18:02 An 8 Kilobyte Mode 7 Demo for the Apple II

by Vincent M. Weaver

While making an inside-joke filled game for my favorite machine, the Apple][, I needed to create a Final-Fantasy-esque flying-over-the-planet sequence. I was originally going to fake this, but why fake graphics when you can laboriously spend weeks implementing the effect for real. It turns out the Apple][is just barely capable of generating the effect in real time.

Once I got the code working I realized it would be great as part of a graphical demo, so off on that tangent I went. This turned out well, despite the fact that all I knew about the demoscene I had learned from a few viewings of the Future Crew *Second Reality* demo combined with dimly remembered Commodore 64 and Amiga usenet flamewars.

While I hope you enjoy the description of the demo and the work that went into it, I suspect this whole enterprise is primarily of note due to the dearth of demos for the Apple][platform. For those of you who would like to see a truly impressive Apple][demo, I would like to make a shout out to FrenchTouch whose works put this one to shame.

The Hardware

CPU, RAM and Storage:

The Apple][was introduced in 1977 with a 6502 processor running at roughly 1.023MHz. Early models only shipped with 4k of RAM, but in later years, 48k, 64k and 128k systems became common. While the demo itself fits in 8k, it decompresses to a larger size and uses a full 48k of RAM; this would have been very expensive in the seventies.

In 1977 you would probably be loading this from cassette tape, as it would be another year before Woz's single-sided $5\frac{1}{4}$" Disk II came around. With the release of Apple DOS3.3 in 1980, it offered 140k of storage on each side.

Sound:

The only sound available in a stock Apple][is a bit-banged speaker. There is no timer interrupt; if you want music, you have to cycle-count via the CPU to get the waveforms you needed.

The demo uses a Mockingboard soundcard, first introduced in 1981. This board contains dual AY-3-8910 sound generation chips connected via 6522 I/O chips. Each sound chip provides three channels of square waves as well as noise and envelope effects.

Graphics:

It is hard to imagine now, but the Apple][had nice graphics for its time. Compared to later competitors, however, it had some limitations: No hardware sprites, user-defined character sets, blanking interrupts, palette selection, hardware scrolling, or even a linear framebuffer! It did have hardware page flipping, at least.

The hi-res graphics mode is a complex mess of NTSC hacks by Woz. You get approximately 280x192 resolution, with 6 colors available. The colors are NTSC artifacts with limitations on which colors can be next to each other, in blocks of 3.5 pixels. There is plenty of fringing on edges, and colors change depending on whether they are drawn at odd or even locations. To add to the madness, the framebuffer is interleaved in a complex way, and pixels are drawn least-significant-bit first. (All of this to make DRAM refresh better and to shave a few 7400 series logic chips from the design.) You do get two pages of graphics, Page 1 is at `$2000` and Page 2 at `$4000`.[1] Optionally four lines of text can be shown at the bottom of the screen instead of graphics.

The lo-res mode is a bit easier to use. It provides 40×48 blocks, reusing the same memory as the 40×24 text mode. (As with hi-res you can switch to a 40×40 mode with four lines of text displayed at the bottom.) Fifteen unique colors are available, plus a second shade of grey. Again the addresses are interleaved in a non-linear fashion. Lo-res Page 1 is at `$400` and Page 2 is at `$800`.

Some amazing effects can be achieved by cycle counting, reading the floating bus, and racing the beam while toggling graphics modes on the fly.

[1] On 6502 systems hexadecimal values are traditionally indicated by a dollar sign.

