Bang! Or: "How To Make A Demo"

"Use your faults, use your defects,
then you're gonna be a star!"

A look inside a price winning demo for
the Atari 2600 VCS

Hackover 2014 - 2014-10-26 - 13:00

What Is A Demo?

Aesthetic appealing program

Definitively an art form

Some kind of digital graffiti,
only with animation and music

Usually with none or very few interactions

Let's try an example: the demo in question

What Is Needed For A Demo?

- Music
  - Very important
  - Had luck, a very good musician found me
- Graphics
  - Hard to separate from code on the VCS
  - There were a lot of guys helping out here
- Code
- Coordinator / "Director"

What To Do With A Demo?

Demos are usually coded to be released at demo-parties.

"Bang!" was intended for and released at Revision, the biggest demo-only-party in Germany. Bigger venues are not in Germany and not demo-only.

To me it's like the inofficial national championships.

Music

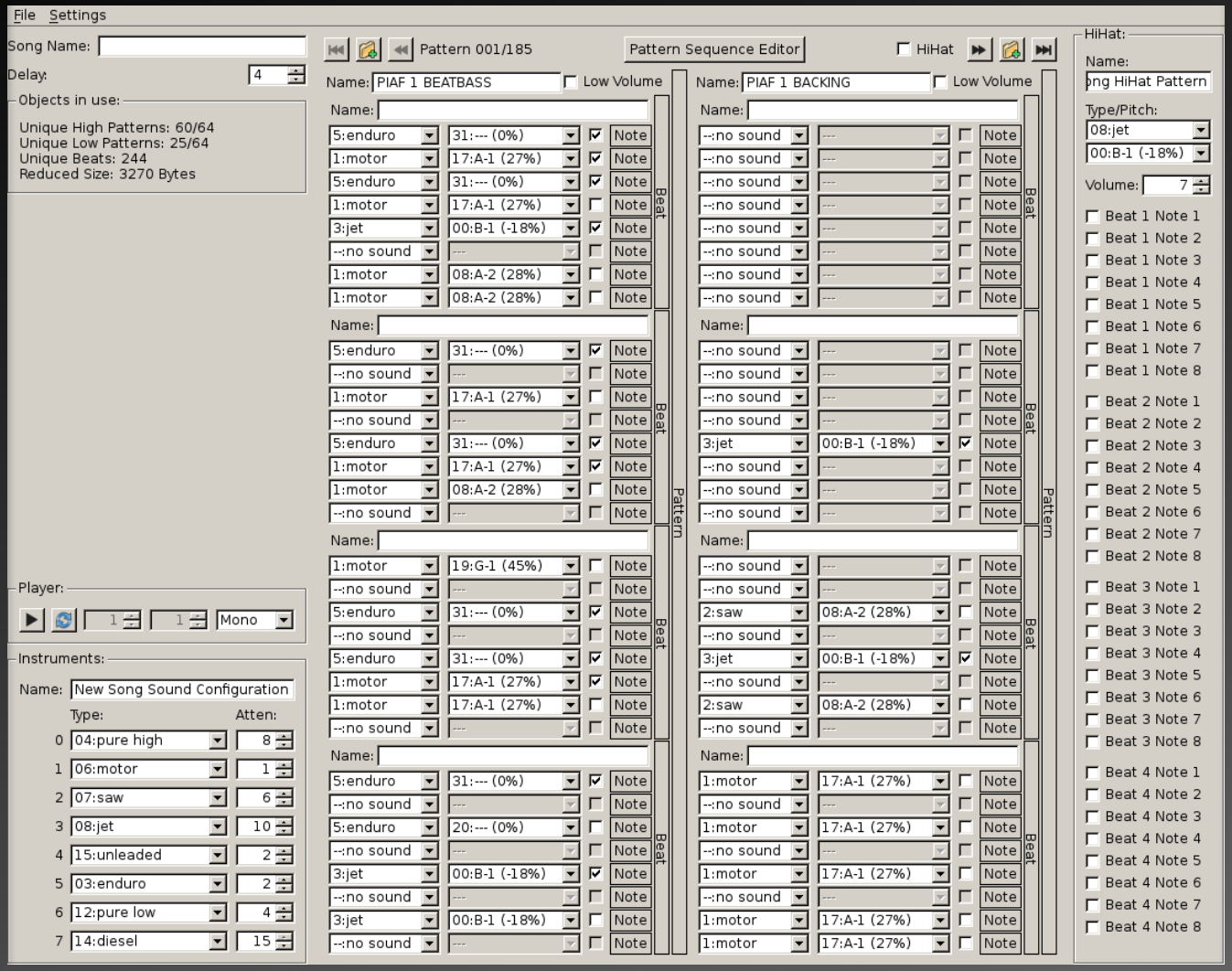
Good music in a demo makes a difference

Music on VCS almost always included coding

There is Music Kit by Paul Slocum:
no coding anymore, but still need to know some details about coding, since the music is still defined in assembler tables

→ Write an editor that creates those tables

Tool #1, "Music Editor": apefat



A Poor Excuse For A Tracker

The Hardware

Before going into the effects of the demo parts, some background info is needed.

What is the hardware capable of?

What are the limitations?

Atari 2600 VCS: The Hardware

The Atari 2600 VCS mainly consists of three chips

CPU, a MOS 6507 (6502 with less pins)

TIA, the Television Interface Adapter

RIOT, RAM/Input/Output/Timer

→ RAM: 128 bytes, 1 timer, no interrupts

Bang! utilized an additional 128 bytes of RAM hosted in the cartridge

Called SuperChip-RAM (SC-RAM)

Graphical Capabilities

Graphic is done by TIA

TIA does only generate graphics but does not "run" anything

TIA is driven by the CPU via memory mapped I/O

CPU runs at ~1.19MHz

The CPU: 6507

6 registers

A: multi-purpose accumulator (8 bit)

X: index register (8 bit)
Y: index register (8 bit)

PC: program counter (16 bit internal, little endian, external bus is 13 bit only)

SP: stack pointer (8 bit) (offset to $0100)

ST: processor status (8 bit)

Instructions take 1 - 3 bytes and 2 - 7 clock cycles

Basic Trick #1:
Fast Mathematical Functions

Typical mathematical function *f:* n → m

In our case n and m are 8bit values, CPU registers

Fastest implementation: table of 256 bytes

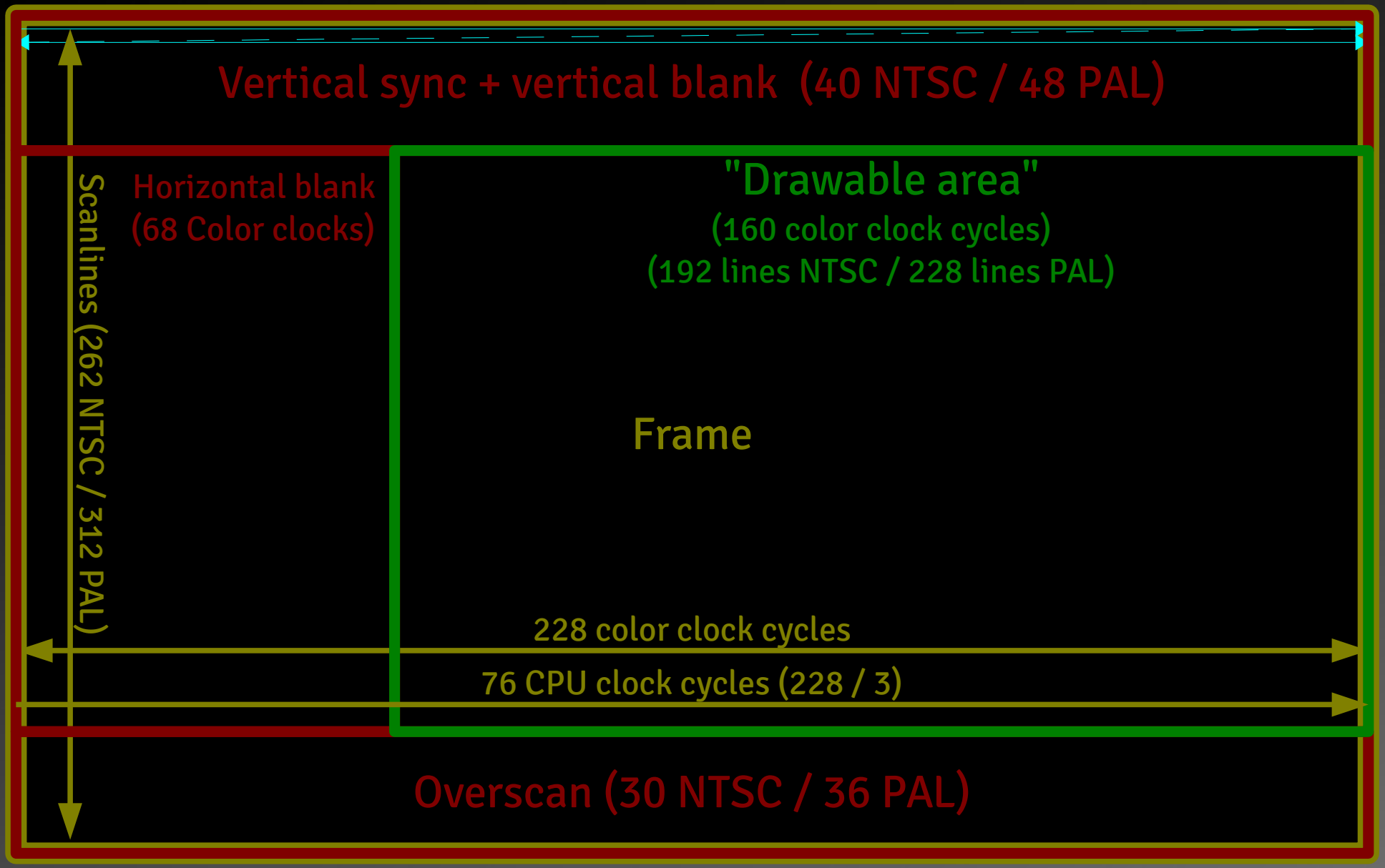
Typical example: sine/cosine wave

LDA cosine_table,X can be read as A=cos(X)

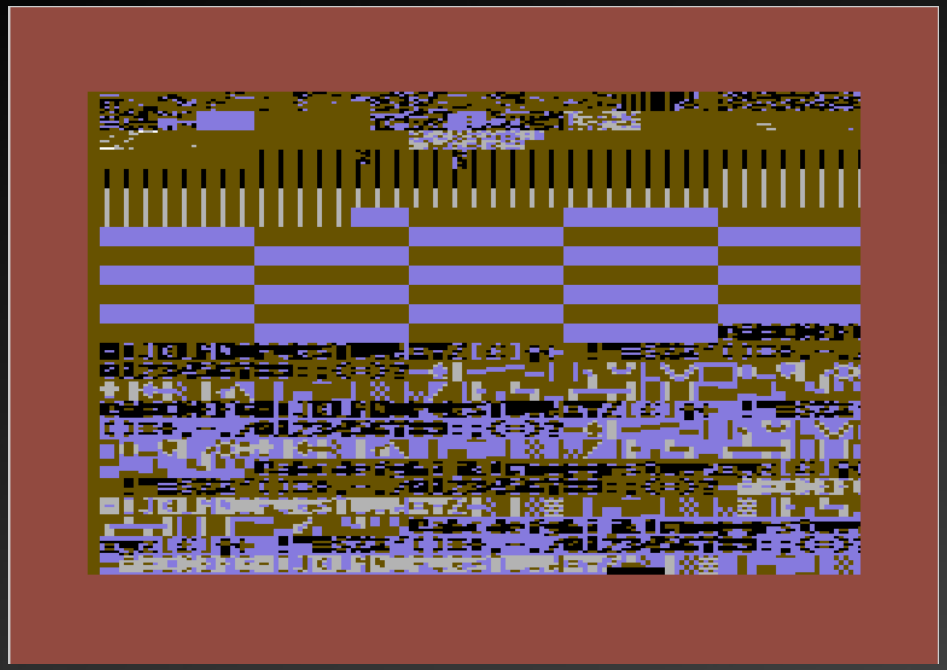
Disadvantage: costs expensive memory (ROM)

Advantage: ~~very~~ extremely fast

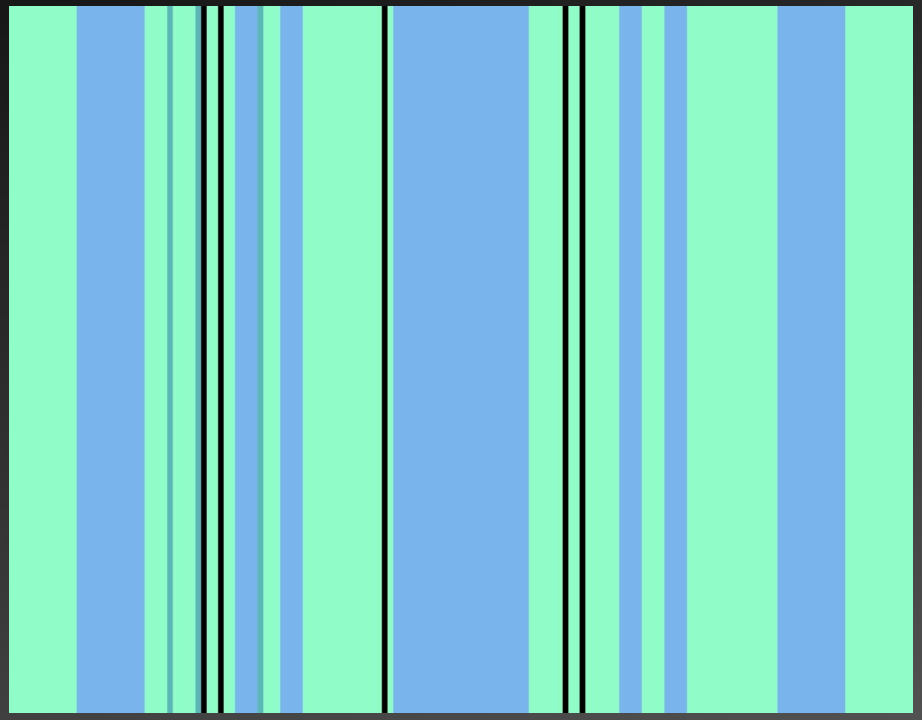
Display


Vertical sync + vertical blank (40 NTSC / 48 PAL)
Horizontal blank
(68 Color clocks)
"Drawable area"
(160 color clock cycles)
(192 lines NTSC / 228 lines PAL)
Scanlines (262 NTSC / 312 PAL)
Frame
228 color clock cycles
76 CPU clock cycles (228 / 3)
Overscan (30 NTSC / 36 PAL)

The Big Difference



Commodore 64



Atari 2600 VCS

These images show an experiment: write random data to graphics hardware and run an endless loop.

"Racing the beam"

Instead of "running" the graphics frame by frame, the image is drawn line by line

If nothing is changed, the next line is drawn like the one before

There are no registers for Y-components

Example: a player sprite size is 8 bit wide and as high as the screen

You need to tell the TIA what to paint while it is painting! This is called "Racing the beam"

Playfield graphics (1)

Resolution: 40 bits – 4 color clock cycles per bit

Registers responsible for playfield generation:

COLUPF, COLUBK: color

PF0, PF1, PF2: data

How to squeeze this 40 bit resolution into 3 bytes?

