A version of this document appeared in PoC || GTFO 0x18

Making an 8k Low-resolution Graphics Demo for the Apple II

by DEATER, AKA Vincent M. Weaver

1 Why would anyone do this?

While making an inside-joke filled game for my retro system of choice, the Apple II, 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 II 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 II platform. If you are truly interested in seeing impressive Apple II demos, I would like to make a shout out to FrenchTouch whose works put this one to shame.

2 The Hardware

The Apple II was introduced in 1977. In theory this demo will run on hardware that old, although I do not have access to a system of that vintage. I like to troll Commodore fans by noting this predates the Commodore 64 by five years.

CPU, RAM and Storage:

The Apple II has a 6502 processor running at roughly 1.023MHz. Early models only shipped with 4k of RAM, but later 48k, 64k, and 128k systems were 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 1977. See Figure 7 for a diagram of the memory map.

Also in 1977 you would probably be loading this from cassette tape. It would be another year before Woz's single-sided $5\frac{1}{4}$” Disk II came about (eventually offering 140k of storage per side with the release of Apple DOS3.3 in 1980).

Sound:

The only sound available in a stock Apple II is a bit-banged speaker. There was no timer interrupt; if you wanted music you had to cycle-count via the CPU to get the waveforms you needed.

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

Graphics:

It is hard to imagine now, but the Apple II had nice graphics for its time. Compared to later competitors, however, it had some limitations.

Hardware Sprites	No
User-defined charset	No
Blanking interrupts	No
Palette selection	No
Linear framebuffer	No
Hardware scrolling	No
Hardware page flip	Yes

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 get DRAM refresh for free and to shave a few 7400 series logic chips from the design). You do get two pages of graphics, Page 1 is at $2000[1] and Page 2 at $4000. Optionally 4 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 40x48 blocks, reusing the same memory as the 40x24 text mode. (As with hi-res you can switch to a 40x40 mode with four lines of text displayed at the bottom). Fifteen colors are available (there are two greys which are indistinguishable). 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. Unfortunately for you this demo does not do any of those things so you will not be reading about that today.

3 Development Setup

I do all of my coding under Linux, using the nano text editor. I use the ca65 assembler from the cc65 project, which I find to be a reasonable tool although many "real" Apple II programmers look down on it for some reason. I cross-compile the code, constructing Apple DOS3.3 disk images using custom tools I have written. I test using emulators: AppleWin (run under the wine emulator) 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 the actual Apple II (an Apple IIe platinum edition).

Figure 1: VMW logo hidden in the executable data.

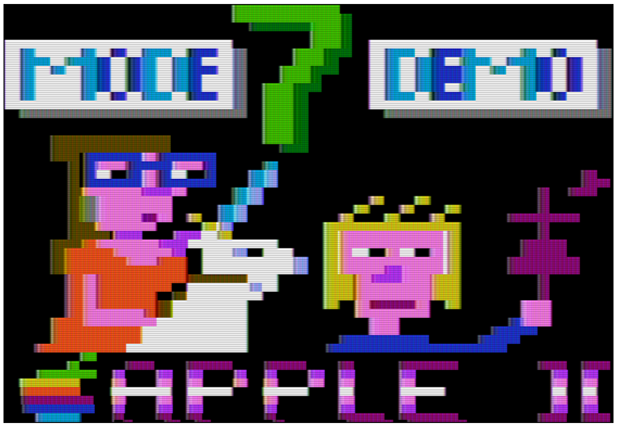


Figure 2: The title screen.

4 The Demo

4.1 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. This conveniently fills all 8k of the display RAM, or would have if we had POKEd the right soft-switch to turn off the bottom 4 lines of text.

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

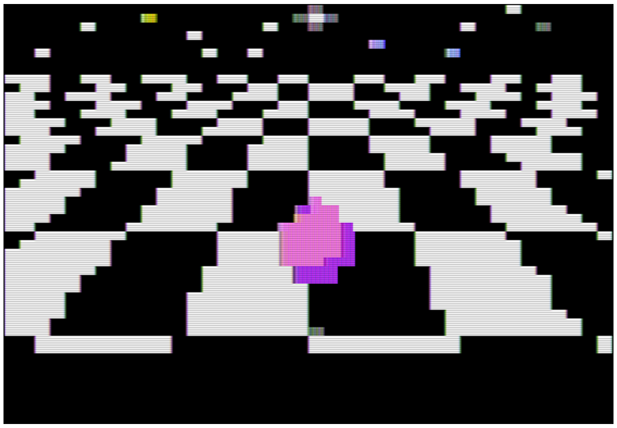


Figure 3: Bouncing ball on infinite checkerboard.