Figure 1. Colorful View of Executable Code

```
 -------------  $ffff
|    ROM/IO    |
 -------------  $c000
|              |
| Uncompressed|
| Code/Data    |
|              |
 -------------  $4000
| Compressed   |
|   Code       |
 -------------  $2000
|   free       |
 -------------  $1c00
|   Scroll     |
|    Data      |
 -------------  $1800
|  Multiply    |
|   Tables     |
 -------------  $1000
| LORES pg 3   |
 -------------  $0c00
| LORES pg 2   |
 -------------  $0800
| LORES pg 1   |
 -------------  $0400
|free/vectors |
 -------------  $0200
|    stack     |
 -------------  $0100
|   zero pg    |
 -------------  $0000
```

Figure 2. Memory Map

Development Toolchain

I do all of my coding under Linux, using the ca65 assembler from the cc65 project. I cross-compile the code, constructing AppleDOS 3.3 disk images using custom tools I have written. I test first in emulation, where AppleWin under Wine is the easiest to use, but until recently MESS/MAME had cleaner sound.

Once the code appears to work, I put it on a USB stick and transfer to actual hardware using a CFFA3000 disk emulator installed in an Apple IIe platinum edition.

Bootloader

An Applesoft BASIC "HELLO" program loads the binary automatically at bootup. This does not count towards the executable size, as you could manually BRUN the 8k machine-language program if you wanted.

To make the loading time slightly more interesting the HELLO program enables graphics mode and loads the program to address `$2000` (hi-res page1). This causes the display to filled with the colorful pattern corresponding to the compressed image. (Figure 1.) This conveniently fills all 8k of the display RAM, or would have if we had `poke`d the right soft-switch to turn off the bottom four lines of text. After loading, execution starts at address `$2000`.

Decompression

The binary is encoded with the LZ4 algorithm. We flip to hi-res Page 2 and decompress to this region so the display now shows the executable code.

The 6502 size-optimized LZ4 decompression code was written by qkumba (Peter Ferrie).[2] The program and data decompress to around 22k starting at `$4000`. This overwrites parts of DOS3.3, but since we are done with the disk this is no problem.

If you look carefully at the upper left corner of the screen during decompression you will see my triangular logo, which is supposed to evoke my VMW initials. To do this I had to put the proper bit pattern inside the code at the interleaved addresses of `$4000`, `$4400`, `$4800`, and `$4C00`. The image data at `$4000` maps to (mostly) harmless code so it is left in place and executed.

[2] `http://pferrie.host22.com/misc/appleii.htm`

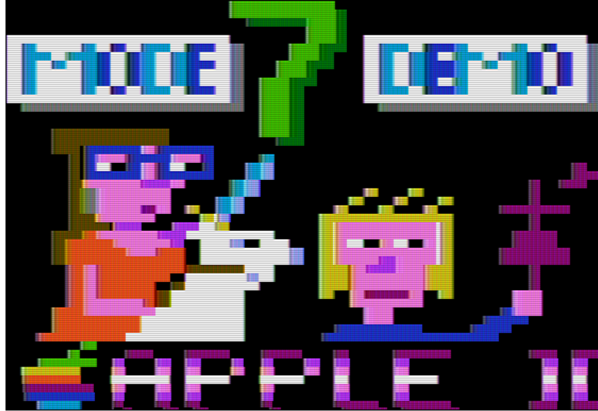


Figure 3. The title screen.

Optimizing the code inside of a compressed image (to fit in 8k) is much more complicated than regular size optimization. Removing instructions sometimes makes the binary *larger* as it no longer compresses as well. Long runs of a single value, such as zero padding, are essentially free. This became an exercise of repeatedly guessing and checking, until everything fit.

Title Screen

Once decompression is done, execution continues at address `$4000`. We switch to low-res mode for the rest of the demo.

FADE EFFECT: The title screen fades in from black, which is a software hack as the Apple][does not have palette support. This is done by loading the image to an off-screen buffer and then a lookup table is used to copy in the faded versions to the image buffer on the fly.

TITLE GRAPHICS: The title screen is shown in Figure 3. The image is run-length encoded (RLE) which is probably unnecessary in light of it being further LZ4 encoded. (LZ4 compression was a late addition to this endeavor.)

