

# psychedelia syndrome

the pixels and code  
of a new kind  
of videogame

**rob hogan**



**psychedelia**  
**syndrome**

For my beloved wife Edna.  
Thank you for your love and understanding.

© Rob Hogan 2025, All Rights Reserved.

**Edition Date:** Sunday 25<sup>th</sup> May, 2025

This work is licensed under a Creative Commons  
"Attribution-NonCommercial-ShareAlike 3.0 Unported"  
license.



# Contents

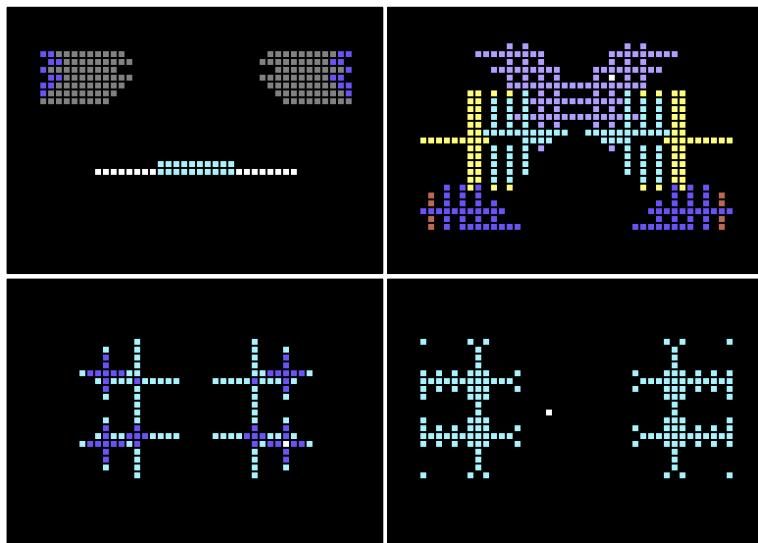
<b>1 we hope you like typing</b>	<b>11</b>
<b>2 the soul of a small light machine</b>	<b>21</b>
<b>3 let's pretend we can read code</b>	<b>37</b>
<b>4 increasing the dose</b>	<b>75</b>
<b>5 all the pretty patterns</b>	<b>99</b>
<b>6 fearful symmetries</b>	<b>125</b>
<b>7 beatific bursts</b>	<b>139</b>
<b>8 sensitive sequencer</b>	<b>157</b>
<b>9 particular presets</b>	<b>169</b>
<b>10 dials buttons switches</b>	<b>191</b>
<b>11 after effects</b>	<b>221</b>

<b>12 developing a bit of a complex</b>	<b>241</b>
<b>13 painting pixels is so complicated</b>	<b>245</b>
<b>14 all the little lightforms</b>	<b>275</b>
<b>15 plentiful presets</b>	<b>293</b>
<b>16 colourflow</b>	<b>321</b>

# about this book

This is a book about a toy. A toy that was the first of its kind. A kind of toy that has never really caught on, though maybe its moment lies some way in the future.

A static picture does the toy no justice, but we will show a few anyway. If you do not know what *Psychedelia* is, they might at least give you a flavour of this game that is not a game.



*Psychedelia* is a kind of interactive pixel painter and was the first of its kind. The best description of what this toy is about is in the words of its creator, Jeff Minter, who at the time was a 25 year old programmer of eccentric video games for the Commodore 64:

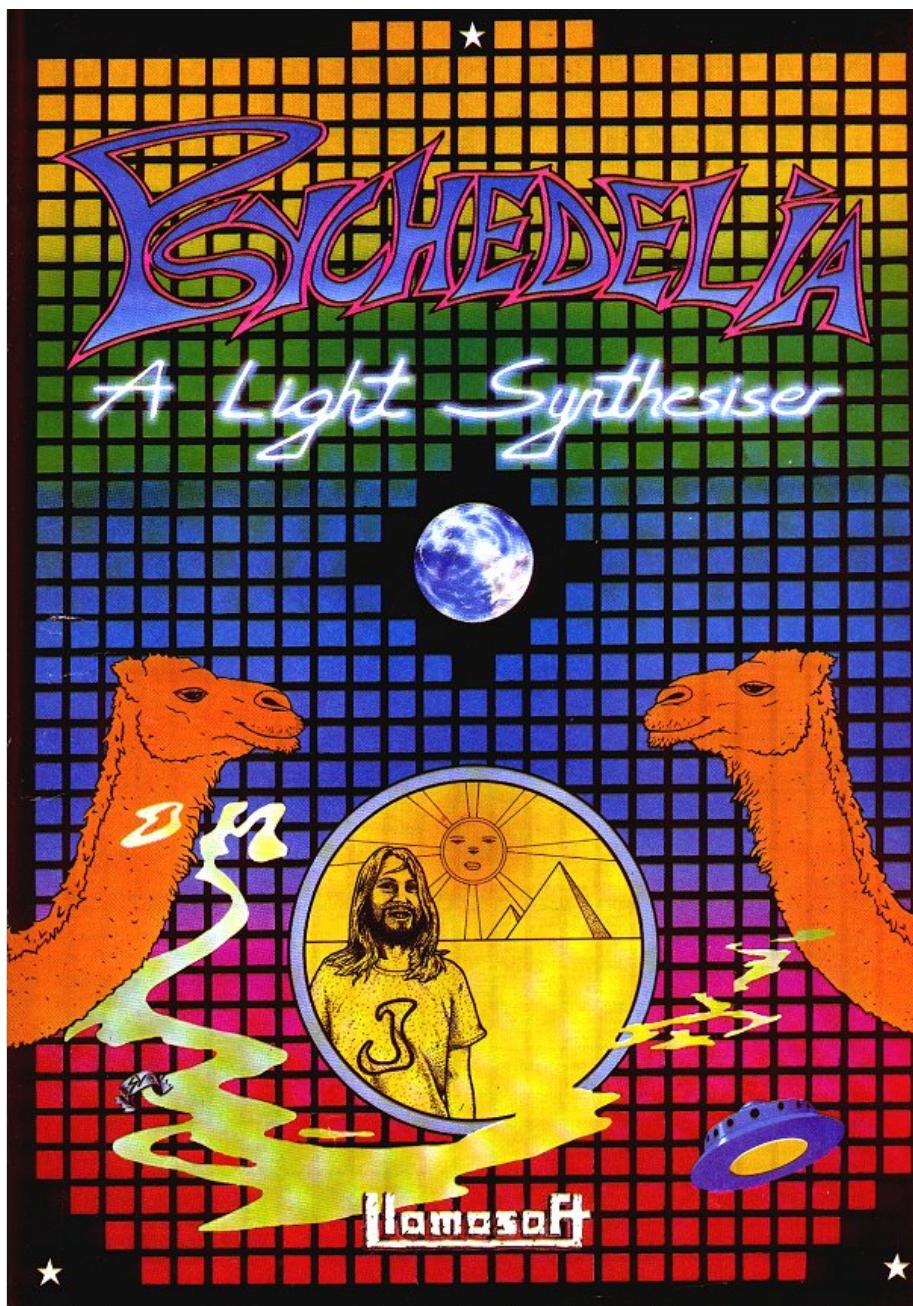


Figure 1: Cover art by Steinar Lund for the commercial edition of *Psychedelia*

### Jeffrey Says



It just felt wonderfully new, and somehow primal... it was like the patterns and mandalas that have fascinated humans for millennia, but come to life, under your control... in fact I was so moved by what I'd found that at first I refused to make it commercial... I felt that something so basic and lovely deserved more than just being another thing to be sold and profited from. I actually gave that first algorithm away in listing form to a computer magazine. But my parents argued successfully that there was no shame in making some money from it, and so in due course I created a somewhat expanded version with more patterns and control options, and that is what was released as *Psychedelia*.

I called the program a "light synthesiser", and advertised it as a new, non-competitive form of entertainment... no enemies, no killing, just light and colour.

This book is not about the history of this game. This book is an exploration of the computer code that makes the game tick. We discuss the algorithm used to generate all these pretty pictures in detail and dissect the pictures themselves. We pick over the code in sometimes minute detail to try to figure out its workings.

We are going to take apart the toy to see how it works. The pieces are all over the floor and they seem to be made of strange-looking computer instructions, I hope we know what we're doing...

### note on the text

The version you are reading is still a few revisions away from a finished product. If you find the writing hard going or the attempts to explain things difficult to follow, by all means [leave me a note](#) and I will gratefully accept your complaint. In the meantime, please skip over any blemishes to the next pretty picture or promising-looking block of text.

The full source code is available in [its own Github repository](#). You should find that it matches exactly the snippets of code provided in the book, though in some cases the extracts in the book have been edited and reformatted for brevity.

Rob Hogan (@mwenge)  
Dublin 2025



*psychedelia*  
**syndrome**



# **we hope you like typing**

Gather round children and let me tell you about a time when kids not unlike you were so determined to see colors on a screen that they were prepared to spend hours punching numbers into a computer terminal with only the very slim prospect that their efforts would not be utterly in vain.

This strange phenomenon was known as a 'listing'. Print magazines of the 1980s would run long lists of computer code for their apparently willing, and presumably adolescent, customers to type in to their home computers on the promise of hours of fun. In an era that preceded widespread, practically-free forms of distribution such as the internet and where purchasing a game on cassette or disk could run to as much as 60 pounds sterling, this was an inexpensive way to obtain something resembling a computer game at a bargain price - paying for the pleasure only at the expense of free time and patience.

Prior to the release of Psychedelia in the Christmas of 1984, Jeff Minter made a heavily cut-down version of the game available as just such a type-in listing in an English title by the name of 'Popular Computing Weekly'. This magazine was not prestigious. In the manner of the time it consisted mostly of ads and poorly printed black and white copy. In between the tightly-printed classified notices for dubious-sounding software packages available for mail order from Post Office boxes in places such as Slough and Dudley, there were brief game reviews ('BORING') and coverage of the bewildering array of overpriced 8-bit breadboards that flooded the home computer market in the 1980s (something called the JVC HC-7 that looks like a defective point-of-sale terminal was available for the modern equivalent of £1600). The full issue is worth a read in its own right, if only to travel back to much less innocent time when paying customers were openly preyed upon by unscrupulous mail-order business and fly-by-night game companies advertising products that could never live up to their colourful promise.

## Commodore 64

### Light Fantastic

This week a sneak preview of Lamasoft's latest program — Psychedelia — written and conceived by Jeff Minter

**J**eff Minter's new program *Psychedelia* is going to come as something of a surprise to those expecting another last-blitz action game. In an industry that uses phrases such as 'innovative', 'unique', and 'totally original', with the abandon usually reserved for confetti at weddings, he has produced something that actually is a little different.

Jeff calls it a Light Synthesiser, and that is a fair enough description. If you've ever been to a laser light show you may find it easier to visualise if you imagine yourself at the controls of one of those machines. Listen

to your favourite track on your stereo while you create a kaleidoscope of swirling and changing colours on your TV. This roughly speaking is what *Psychedelia* is all about.

The concept is quite simple. You control an on-screen cursor with a joystick. On pressing Fire, a particular shape is plotted, 'decaying' eventually back to black, through different colours with time. If the joystick is moved while the fire button is held down, the shape is continually replotted, leaving a trail of changing colour.

It is difficult to do the program justice in words, so this week we present a short

version of Jeff's original idea that sparked the whole thing off.