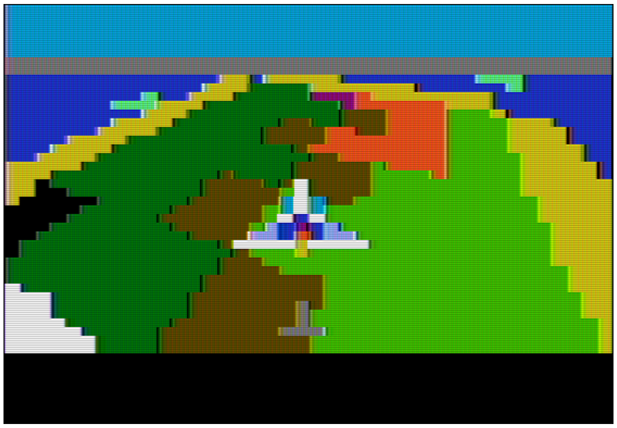


Figure 4: Spaceship flying over an island.

Upon loading, execution starts at address `$2000`.

4.2 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). The program and data decompress to around 22k starting at `$4000`. This over-writes parts of DOS3.3, but since we are done with the disk this is not an issue.

If you look carefully at the upper left corner of the screen during decompress you will see my triangular logo, which is supposed to evoke my VMW initials (see Figure 1). 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. Making this work turned out to be more trouble than it was worth, especially as the logo is not visible in the youtube capture of the demo (the video compression does not handle screens full of seemingly random noise well).

The demo was optimized to fit in 8k. Optimizing code inside of a compressed image 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 values (such as 0 padding) are essentially free. This mostly turned into an exercise of guess-and-check until everything fit.

4.3 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. This is a software hack as the Apple II does not have palette support. The image is loaded to an off-screen buffer and a lookup table is used to copy in the faded versions on the fly.

TITLE GRAPHICS: The title screen is shown in Figure 2. The image is run-length encoded (RLE) which is probably unnecessary in light of it being further LZ4 encoded. (The LZ4 compression was a late addition to this endeavor).

Why not save some space and just load our demo at `$400` and negate the need to copy the image in place? Remember the graphics are 40x48 (shared with the text display region). It might be easier to think of it as 40x24 characters, with the top / bottom 4-bits 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

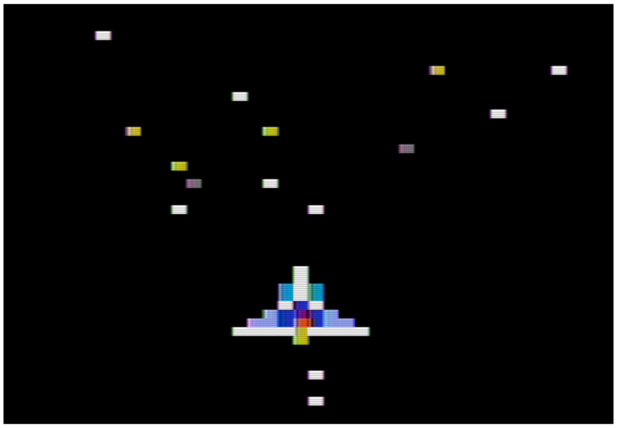
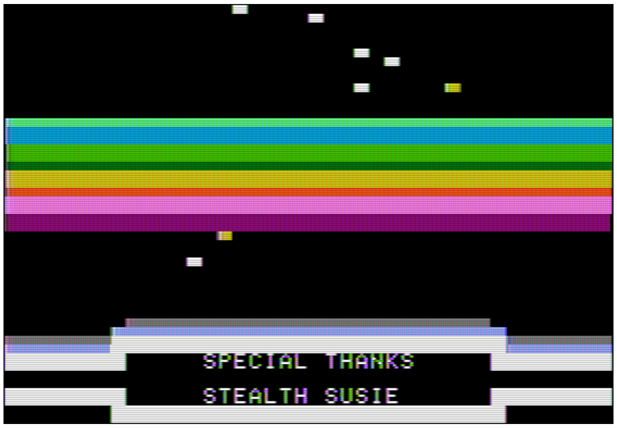


Figure 5: Spaceship with starfield.