CTRLPF: control register

- Bit 0: 1=reflect playfield, 0=repeat playfield
- Bit 1: 1=use player colors, 0=use playfield color
- Bit 2: 1=playfield over sprites, 0=sprites over playfield

Playfield graphics (2)

The data registers in depth:

- PF0: `ABCD ----`
- PF1: `EFGH IJKL`
- PF2: `MNOP QRST`

So the playfield data are only 20 bits that can be

Mirrored: `DCBAEFGHIJKLTSRQPONMMNOPQRSTLKJIHGFEABCD`

Repeated: `DCBAEFGHIJKLTSRQPONMDCBAEFGHIJKLTSRQPONM`

Changed: `DCBAEFGHIJKLTSRQPONMdcbaefghijkltsrqponm`

Note: Intuitive and straight forward to code for, well this isn't

Sprites

The TIA has 5 sprites:

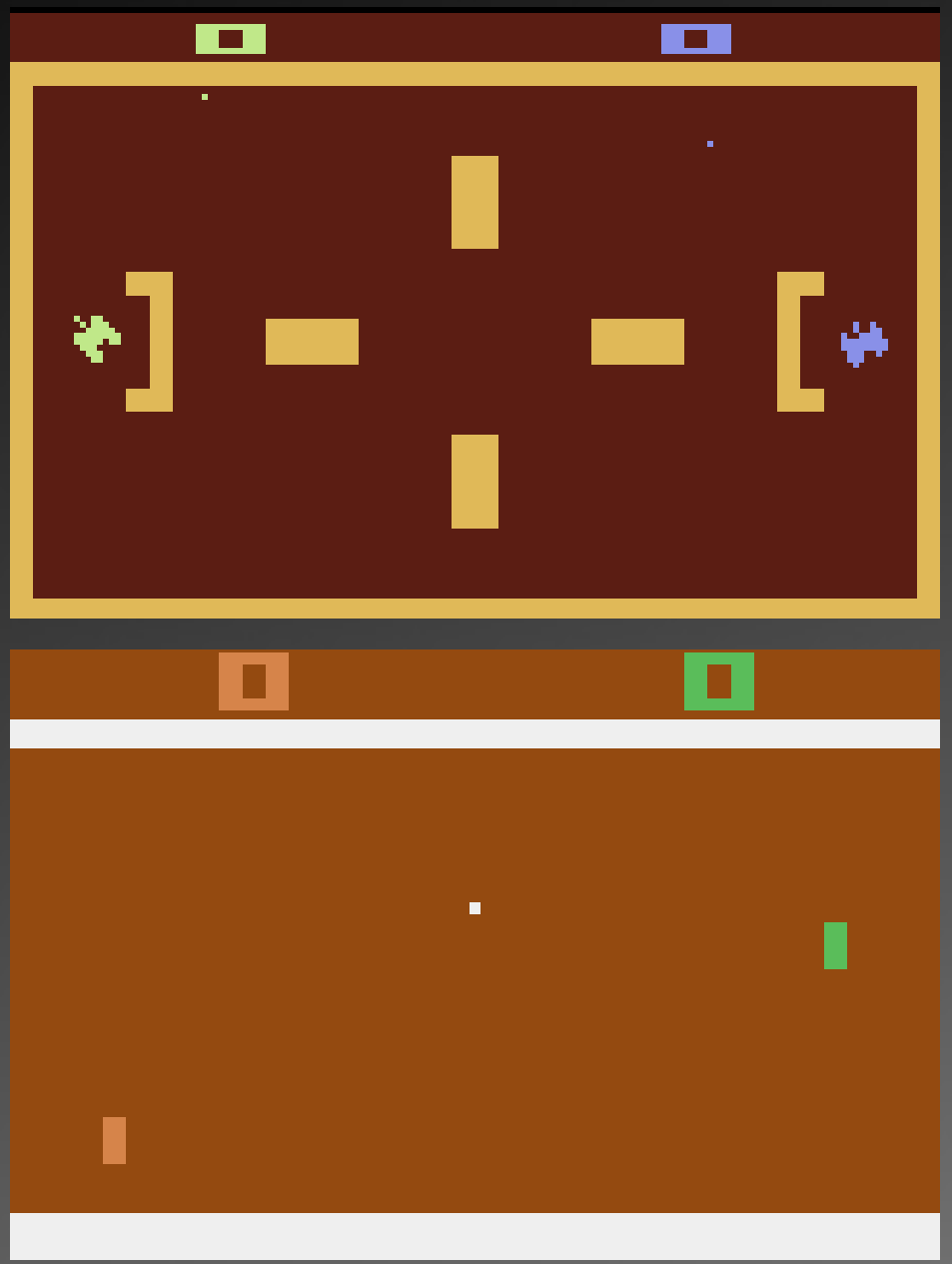
- 2 player sprites (8 bit data)
- 2 missile sprites (1 bit on/off)
- 1 ball sprite (1 bit on/off)

Missile sprite positions can be linked to player positions or positioned independently

Hardware was designed for running

Tank (Combat)

Pong (Video Olympics)

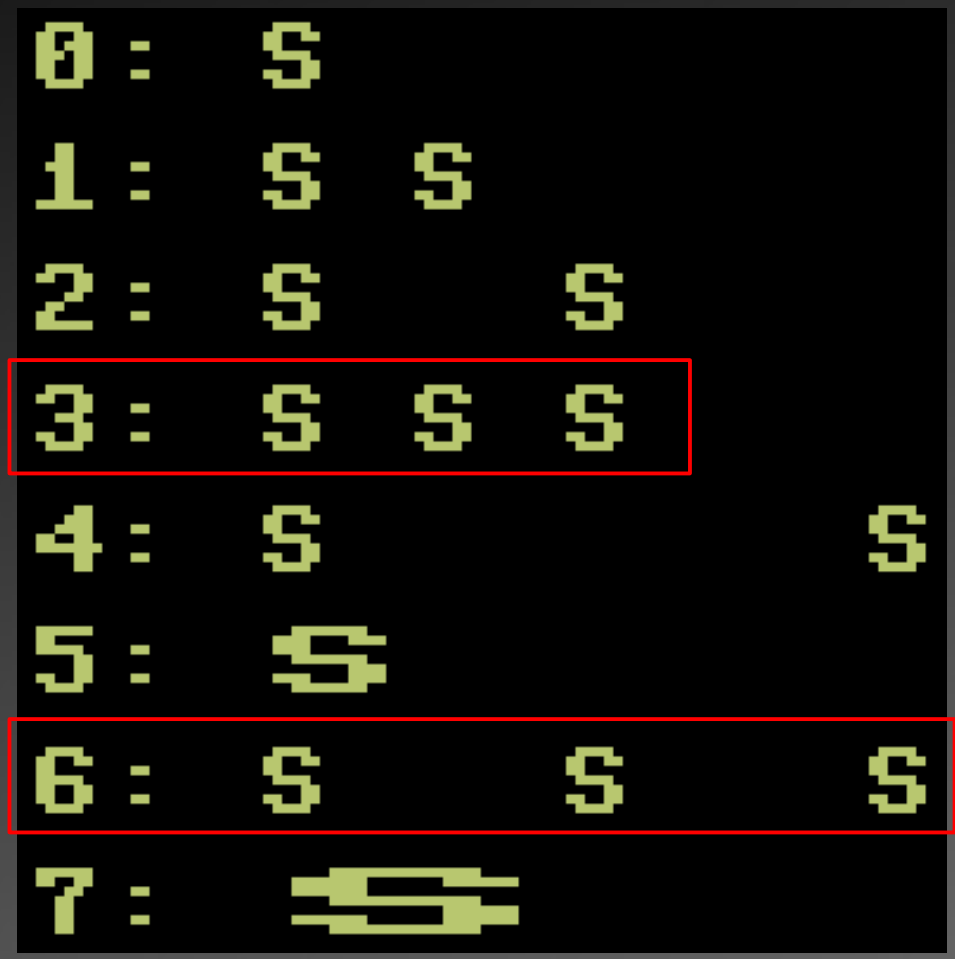


Sprites: size and repetition

The player sprites can be repeated or stretched in 7 different ways

Mirroring of player sprites is also possible

Ball and missile sprites can be defined being in size of 1, 2, 4 or 8 clock cycles



48 Pixel Sprite (1)

We have two player sprites, each 8 pixels wide

Each can be repeated three times with an 8 pixel gap

They can be positioned to form one big 48 pixel sprite

The biggest problem is to change the graphics data at the correct time

Using interlacing even 96 pixels are possible, but flickering occurs

Sprites placement (1)

How are sprites placed on the screen?

Y: enable before beam reaches position

X: more complicated, though

There are registers to reset the sprite position, no value taken

"Reset" has a slightly different interpretation here:

Not reset to position 0, but to current X position of beam

Sprites placement (2)

TIA clock 3 times as fast as CPU clock

Fine-tuning the position:

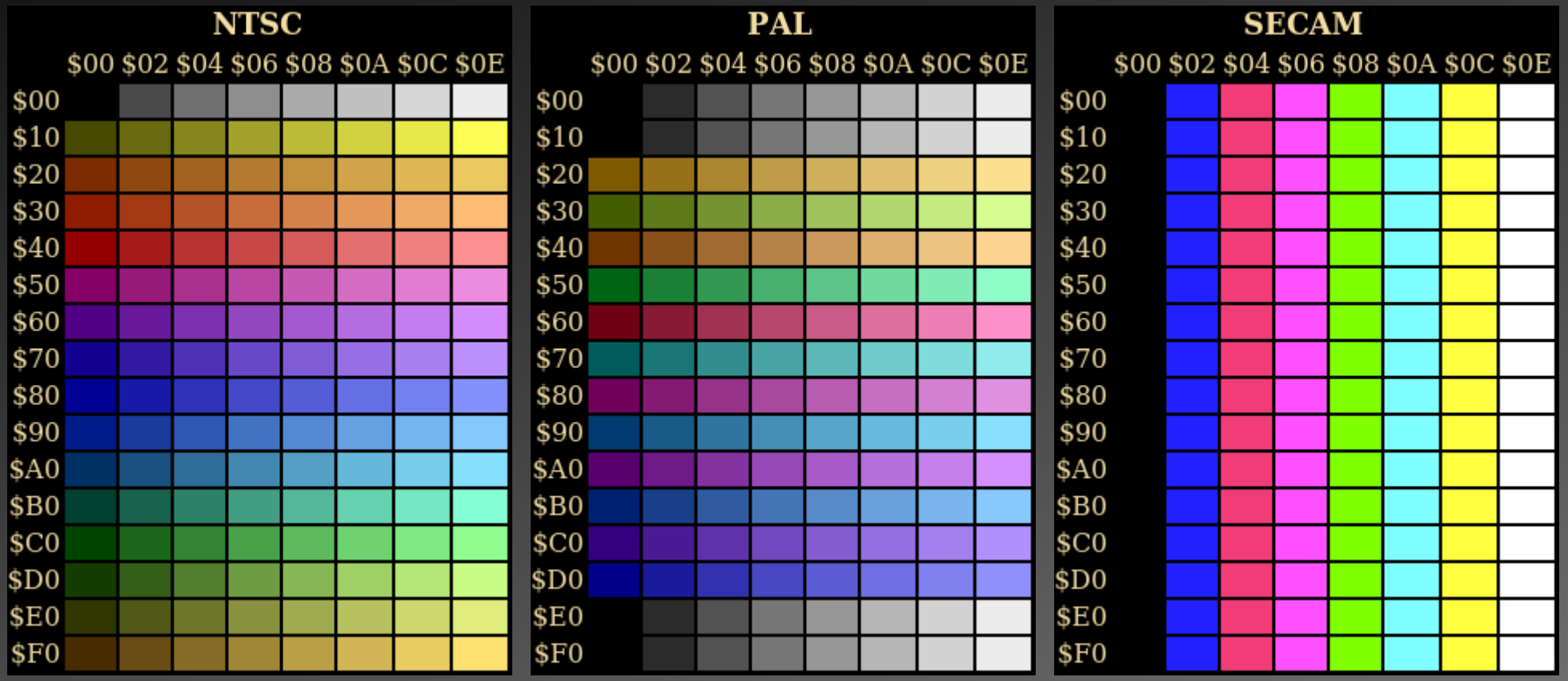
4 bit signed motion registers (1 per sprite)
- can move -8 to +7 color clock cycles
- negative moves right, positive left
- only prepares moving the sprite

Writing to another register moves all sprite at once
- as implied by motion registers

Colors (1)

4 Color registers: background, playfield, 2 players

Each color can be picked out of a palette of 128



Colors (2)

COLUBK
– background

COLUPF
– playfield, ball

COLUP0
– player 0, missile 0
– playfield left half (score mode)

COLUP1
– player 1, missile 1
– playfield right half (score mode)

A Good Atari 2600 VCS Demo?

Before we now through the parts of the demo, let's ask this difficult question:

What makes Atari 2600 VCS demo a good demo?

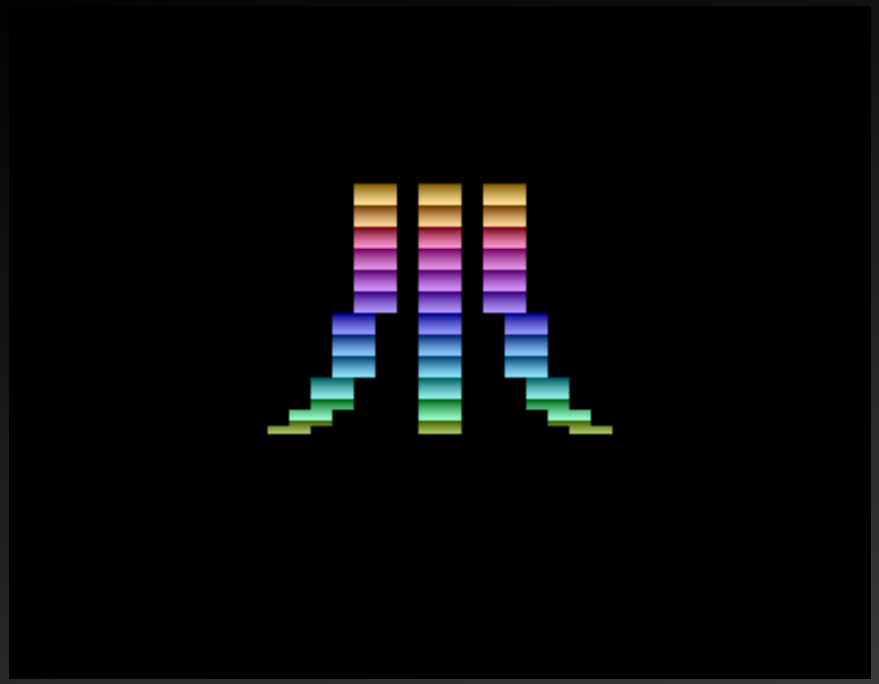
"A good Atari 2600 VCS demo doesn't look like an Atari 2600 VCS demo at all!" – Jac!