We emphasise, this is just a taster of the main program. *Psychedelia* itself is considerably more complex with many pre-set shapes. It allows you to create your own. You can change the colour decay sequence of the 'trail', as well as its band width and pulse length. Stored sequences can be recalled on-screen and used as a background for more real-time creations.

However, the full-blown *Psychedelia* program will not be available (versions for the C64, C16 and Vic 20) from Lamasoft until December, price £7.50 for the 64, £6.00 for the others.

Until then, type in the Basic listing (paying special note to Lines 30 and 40 before you start), plug in the joystick, turn up your hi-fi and play that TV ...

```

10 REM *** A JEFF MINTER PRODUCTION ***
20 REM *** BASIC PROG. BY KEVIN BERGIN
***  

30 REM *** ENTER THE FOLLOWING IN DIREC  
T MODE BEFORE YOU START!! ***
40 REM *** POKE 43,1:POKE 44,64:POKE 16  
384,0:CLR:NEW ***
50 PRINT"** HANG ON A SEC...."  
60 POKE53281,0:POKE53280,0:FORA=2049TO3  
457
70 READX:J=J+X:POKEA,X
80 NEXTX:IFJ<>98035THEN180
90 PRINT"** WONDERFUL COLOURS COMING..."  
100 PRINT"** BUT FIRST HAVE YOU SAVED T  
HIS ?"  
110 GETA$:IFA$<>"Y"ANDA$<>"N"THEN110
120 IFA$="N"THENPRINT"** SAVE IT NOW TH  
EN":STOP
130 PRINT"** TO SAVE THE CODE":PRINT"**  
PRESS RUN/STOP & RESTORE"
140 PRINT"** THEN SAVE AS NORMAL (THAT  
IS AFTER YOU EXIT THIS PROG
150 POKE198,0:PRINT"** PRESS A KEY TO S  
TART.":WAIT198,1
160 PRINT"** CLR:NEW":PRINT"** P43,1:P  
44,8":PRINT"** RUN":CHR$(19)
170 POKE631,13:POKE632,13:POKE633,13:PO  
KE198,3:END
180 PRINT"** DATA ERROR TRY AGAIN!":STO  
P
190 DATA11,8,10,0,158,50,48
200 DATA54,52,0,0,0,0,0
210 DATA0,169,64,133,252,169,8
220 DATA133,254,169,0,133,251,169
230 DATA53,133,253,160,0,162,6
240 DATA177,253,145,251,136,208,249
250 DATA230,252,230,254,202,208,242
260 DATA76,0,64,169,0,141,32
270 DATA208,141,53,208,169,216,133
280 DATA252,169,0,133,251,162,0
290 DATA165,252,157,96,3,165,251
300 DATA157,64,3,24,105,40,133
310 DATA251,165,252,105,0,133,252
320 DATA232,224,25,208,230,32,40
330 DATA69,76,238,66,162,0,169
340 DATA207,157,0,4,157,0,5
350 DATA157,0,6,157,0,7,169
360 DATA0,157,0,216,157,0,217
370 DATA157,0,218,157,0,219,202
380 DATA208,225,96,0,6,2,4
390 DATAS,3,7,1,166,3,189
400 DATA64,3,133,5,189,96,3
410 DATA133,6,164,2,96,165,2
420 DATA441,128,208,249,165,2,201
430 DATA40,16,243,165,3,41,128
440 DATA208,237,165,3,201,24,16
450 DATA231,32,92,64,177,5,41
460 DATA7,162,0,221,84,64,240
470 DATA5,232,224,9,268,246,138
480 DATA133,253,166,4,232,226,253
490 DATA240,3,16,1,96,166,4
500 DATA189,84,64,145,5,96,32
510 DATA86,65,160,0,165,4,201
520 DATA7,208,1,96,169,7,141
530 DATA76,65,165,2,133,8,165
540 DATA3,133,9,165,8,24,121
550 DATA254,64,133,2,165,9,24
560 DATA121,37,65,133,3,152,72
570 DATA32,86,65,104,168,200,185
580 DATA254,64,201,85,208,225,206
590 DATA76,65,173,76,65,197,4
600 DATA240,8,201,1,240,4,200
610 DATA76,196,64,165,8,133,2
620 DATA165,9,133,3,96,0,1
630 DATA1,1,0,255,255,255,0,5
640 DATA2,0,254,85,0,3
650 DATA0,253,85,0,4,0,252
660 DATA85,255,1,5,5,1,255
670 DATA251,251,85,0,7,0,249
680 DATA85,85,255,255,0,1,1
690 DATA0,0,255,85,254,0,2
700 DATA0,85,253,0,3,0,85
710 DATA252,0,4,0,85,251,251
720 DATA225,1,5,5,1,255,85
730 DATA249,0,7,0,85,85,1
740 DATA189,153,225,238,78,65,96
750 DATA0,0,165,2,72,165,3
760 DATA72,32,107,64,173,149,65
770 DATA208,8,184,133,3,104,133
780 DATA2,96,96,169,40,56,229
790 DATA2,133,2,32,107,64,173
800 DATA149,65,201,1,240,231,169
810 DATA24,56,229,3,133,3,32
820 DATA107,64,104,168,104,133,2
830 DATA152,72,32,107,64,104,133
840 DATA3,96,1,15,14,13,12
850 DATA11,10,9,4,5,6,7
860 DATA8,9,10,11,12,13,14
870 DATA15,16,17,18,19,20,21
880 DATA22,23,20,19,18,17,16
890 DATA0,0,0,0,0,0,0,0
900 DATA0,0,0,0,0,0,0,0

```

Figure 1.1: You are expected to type all of this.

But this issue contained a saving grace that was guaranteed to have issues flying from the shelves ('Only 40p'). The cover needed only two words to communicate the delights awaiting: 'MINTER INSIDE'. On page 32 there was nothing less than a 'Jeff Minter special - preview of *Psychedelia*, Llamasoft's new release'. 1984 had been another busy year for Minter, he had completed two fairly large, but only moderately successful, games in 'Sheep in Space' and 'Ancipital', as well as a clatter of lesser works. During the development of another game, 'Mama Llama', he was struck by the inspiration for something completely novel. This was not a game but 'some kind of light synthesizer'.

**Jeffrey Says**



"I'd long imagined some kind of light synthesiser I'd like to see at a gig or a party, something with which you could visualise the very feel of a good piece of music. I didn't imagine I could ever put such a concept successfully onto a home micro, but one Sunday I got back from running and decided to have a little tinker on my '64. An algorithm was in my head, it came from God knows where, it was just there, complete. Maybe my subconscious made it. Maybe my brain was leaking 3 months into the future. I coded it up, assembled, SYSED in, picked up the joystick and blew my brains out."

The rest of the game was completed in a two-week blitz in September 1984, and Minter immediately converted it to the VIC20 and C16 while he was at it. The listing in 'Popular Computing Weekly' was born of an eagerness to get the end result into people's hands. But in doing so, Minter was handing over his work for a fairly modest consideration. 'Popular Computing Weekly' would have paid Jeff Minter a fee, probably in the range of £50-£100 (the customers meanwhile paid with their wits), but this hardly seems a fair exchange for a work that was the first of its kind. '*Psychedelia*' after all was the first time anyone had written an interactive music visualizer for any kind of home computer. The only real predecessor of any kind was 'Atari Video Music' released in 1977. This was a physical appliance you plugged into your hi-fi and your TV and which would display programmed visualizations with the ability to alter the display through physical switches and dials on the console itself.

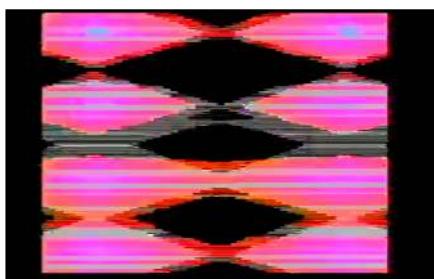


Figure 1.2: Atari Video Music in action.