SPECIAL THANKS
STEALTH SUSIE

Figure 6: Rasterbars, stars, and credits. Stealth Susie was a particularly well-traveled guinea pig.

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. To this end 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 40x4 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 not a reliable cross-platform way to get the VBLANK info on Apple II machines, especially the older models. This is even more noticeable in the recorded video, as the capture card and video encoding conspire to make this look worse than things look in person.

4.4 MOCKINGBOARD 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 time waiting 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 rearranged 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 14 register values from each frame in an interrupt handler.

Writing out the registers quickly enough is a challenge on the Apple II. For each register you have to do a handshake 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) it is sometimes 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.

Our code detects the Mockingboard at startup; we are lazy and only support finding the card in Slot 4 (which is a fairly typically location). The board is initialized, and then one of the 6522 timers is set to interrupt at 25Hz. Why 25Hz and not 50Hz? At 50Hz with 14 registers you use 700 bytes/s. So a 2 minute song would take 84k of RAM, which is much more than is available. Also the Disk II requires hard real-time response involving the full CPU to read from disk, so it is not possible to read more data while the demo is running. To allow the song to fit in memory (without the fancy circular buffer decompression routine utilized in my VMW Chiptune music-disk demo) we have to reduce the size. First the music is changed so it only needs to be updated at 25Hz. Then the register data is compressed from 14 bytes to 11 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.

4.5 MODE7 BACKGROUND

"Mode7" 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. It 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 x and y coordinates (*cx* and *cy*).

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

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

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

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

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

$$dx = -sin(angle) \times h$$

$$dy = cos(angle) \times h$$

The leftmost position in the tile lookup is calculated:

$$tilex = cx + (d * cos(angle) - (width/2) * dx$$

$$tiley = cy + (d * sin(angle) - (width/2) * dy$$

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

$$color = tilelookup(tilex, tiley)$$

$$plot(x, y)$$

$$tilex+ = dx, tiley+ = dy$$

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). We also save some cycles 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 40x20 resolution and in the end we improved this to 5.7fps in 40x40 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 16bit x 16bit = 32bit multiplies, something that is *really* slow on the 6502. A typical implementation takes around 700 cycles for a 8.8 x 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

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

For 8-bit values if you create a table of squares from 0 to 511 (all 8-bit a+b and a-b fall in this range) then you can convert a multiply into two table lookups and a subtraction. This does have the downside of requiring 2kB of lookup tables (which can be generated at startup) but it reduces the multiply cost to the order of 250 cycles or so.

4.6 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 16 sprites that were captured from a 20 year old OpenGL game engine. Screenshots were taken then reduced to the proper size and color limitations. The shadows are also just sprites. Note that the Apple II has no dedicated sprite hardware, so 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.

4.7 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.

4.8 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

$$dx = \frac{x}{z}, dy = \frac{y}{z}$$

In order to get a good frame rate and not clutter the lo-res screen only 16 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 II. Originally we had a 256-byte blob of pre-generated "random" values included in the code. This wasted space, so now instead we just use our code at address at $5000 as if it were a block of random numbers. This was arbitrarily chosen, and it is not as random as it could be as seen when the ship enters hyperspace and the lower-right quadrant is distressingly star-free.

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.