Why not save some space and just loading our demo at `$400`, negating the need to copy the image in place? Remember the graphics are 40×48 (shared with the text display region). It might be easier to think of it as 40×24 characters, with the top / bottom nybbles of each ASCII character being interpreted as colors for a half-height block. If you do the math you will find this takes 960 bytes of space, but the memory map reserves 1k for this mode. There are "holes" in the address range that are not displayed, and various pieces of hardware can use these as scratchpad memory. This means just overwriting the whole 1k with data might not work out well unless you know what you are doing. Our RLE decompression code skips the holes just to be safe.

SCROLL TEXT: The title screen has scrolling text at the bottom. This is nothing fancy, the text is in a buffer off screen and a 40×4 chunk of RAM is copied in every so many cycles.

You might notice that there is tearing/jitter in the scrolling even though we are double-buffering the graphics. Sadly there is no reliable cross-platform way to get the VBLANK info on Apple][machines, especially the older models.

Mockingbird Music

No demo is complete without some exciting background music. I like chiptune music, especially the kind written for AY-3-8910 based systems. During the long wait for my Mockingboard hardware to arrive, I designed and built a Raspberry Pi chiptune player that uses essentially the same hardware. This allowed me to build up some expertise with the software/hardware interface in advance.

The song being played is a stripped down and re-arranged version of "Electric Wave" from CC'00 by EA (Ilya Abrosimov).

Most of my sound infrastructure involves YM5 files, a format commonly used by ZX Spectrum and Atari ST users. The YM file format is just AY-3-8910 register dumps taken at 50Hz. To play these back one sets up the sound card to interrupt 50 times a second and then writes out the fourteen register values from each frame in an interrupt handler.

Writing out the registers quickly enough is a challenge on the Apple][, as for each register you have to do a handshake and then set both the register number and the value. It is hard to do this in less than forty 1MHz cycles for each register. With complex chiptune files (especially those written on an ST with much faster hardware), sometimes it is not possible to get exact playback due to the delay. Further slowdown happens as you want to write both AY chips (the output is stereo, with one AY on the left and one on the right). To help with latency on playback, we keep track of the last frame written and only write to the registers that have changed.

The demo detects the Mockingboard in Slot 4

at startup. First the board is initialized, then one of the 6522 timers is set to interrupt at 25Hz. Why 25Hz and not 50Hz? At 50Hz with fourteen registers you use 700 bytes/s. So a two minute song would take 84k of RAM, which is much more than is available! To allow the song to fit in memory, without a fancy circular buffer decompression routine, we have to reduce the size.[3]

First the music is changed so it only needs to be updated at 25Hz, and then the register data is compressed from fourteen bytes to eleven bytes by stripping off the envelope effects and packing together fields that have unused bits. In the end the sound quality suffered a bit, but we were able to fit an acceptably catchy chiptune inside of our 8k payload.

Drawing the Mode7 Background

Mode 7 is a Super Nintendo (SNES) graphics mode that takes a tiled background and transforms it by rotating and scaling. The most common effect squashes the background out to the horizon, giving a three-dimensional look. The SNES did these transforms in hardware, but our demo must do them in software.

Our algorithm is based on code by Martijn van Iersel which iterates through each horizontal line on the screen and calculates the color to output based on the camera height (`spacez`) and `angle` as well as the current coordinates, x and y.

First, the distance d is calculated based on fixed scale and distance-to-horizon factors. Instead of a costly division operation, we use a pre-generated lookup table for this.

$$d = \frac{z \times \texttt{yscale}}{y + \texttt{horizon}}$$

Next we calculate the horizontal scale (distance between points on this line):

$$h = \frac{d}{\texttt{xscale}}$$

Then we calculate delta x and delta y values between each block on the line. We use a pre-computed sine/-cosine lookup table.

$$\Delta x = -\sin(\texttt{angle}) \times h$$

$$\Delta y = \cos(\texttt{angle}) \times h$$

[3]For an example of such a routine, see my Chiptune music-disk demo.