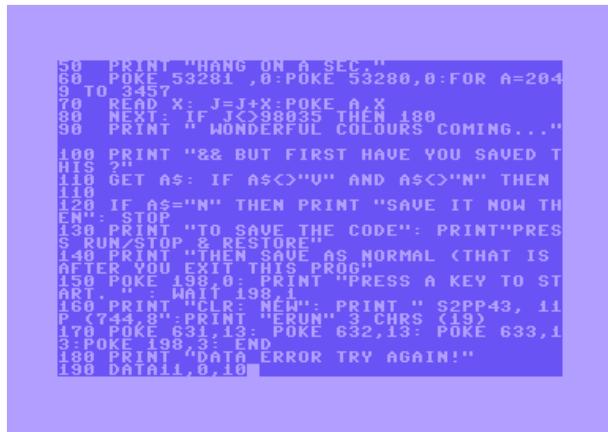
## Commodore 64

```
910 DATA0,0,0,0,0,0,0  
920 DATA0,0,0,0,0,0,0  
930 DATA0,0,0,0,12,13,14  
940 DATA15,15,15,14,4,4,4  
950 DATA4,4,4,4,4,5,6  
960 DATA7,8,9,10,11,12,13  
970 DATA13,13,13,7,8,9,10  
980 DATA11,0,0,0,0,0,0  
990 DATA0,0,0,0,0,0,0  
1000 DATA0,0,0,0,0,0,0  
1010 DATA0,0,0,0,0,0,0  
1020 DATA0,0,0,0,255,255  
1030 DATA255,255,255,255,255,255,255  
1040 DATA255,255,255,255,255,255,255  
1050 DATA255,255,255,255,255,255,255  
1060 DATA255,255,255,255,255,255,255  
1070 DATA255,255,0,0,0,0  
1080 DATA0,0,0,0,0,0,0  
1090 DATA0,0,0,0,0,0,0  
1100 DATA0,0,0,0,0,0,0  
1110 DATA0,0,0,0,0,0,12  
1120 DATA12,12,12,12,12,12,12  
1130 DATA12,12,12,12,12,12,12  
1140 DATA12,12,12,12,12,12,12  
1150 DATA12,12,12,12,12,12,12  
1160 DATA12,12,12,0,0,0,0  
1170 DATA0,0,0,0,0,0,0  
1180 DATA0,0,0,0,0,0,0  
1190 DATA0,0,0,0,0,0,0  
1200 DATA0,0,0,0,0,0,0  
1210 DATA4,7,1,2,3,6,7  
1220 DATA6,12,2,3,6,7,1  
1230 DATA2,2,4,4,7,1,2  
1240 DATA3,6,7,12,2,3,2  
1250 DATA3,7,1,2,0,0,0  
1260 DATA0,0,0,0,0,0,0  
1270 DATA0,0,0,0,0,0,0  
1280 DATA0,0,0,0,0,0,0  
1290 DATA0,0,0,0,0,0,0  
1300 DATA0,162,0,138,157,150,65  
1310 DATA157,214,65,157,22,66,157  
1320 DATA86,66,157,150,66,232,224  
1330 DATA64,208,236,96,32,214,66  
1340 DATA32,44,67,238,43,67,173  
1350 DATA43,67,45,53,68,141,43  
1360 DATA67,179,222,150,66,208,34  
1370 DATA189,86,66,157,150,66,189  
1380 DATA22,66,201,255,249,21,133  
1390 DATA4,189,156,65,133,2,189  
1400 DATA214,65,133,3,32,171,64  
1410 DATA174,43,67,222,22,66,76  
1420 DATA244,66,8,120,169,75,141  
1430 DATA20,3,169,67,141,21,3  
1440 DATA169,10,141,50,68,141,51  
1450 DATA68,169,1,141,21,208,141  
1460 DATA39,208,88,96,1,0,206  
1470 DATA73,67,240,3,76,49,234  
1480 DATA169,2,141,73,67,169,0  
1490 DATA133,12,32,42,68,173,0  
1500 DATA220,41,3,201,3,240,37  
1510 DATA201,2,240,6,238,51,68  
1520 DATA238,51,68,206,51,68,173  
1530 DATA51,68,201,255,208,8,169  
1540 DATA23,141,51,68,76,141,67  
1550 DATA201,24,208,5,169,0,141  
1560 DATA51,68,173,0,220,41,12  
1570 DATA201,12,240,37,201,8,240  
1580 DATA6,238,50,68,238,50,68  
1590 DATA206,50,68,173,50,68,201  
1600 DATA255,208,8,169,39,141,50  
1610 DATA68,76,187,67,201,40,208  
1620 DATA5,169,0,141,50,68,173  
1630 DATA0,220,41,16,240,8,169  
1640 DATA0,141,54,68,76,5,68  
1650 DATA173,55,68,240,8,173,54  
1660 DATA68,208,49,238,54,68,238  
1670 DATA52,68,173,52,68,45,53  
1680 DATA68,141,52,68,170,189,22  
1690 DATA66,201,255,208,26,173,50  
1700 DATA68,157,150,65,173,51,68  
1710 DATA157,214,65,169,7,157,22  
1720 DATA66,173,56,68,157,86,66  
1730 DATA157,150,66,32,25,68,177  
1740 DATA10,41,7,141,74,67,169  
1750 DATA1,133,12,32,42,68,76  
1760 DATA49,234,174,51,68,189,64  
1770 DATA3,133,10,189,96,3,133  
1780 DATA11,172,50,68,96,32,25  
1790 DATA68,165,12,145,10,96,30  
1800 DATA13,26,31,0,0,12,0  
1810 DATA0,0,0,0,0,0,0  
1820 DATA0,0,0,0,0,0,0  
1830 DATA0,0,0,0,0,0,0  
1840 DATA0,0,0,0,0,0,0  
1850 DATA0,0,0,0,0,0,0  
1860 DATA0,0,0,0,0,0,0  
1870 DATA0,0,0,0,255,0,0  
1880 DATA0,0,0,0,0,0,0  
1890 DATA0,0,0,0,0,0,0  
1900 DATA0,0,0,0,0,0,0  
1910 DATA0,0,0,0,0,0,0  
1920 DATA0,0,0,0,0,0,0  
1930 DATA0,0,0,0,0,0,0  
1940 DATA0,0,0,0,0,0,0  
1950 DATA0,0,0,0,0,0,0  
1960 DATA0,0,0,0,0,0,0  
1970 DATA0,0,0,0,0,0,0  
1980 DATA0,0,0,0,0,0,0  
1990 DATA0,0,0,0,0,0,0  
2000 DATA0,0,0,0,0,0,0  
2010 DATA0,0,0,0,0,0,0  
2020 DATA0,0,0,0,0,0,0  
2030 DATA0,0,0,0,0,0,0  
2040 DATA0,0,0,0,0,0,0  
2050 DATA0,0,0,0,0,0,0 2060 DATA0,0,0,0,0,0,  
2070 DATA0,0,0,0,0,0,0 2080 DATA0,0,0,0,0,0,  
2090 DATA0,0,16,19,25,3,8 2100 DATA5,4,5,12,  
2110 DATA46,46,1,32,6,15,18  
2120 DATA5,20,1,19,20,5,32  
2130 DATA2,25,32,10,5,6,6  
2140 DATA32,13,9,14,20,5,18  
2150 DATA32,50,64,162,40,189,255  
2160 DATA68,157,191,7,169,12,157  
2170 DATA191,219,282,208,242,96,0  
2180 DATA0,0,191,0,157,0,255  
2190 DATA0,255,0,255,0,255,0  
2200 DATA223,255,255,255,255,0,255
```

Figure 1.3: And this.

Realizing all this in software was something completely new and here was Minter giving it away more or less for free. Admittedly this was a much reduced version of the commercial product. In order to fit it into a mere two pages of arduous number typing Minter had removed nearly all the configurability that makes Psychedelia really compelling, but nevertheless the core of the game is in there, especially the fundamental algorithm that came to him while he was out for a jog that Sunday. Over the next few chapters we'll break down this humble listing and get to understand it in all its parts. It will be the ideal way to understand how Psychedelia works in principle. Once we have that we can look at all of the extras the final game came with and examine the workings of each individually.

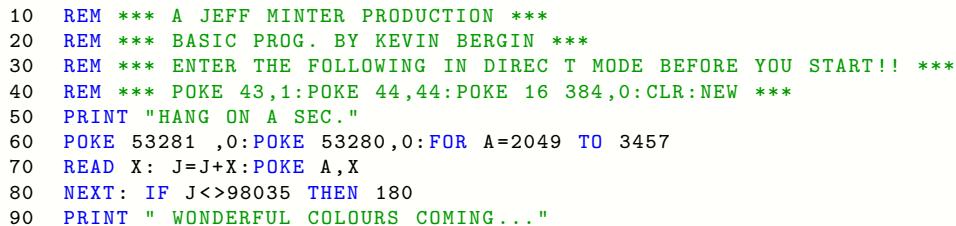
For now though, let's place ourselves in the position of a spotty English teenager settling down in the days before Christmas to get this thing up and running. Here we are, having fun:



```
50 PRINT "HANG ON A SEC."
60 POKE 53281 ,0:POKE 53280,0:FOR A=204
9 TO 3457
70 READ X: J=J+X:POKE A,X
80 NEXT: IF J<>98035 THEN 180
90 PRINT " WONDERFUL COLOURS COMING..."
100 PRINT "&& BUT FIRST HAVE YOU SAVED T
HIS ?"
110 GET A$: IF A$<>"V" AND A$<>"N" THEN
120 IF A$=="N" THEN PRINT "SAVE IT NOW TH
EN": STOP
130 PRINT "TO SAVE THE CODE": PRINT"PRES
S STOP & RESTORE AS NORMAL (THAT IS
AFTER YOU EXIT THIS PROG"
140 POKE 198,0: PRINT "PRESS A KEY TO ST
ART. ":" WAIT 198,1
150 PRINT "CLR: NEW": PRINT " S2PP43, 11
P(744,8":PRINT "ERUN" 3 CHR$ (19)
160 POKE 631,13: POKE 632,13: POKE 633,1
3:POKE 198,3: END
180 PRINT "DATA ERROR TRY AGAIN!"
190 DATA 1,0,10
```

Now the hell will start.

Above, we've got past the easy part - the almost-intelligible preamble that we can sort of understand. Like all type-in listings of its day this consisted of a small BASIC program that loads the raw machine code into memory. This isn't the juice of the program really.



```
10 REM *** A JEFF MINTER PRODUCTION ***
20 REM *** BASIC PROG. BY KEVIN BERGIN ***
30 REM *** ENTER THE FOLLOWING IN DIREC T MODE BEFORE YOU START!! ***
40 REM *** POKE 43,1:POKE 44,44:POKE 16 384,0:CLR:NEW ***
50 PRINT "HANG ON A SEC."
60 POKE 53281 ,0:POKE 53280,0:FOR A=2049 TO 3457
70 READ X: J=J+X:POKE A,X
80 NEXT: IF J<>98035 THEN 180
90 PRINT " WONDERFUL COLOURS COMING..."
```

```
100 PRINT "&& BUT FIRST HAVE YOU SAVED THIS ?"
110 GET A$: IF A$<>"V" AND A$<>"N" THEN 110
120 IF A$="N" THEN PRINT "SAVE IT NOW THEN": STOP
130 PRINT "TO SAVE THE CODE": PRINT"PRESS RUN/STOP & RESTORE"
140 PRINT "THEN SAVE AS NORMAL (THAT IS AFTER YOU EXIT THIS PROG"
150 POKE 198,0: PRINT "PRESS A KEY TO START. " : WAIT 198,1
160 PRINT "CLR: NEW": PRINT " S2PP43, 11P (744,8":PRINT "ERUN" 3 CHR$ (19)
170 POKE 631,13: POKE 632,13: POKE 633,13:POKE 198,3: END
180 PRINT "DATA ERROR TRY AGAIN!"
```

These lines perform the mundane task of reading in all of the numbers in the rest of the listing and, once done, executing those numbers as a machine code program. If you look closely you can see on line 80 there's a little sense-check in there too: all the numbers we type in from the DATA lines must add up to 98,035 or else we've made a mistake somewhere. If that happens we get told DATA ERROR TRY AGAIN!. As we will see a little later it was the grim fate of every child who attempted to type in this listing to hit this error and never recover from it unless they performed some inspired guesswork. This was a very cruel Christmas present.

But first lets understand what all these DATA lines are all about. They are a long string of decimal numbers each representing a byte of machine code. Minter wrote the game in 6502 Assembly Language, something we will be seeing a lot of in future chapters, and using a program called an Assembler converted these language instructions into a sequence of values between 0 and 255. This sequence forms the program itself. When stored on disk or cassette it is usually termed a PRG file because that is the conventional suffix used for such files when naming them. In order to make it into a type-in-listing this binary file has been converted into a simple list of decimal numbers, each one between 0 and 255. If you ever complete typing in this listing, the numbers will get written to a location in the C64's memory and executed as machine code.

So what we're interested in is the assembly instructions that underlie that long list of numbers. Since 6502 Assembly Language is so low-level this is surprisingly easy to do.

To give you an idea of how we get from raw numbers to an assembly program in practice I'll show you how a little routine I call PaintPixel appears in the listing. Here it is as it appears in print:

```

400 DATA64,3,133,5,189,96,3
410 DATA133,6,164,2,96,165,2
420 DATA41,128,208,249,165,2,201
430 DATA40,16,243,165,3,41,128
440 DATA208,237,165,3,201,24,16
450 DATA231,32,92,64,177,5,41
460 DATA7,162,0,221,84,64,240
470 DATA5,232,224,8,208,246,138
480 DATA133,253,166,4,232,228,253
490 DATA240,3,16,1,96,166,4
500 DATA189,84,64,145,5,96,32
510 DATA86,65,160,0,165,4,201
520 DATA7,208,1,96,169,7,141
END DATA

```

Figure 1.4: These numbers contain a complete machine-code routine we've called PaintPixel.

Our PaintPixel routine starts towards the end of line 410:

```

410 DATA 165,2
420 DATA 41,128,208,249,165,2,201
430 DATA 40,16,243,165,3,41,128
440 DATA 208,237,165,3,201,24,16
450 DATA 231,32,92,64,177,5,41
460 DATA 7,162,0,221,84,64,240
470 DATA 5,232,224,8,208,246,138
480 DATA 133,253,166,4,232,228,253
490 DATA 240,3,16,1,96,166,4
500 DATA 189,84,64,145,5,96

```

Now let's look at the four stages we go through to transform this list of decimal numbers back into the original assembly instructions. If you read left to right below, you can see the decimal values given in the listing translated to hexadecimal, then to the meaning of those hexadecimal values in 6502 assembly language.