4.9 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 II, 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

```
 -------------  $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 7: Memory Map (not to scale)

recognition, and I am still not totally happy with the centering of the text.

5 Obtaining the Code

More details, disk image, and full source can be found at the website: `http://www.deater.net/weave/vmwprod/mode7_demo/`

A The Lores Memory Map

A.1 Why is it so weird?

The Apple II is very much a TV-typewriter video-terminal that happens to have a 6502 processor attached to give the display something to do. (This makes it similar to the SoC in a Raspberry Pi, which is a large GPU with a small helper ARM processor tacked onto the side.)

The Apple II video display is so central, that it even affects the CPU timings. The CPU clock usually runs at 978ns, but every 65th cycle it is extended to 1117ns to keep the video output in sync with the colorburst. This is why the 6502 runs at the somewhat unusual average speed of 1.020484MHz.

Text mode and low-resolution graphics share the same 1k region of memory from addresses `$400` to `$800` for Page1. A straightforward setup would have a linear memory map where location (0,0) would map to address `$400`, location (39,0) would map to `$427`, and location (0,1) would be at `$428`. That would make too much sense.

For low-res, the first complication is what is represented by each memory byte. In text mode this is the ASCII value you wish to display, or-ed with $80 so the high bit is set. Leaving the high bit clear does weird things like enable inverse (black-on-white) or flashing characters. Setting address `$400` to `$C1` would put an 'A' (ASCII `$41`) in the upper left corner of the screen. In low-res graphics mode the two 4-bit nibbles are split and interpreted as two blocks, one above each other. In this case the the `$C1` would be a color 1 (red) block on top and a color 12 (light green) block on the bottom. The colors are NTSC artifact colors, formed by outputting the raw bit pattern out to the screen with the color burst enabled. You can try this out yourself from BASIC by running `TEXT:HOME:POKE 1024,193` to see the text result, and `GR:POKE 1024,193` to see the graphics result.