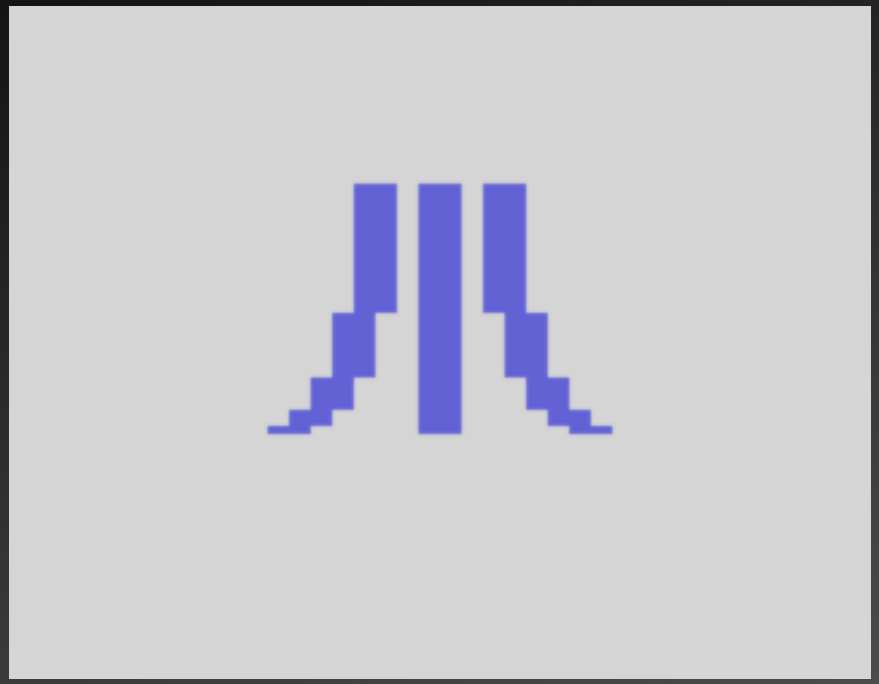
Stella Developer Colors

- Each screen is shown twice
  - Correct colors (read from color registers)
  - Colors describing what hardware is used

	Dark	Bright
Red	Player 0 (8 bit)	Missile 0 (1 bit)
Yellow	Player 1 (8 bit)	Missile 1 (1 bit)
Blue	Playfield (20+1 bit)	Ball (1 bit)
Grey/White	Background	"Bad Lines"

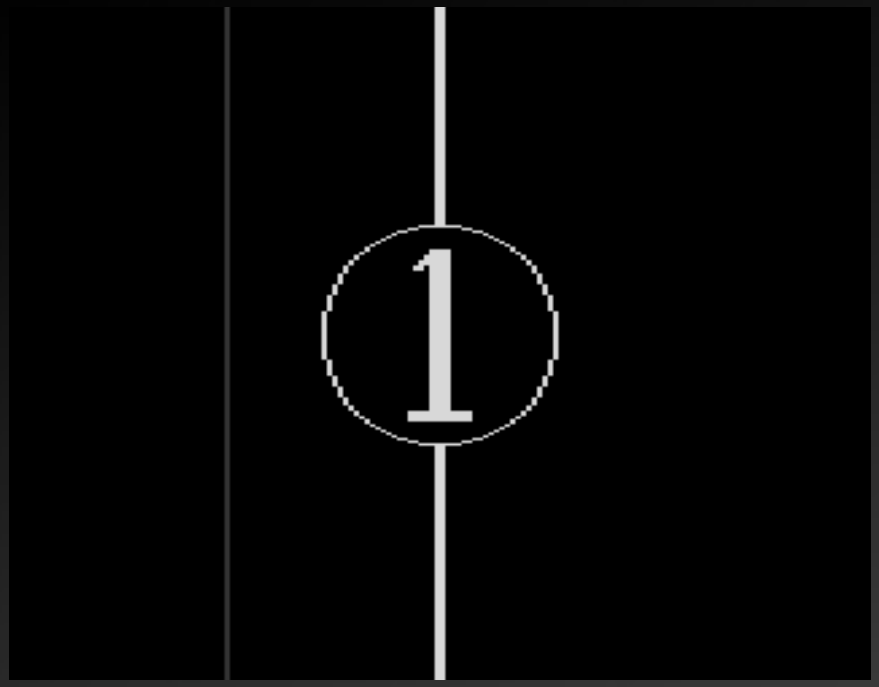
Intro 1: Gameboy-Like Logo

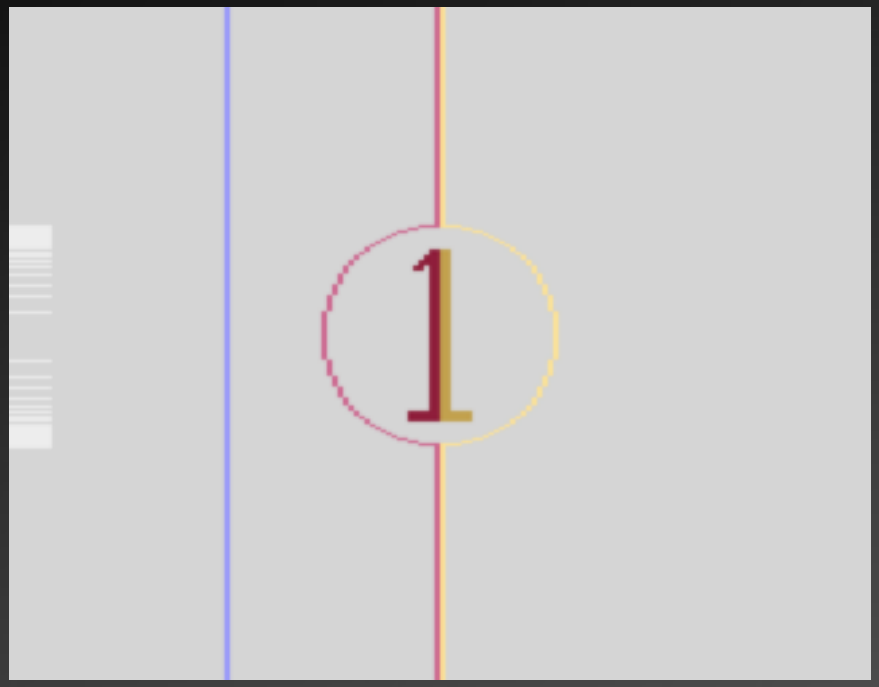




Graphics calculated on first frame, 127 bytes in ROM, color values are calculated on the fly

Intro 2: Film Countdown





Using all sprite hardware, ball sprite for lens-fuzz, missile sprites for line, player sprites for number. Missile sprites are not symmetric because of sprite width.

Intro 3: Edith Piaf

Moving sprites 8 pixels back and forth, using a corner case of the TIA for moving left.

Intro 4: AtariAge Logo

Basically same as "Film Countdown", playfield graphics used as background, sprites overlapping to "smooth" edges.

Intro 5: Pouët.net Logo

Using sprites over playfield to create the illusion of a higher resolution.

Intro 6: MEGA Logo

49(!) pixel sprite, additionally using ball sprite. Of all intro logos, this is the most complex code, best technical achievement, only real animation, and closest to the original logo.

Intro 7: XAYAX Logo

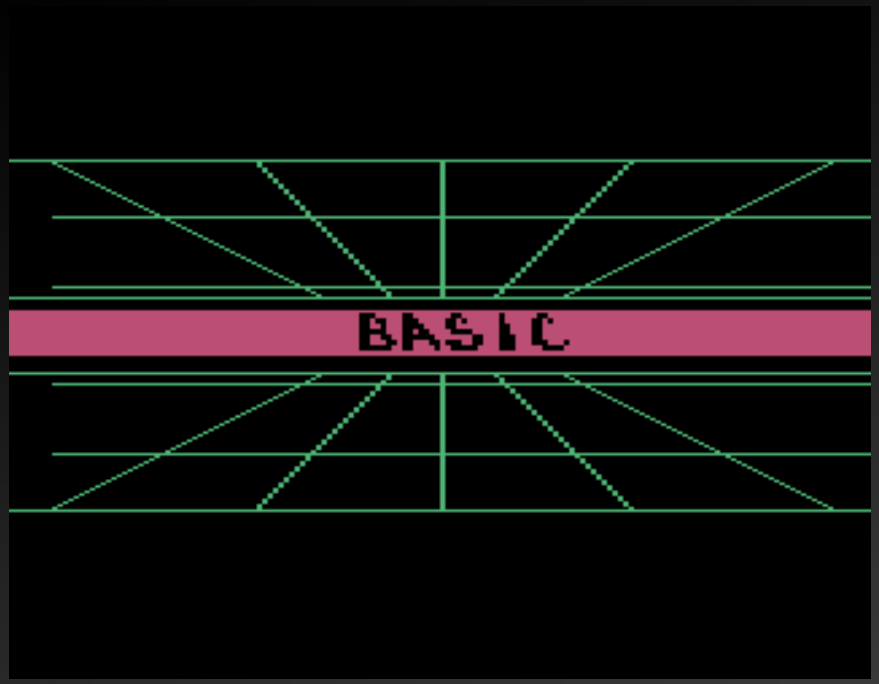
Using sprites and replication to create the illusion of high resolution graphics.

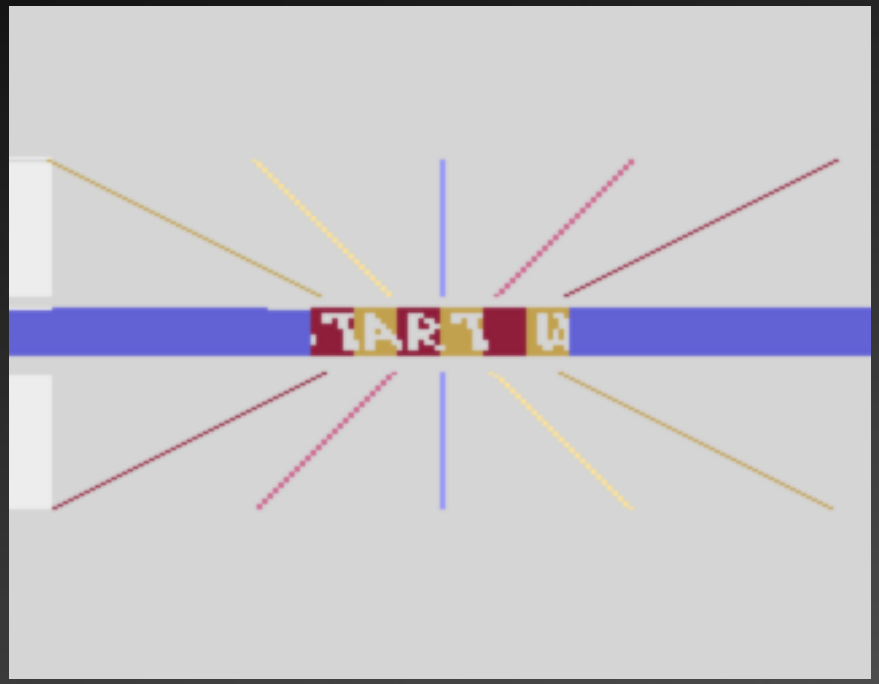
The name XAYAX was chosen because it's very easy to display on the VCS.

Intro 8: Bang! Title

Using SC-RAM for the "falling" sprite to drop the need for a compare, since the playfield graphics take up most of the rastertime.

Chapter 1-1: 48 Pixel Scroll

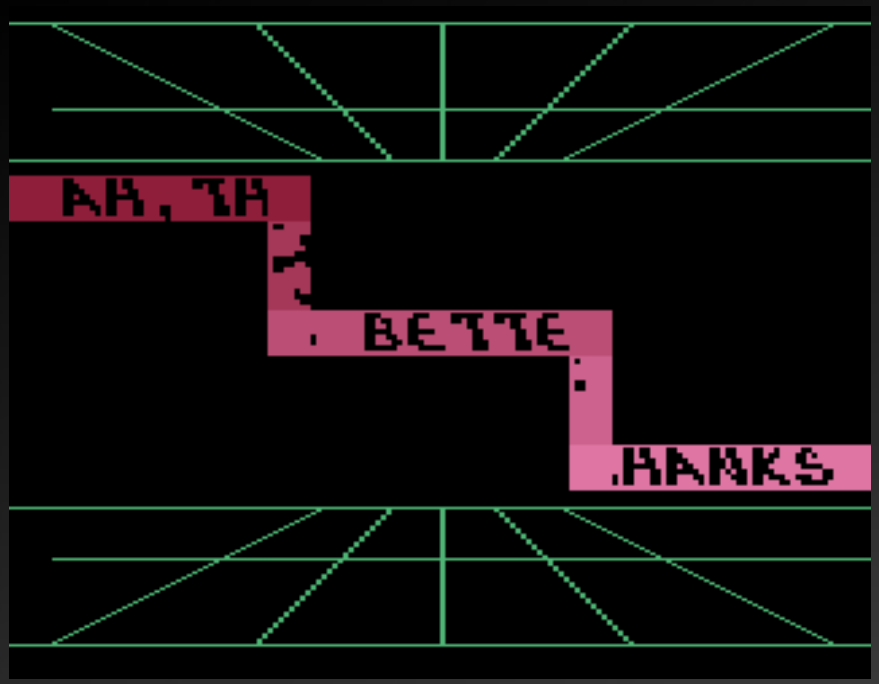


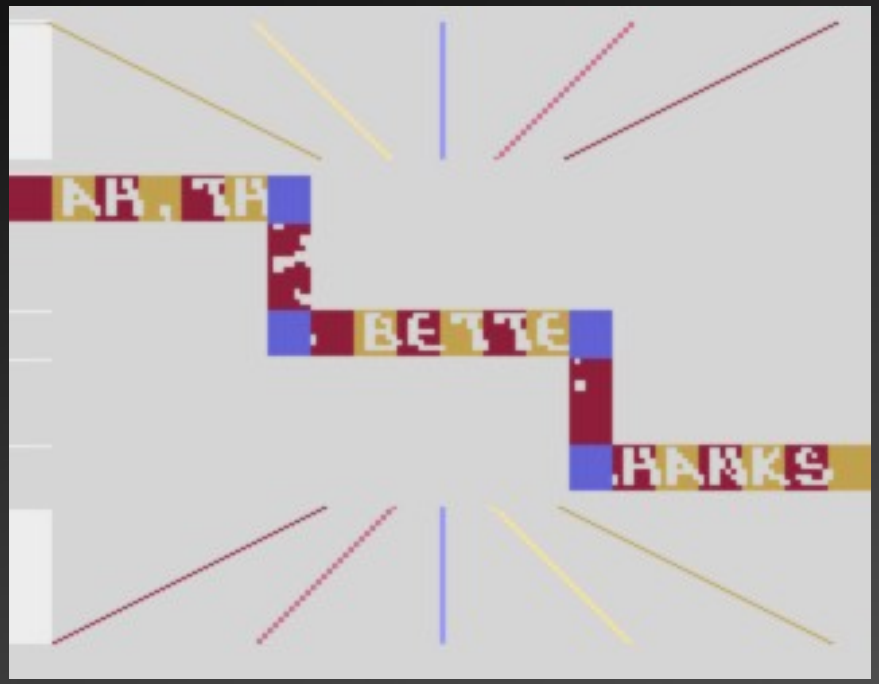


Starting of as a rather typical part.

Typical 48 pixel scroller.