Decimal Data	Hex	6502 Assembly	Pretty Assembly with labels
<hr/>			
165 2	A5 02	LDA \$02	LDA pixelXPosition
41 128	29 80	AND #\$80	AND #\$80
208 249	D0 F9	BNE \$089F	BNE ReturnEarly
165 2	A5 02	LDA \$02	LDA pixelXPosition
201 40	C9 28	CMP #\$28	CMP #NUM_COLS
16 243	10 F3	BPL \$089F	BPL ReturnEarly
165 3	A5 03	LDA \$03	LDA pixelYPosition
41 128	29 80	AND #\$80	AND #\$80
208 237	D0 ED	BNE \$089F	BNE ReturnEarly
165 3	A5 03	LDA \$03	LDA pixelYPosition
201 24	C9 18	CMP #\$18	CMP #NUM_ROWS
16 231	10 E7	BPL \$089F	BPL ReturnEarly
32 145 8	20 91 08	JSR \$0891	JSR LoadXAndYPosition
177 5	B1 05	LDA (\$05),Y	LDA (currentLineColorRamLoPtr),Y
41 7	29 07	AND #\$07	AND #COLOR_MAX
162 0	A2 00	LDX #\$00	LDX #\$00

Decimal Data	Hex	Assembly	Pretty Assembly with labels
<hr/>			
221 137 8	DD 89 08	CMP \$0889,X	CheckPresetsLoop CMP presetColorValuesArray,X
240 5	F0 05	BEQ \$08CB	BEQ MaybePaintPixel
232	E8	INX	INX
224 8	E0 08	CPX #\$08	CPX #COLOR_MAX + 1
208 246	D0 F6	BNE \$08C1	BNE CheckPresetsLoop
<hr/>			
138	8A	TXA	MaybePaintPixel TXA
133 253	85 FD	STA \$FD	TXA STA indexOfCurrentColor
166 4	A6 04	LDX \$04	STA indexOfCurrentColor LDX colorIndexForCurrentPixel
232	E8	INX	INX
228 253	E4 FD	CPX \$FD	CPX indexOfCurrentColor
240 3	F0 03	BEQ \$08D8	BEQ ActuallyPaintPixel
16 1	10 01	BPL \$08D8	BPL ActuallyPaintPixel
96	60	RTS	RTS
<hr/>			
166 4	A6 04	LDX \$04	ActuallyPaintPixel LDX colorIndexForCurrentPixel
189 137 8	BD 89 08	LDA \$0889,X	LDX colorIndexForCurrentPixel LDA presetColorValuesArray,X
145 5	91 05	STA (\$05),Y	LDA presetColorValuesArray,X STA (currentLineColorRamLoPtr),Y
96	60	RTS	RTS

In the last column I've applied a final transformation where I invent names for each of the variables and labels of loops and subroutines. This final bit of information exists only as raw numbers and addresses in the machine code, so this last step is in a way pure invention on my part. When writing Psychedelia there's no way Jeff Minter would have used such verbose names. They would have taken up too much space on disk and too much real estate on the screen, not to mention that Camel-Casing was not possible on the computers of the day which tended to display everything exclusively in an upper-case font.

Comparing each column in the listing you quickly get a sense of what each decimal value maps to. For example, in the very first line 165 maps to \$A5, which maps to the LDA instruction. This instruction loads the value that follows it (\$02) into the Accumulator register, one of three little pigeon-holes the 6502 CPU has for storing values while doing its processing. Along with the X and Y registers, these little pigeon-holes are used for nearly all temporary storage and indexing by 6502 assembly. If you want to add anything up, compare one thing to another, or reference an array with an index number you are going to have to make use of one or more of these registers.

But these kinds of details are all before us. For now, we are still stuck in our bedroom or on the carpet before our living-room television typing in these bloody numbers. After an hour or two of careful dictation we think we might be done. We've typed everything in, but towards the end we noticed something funny:

```
2050 DATA0,0,0,0,0,0,0 2060 DATA0,0,0,0,0,0,  
2070 DATA0,0,0,0,0,0,0 2080 DATA0,0,0,0,0,0,  
2090 DATA0,0,16,17,25,3,8 2100 DATA5,4,5,12,  
2110 DATA46,46,1,32,6,15,18
```

Figure 1.5: No more room on the page so just stick them in there beside those others.

This does not look right. There are numbers missing from the end of those lines! Any-one who didn't notice this was doomed. There was no way of typing in what was on the page and passing the consistency check built into the little BASIC program at the top of the listing. However, since we did notice it we can apply a little enterprise and a little guesswork to retrieve the situation.

Here are the corrupted lines with an X indicating each of our missing bytes:

```
2050 DATA 0, 0, 0, 0, 0, 0, 0
2060 DATA 0, 0, 0, 0, 0, 0, X
2070 DATA 0, 0, 0, 0, 0, 0, 0
2080 DATA 0, 0, 0, 0, 0, 0, X
2090 DATA 0, 0, 16, 19, 25, 3, 8
2100 DATA 5, 4, 5, 12, X, X, X
2110 DATA 46, 46, 1, 32, 6, 15, 18
```

Lines 2060 and 2080 are easy to guess: both are almost certainly 0. Why on earth the listing contains such a long sequence of redundant zeroes is another matter.

This leaves us with three bytes to fill. Is it too much to expect anyone to notice the pattern in these bytes that allows us to fill the gaps, least of all a bewildered teenager on their living room carpet? Can you?

What if we do something as simple as assume each of these values refers to a letter in the alphabet? Doing so gives us:

16	19	25	3	8	5	4	5	12	X	X	X	46	46	1	32	6	15	18
P	S	Y	C	H	E	D	E	L	X	X	X	.	.	A	F	O	R	

Genius at work.

We are a genius. This must be text for the title screen or something. We can guess that the first two missing letters are I and A, completing the word PSYCHEDELIA. No prizes for that one. However the final missing letter could be either a space or a '.' (a full stop). Let's try a full stop so that our corrected listing reads:

```
2050 DATA 0, 0, 0, 0, 0, 0, 0
2060 DATA 0, 0, 0, 0, 0, 0, 0
2070 DATA 0, 0, 0, 0, 0, 0, 0
2080 DATA 0, 0, 0, 0, 0, 0, 0
2090 DATA 0, 0, 16, 19, 25, 3, 8
2100 DATA 5, 4, 5, 12, 12, 9, 46 ; We add '12, 9, 46'
2110 DATA 46, 46, 1, 32, 6, 15, 18
```

We type 'RUN'. And then:



Figure 1.6: The fruit of hours of pubescent toil.

It works. My God, it actually works.

The question this book asks is 'How does it work?' For that, we will need to pick this listing apart again and figure out its component parts. We will start 'in the middle', by looking at the core algorithm that powers the psychedelic displays. This is the algorithm that came to Minter on his Sunday run.

# the soul of a small light machine

## Jeffrey Says



It was a simple algorithm, just seeding patterns along a path; the patterns were to expand and change shape and colour over time. I got back from my run and coded up the algo – it fit in about 1K of 6502 assembler code. I ran the code and a white cursor appeared on the screen. I picked up the joystick, moved the cursor, and pressed down the fire button.

If you look at Psychedelia at work for even a short time it is possible to guess at the fundamentals of its operation. It takes the position of the cursor, paints a pattern with the cursor as its centre, and then cycles each pixel it has painted through a series of colors.

As we look some more we might notice that the color a pixel changes to depends on what color it has already. This becomes more noticeable as we move the cursor around. If we're paying very close attention we might notice that the number of available colors is fairly limited, perhaps as low as 7 or 8.

To improve our understanding any further we will need to look at the code. We might have to take this part slowly. Although the code is very compact it might be hard to disentangle. Scanning through the 800 or so lines of assembly we notice something relevant to our interests:

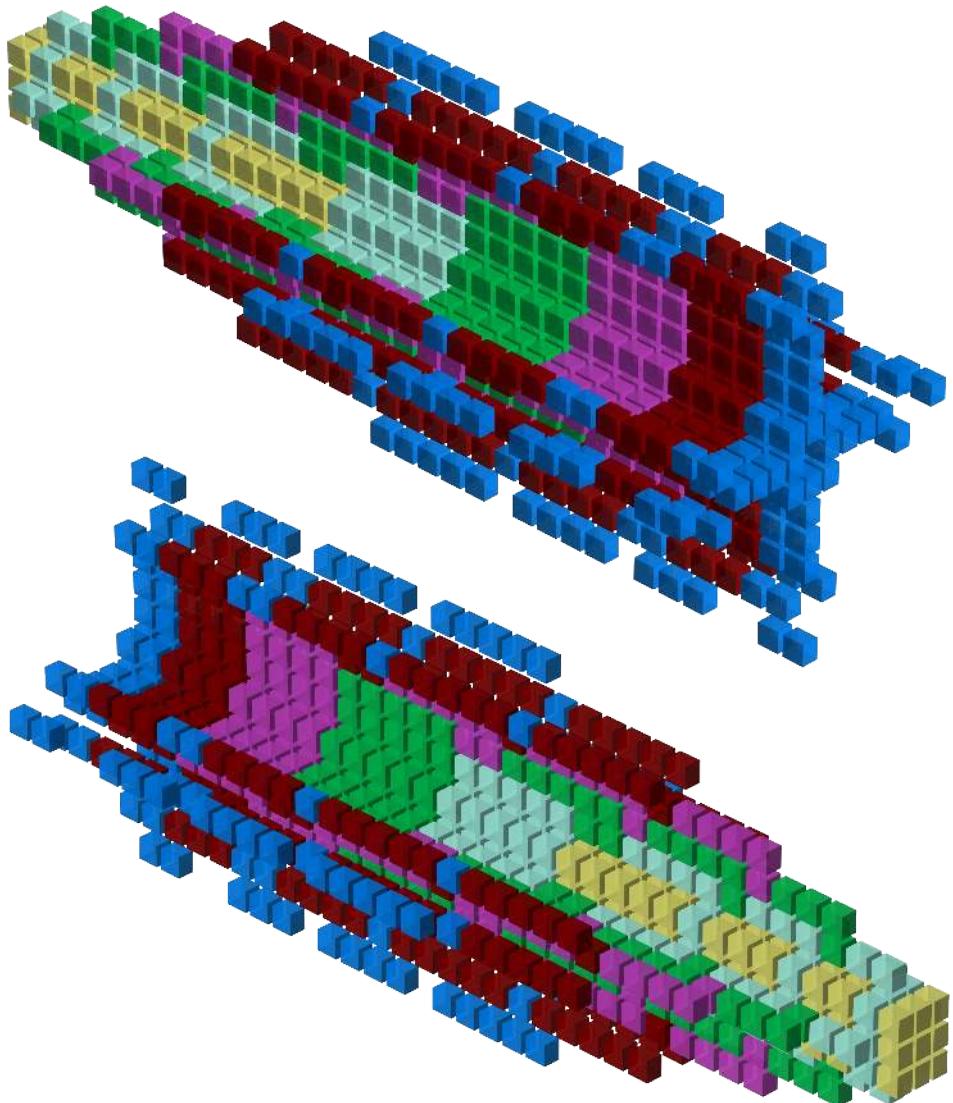


Figure 2.1: Evolution of the 'Star One' pattern.

```

starOneXPosArray
.BYTE $00,$01,$01,$01,$00,$FF,$FF,$FF,$55 ;      5
.BYTE $00,$02,$00,$FE,$55 ;                      ;
.BYTE $00,$03,$00,$FD,$55 ;                      ;      4 4
.BYTE $00,$04,$00,$FC,$55 ;                      ;      3
.BYTE $FF,$01,$05,$05,$01,$FF,$FB,$FB,$55 ;      2
.BYTE $00,$07,$00,$F9,$55 ;                      ;      1
.BYTE $55 ;                      ;      4 000 4
starOneYPosArray ; 5 3210 0123 5
.BYTE $FF,$FF,$00,$01,$01,$01,$00,$FF,$55 ;      4 000 4
.BYTE $FE,$00,$02,$00,$55 ;                      ;      1
.BYTE $FD,$00,$03,$00,$55 ;                      ;      2
.BYTE $FC,$00,$04,$00,$55 ;                      ;      3
.BYTE $FB,$FB,$FF,$01,$05,$05,$01,$FF,$55 ;      4 4
.BYTE $F9,$00,$07,$00,$55 ;                      ;
.BYTE $55 ;                      ;      5

```

Listing 2.1: Source code for the Star.