The leftmost position in the tile lookup is calculated:

$$\texttt{tilex} = x + \left(d\cos(\texttt{angle}) - \frac{\texttt{width}}{2}\right)\Delta x$$

$$\texttt{tiley} = y + \left(d\sin(\texttt{angle}) - \frac{\texttt{width}}{2}\right)\Delta y$$

Then an inner loop happens that adds Δx and Δy as we lookup the color from the tilemap (just a wrap-around array lookup) for each block on the line.

$$\texttt{color} = \texttt{tilelookup}(\texttt{tilex}, \texttt{tiley})$$

$$\texttt{plot}(x, y)$$

$$\texttt{tilex} \mathrel{+}= \Delta x, \texttt{tiley} \mathrel{+}= \Delta y$$

Optimizations: The 6502 processor cannot do floating point, so all of our routines use 8.8 fixed point math. We eliminate all use of division, and convert as much as possible to table lookups, which involves limiting the heights and angles a bit.

Some cycles are also saved by using self-modifying code, most notably hard-coding the height (z) value and modifying the code whenever this is changed. The code started out only capable of roughly 4.9fps in 40×20 resolution and in the end we improved this to 5.7fps in 40×40 resolution. Care was taken to optimize the innermost loop, as every cycle saved there results in 1280 cycles saved overall.

Fast Multiply: One of the biggest bottlenecks in the mode7 code was the multiply. Even our optimized algorithm calls for at least seven 16-bit by 16-bit to 32-bit multiplies, something that is *really* slow on the 6502. A typical implementation takes around 700 cycles for an 8.8×8.8 fixed point multiply.

We improved this by using the ancient quarter-square multiply algorithm, first described for 6502 use by Stephen Judd.

This works by noting these factorizations:

$$(a+b)^2 = a^2 + 2ab + b^2$$

$$(a-b)^2 = a^2 - 2ab + b^2$$

If you subtract these you can simplify to

$$a \times b = \frac{(a+b)^2}{4} - \frac{(a-b)^2}{4}$$

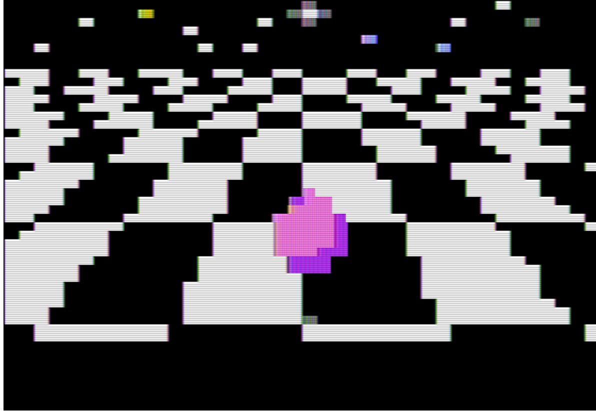


Figure 4. Bouncing ball on infinite checkerboard.

Figure 5. Spaceship flying over an island.

For 8-bit values if you create a table of squares from 0 to 511, then you can convert a multiply into two table lookups and a subtraction.[4] This does have the downside of requiring two kilobytes of lookup tables, but it reduces the multiply cost to the order of 250 cycles or so and these tables can be generated at startup.

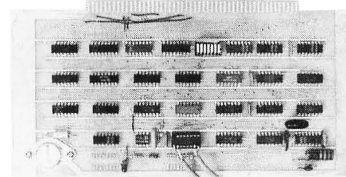
BALL ON CHECKERBOARD

The first Mode7 scene transpires on an infinite checkerboard. A demo would be incomplete without some sort of bouncing geometric solid, in this case we have a pink sphere. The sphere is represented by sixteen sprites that were captured from a twenty year old OpenGL example. Screenshots were reduced to the proper size and color limitations. The shadows are also sprites, and as the Apple][has no dedicated sprite hardware, these are drawn completely in software.

The clicking noise on bounce is generated by accessing the speaker port at address `$C030`. This gives some sound for those viewing the demo without the benefit of a Mockingboard.