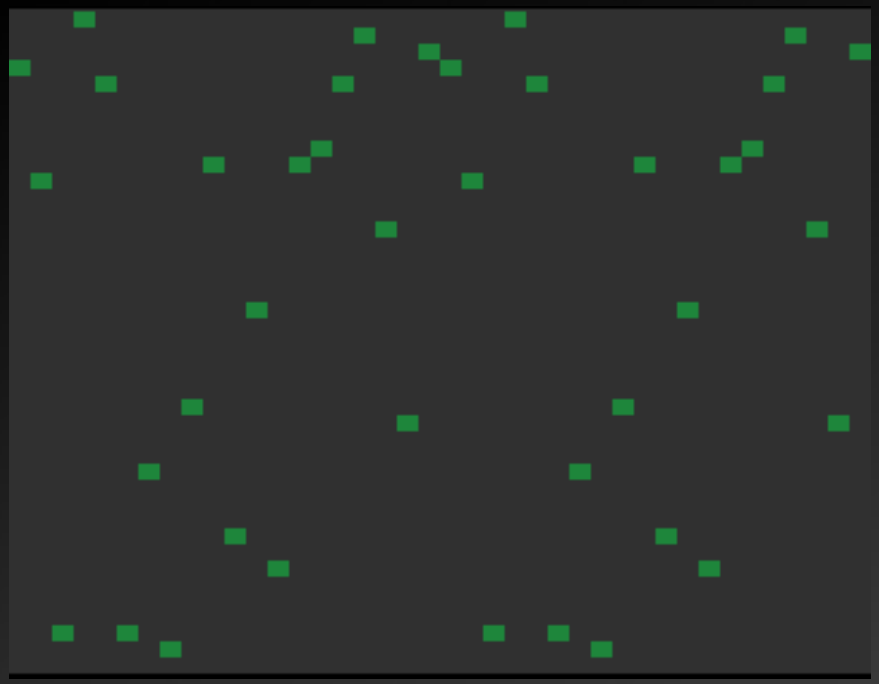
Chapter 1-2: 160(!) Pixel Scroll

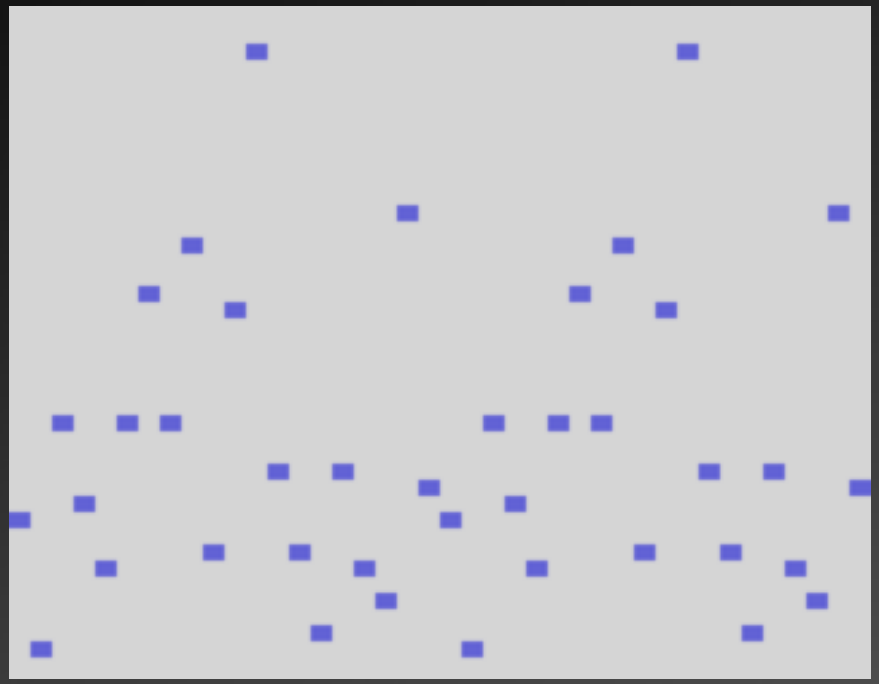




Since only 48 pixels can be shown per line, use 3 of those on different positions to form a some kind of stairs. A bit tricky was the "removing" of playfield data used as "turning blocks". SC-RAM needed.

Chapter 2-1: Matrix



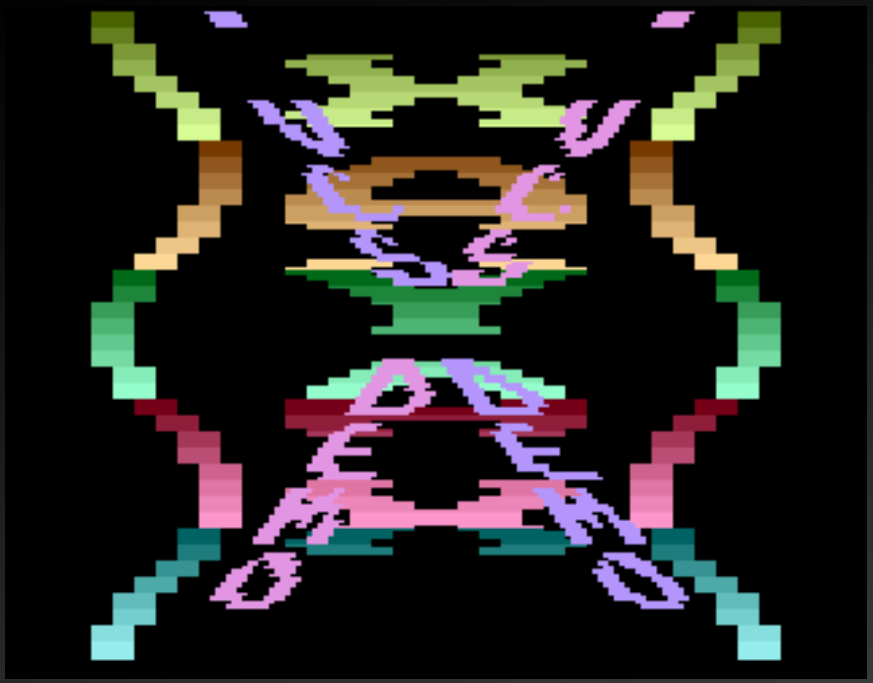


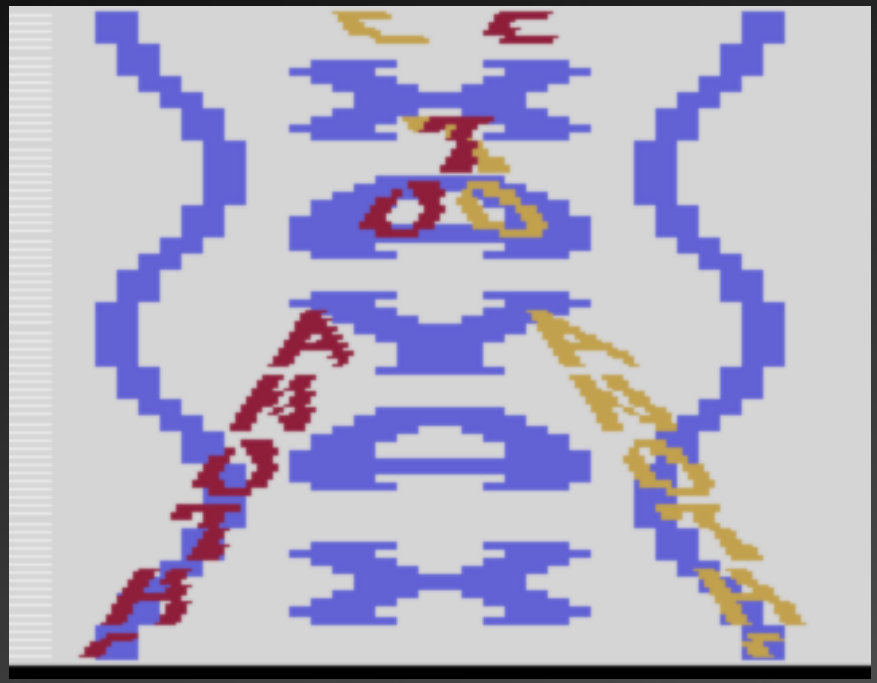
Height: 41 lines

Using disadvantages of the VCS to my advantage

Calculating next line while leaving registers unchanged, using playfield repeat

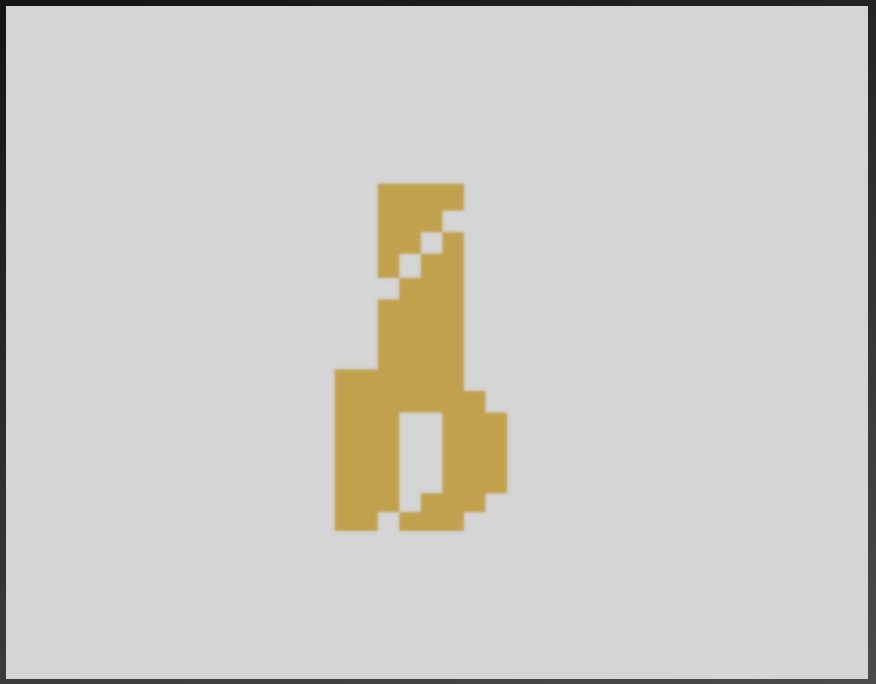
Chapter 2-2: Bresenham





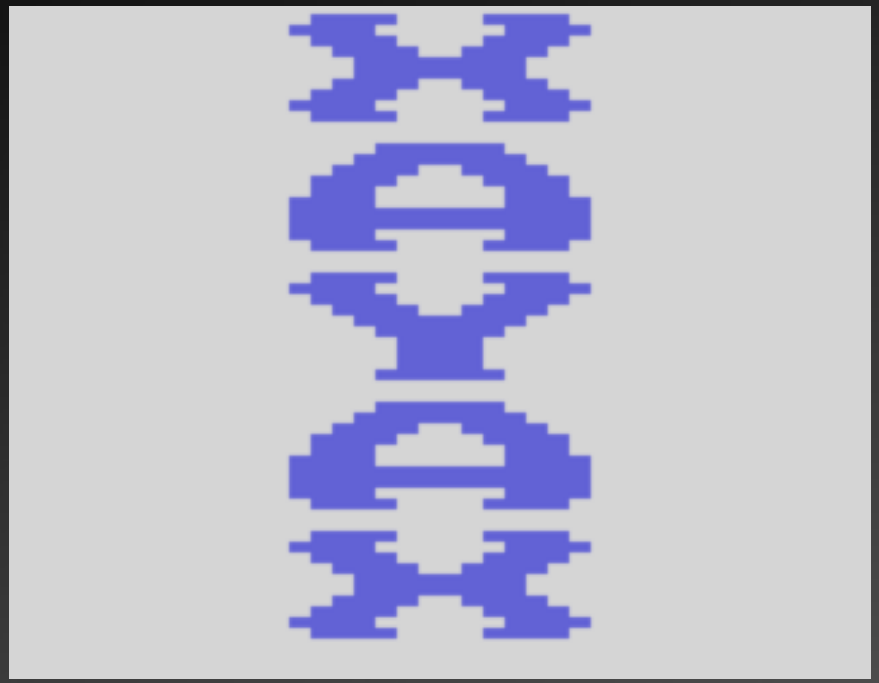
Implementation of the Bresenham line algorithm VCS style: made sure that the angle is not > 45°, using HMOVE to move both sprites at once. XAYAX down-logo is written to SC-RAM, because it's used later on with different spacing.

Chapter 3: Rotate Text



Complex mathematics done using macros and tables. Expanded player sprite used for letters.

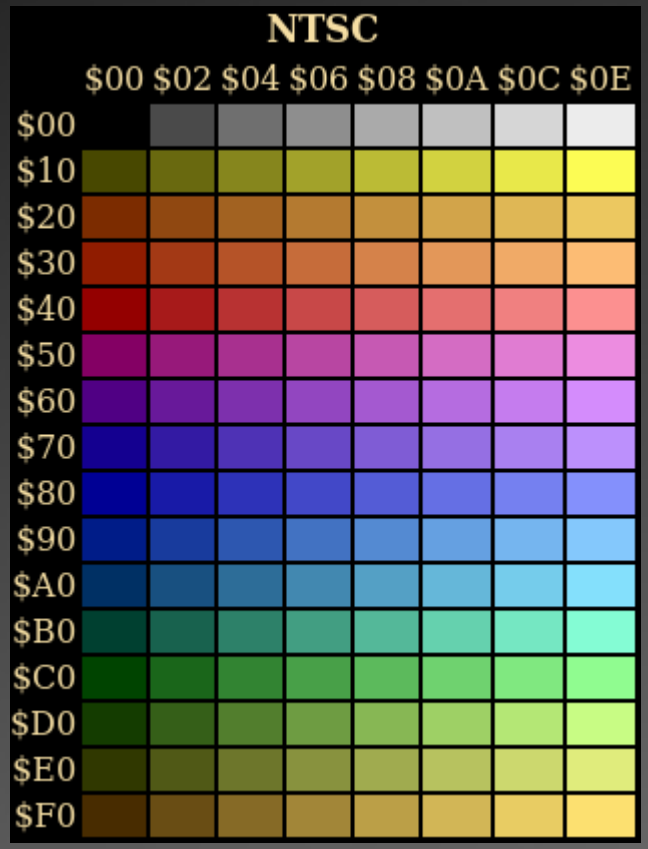
Chapter 4-1: 1D Plasma



A fixed point cosine table, and a good color table, add different phases convert the result into a color, all tables are 256 bytes to utilize the "rollover" of index registers. Again "XAYAX" can be displayed easily, read from SC-RAM.