Somehow this data is encoding the pattern that Psychedelia is drawing. It looks to be a list of X and Y co-ordinates with the X co-ordinates stored in starOneXPosArray and the Y co-ordinates in starOneYPosArray. The only problem is they don't look much like co-ordinates. After puzzling for a little while I realized what's going on here. The very first pair of bytes is \$00 from starOneXPosArray and \$FF from starOneYPosArray. That can't mean that the X,Y position is (0,255), so what if \$FF means something else? What about -1? That would mean \$FE means -2 and so on. Now things start to make sense. If we treat 'C' as our origin the first line of each array actually contains a unit distance plot around the origin. We can represent it with 0s around the origin as follows:

000
0C0
000

Putting our rudimentary diagram side by side with our data arrays we have:

```

starOneXPosArray
.BYTE $00,$01,$01,$01,$00,$FF,$FF,$FF,$55 ;      000
starOneYPosArray ;          0C0
.BYTE $FF,$FF,$00,$01,$01,$01,$00,$FF,$55 ;      000

```

The \$55 at the end of each line looks like a kind of line-break splitting up the array, so we don't use it for co-ordinates. When we add in the second line from each array, plotting the positions with a 1 this time, we then get:

```
starOneXPosArray
    .BYTE $00,$01,$01,$01,$00,$FF,$FF,$FF,$55      ;
    .BYTE $00,$02,$00,$FE,$55                          ; 1
                                                    ; 000
                                                    ; 10 01
                                                    ; 000
starOneYPosArray
    .BYTE $FF,$FF,$00,$01,$01,$01,$00,$FF,$55      ; 1
    .BYTE $FE,$00,$02,$00,$55                          ;
```

Listing 2.2: Source code for the Star.

And so on, until we build up the picture we found in the initial code above. For some reason the very last 'line' in the array is empty, it just contains the delimiter value \$55:

```
starOneXPosArray
...
    .BYTE $55
starOneYPosArray
...
    .BYTE $55
```

It's as though it is just there to make up the numbers, giving us seven lines of data in total. Let's hold that thought for now.

So we kind of understand an important data structure used by the code. It seems designed to build the pattern incrementally, one pair of lines from each array at a time. This matches our intuition of how the painting ought to work given the way we see it operate in practice. The pattern isn't drawn all at once but accumulates on-screen in a way that matches what we're seeing in the way the pattern's co-ordinates are structured in starOneXPosArray and starOneYPosArray.

The next thing we might think of looking for is the way the colors are defined. We find them towards the beginning of the file:

```
presetColorValuesArray
    .BYTE BLACK,BLUE,RED,PURPLE,GREEN,CYAN,YELLOW,WHITE
```

There are eight colors defined in total, more or less as we suspected. This number is a strange limitation, even for a computer as old as the Commodore 64 which after all has a total of 16 available. If we look a little closer, the number is even fewer: the first one listed there is BLACK so it doesn't even count as a color. The screen is black already after all, unless we've painted it with something. Indeed, putting BLACK at the start like that suggests it is being used to clear a 'pixel' we've already painted. This leaves us with seven colours in total.

So after a brief inspection of just the data we seem to have a system that consists of

patterns with seven lines of data and a palette of seven colors. We would be right to suspect that this is not a coincidence. But we have to dive into the code to see why.

The best place to read code is at the beginning. This doesn't mean from the top of the file but from the place that controls everything else in the program. In nearly all programs this is the 'main loop', a usually short section of code that runs repeatedly in a loop and controls the operation of the rest of the program. As it happens, Pyschedelia has two of these!

The first is what we conventionally understand by a 'main loop': the program starts executing this and never stops. It just keeps going around in a circle executing the instructions over and over again. Here is a slightly chopped-down picture of Psychedelia's MainPaintLoop. We'll see the rest of it later on but this is enough to help you see how it just goes round-and-round:

```
MainPaintLoop
; Here and elsewhere '...' indicates we've removed some code for
; brevity.
...
; 'X' below is the current index value, the loop increments it at
; every iteration and ensures it is always a value between 0 and 31.
; We've cut out the bits that do all that for now to aid clarity.
...
; Load the value from the current index position in
; currentColorIndexArray and store it in colorIndexForCurrentPixel.
; The values in currentColorIndexArray array are always between 0
; and 7.
LDA currentColorIndexArray,X
...
STA colorIndexForCurrentPixel

; Get the X and Y position stored in the current position of each
; array.
LDA pixelXPositionArray,X
STA pixelXPosition
LDA pixelYPositionArray,X
STA pixelYPosition

; Where the magic happens.
JSR PaintStructureAtCursorPosition

; Decrement the value stored in currentColorIndexArray at the
; current position, for example from 7 to 6.
DEC currentColorIndexArray,X

JMP MainPaintLoop ; And back to the start again!
```

What this loop does at every turn is choose a position on the screen and paint the pattern around it. Basically the operation we intuited when we looked at the starOneXPosArray and starOneYPosArray data structures above. For this it has

a virtual list of 32 positions which it reads from. This list is made up of two actual lists (also known as arrays), one containing the X co-ordinate of the positions (`pixelXPositionArray`) and one containing the Y co-ordinates (`pixelYPositionArray`). It reads these from beginning to end and then back to the start again. There's another list it reads from at the same time, `currentColorIndexArray`. This is also 32 elements long and the values it contains are always between 0 and 7. The reason for this is that the values are an index into `presetColorValuesArray`. So what each value actually tells us is the color that was used as the starting point for painting colors the last time we painted a pattern using this position in the arrays. For example a value of 5 would indicate CYAN since counting from zero bring us to that element in the list.

```
presetColorValuesArray
;          0      1      2      3      4      5      6      7
.BYTE BLACK, BLUE, RED, PURPLE, GREEN, CYAN, YELLOW, WHITE
```

So we have three main arrays in use here. Two giving us a list of positions on the screen (`pixelXPositionArray`, `pixelYPositionArray`) and one giving us a list of colors (`currentColorIndexArray`). I've indicated in the listing above that the magic happens in `PaintStructureAtCursorPosition`. This is where the values retrieved from each of the three lists are put to magical use and colourful things are drawn on the screen.

```
pixelXPositionArray
.BYTE $0F,$0E,$0D,$0C,$0B,$0A,$09,$04
.BYTE $05,$06,$07,$08,$09,$0A,$0B,$0C
.BYTE $0D,$0E,$0F,$10,$11,$12,$13,$14
.BYTE $15,$16,$17,$14,$13,$12,$11,$10
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
pixelYPositionArray
.BYTE $0C,$0D,$0E,$0F,$0F,$0E,$04
.BYTE $04,$04,$04,$04,$04,$04,$04,$05
.BYTE $06,$07,$08,$09,$0A,$0B,$0C,$0D
.BYTE $0D,$0D,$0D,$07,$09,$09,$0A,$0B
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
currentColorIndexArray
.BYTE $FF,$FF,$FF,$FF,$FF,$FF,$FF,$FF
.BYTE $FF,$FF,$FF,$FF,$FF,$FF,$FF,$FF
.BYTE $FF,$FF,$FF,$FF,$FF,$FF,$FF,$FF
.BYTE $FF,$FF,$FF,$FF,$FF,$FF,$FF,$FF
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
```

Listing 2.3: The arrays have a larger capacity than is actually used: each one above is 64 bytes long.

Before we delve into how `PaintStructureAtCursorPosition` achieves this, we should answer a question that might be occurring to you: how are the three lists above populated? There's no sign of anything happening to them in `MainPaintLoop`. It just reads them over and over again forever. The answer to this lies in the program's second 'main loop' that we mentioned earlier. Strictly speaking it's not a loop at all but what is known as a 'hardware interrupt'. A hardware interrupt is when a computer wakes up periodically (usually many times every second!) and executes some code. This is useful because programmers can nominate some code for it to run each time and as a bonus specify exactly when the code should run, for example just after the screen has been painted or even at a specific point during the painting of the screen.

Psychedelia doesn't get too fancy in this regard. The code it nominates is specified to run once per screen paint. This code concerns itself with one thing: collecting the user's key presses, and most importantly from our point of view the key presses that move the cursor around the screen. This is the stuff that feeds our `pixelXPositionArray` and `pixelYPositionArray`.

```

MainInterruptHandler
...
; Increment our index.
INC indexIntoArrays
LDA indexIntoArrays
; Ensure indexIntoArrays loops back to zero once
; if reaches MAX_INDEX_VALUE (i.e. 31).
AND MAX_INDEX_VALUE
STA indexIntoArrays
...
; With an update counter stored in the 'A' register,
; move it to 'X' so we can use it below.
TAX
...
; Feed our three beasts!
LDA cursorXPosition
STA pixelXPositionArray,X
LDA cursorYPosition
STA pixelYPositionArray,X
LDA #COLOR_MAX           ; COLOR_MAX = 7
STA currentColorIndexArray,X

JMP RETURN_FROM_INTERRUPT

```

As you can see in the last few lines above we periodically feed the cursor position into `pixelXPositionArray` and `pixelYPositionArray`. We also feed a 7 (`COLOR_MAX`) into the `currentColorIndexArray`. So if we stay absolutely still and don't move the cursor at all, after a short while the three arrays will end up looking like this. With \$1E (30) as the X position in each slot and \$0D (13) as the Y position in each slot.

```
pixelXPositionArray
    .BYTE $1E,$1E,$1E,$1E,$1E,$1E,$1E
    .BYTE $1E,$1E,$1E,$1E,$1E,$1E,$1E
    .BYTE $1E,$1E,$1E,$1E,$1E,$1E,$1E
    .BYTE $1E,$1E,$1E,$1E,$1E,$1E,$1E
    ...
pixelYPositionArray
    .BYTE $0D,$0D,$0D,$0D,$0D,$0D,$0D
    .BYTE $0D,$0D,$0D,$0D,$0D,$0D,$0D
    .BYTE $0D,$0D,$0D,$0D,$0D,$0D,$0D
    .BYTE $0D,$0D,$0D,$0D,$0D,$0D,$0D
    ...
currentColorIndexArray
    .BYTE $07,$07,$07,$07,$07,$07,$07
    .BYTE $07,$07,$07,$07,$07,$07,$07
    .BYTE $07,$07,$07,$07,$07,$07,$07
    .BYTE $07,$07,$07,$07,$07,$07,$07
    ...
    ...
```

Armed with an understanding of the data that is feeding into the pixel generation of Psychedelia we can now move on to getting our heads around the code that implements the pixel generation itself. This starts in the routine that reads in the pattern's data structure, `PaintStructureAtCursorPosition`, called from our main loop every time we read in from our three arrays above.