That is not too bad so far. The next complication is packing the 40-columns of characters into video memory. Sadly 40 is not a nice power of two, so any packing is going to be inefficient somehow with respect to addressing bits. The compromise is to pack three 40-byte columns into 128 bytes, wasting 8 bytes (the "screen holes").

This still might not be that weird, but then the address interleaving comes into play. Note that row 0 starts at `$480`, but row 1 starts at `$480` (a diff of 128), not `$428` (a diff of 40) as you might expect. Address `$428` actually corresponds to row 16.

For example, see the sample image in Table 1 and how the address values are interleaved. This same image is shown in Table 2 as it would appear if memory was read linearly. To make things even more confusing, the image is scattered even more completely across the physical RAM chips for reasons we will describe below.

The reason for this craziness, as with most oddities on the Apple II, turns out to be Steve Wozniak being especially clever. Early home computers often used static RAM (SRAM). SRAM is easy to use, you just hook up the CPU address and read/write lines to the memory chips and read and write bytes as needed.

The Apple II instead uses dynamic RAM (DRAM), where each bit is stored in a capacitor whose value will leak away to zero unless you refresh it periodically. Why would you use memory that did that? Well SRAM uses 6 transistors to store a bit, DRAM uses only 1. So in theory you can fit 6 times the RAM in the same space, leading to much cheaper costs and much better density.

To avoid losing DRAM contents, you must regularly refresh the capacitors. This involves reading each memory value out faster than it leaks away. DRAM reads are destructive, so a read operation always reads out, recharges, then writes back the original value. Because of this you can avoid explicitly refreshing DRAM with a dedicated circuit if you can guarantee you perform a read of each memory row in the required timeframe.

Many systems could not do this, so there was separate hardware to conduct the refresh. Often this hardware would take over the memory bus and halt the CPU while it was happening, slowing down the whole system. This is true of the original IBM PC; if you ever look at cycle-level optimization on the PC you will notice the coders have to take into account pauses caused by memory refresh (the refresh tended to be conservative so some coders chose to live dangerously and make refresh happen less often to increase performance).