TFV SPACESHIP FLYING

This next scene has a spaceship flying over an island. The Mode7 graphics code is generic enough that only one copy of the code is needed to generate both the checkerboard and island scenes. The spaceship, water splash, and shadows are all sprites. The path the ship takes is pre-recorded; this is adapted from the Talbot Fantasy 7 game engine with the keyboard code replaced by a hard-coded script of actions to take.



The Tarbell Cassette Interface

- Plugs directly into your IMSAI or ALTAIR
- Fastest transfer rate: 187 (standard) to 540 bytes/second
- Extremely Reliable—Phase encoded (self-clocking)
- 4 Extra Status Lines, 4 Extra Control Lines
- 25-page manual included
- Device Code Selectable by DIP-switch
- Capable of Generating BYTE/LANCASTER tapes also.
- No modification required on audio cassette recorder
- Complete kit $120, Assembled $175, Manual $4

TARBELL ELECTRONICS

144 Miraleste Drive #106, Miraleste, Calif. 90732
(213) 832-0182
California residents please add 6% sales tax

[4] All 8-bit $a + b$ and $a - b$ fall in this range.

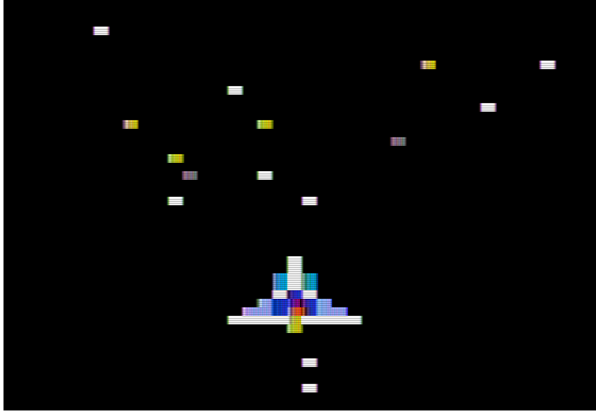


Figure 6. Spaceship with starfield.

Figure 7. Rasterbars, stars, and credits.

STARFIELD

The spaceship now takes to the stars. This is typical starfield code, where on each iteration the x and y values are changed by

$$\Delta x = \frac{x}{z}, \Delta y = \frac{y}{z}$$

In order to get a good frame rate and not clutter the lo-res screen only sixteen stars are modeled. To avoid having to divide, the reciprocal of all possible z values are stored in a table, and the fast-multiply routine described previously is used.

The star positions require random number generation, but there is no easy way to quickly get random data on the Apple][. Originally we had a 256-byte blob of pre-generated "random" values included in the code. This wasted space, so instead we use our own machine code at address at $5000 as if it were a block of random numbers!

A simple state machine controls star speed, ship movement, hyperspace, background color (for the blue flash) and the eventual sequence of sprites as the ship vanishes into the distance.

RASTERBARS/CREDITS

Once the ship has departed, it is time to run the credits as the stars continue to fly by.

The text is written to the bottom four lines of the screen, seemingly surrounded by graphics blocks. Mixed graphics/text is generally not be possible on the Apple][, although with careful cycle counting and mode switching groups such as FrenchTouch have achieved this effect. What we see in this demo is the use of inverse-mode (inverted color) space characters which appear the same as white graphics blocks.

The rasterbar effect is not really rasterbars, just a colorful assortment of horizontal lines drawn at a location determined with a sine lookup table. Horizontal lines can take a surprising amount of time to draw, but these were optimized using inlining and a few other tricks.

The spinning text is done by just rapidly rotating the output string through the ASCII table, with the clicking effect again generated by hitting the speaker at address `$C030`. The list of people to thank ended up being the primary limitation to fitting in 8kB, as unique text strings do not compress well. I apologize to everyone whose moniker got compressed beyond recognition, and I am still not totally happy with the centering of the text.

A Parting Gift

Further details, a prebuilt disk image, and full source code are available both online and attached to the electronic version of this document.[5] [6]

[5] `unzip pocorgtfo18.pdf mode7.tar.gz`
[6] `http://www.deater.net/weave/vmwprod/mode7_demo/`