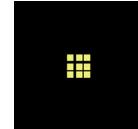
Remember that our data structure has seven lines in it, each delimited by a \$55 at its end. This routine will read and paint a number of these lines. The number is determined by the value we read from `currentColorIndexArray` and stored in `currentValueInColorIndexArray`. For example if we read 6 it will only paint the first line from each of `starOneXPosArray` and `starOneYPosArray` (2 lines in total):

```
.BYTE $00,$01,$01,$01,$00,$FF,$FF,$FF,$55
.Byte $FF,$FF,$00,$01,$01,$01,$00,$FF,$55
```



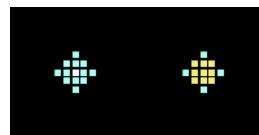
If we read 5 it will only paint the first 2 lines of each array (4 lines in total). Our first one again:

```
.BYTE $00,$01,$01,$01,$00,$FF,$FF,$FF,$55
.Byte $FF,$FF,$00,$01,$01,$01,$00,$FF,$55
```



Then:

```
.BYTE $00,$02,$00,$FE
.BYTE $FE,$00,$02,$00
```



We can see how this logic is implemented in this heavily abbreviated version of `PaintStructureAtCurrentPosition`.

```
PaintStructureAtCurrentPosition
..
LDA #NUM_ARRAYS ; NUM_ARRAYS = 7
STA countToMatchcurrentIndex

..
; Y starts out at zero here. We use it as an index
; for reading in starOneXPosArray and starOneYPosArray
; one byte at a time.
LineReadLoop
LDA initialPixelXPosition
CLC
ADC starOneXPosArray,Y
STA pixelXPosition

LDA initialPixelYPosition
CLC
ADC starOneYPosArray,Y
STA pixelYPosition
...
JSR PaintPixelForCurrentSymmetry
...
INY

; Loop if we haven't reached the end of the current
; line in starOneXPosArray
LDA starOneXPosArray,Y
CMP #$55
BNE LineReadLoop

; Check if we've reached the last line in the structure
; for this visit. This will always be one more than our
; last visit.
DEC countToMatchcurrentIndex
LDA countToMatchcurrentIndex
CMP currentValueInColorIndexArray
BEQ Return
CMP #$01
BEQ Return

; Move to the start of the next line in starOneXPosArray/
; starOneYPosArray.
INY
JMP LineReadLoop
```

```
Return  
...  
RTS
```

It initializes countToMatchCurrentIndex to 7:

```
LDA #NUM_ARRAYS ; NUM_ARRAYS = 7  
STA countToMatchcurrentIndex
```

and decrements this value after reading each line

```
DEC countToMatchcurrentIndex
```

until it matches the value in currentValueInColorIndexArray or we have reached the last line in the structure.

```
LDA countToMatchcurrentIndex  
CMP currentValueInColorIndexArray  
BEQ Return  
CMP #$01  
BEQ Return
```

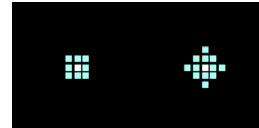
We can see how this works in practice with a few examples. In our very first visit to PaintStructureAtCursorPosition we have a value of \$06 in currentValueInColorIndexArray. So we paint one phase from starOneX/YPosArray:

currentColorIndexArray Bytes 26-32:  
.BYTE \$FF,\$FF,\$06,\$FF,\$FF,\$FF



The next time we visit currentValueInColorIndexArray is \$05, so we paint two phases:

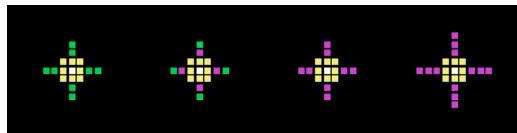
currentColorIndexArray Bytes 26-32:  
.BYTE \$FF,\$FF,\$05,\$07,\$FF,\$FF



Notice that the value of \$05 determines two things: that we paint two phases of the overall pattern and the color we paint them is CYAN (because 05 points to CYAN in presetColorValuesArray). We'll see how this is done shortly.

In yet another example for later on in the pattern's evolution we can see this:

currentColorIndexArray Bytes 26-32:  
.BYTE \$FF,\$FF,\$03,\$05,\$06,\$FF,\$FF



Here \$03 points to PURPLE in `presetColorValuesArray` so each phase is painted in purple. And again because `currentValueInColorIndexArray` is \$03 we paint four phases ( $7 - 3 = 4$ ). For some reason though, we didn't paint any purple in the first phase - notice the pattern has stayed yellow and green.

To understand why this is so we will have to look into how the actual painting is handled by `PaintPixelForCurrentSymmetry`. This is called for each pair of positions read in from `starOneXPosArray` and `starOneYPosArray` and stored in `pixelXPosition` and `pixelYPosition`. In its simplest use `PaintPixelForCurrentSymmetry` just ends up calling another routine `PaintPixel` directly. That's the use we'll look at here, so we can skip straight to `PaintPixel` instead.

As we saw above like `PaintStructureAtCursorPosition` its behaviour is controlled by `currentValueInColorIndexArray`. But whereas there it was used to control how many stages of the structure to paint, here it is used to figure out which color to paint the pixel or whether to paint the pixel at all.

The core of this operation lies as you might imagine in seeing what color the pixel is already. The first step of this involves using the `pixelXPosition` and `pixelYPosition` to load the color information for the position on screen we might want to paint:

```

PaintPixel
...
JSR LoadXAndYPosition
; Y now contains the pixelYPosition
LDA (currentLineForPixelInColorRamLoPtr),Y
; Make sure the color we get is addressable by
; presetColorValuesArray.
AND #COLOR_MAX

```

Now while this value will be between 0 and 7, representing one of the seven colors we paint with, it isn't one we can directly compare with `currentValueInColorIndexArray`. This is because `currentValueInColorIndexArray` is an index rather than a color value. So what we need to do with the color we've retrieved from the screen is figure out what index in `presetColorValuesArray` it refers to:

```

presetColorValuesArray
;      0      1      2      3      4      5      6      7
.BYTE BLACK, BLUE, RED, PURPLE, GREEN, CYAN, YELLOW, WHITE

PaintPixel

```

```
...
; Translate the color at the current pixel to an
; index value so that it can be compared with
; currentValueInColorIndexArray
LDX #$00
FindIndexLoop
    CMP presetColorValuesArray,X
    BEQ b4096
    INX
    CPX #COLOR_MAX + 1
    BNE FindIndexLoop
```

The snippet of code above is what does this. For example where it retrieves the color PURPLE from the position on screen it converts it to a value of 3 (the index position of PURPLE in presetColorValuesArray).

Once we have that we can now decide whether or not to paint the pixel with the color referenced by currentValueInColorIndexArray. What decides this for us is whether the current color at the position in the screen is any of:

- black, i.e. nothing has been painted there yet;
- the same as the color we want to paint;
- or the one just above it in presetColorValuesArray.

So if the color we want to paint is PURPLE, we will only paint on black cells, purple cells, or green cells. Reminding ourselves again that GREEN appears after PURPLE in presetColorValuesArray.

```
pressetColorValuesArray
;          0      1      2      3      4      5      6      7
.BYTE BLACK, BLUE, RED, PURPLE, GREEN, CYAN, YELLOW, WHITE
```

This explains what we saw a little earlier. When painting purple we didn't paint anything in the first phase because the cells at that position were yellow. However in the phases after that we could paint over the green and black cells.

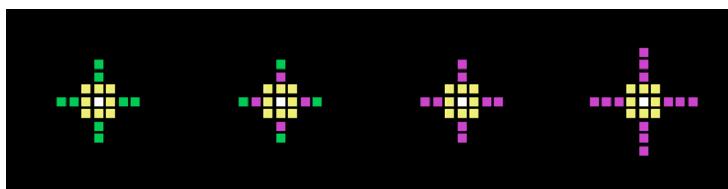


Figure 2.2: Painting only green and black cells, ignoring yellow.

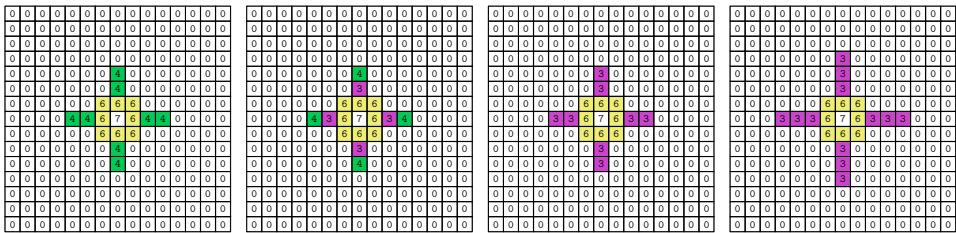


Figure 2.3: The same as above but this time with the index values for each cell. We can only paint in a cell that is black, the same color as the one we want to paint, or just below it in presetColorValuesArray.

In practice this means checking whether the color referenced by currentValueInColorIndexArray is less than or equal to the one we want to paint (indexOfCurrentColor). In CheckIfShouldPaintPixel we implement this check:

```

PaintPixel
...
; If the color value at the current pixel (indexOfCurrentColor)
; is the one 'after' the new one in presetColorValuesArray
; (currentValueInColorIndexArray) then we paint it,
; otherwise we ignore it.
CheckIfShouldPaintPixel
    TXA
    STA currentValueInColorIndexArray
    LDX currentValueInColorIndexArray
    INX
    CPX currentValueInColorIndexArray
    BEQ ActuallyPaintPixel
    BPL ActuallyPaintPixel
    RTS

ActuallyPaintPixel
    LDX currentValueInColorIndexArray
    LDA presetColorValuesArray,X
    STA (currentLineForPixelInColorRamLoPtr),Y
    RTS

```

We do it by first loading currentValueInColorIndexArray to the X register:

```
LDX currentValueInColorIndexArray
```

Then incrementing it:

```
INX
```

Then seeing if our current colour is equal to (BEQ) or greater (BPL) than it:

```
LDX currentValueInColorIndexArray
```

```
CPX indexOfCurrentColor  
BEQ ActuallyPaintPixel  
BPL ActuallyPaintPixel
```

If either of these hold we can actually paint the pixel:

```
LDX currentValueInColorIndexArray  
LDA presetColorValuesArray,X  
STA (currentLineForPixelInColorRamLoPtr),Y  
RTS
```

How, you might wonder, do any of those four lines actually paint a pixel? For that, we will have to look at the code in a little more detail and explain the mechanics of the C64 screen painting mechanism while we're at it. This we will do in the next chapter. But for now, here are the fruits of `PaintPixel` when running through the full sequence of painting a star-shaped thing following the algorithm we've just described.