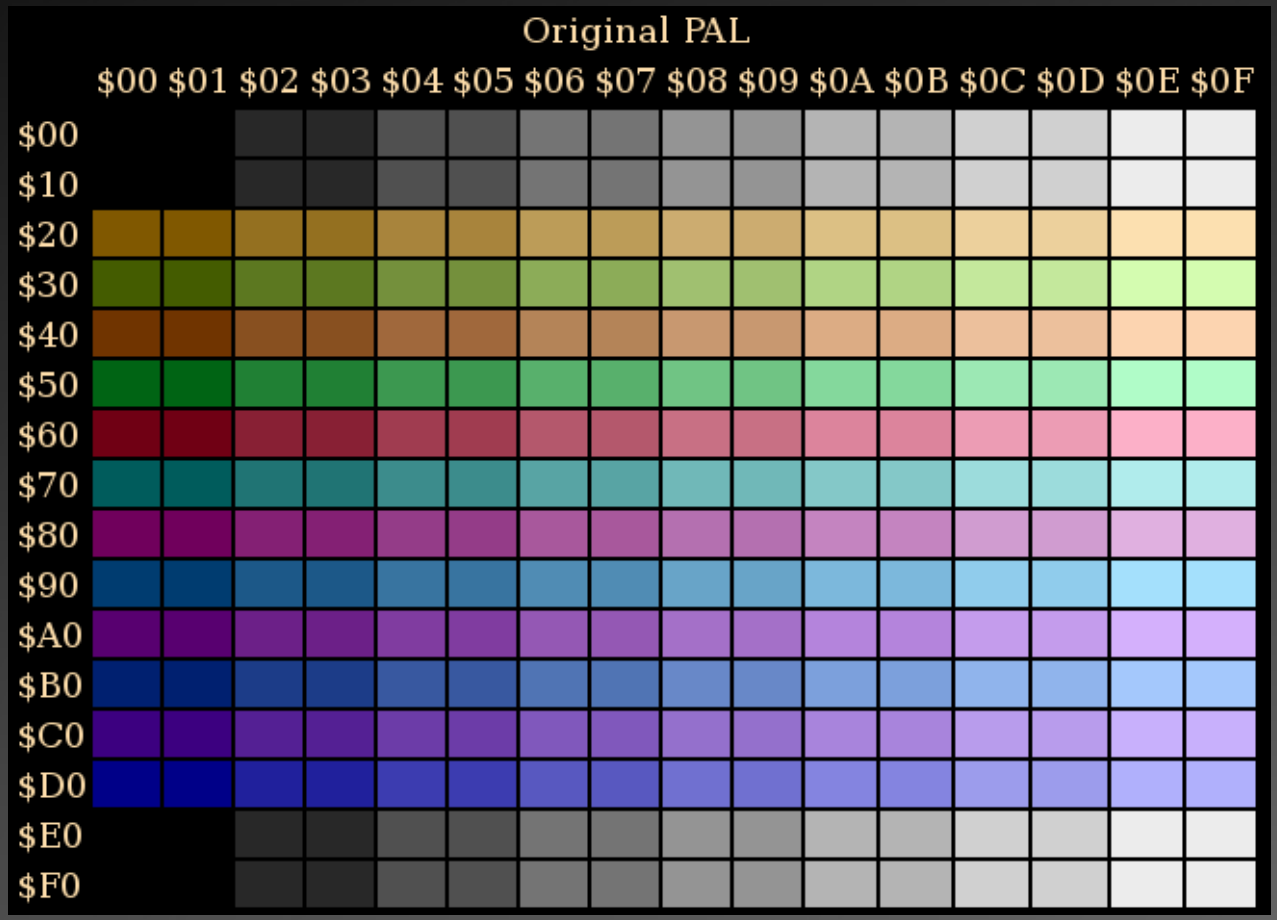
Basic Trick #2: Color Table

Let's take a look at the available colortable:
the 104 colors of PAL are okay, but order is bad
compared to NTSC



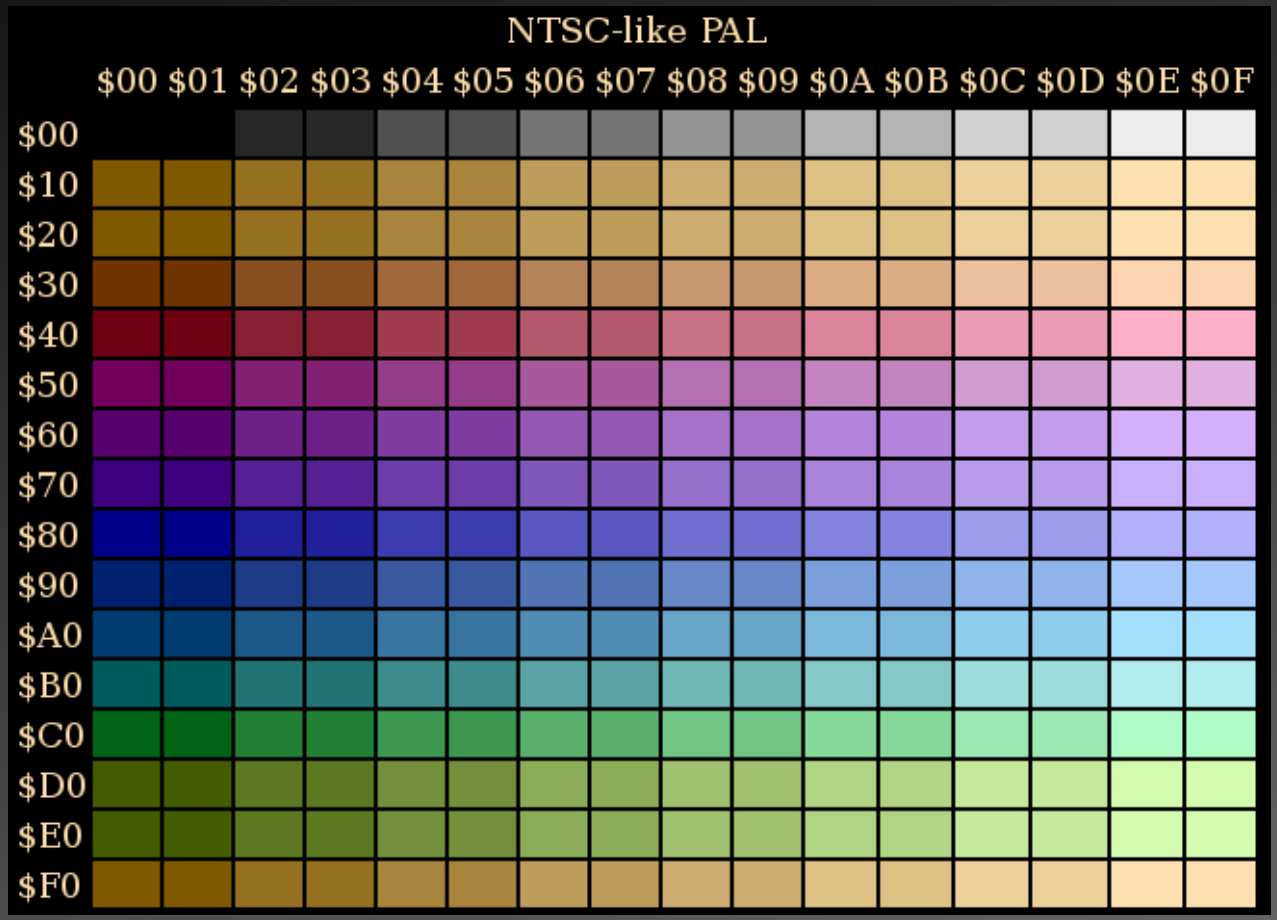
A Not So Good Color Table

All 256 possible values for colors, a lot of gray and the ordering is unhandy compared to NTSC



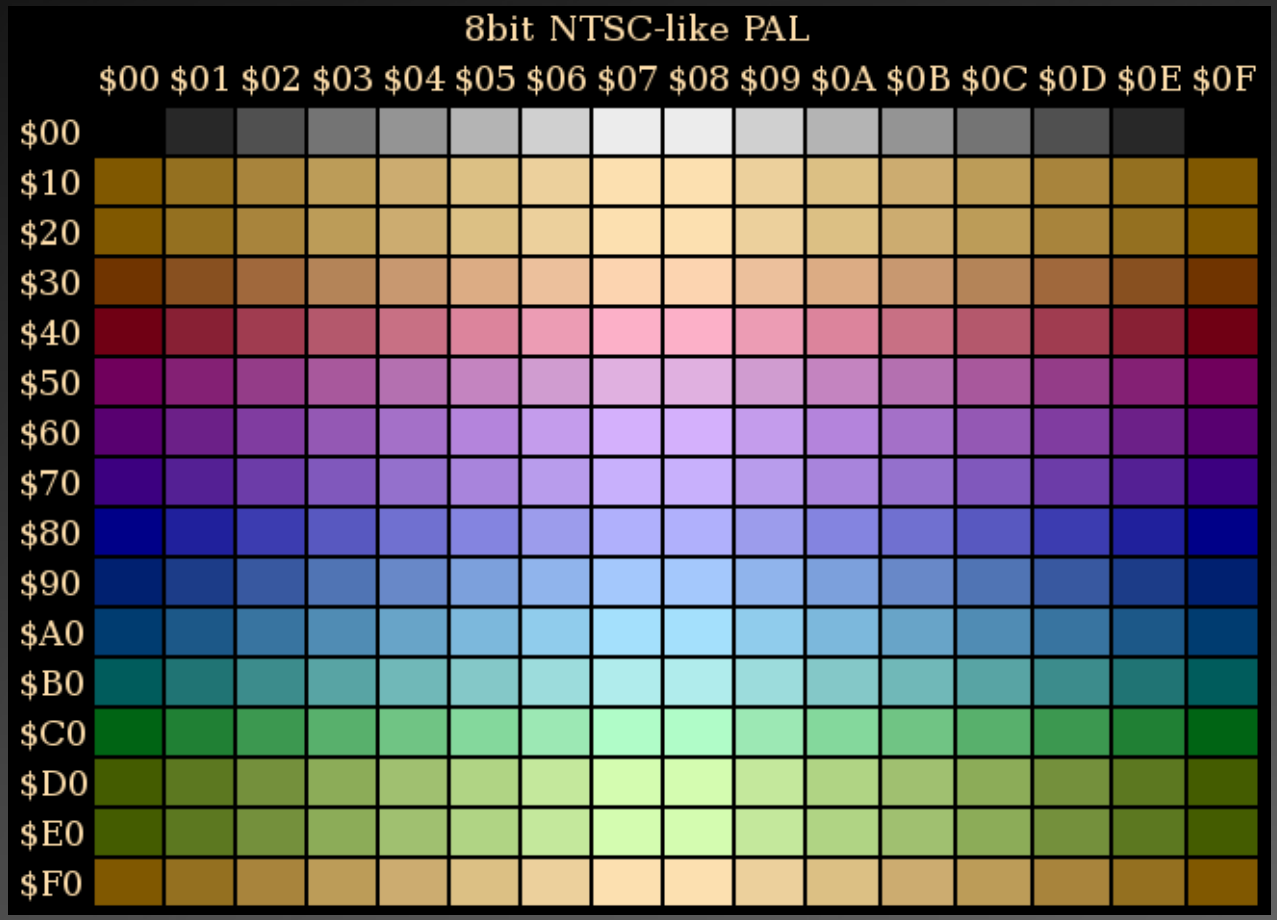
A Better Color Table

Reorder colors (hi-nibble) to resemble the NTSC "layout" using base colors that are closest available

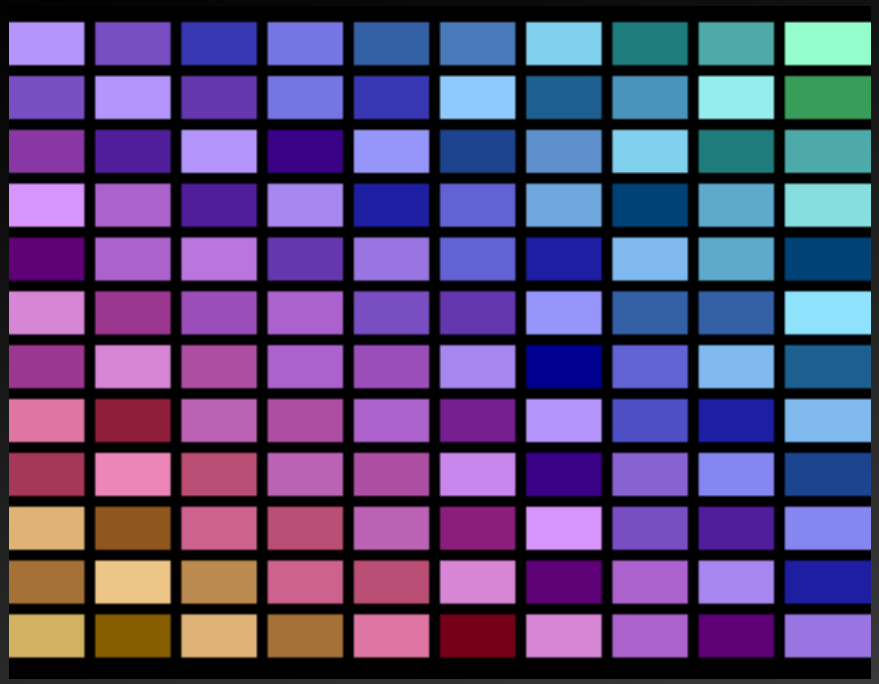


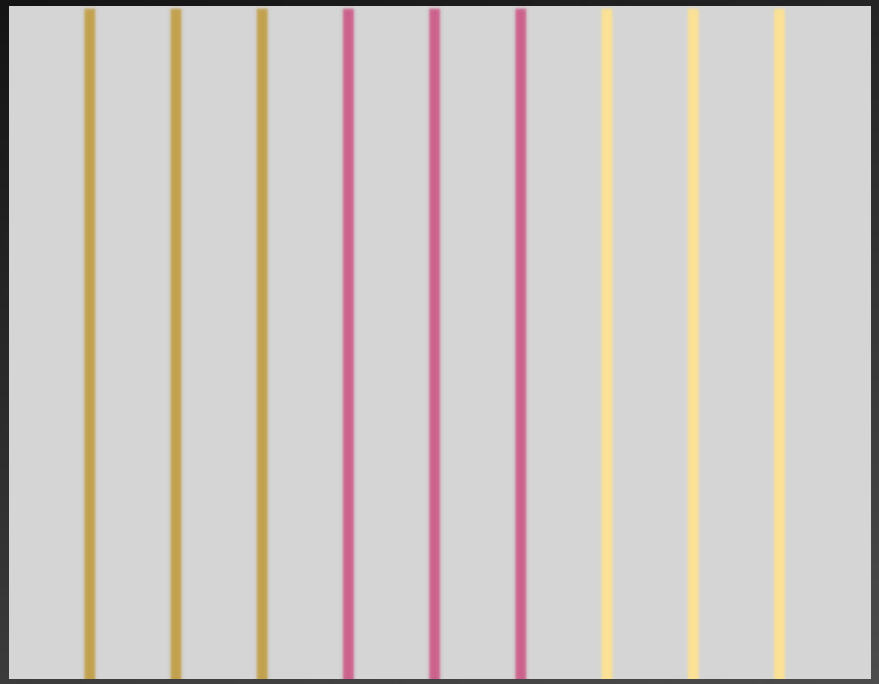
Best Color Table So Far

Reorder colors (lo-nibble) for a "fade in and out"-effect instead of colors just getting brighter



