MSX cannot spare one per lamp. So the converter records each lamp as runs of pixels with an "on" colour and an "off" colour, and the MSX rewrites those pixels when a lamp changes.
- **Flippers as cells.** Each flipper frame becomes a 48×49 cell in hidden video memory. Pixels no frame changes are left transparent, so the blitter's transparent copy only touches the flipper itself; black flipper pixels use a second black so they are not treated as transparent.
- **Ball hiding in the pixels.** The table uses 6 bits of each byte for colour. The two spare bits store the ball's two hide masks, so the ball code reads "is the ball under a ramp here?" from the same byte as the colour.
- **Menu and intro.** These pictures go to a 640-pixel-wide 16-colour mode. Pictures with 16 colours or fewer keep their exact colours; the others get the same Lab fit. One picture, the 21st Century logo, used the Amiga's "extra half-brite" trick (32 extra colours at half brightness), which the converter rebuilds explicitly.

The converted screens were checked against the PC's own frames: a tool compares the V9990 picture with the PC screen for the same moment of play and reports every pixel that differs.

STEP 5

Rebuilding the screen for the V9990

The PC draws into VGA memory. The MSX version does not emulate VGA; it rebuilds the picture with what the GFX9000 does well.

- **The table** is a 64-colour bitmap converted from the PC artwork, scrolled by the V9990's hardware scroll registers.
- **The marquee and score panel** stay fixed at the top through a raster split. The V9990 reads the high scroll register only once per frame, so the table is stored twice in video memory and the split switches between copies. The split is written during the horizontal blank, so no line shows a mix of both.
- **The ball** is drawn by the blitter. Spare pixel bits mark the areas where it passes under ramps.
- **The flippers and plunger** are pre-drawn frames kept in hidden video memory and copied by the blitter.
- **The lamps** are written as short pixel runs, only where they change.
- **The dot-matrix marquee** (160 × 16 dots) is updated dot by dot against a copy of what is on screen; when the game scrolls text, the blitter shifts the whole panel instead.

Blitter commands go into a queue that an interrupt feeds to the chip each time the previous command ends, so the CPU never waits for the blitter unless it must.

STEP 6

The music and sound effects

Each table's music and sound effects live in one file, TABLEn.MOD: a standard 4-channel ProTracker module. The sound effects are not separate files; they are samples in the same module, played as notes on the fourth channel.

The tricky part was how the game uses the module. The PC sound drivers (.SDR files) are compressed with Microsoft's FXPACK, so an unpacker was written first

mes) are compressed with Microsoft's LHA/FAH, so an unpacker was written first, then the Sound Blaster driver was disassembled to document its interface: start, stop, volume, "jump to this position" (used to switch tunes between game modes), and "play this sample as a note on channel N" (the sound effects). The game also hooks one effect command (Bxx) to count its jingles.

The conversion then works like this:

- **Samples** are the module's 8-bit signed samples, copied as they are into the MoonSound's own sample memory, each with a 12-byte OPL4 wave header giving its start, loop and end. The OPL4 plays 8-bit signed audio natively, so nothing is resampled.
- **Pitch** is precomputed: a table turns every ProTracker period into the OPL4's octave and frequency registers.
- **Volume** goes through a table from ProTracker's 0–64 scale to the OPL4's attenuation steps.
- **Patterns** are pre-decoded into fixed 16-byte rows and stay in the cartridge ROM; the player reads one row per tick. Only the effects the driver actually implements are kept.

A new player written in Z80 drives the OPL4 50 times a second, the same tick rate as the PC driver.

The player was first written as a reference in Python, then the Z80 version was checked against it: a recorded trace of the PC game's sound calls and all 64 pattern orders gave identical output.

STEP 7

Four tables, an intro and a menu in one cartridge

Once Party Land worked, the tooling was made table-independent. About 200 hard-coded addresses were moved into a per-table configuration, then Speed Devils, Billion Dollar Gameshow and Stones 'n' Bones went through the same pipeline: translate, lockstep, convert the art. The intro and the menu were rebuilt natively in

a 640-pixel-wide 16-colour mode, keeping the PC layout.

Everything fits in a 4 MB ASCII16-X cartridge. Each table's program data is compressed with ZX0 and unpacked into RAM when the table is chosen.

PART	HOW IT WAS MADE	HOW IT WAS CHECKED
Ball physics	Hand port to Z80	Byte-exact lockstep vs PC
Table rules	x86 → Z80 static translator	Lockstep, fuzzed coverage
Hot routines	Hand-written natives	Lockstep vs the original routine
Graphics	Converted PC art + V9990 blitter	Screen compared to PC frames, VRAM dumps
Music	Z80 MOD player for OPL4	Python reference, all orders
Intro, menu	Rebuilt natively	Headless MSX harness checks

STEP 8

Making it run at full speed

A correct port is only half the job; pinball has to be smooth. Speed was measured in the openMSX emulator, headless, with breakpoints timing every step of every frame.

- **Code moved into RAM.** On the turbo R, code running from the cartridge runs at roughly standard-Z80 speed. Copying the per-frame code into internal RAM cut one display pass from 145 scan lines to 21.
- **Priorities set by the player's eye.** Scrolling, the ball and the music must never slow down; lamps and the marquee may lag a frame. Each frame now starts with the ball, right after the vertical blank, and the lamp and marquee updates stop when the turbo R's hardware timer says the frame is running late. They resume on the next frame.
- **Blits in the right order.** Flipper redraws were moved behind the ball in the blitter queue, so the ball is never drawn while the beam is passing over it.

MEASUREMENT (OPENMSX)	BEFORE	AFTER
Display pass, per-frame code	145 lines	21 lines
Worst game-logic frame	70 ms	26 ms
Party Land: frames over budget in 27 s of play	35	2
Ball drawn under the beam (all 4 tables)	yes, at launch	0 times
Split writes landing in the horizontal blank	—	1074 / 1078

LESSONS

What the project taught

- **Verify against the original, not against expectations.** Most bugs were found by a lockstep comparison, not by watching the screen.
- **Translate rather than rewrite** when the logic is large and exactness matters.
- **Measure on the real timing model.** The project's own fast headless model said a pass took 2 frames; openMSX said 1.01. Only emulator measurements were trusted for speed.
- **Emulators differ.** WebMSX only raises the V9990's command-end interrupt when the status port is read, so the wait loops read it. That one change fixed a freeze there.

Current build: b0129, a demo for testing only, not for commercial use. Requires an MSX turbo R with a GFX9000; a MoonSound is detected at start-up for music. The port is built from the user's own copy of the PC game.