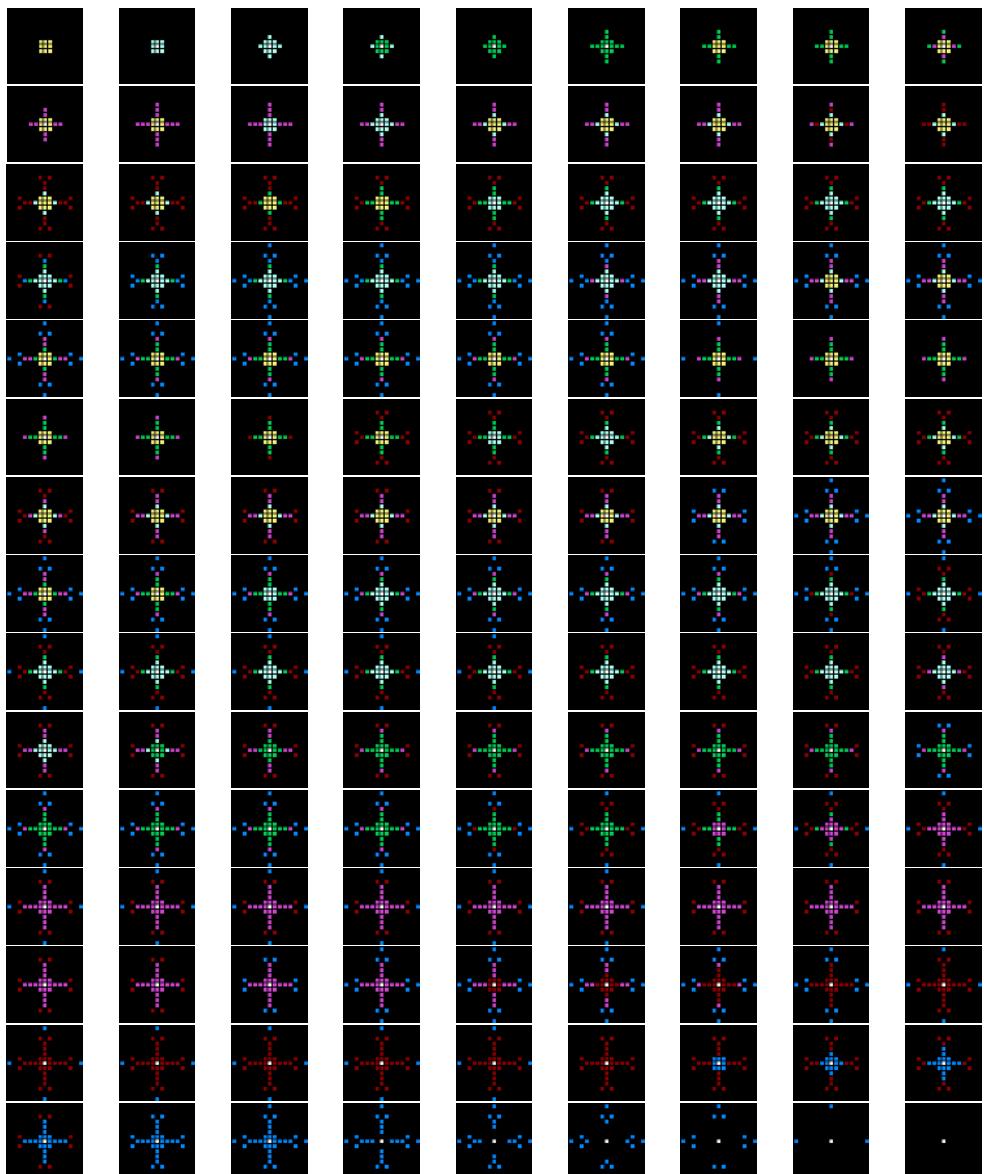


Figure 2.4: The full standard sequence of a star-shaped thing.



# **let's pretend we can read code**

The listing is so short that it possible (and hopefully not too tedious) for us to review it in its entirety. This will allow us to understand the basic workings of a small 'game' such as this one, from booting to running; it will also give us an accurate idea of how the final 'Psychedelia' product worked. The difference between the released game and the listing that appeared in Popular Computing Weekly is not qualitative - the underlying structure is the same, the core routines we reviewed in the previous chapter are more or less identical. The differences lie only in the the wealth of options available, so it turns out that the listing is an ideal way for us to get our heads around the core mechanics of the light synthesizer. In the chapters that follow we can look at some of the features added to the game in more detail and concentrate on the individual routines that made them work. Here we will focus instead on the high-level operation of the game and hopefully obtain a sense of how the thing hangs together.

For the most part I try to present the code in the order it appears in the game binary itself, but I have moved some routines around here and there so that we're reading the code in an order that makes sense to the reader.

Before we drive into the weeds, let's take an aerial view of the code over the next few pages. This gives us a proper sense of how small the program is and also where things are in the listing, how it is ordered, and what, if any, structure it possesses.

## let's pretend we can read code

---

```

;-----[CopyCodeToRAM]-----[SYS 2064]
; Start program at CopyCodeToRAM
; (SYS 2064)
;-----[CopyCodeToRAM]-----[SYS 2064]

* = $0801
.BYTE $00,$08
.BYTE $0A,$00
.BYTE $9E,$22,$30,$36
.BYTE $34,$00,$00,$00,$00,$00,$00,$00

;-----[CopyCodeToRAM]-----[SYS 2064]
CopyCodeToRAM
LDA #$40
STA RAM4000HiPtr
LDA #$08
STA RAM0835HiPtr
LDA #$00
STA RAM4000LoPtr
LDA #$_35
STA RAM0835LoPtr
LDY #$00
LDX #$_06
.Loop LDA (RAM0835LoPtr),Y
STA (RAM4000LoPtr),Y
DEY
BNE _Loop
INC RAM4000HiPtr
INC RAM0835HiPtr
DEX
BNE _Loop
JMP InitializeProgram

NUM_COLS = $28
NUM_ROWS = $18
;-----[InitializeProgram]-----[SYS 2064]
InitializeProgram
LDA #$_00
STA $D020 ;Border Color
STA $D021 ;Background Color 0

LDA #$_COLOR_RAM
STA colorRamHiPtr
LDA #$_COLOR_RAM
STA colorRamLoPtr

LDX #$_00
.Loop
LDA colorRamHiPtr
STA colorRAMLinesTableHiPtrArray,X
LDA colorRamLoPtr
STA colorRAMLinesTableLoPtrArray,X
CLC
ADC #$_UM_COLS
STA colorRamHiPtr
LDA colorRamHiPtr
ADC #$_00
STA colorRamHiPtr
INX
CPX #$_UM_ROWS+1
BNE _Loop

JSR InitializeScreenAndText
JMP LaunchPsychedelia

;-----[InitializeScreenWithInitCharacter]-----[SYS 2064]
InitializeScreen
LDI #$_00
.Loop
LDA #$_CF
STA SCREEN_RAM + $0000,X
STA SCREEN_RAM + $0100,X
STA SCREEN_RAM + $0200,X
STA SCREEN_RAM + $0300,X
LDA #$_00
STA COLOR_RAM + $0000,X
STA COLOR_RAM + $0100,X
STA COLOR_RAM + $0200,X
STA COLOR_RAM + $0300,X
DEX
BNE _Loop
RTS

presetColorValuesArray
.BYTE BLACK,BLUE,RED,PURPLE,GREEN,
CYAN,YELLOW,WHITE
;-----[LoadAndYPosition]-----[SYS 2064]
LoadAndYPosition
LDX pixelYPosition
LDA colorRAMLinesTableLoPtrArray,X
STA curLineInColorRamLoPtr
LDA colorRAMLinesTableHiPtrArray,X
STA curLineInColorRamHiPtr
LDY pixelYPosition
ReturnEarly
RTS

```

```

;-----[PaintPixel]-----[SYS 2064]
PaintPixel
LDA pixelXPosition
AND #$_80
BNE ReturnEarly
LDA pixelXPosition
CMP #$_NUM_COLS
BPL ReturnEarly
LDA pixelYPosition
AND #$_80
BNE ReturnEarly
LDA pixelYPosition
CMP #$_NUM_ROWS
BPL ReturnEarly

JSR LoadAndYPosition
LDA (curLineInColorRamLoPtr),Y
AND #$_COLOR_MAX

LDX #$_00
.Loop
CMP presetColorValuesArray,X
BEQ MaybePaintPixel
INX
CPX #$_COLOR_MAX + 1
BNE _Loop

MaybePaintPixel
TIA
STA currentColorValueOfPixel
LDX colorIndexForCurrentPixel
INX
CPX currentColorValueOfPixel
BEQ ActuallyPaintPixel
BPL ActuallyPaintPixel
RTS

ActuallyPaintPixel
LDX colorIndexForCurrentPixel
LDA presetColorValuesArray,X
STA (curLineInColorRamLoPtr),Y
RTS

;-----[PaintStructureAtCurrentPosition]-----[SYS 2064]
PaintStructureAtCurrentPosition
JSR PaintPixelForCurrentSymmetry
LDY #$_00
LDA colorIndexForCurrentPixel
CMP #$_07
BNE CanLoopAndPaint
RTS

CanLoopAndPaint
LDA #$_07
STA countToMatchCurrentIndex

LDA pixelXPosition
STA initialPixelXPosition
LDA pixelYPosition
STA initialPixelYPosition

PixelPaintLoop
LDA initialPixelXPosition
CLC
ADC starOneXPosArray,Y
STA pixelXPosition

LDA initialPixelYPosition
CLC
ADC starOneYPosArray,Y
STA pixelYPosition

TIA
PHA
JSR PaintPixelForCurrentSymmetry
PLA
TAY
PLA
STA pixelYPosition
PLA
STA pixelXPosition
RTS

PaintPixelForCurrentSymmetry
LDA pixelXPosition
PHA
JSR PaintPixel
PLA
STA pixelYPosition
PLA
STA pixelXPosition
RTS

HasSymmetry
LDA #$_NUM_COLS
SEC
SBC pixelXPosition
STA pixelXPosition

JSR PaintPixel
LDA currentSymmetrySettingForStep
CMP #$_01
BEQ CleanUpAndReturnFromSymmetry

LDA #$_NUM_ROWS
SEC
SBC pixelYPosition
STA pixelYPosition
JSR PaintPixel

PLA
TAY
PLA
STA pixelXPosition
TIA
PHA
JSR PaintPixel
PLA
TAY
PLA
STA pixelYPosition
RTS

currentSymmetrySettingForStep
.BYTE $01
pixelXPositionArray
.BYTE $0F,$0E,$0D,$0C,$0B,$0A,$09,$04
.BYTE $05,$06,$07,$08,$09,$0A,$0B,$0C
.BYTE $0D,$0E,$0F,$0G,$0H,$0I,$0J,$0K
.BYTE $15,$16,$17,$18,$13,$12,$11,$10
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
pixelYPositionArray
.BYTE $0C,$0D,$0E,$0F,$0G,$0H,$0B,$04
.BYTE $04,$04,$04,$04,$04,$04,$04,$04
.BYTE $06,$07,$08,$09,$0A,$0B,$0C,$0D
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00

```

**Lines 1-400.** The first 400 lines or so of the listing opposite contain the main meat of the painting engine we covered in the previous chapter. In the second and third columns opposite we find the PaintPixel, PaintStructureAtCursorPosition, and PaintPixelForCurrentSymmetry routines, in that order.

The start of the program, in the first columns, contains a number of bookkeeping routines used at program initialization. You'll note at the very start of the program the clause:

```
* = $0801
```

This indicates that the program is loaded to position \$0801 in RAM. If you imagine the working memory of the C64 as a long tape 65,536 segments long (\$0000 to \$FFFF) then this little program nestles modestly near the very start of the tape occupying a mere 1415 bytes. So what you are looking at opposite is approximately 700 or so of these bytes in their assembly language form.

Near the end of the third column we come to the end of the 'brain' of our truncated version of Psychedelia and start into a short section of data. We encountered the use of these arrays in the previous chapter and together with the main routines they are responsible for working out what pixels to paint and where. The hard work is done on the next page, but it is here that the art takes place.