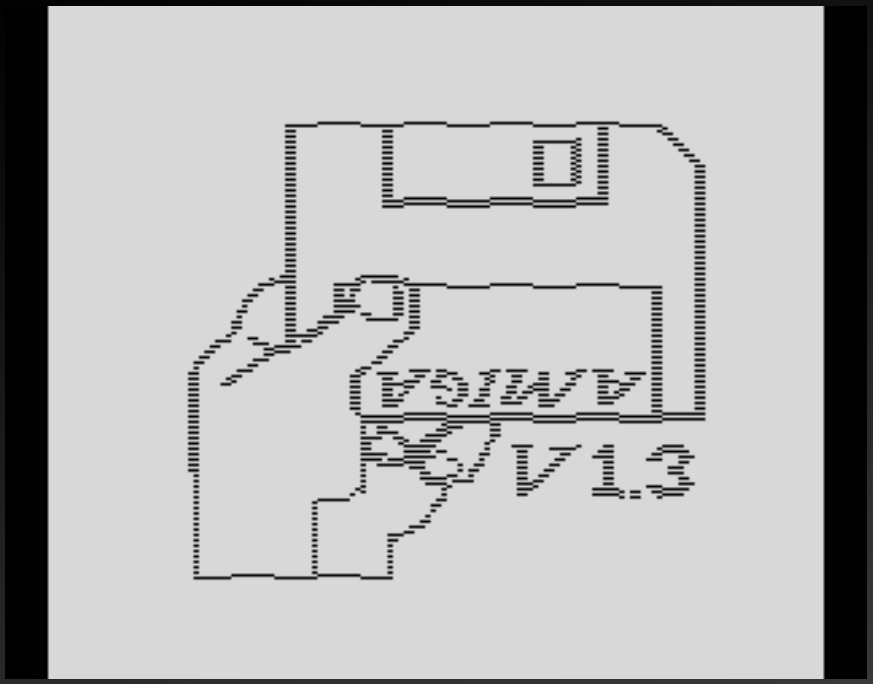
Chapter 4-2: 2D Plasma

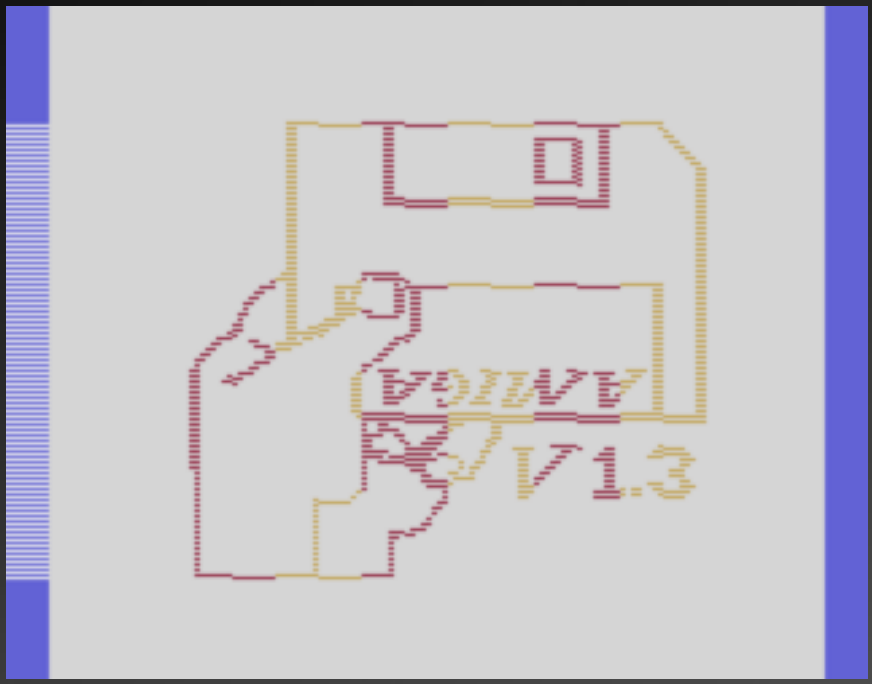




Part of the code is copied into RAM for modification and run there for speed purpose. Illusion of panels being 16 pixel wide, black lines hide the fact that they are 15, 18 and 15 pixels wide.

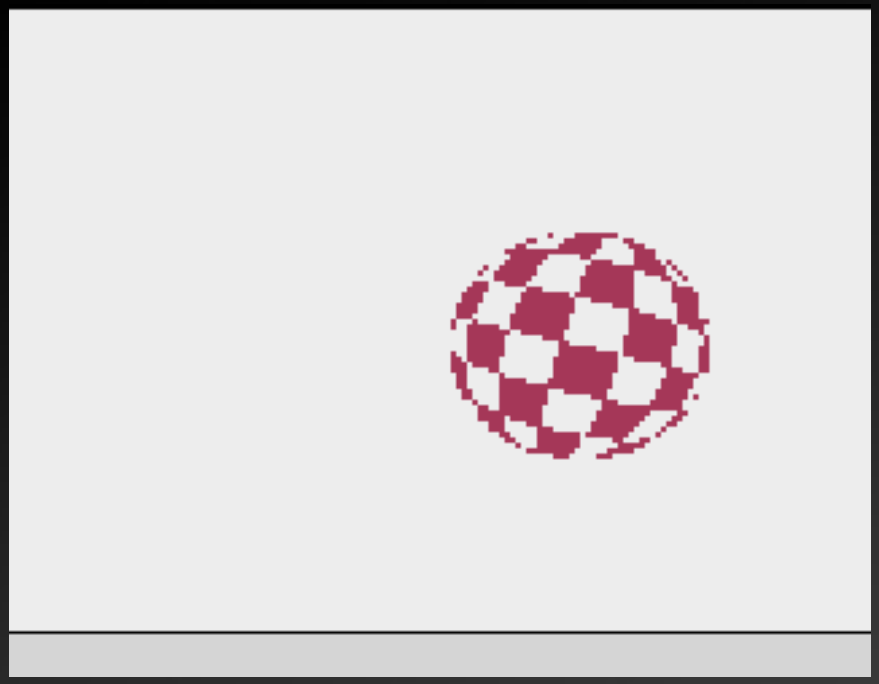
Chapter 5-1: Amiga Disk

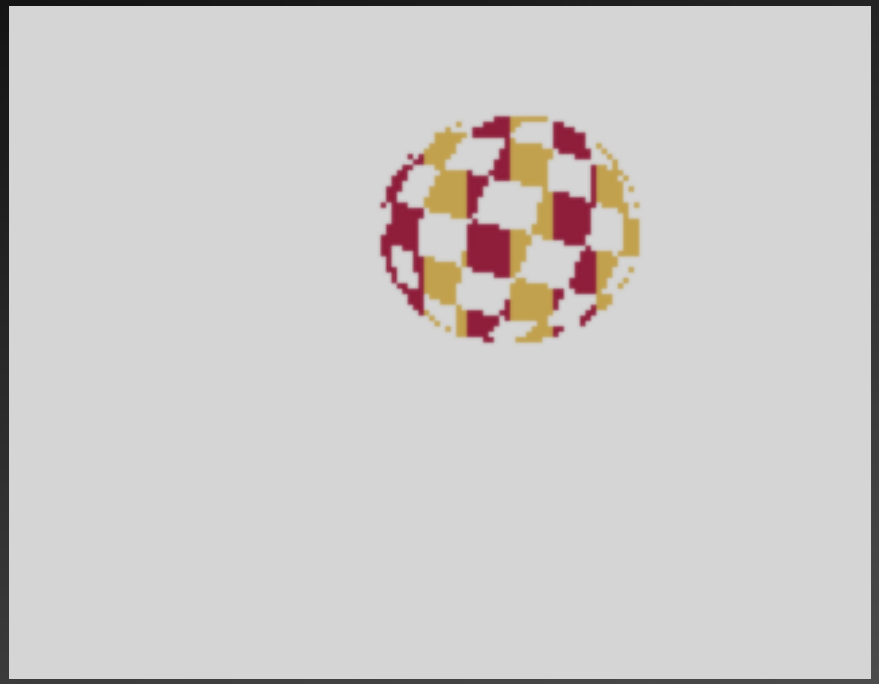




Typical 96 pixel interlace sprite, using same routine as for Edith Piaf. Note the hiding of the "sprite move lines" by using black playfield graphics.

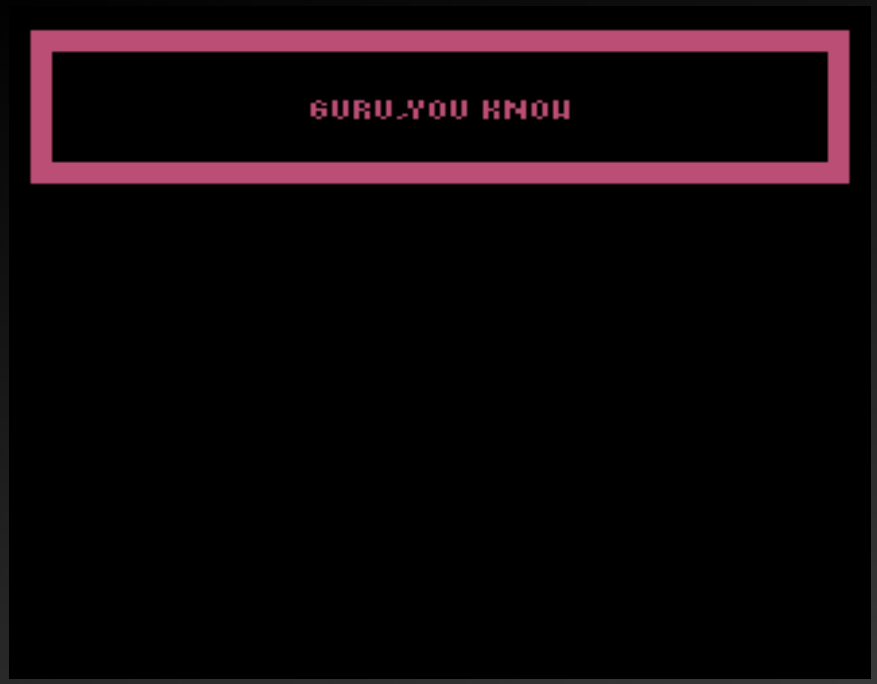
Chapter 5-2: Amiga Ball

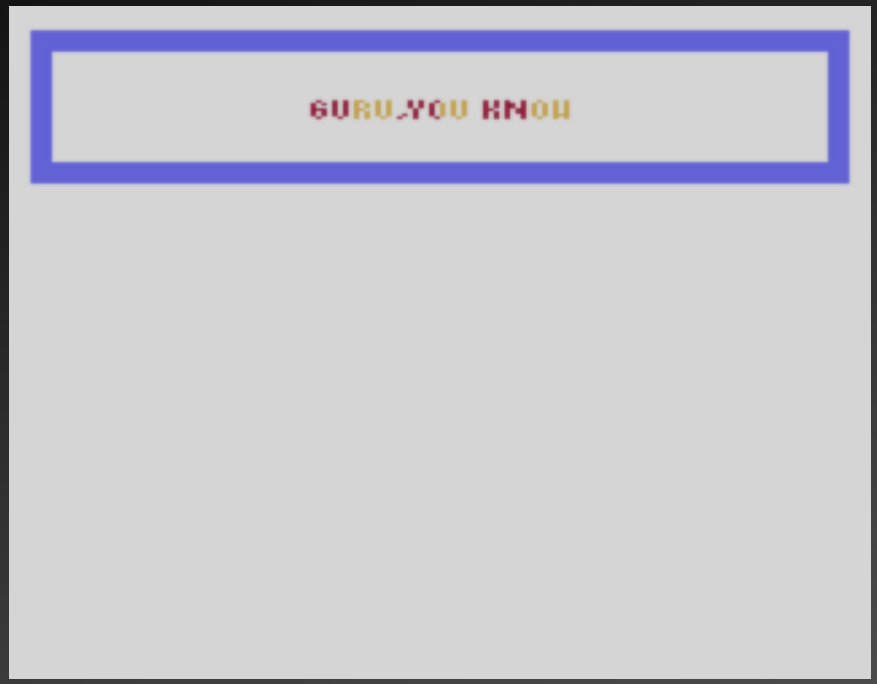




48 pixel sprite with 6 animation frames + 1 for Revision logo. Cosine is pre-calculated this time.

Chapter 5-3: Amiga Guru



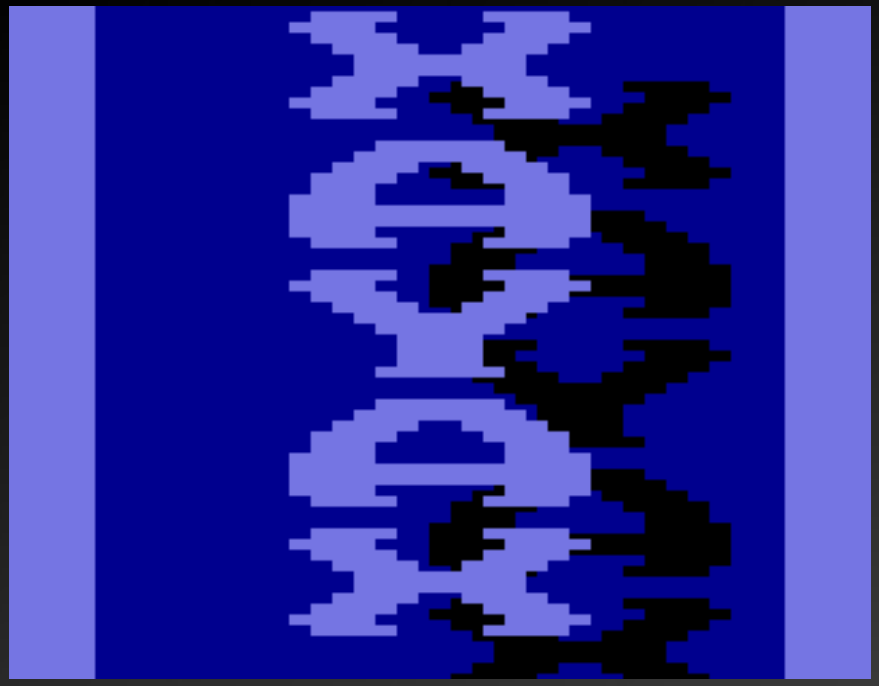


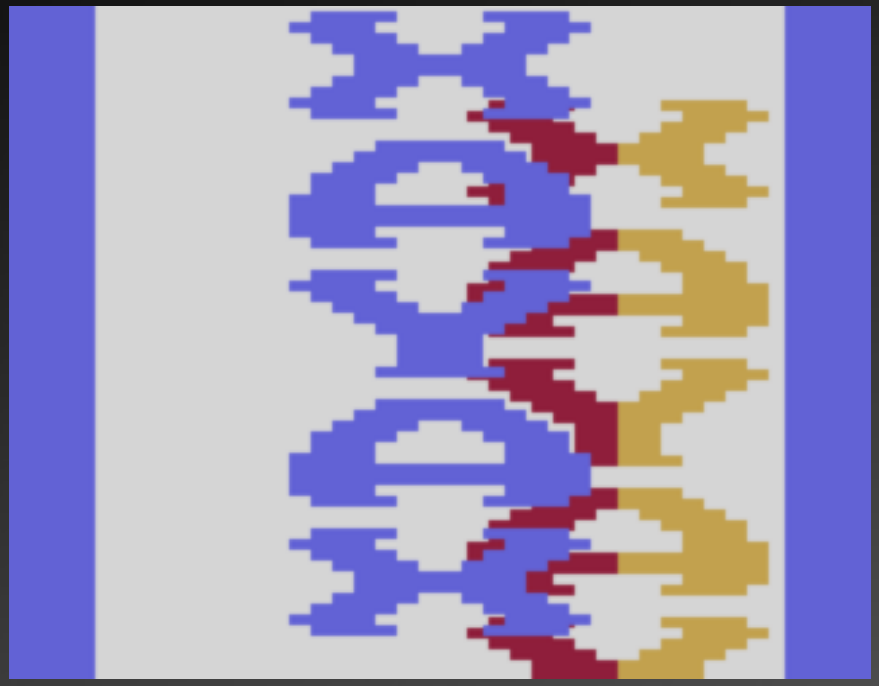
Originally intended as "Guru Meditation", there was no way to squeeze the message into 48 pixels, so I tried to cover this with a pun. Code shared with C64 load.

Chapter 6-1: C64 Loading

It's a hard time to create something resembling the C64 without having bitmaps, using playfield to cover up "unreachable" areas. Tape loading is an exact replication.