Steve Wozniak realized that he could avoid stopping the CPU for refresh. The 6502 clock has two phases: during first phase processor is busy with internal work and the memory bus is idle. The CPU only accesses memory in the second phase. The Apple II uses the idle phase to step through the video memory range and updates the display. To refresh the 16k (model 4116) DRAM chips you need to read each 128-wide row at least once every 2ms. By carefully selecting the way that the CPU address lines map to the RAS/CAS lines into the DRAM you can have the video scanning circuitry walk through each row of the DRAMs fast enough to conduct the refresh for free. This works beautifully, but as a side effect you end up with the Apple II's weird interleaved memory maps.

Wozniak said in a later interview that in retrospect he could have gotten a linear video memory map at the expense of two more chips on the circuit board. Apparently when designing the Apple II he thought most people would use BASIC which hid the memory map, and did not realize the interleaving would be such a pain for assembly coders.

This is why low-level text and lowres graphics routines can be complex, using lookup tables and read/shift/mask operations just to do simple plot operations. Fully generic routines have to handle all the corner cases, which is why the Mode7 demo cheats and the sprite drawing code only works at even row offsets (as this makes the code smaller and simpler).

While this seems needlessly complicated, the hi-res graphics mode is even worse that the mess described above.

Table 1: Apple II lores display, 40x48. Note the interleaving of the row addresses. Rows 40-47 are ASCII text being interpreted as graphic blocks.

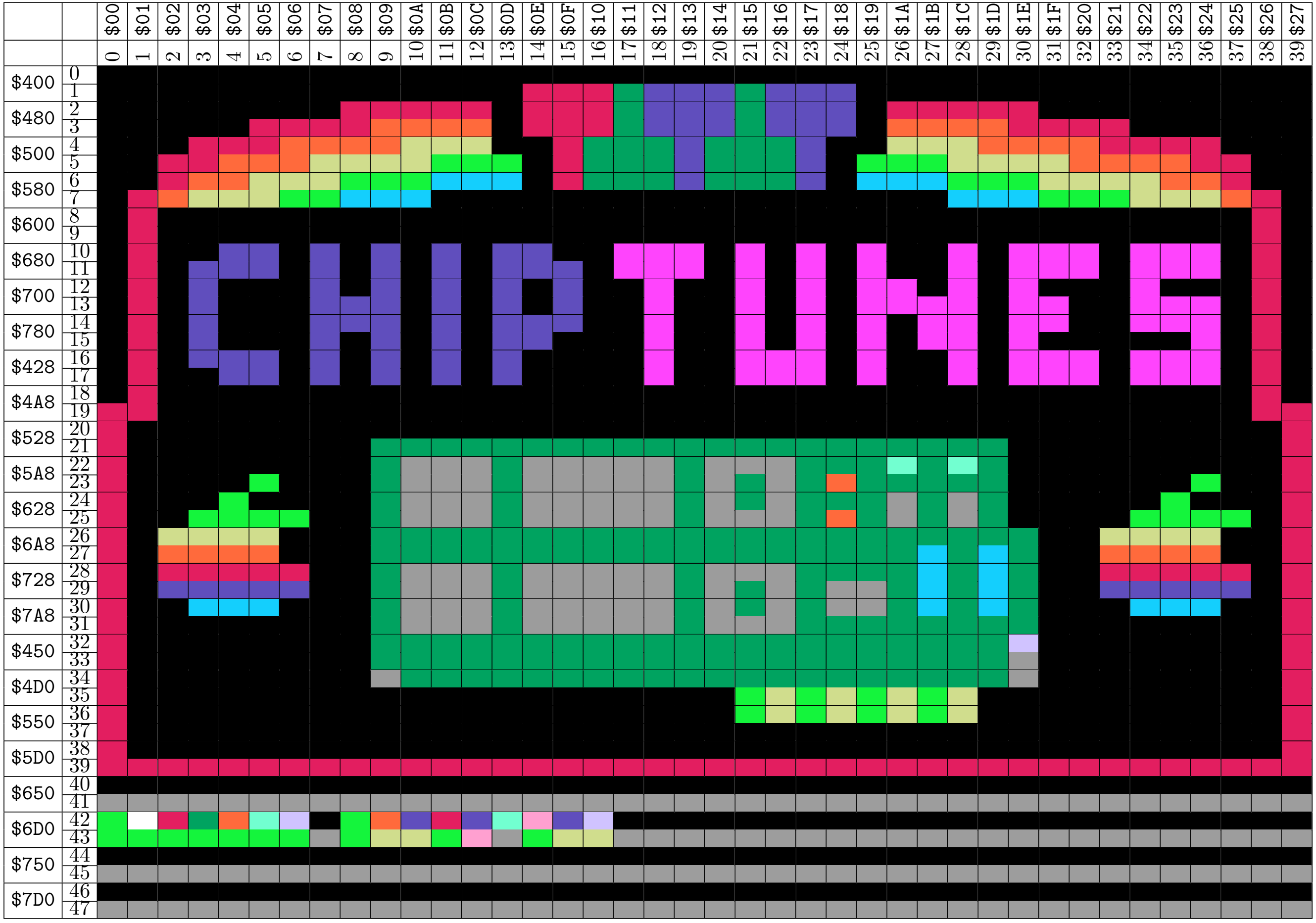
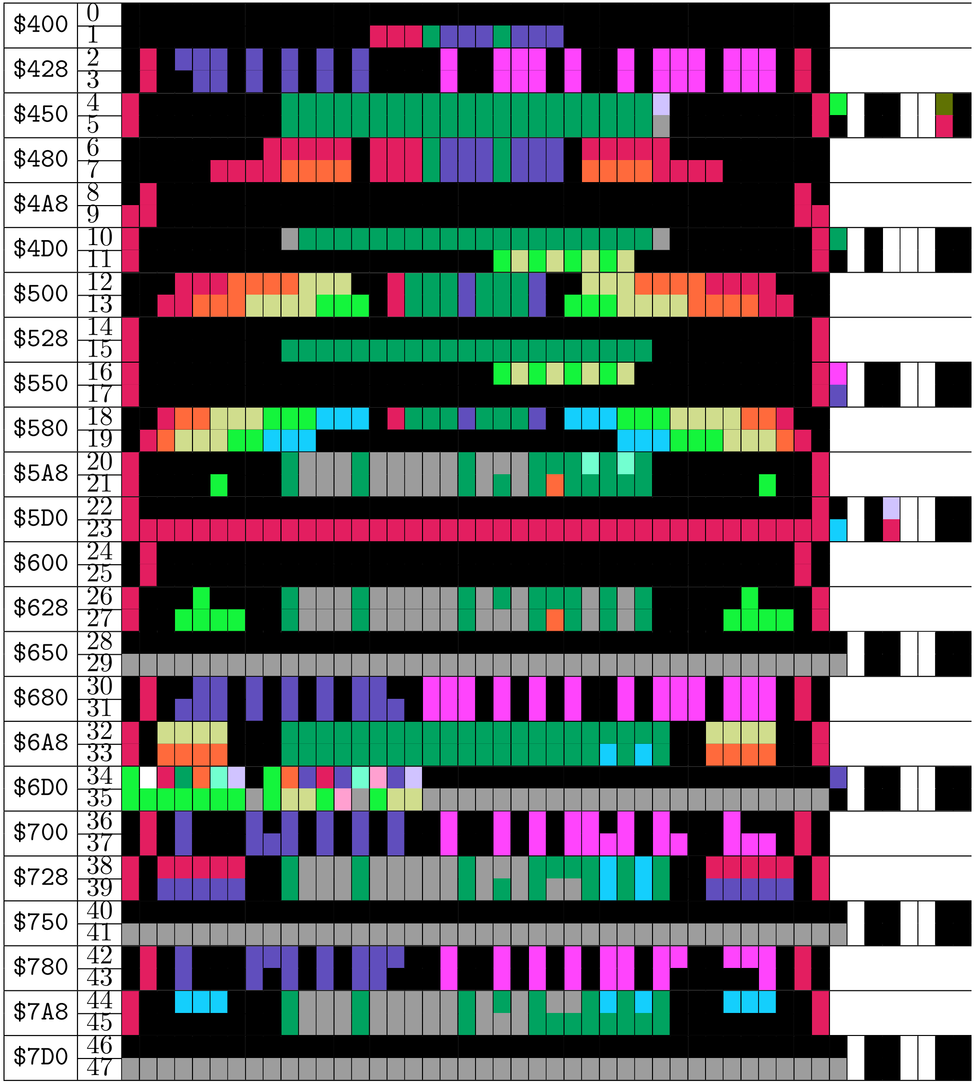


Table 2: The same image from Table 1, but viewed with linear addresses. Note the screen "holes" which pad every third line to a 128 byte boundary. This unused memory can be used by I/O cards.


$400
0
1
$428
2
3
$450
4
5
$480
6
7
$4A8
8
9
$4D0
10
11
$500
12
13
$528
14
15
$550
16
17
$580
18
19
$5A8
20
21
$5D0
22
23
$600
24
25
$628
26
27
$650
28
29
$680
30
31
$6A8
32
33
$6D0
34
35
$700
36
37
$728
38
39
$750
40
41
$780
42
43
$7A8
44
45
$7D0
46
47