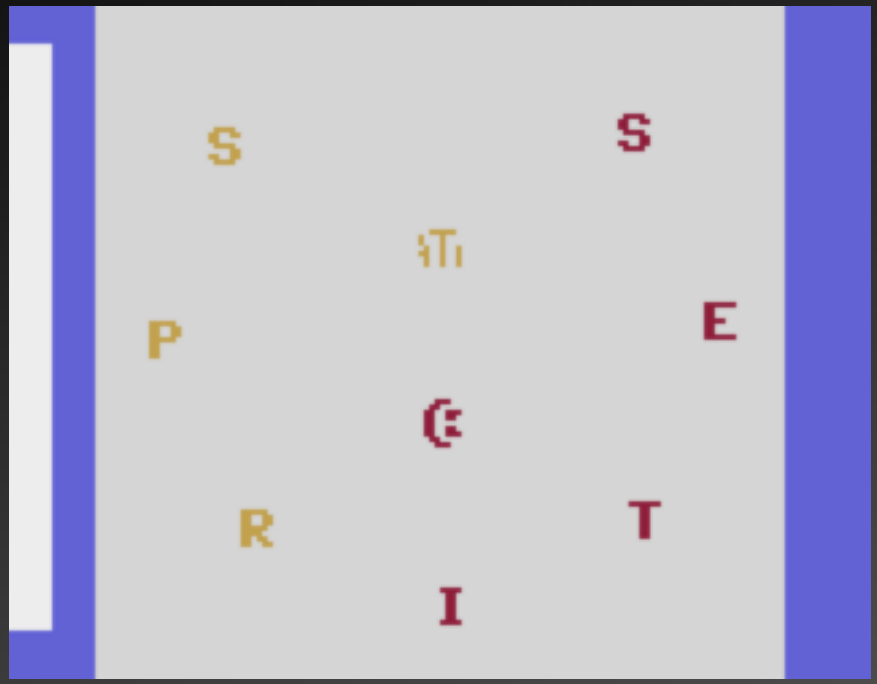
Chapter 6-2: C64 Open Border





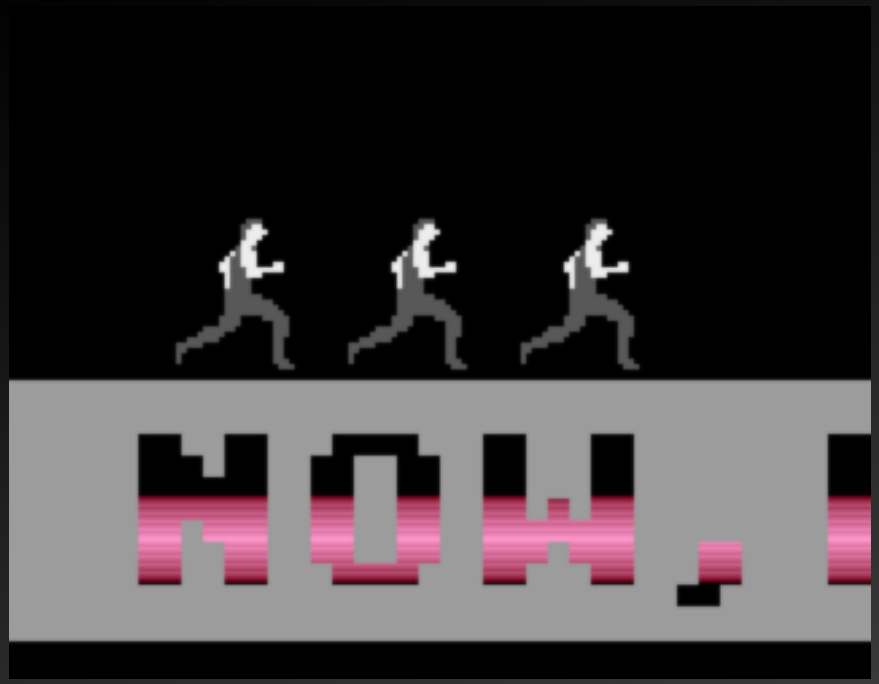
Reusing XAYAX down-logo, again. This time for three registers: 8 bits of playfield and both player sprites, using mirror effects of the VCS.

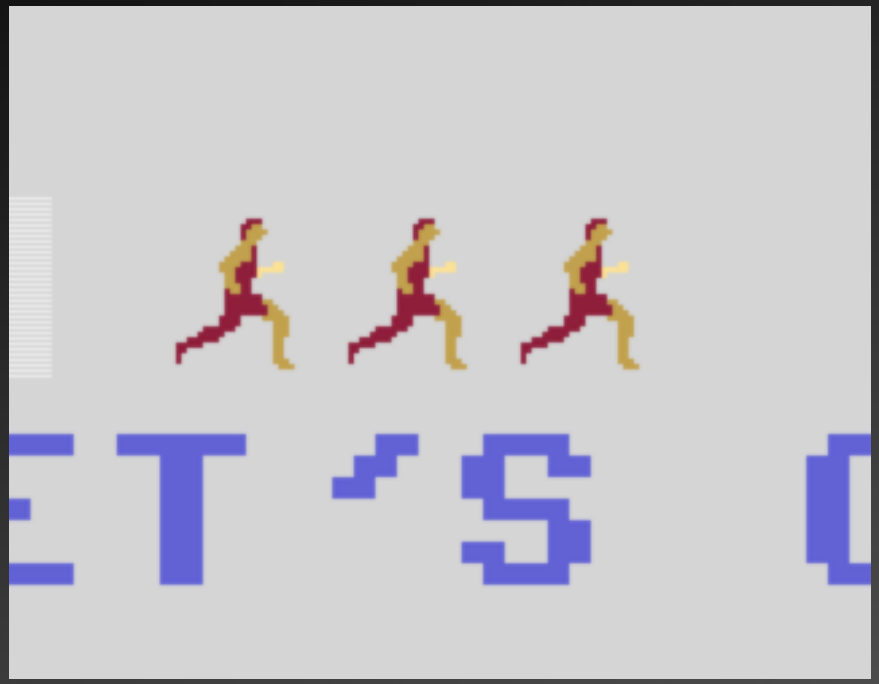
Chapter 6-3: C64 Multiplex



The technically most challenging part of the demo: multiplexing sprites. It took five attempts in three weeks to get this one running. Concession that had to be made: only center 104 pixels can be used.

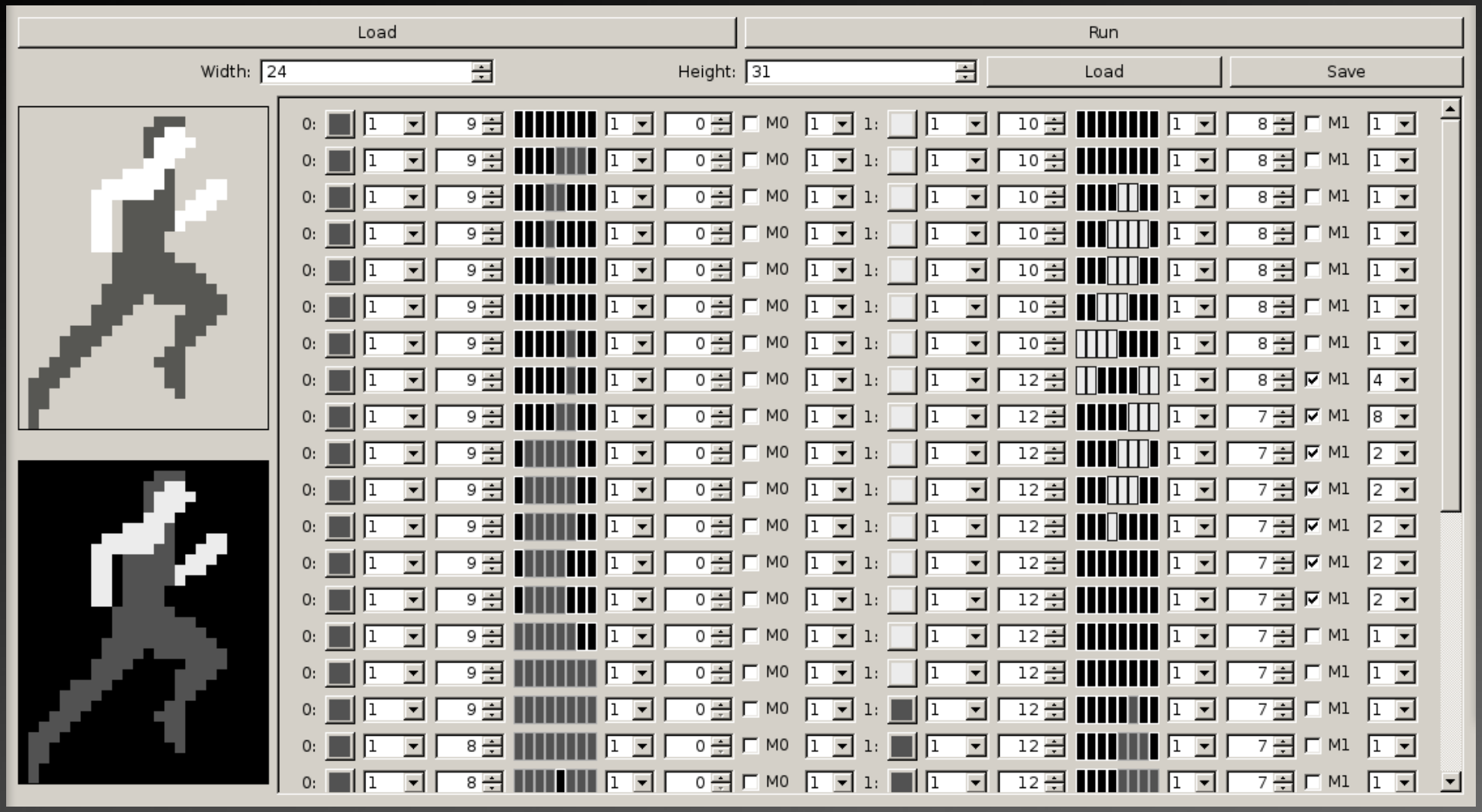
Chapter 7-1: Impossbile Mission Men





Something to hold against the Armalyte sprites of aTaRSI. Big sprites were created by moving and changing sprites line by line (→ need a tool). Three men are shown due to hardware sprite replication.

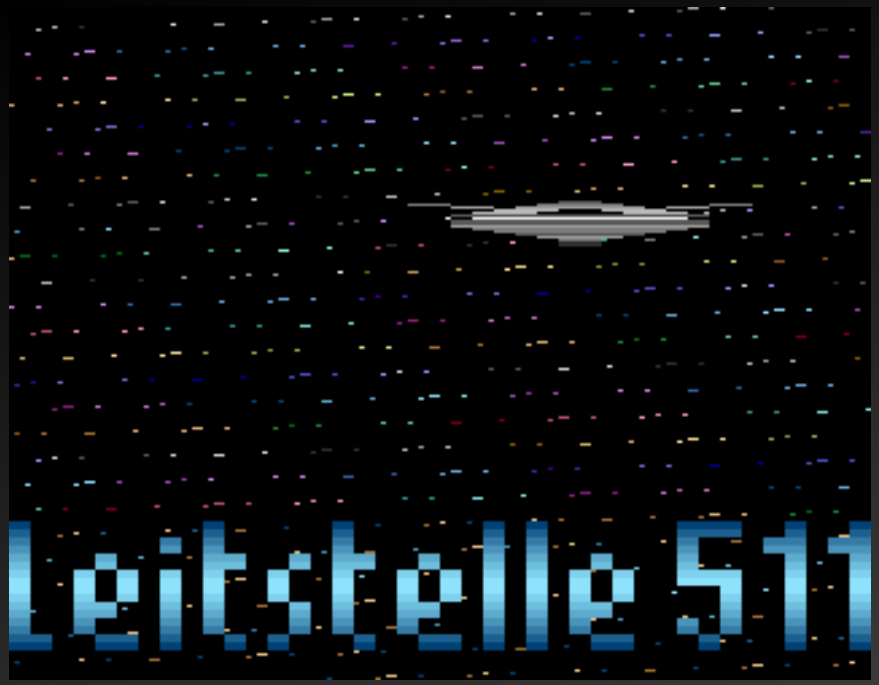
Tool #2, "Graphics Editor"
VCS_GFX_Codegen



Chapter 7-2: Orion Boarding

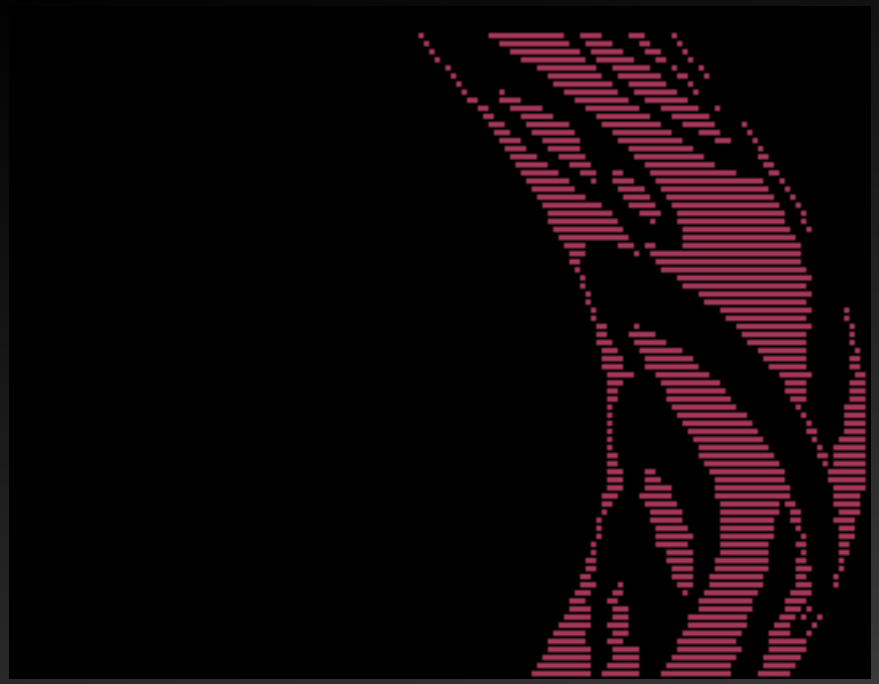
Impossible Mission Men were downsized using a graphics program, post-pixeled by hand. Resembling the starting scene of "Raumpatroullie Orion", a b/w sci-fi TV show.

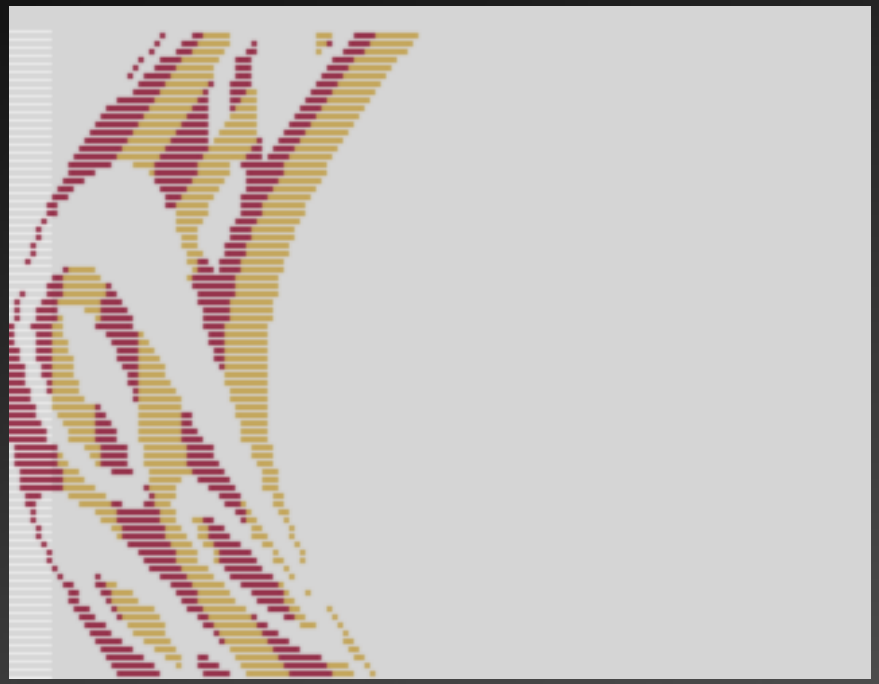
Chapter 7-3: Orion Greetings



Exploiting a hardware bug, first used in Cosmic Ark, using only one sprite. I was proud doing three sprites, then came Thomas Jentzsch in a post of AtariAge using all five. SC-RAM used for "striping" logos in and out.

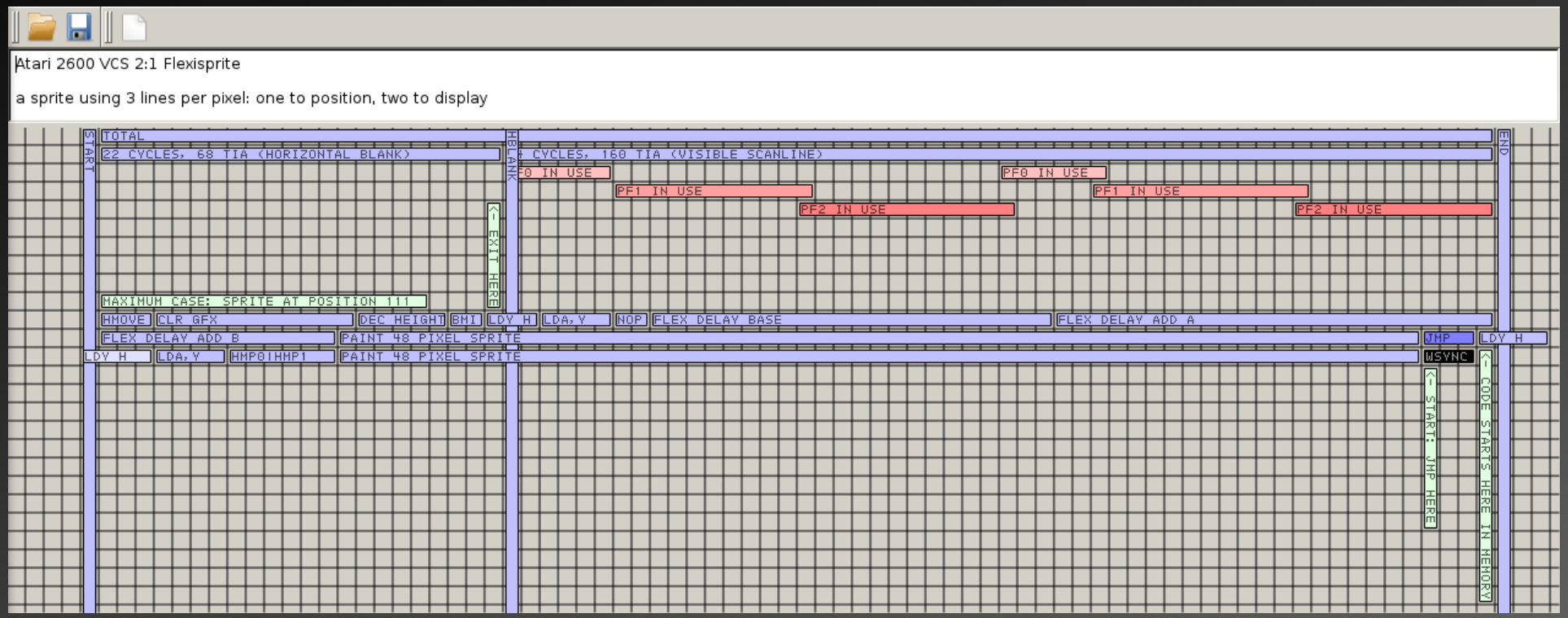
Chapter 8: Snake



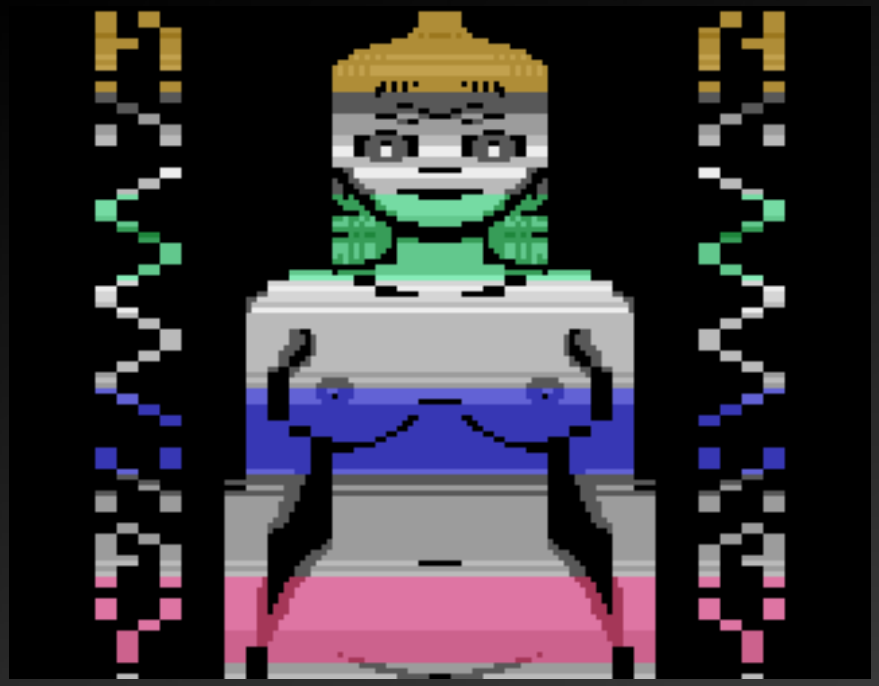


This part shaped itself up over a longer period of time. The question was: how fast can a 48 pixel sprite be repositioned? (→ tool needed) SC-RAM used, because 2x80 bytes are needed to store movement data between lines.

Tool #3, "Rasterpaper": fridgegrid

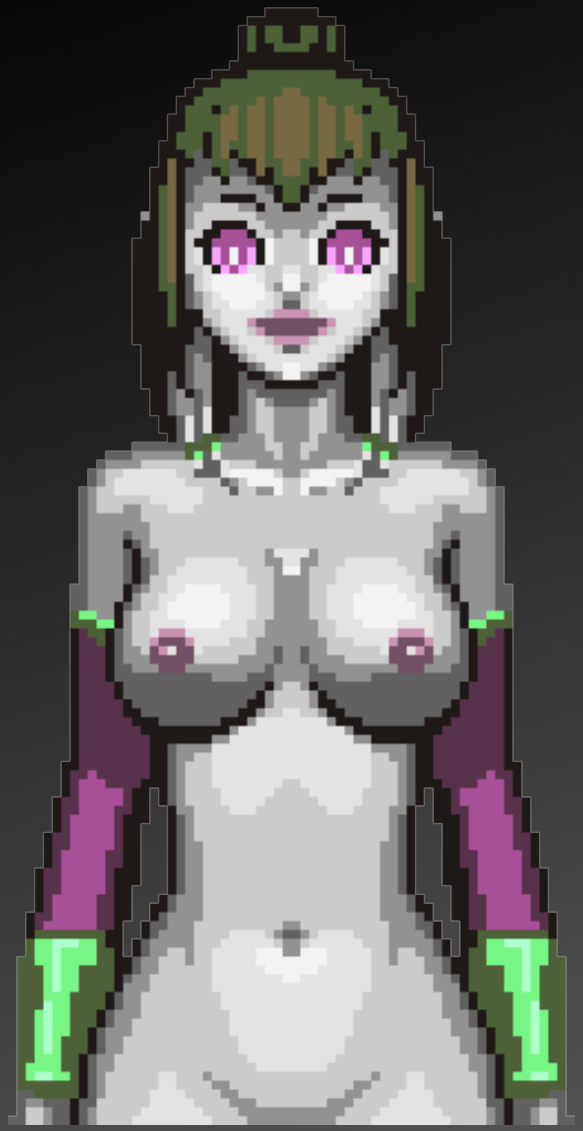


Chapter 9: The Girl, "Donna"



Use playfield graphics, "enhanced" with sprites here and there. Code and graphics data were created hand in hand. Mirrored image in mandatory. Moving colors were pre-calculate and stored in SC-RAM.

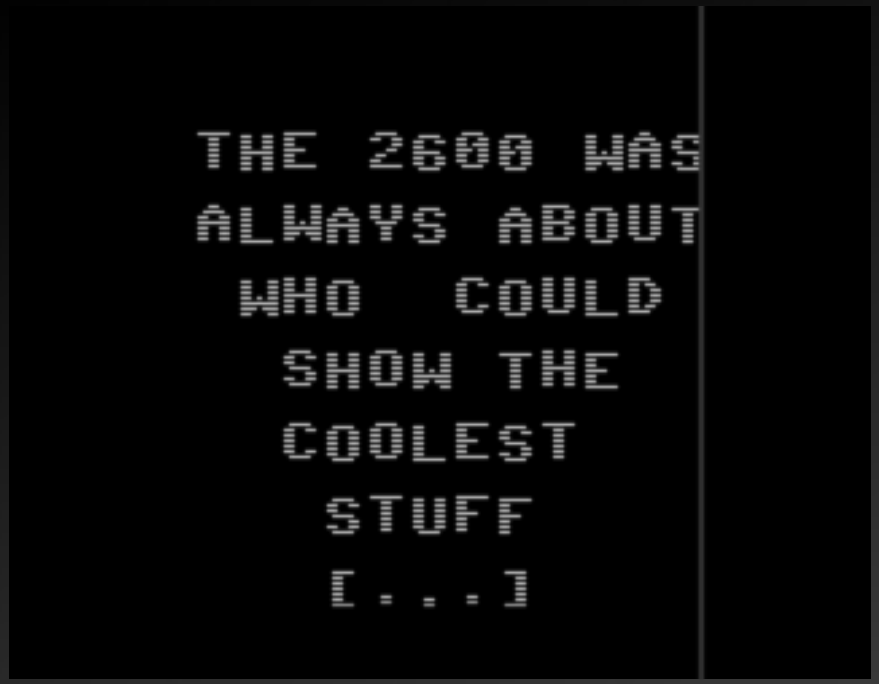
The First Draft And The Last Stage

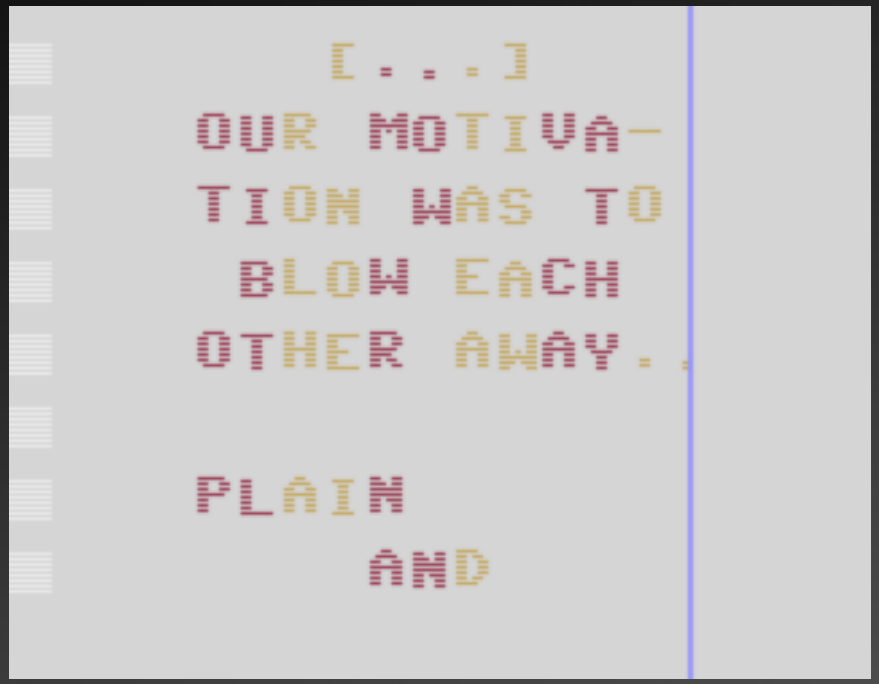


Some pixels are encoded colors or sprite movement, on the right side there is the sprite data.

Deft, I'm sorry!

Outro 1: Film Outro





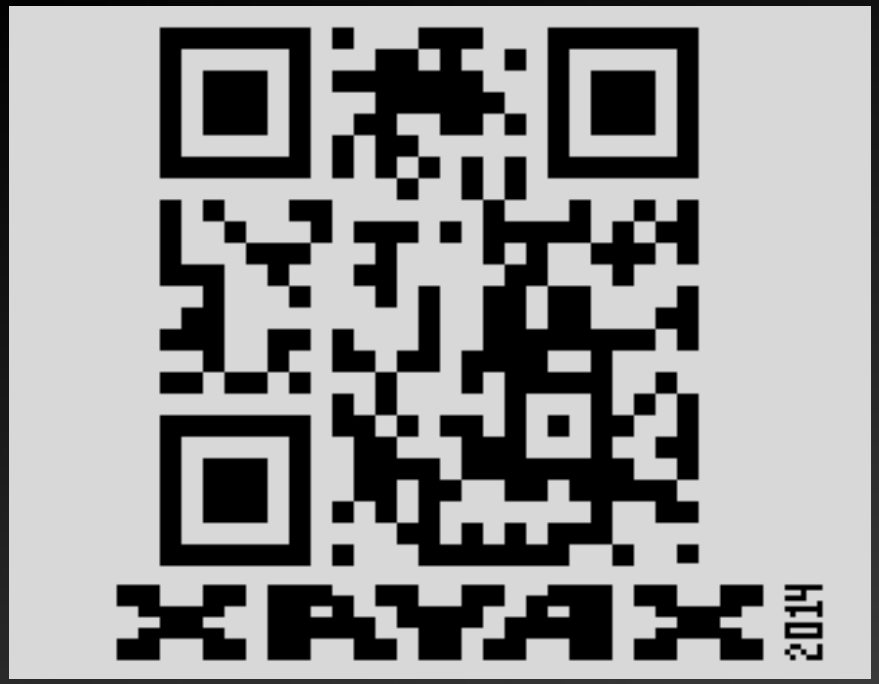
Typical 96 pixel sprite. Lens-fuzz is interesting: moving 8 pixel to left also moves ball sprite, has to be counter-acted with setting sprite movement.

Outro 2: End Of Film Gag

Using 32 pixel sprite for "searching" text, playfield color changed right on time for text color.

Sprite priority changed in the center.

Outro 3: QR Code



Two routines for diplaying playfield data. QR code asymmetrical, XAYAX is mirrored. 2014 is a sprite.

Thanks For Listening

Also thanks to:

Skyrunner, Deft, Veto, Titus
The folks at AtariAge.com
Michael Steil
Ninja / The Dreams

Questions?

Meet me at AtariAge.com,
#vcsdev or at a demo-party.
http://xayax.net/