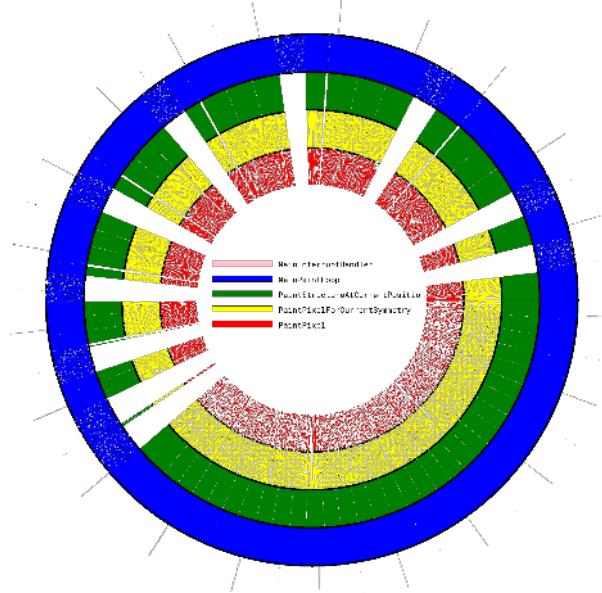


Figure 3.1: The execution map of a full pattern evolution in the listing edition of Psychedelia.

## let's pretend we can read code

---

```

currentColorIndexArray
.BYTE $FF,$FF,$FF,$FF,$FF,$FF,$FF
.BYTE $FF,$FF,$FF,$FF,$FF,$FF,$FF
.BYTE $FF,$FF,$FF,$FF,$FF,$FF,$FF
.BYTE $FF,$FF,$FF,$FF,$FF,$FF,$FF
.BYTE $00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00
initialFramesRemainingToNext
.BYTE $0C,$0C,$0C,$0C,$0C,$0C,$0C,$0C
.BYTE $0C,$0C,$0C,$0C,$0C,$0C,$0C,$0C
.BYTE $0C,$0C,$0C,$0C,$0C,$0C,$0C,$0C
.BYTE $0C,$0C,$0C,$0C,$0C,$0C,$0C,$0C
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
framesRemainingToNextPaint
.BYTE $04,$07,$01,$02,$03,$06,$07,$06
.BYTE $0C,$02,$03,$06,$07,$01,$02,$02
.BYTE $04,$04,$07,$01,$02,$03,$06,$07
.BYTE $0C,$02,$03,$02,$03,$07,$01,$02
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
; ReinitializeSequences
; -----
ReinitializeSequences
LDX #$00
TXA
b42D9
STA pixelXPosition,X
STA pixelYPosition,X
STA currentColorIndexArray,X
STA initialFramesRemainingToNext,X
STA framesRemainingToNextPaint,X
INX
CPX #$40
BNE b42D9
RTS

; -----
; LaunchPsychelia
; -----
LaunchPsychelia
JSR ReinitializeSequences
JSR SetUpInterruptHandlers
; -----
; MainPaintLoop
; -----
MainPaintLoop
LDX currentColorIndexArray,X
LDA currentIndexToPixelBuffers
AND maskForFireOffset
STA currentIndexToPixelBuffers
TAX
DEC framesRemainingToNextPaint,X
BNE GoBackToStartOfLoop
LDA initialFramesRemainingToNext,X
STA framesRemainingToNextPaint,X
LDA currentColorIndexArray,X
CMP #$FF
BEQ GoBackToStartOfLoop
STA colorIndexForCurrentPixel
LDA pixelXPosition,X
STA pixelXPosition
LDA pixelYPosition,X
STA pixelYPosition
JSR PaintScreenAtCurrentPosition
LDA currentColorIndexArray,X
DEC currentColorIndexArray,X
GoBackToStartOfLoop
IMP MainPaintLoop

; -----
; SetUpInterruptHandlers
; -----
SetUpInterruptHandlers
SEI
LDA #MainInterruptHandler
STA $0014 ;IRQ
LDA #$0015 ;IRQ
LDA #$0A
STA cursorXPosition
STA cursorYPosition
LDA #$01
STA $D015 ;Sprite 0 display Enable
STA $D027 ;Sprite 0 Color
CLI
RTS

```

```

currentIndexToPixelBuffers .BYTE $08
stepsToNextInputCheck .BYTE $01
lastColorPainted .BYTE $00
; -----
; MainInterruptHandler
MainInterruptHandler
DEC stepsToNextInputCheck
BEQ b4353
JMP RETURN_FROM_INTERRUPT

b4353 LDA #$02
STA stepsToNextInputCheck
LDA #$00
STA currentColorToPaint

JSR PaintCursorAtCurrentPosition
LDA $DC00 ;CIA1: Data Port
            Register A
AND #$03
CMP #$03
BEO CheckIfCursorMovedLeftOrRight

CMP #$02
BEO PlayerHasPressedDown

INC cursorXPosition
INC cursorYPosition

PlayerHasPressedDown
DEC cursorYPosition
LDA cursorYPosition
CMP #$FF
BNE CheckIfCursorAtBottom

LDA #$_17
STA cursorYPosition
JMP CheckIfCursorMovedLeftOrRight

CheckIfCursorAtBottom
CMP #NUM_ROWS
BNE CheckIfCursorMovedLeftOrRight

LDA #$00
STA cursorYPosition

CheckIfCursorMovedLeftOrRight
LDA $DC00 ;CIA1: Data Port
            Register A
AND #$0C
CMP #$0C
BEO CheckIfPlayerPressedFire

CMP #$08
BEO CheckMovedLeft
; Player has pressed right.
INC cursorPosition
INC cursorXPosition

CursorMovedLeft
DEC cursorXPosition
LDA cursorPosition
CMP #$FF
BNE CheckIfCursorAtExtremeRight

LDA #$_27
STA cursorPosition
JMP CheckIfPlayerPressedFire

CheckIfCursorAtExtremeRight
CMP #NUM_COLS
BNE CheckIfPlayerPressedFire
LDA #$00
STA cursorPosition

CheckIfPlayerPressedFire
LDA $DC00 ;CIA1: Data Port
            Register A
AND #$10
BEO PlayerHasntPressedFire

; Player has pressed fire.
LDA #$00
STA stepsSincePressedFire
JMP DrawCursorAndReturnFromInterrupt

PlayerHasntPressedFire
LDA stepsExceeded255
BEO b43D7
LDA stepsSincePressedFire
BNE DrawCursorAndReturnFromInterrupt

INC stepsSincePressedFire
b43D7
INC seedValueForArrayIndices
LDA seedValueForArrayIndices
AND maskForFireOffset
STA seedValueForArrayIndices

```

```

UpdateColorIndexArray
TAX
LDA currentColorIndexArray,X
CMP #$FF
BNE DrawCursorAndReturnFromInterrupt

LDA cursorXPosition
STA pixelXPositionArray,X
LDA cursorYPosition
STA pixelYPositionArray,X
LDA #COLOR_MAX
STA currentColorIndexArray,X

LDA smoothingDelay
STA initialFramesRemainingToNext,X
STA framesRemainingToNextPaint,X

DrawCursorAndReturnFromInterrupt
JSR LoadAndYOfCursorPosition
LDA (curCursorLineColorRamLoPtr),Y
AND #COLOR_MAX
STA lastColorPainted
LDA #WHITE
STA currentColorToPaint
JSR PaintCursorAtCurrentPosition
JMP RETURN_FROM_INTERRUPT

LoadXandYOfCursorPosition
LDX cursorYPosition
LDA colorRAMLineTableLoPtrArray,X
STA curCursorlineColorRamLoPtr
LDA colorRAMLineTableHiPtrArray,X
STA curCursorlineColorRamHiPtr
LDY cursorXPosition
RTS

; -----
; PaintCursorAtCurrentPosition
; -----
PaintCursorAtCurrentPosition
JSR LoadAndYOfCursorPosition
LDA currentColorToPaint
STA (curCursorLineColorRamLoPtr),Y
RTS

cursorXPosition .BYTE $1E
cursorYPosition .BYTE $0D
seedValueForArrayIndices .BYTE $1A
maskForFireOffset .BYTE $1F
stepsExceeded255 .BYTE $00
smoothingDelay .BYTE $0C
.BYTE $00,$00,$00,$00,$00,$00,$00
.BYTE $5B,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00
.BYTE $FF,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00
.bannerText
.TEXT $00,"PSYCHEDELIA...A FORETASTE
BY JEFF MINTER"

; -----
; InitializeScreenAndText
; -----
InitializeScreenAndText
JSR InitializeScreen

LDX #NUM_COLS
Loop LDA bannerText,X
STA SCREEN_RAM + $03BF,X
LDA #$_0C
STA COLOR_RAM + $03BF,X
DEX
BNE _Loop
RTS

.BYTE $00,$00,$00,$8F,$00,$8D,$00,$FF
.BYTE $00,$FF,$00,$FF,$00,$FF,$00,$8F
.BYTE $FF,$FF,$FF,$FF

```

**Lines 401-720.** This second half of the program contains, towards the bottom of the first column opposite, the main loop (`MainPaintLoop`) that runs round and round in a circle for as long as the program itself is running.

The bulk of the rest of the listing is what we call an 'interrupt handler', a routine that gets called numerous times every second by the C64 and so is a useful place to do things like checking for user input on the keyboard or joystick, which is precisely what `MainInterruptHandler` in fact does.

Towards the end, in the third column, we find a couple of helper functions tagged on: `LoadXAndYOfCursorPosition` and `PaintCursorPositionAtCurrentPosition`. These are used by the interrupt handler so can be considered part of the interrupt handler itself.

The very last part contains some variables used by the handler, e.g `cursorXPosition` and `cursorYPosition`. And then a longish section of unused data, just zero bytes, followed by the text storage for the program's title screen. We encountered this part in our first chapter, where we saw how the magazine listing had been misprinted at this point and left us attempting to piece it back together with the program's title as a useful clue for the missing bytes.

And that is it - a quite tiny program that fits (not quite legibly I'll admit) on just two pages. In the rest of this chapter we'll dig into the details of each of these routines. It may be useful to refer back here every now and then to get a sense of where you are on the map as we pull apart the detail.

let's pretend we can read code

---

### Lines 46-50. Sys2064

```
* = $0801
; -----
; Start program at CopyCodeToRAM (SYS 2064)
; SYS 2064 ($0810)
; This is where execution starts.
; It is a short BASIC program that executes whatever is at address
; $0810 (2064 in decimal). In this case, that's CopyCodeToRAM
; -----
.BYTE $0B,$08          ; Points to EndOfProgram address below
.BYTE $0A,$00          ; Arbitrary Line Number, in this case: 0010
.BYTE $9E              ; SYS
.BYTE $32,$30,$36,$34 ; 2064 ($810), which is CopyCodeToRAM below.
.BYTE $00              ; Null byte to terminate the line above.
.BYTE $00,$00           ; EndOfProgram (all zeroes)
.BYTE $00,$00,$00       ; Filler bytes so that InitializeProgram is
; located at $0810
```

Listing 3.1: The bootstrap routine common to nearly every C64 program

### Lines 56-75. CopyCodeToRam

```
RAM0835LoPtr          = $FD
RAM0835HiPtr          = $FE
RAM4000HiPtr          = $FC
RAM4000LoPtr          = $FB
; -----
; CopyCodeToRAM
; -----
CopyCodeToRAM
    LDA #$40
    STA RAM4000HiPtr
    LDA #$08
    STA RAM0835HiPtr
    LDA #$00
    STA RAM4000LoPtr
    LDA #$35
    STA RAM0835LoPtr
    LDY #$00
    LDX #$06
CopyLoop
    LDA (RAM0835LoPtr),Y
    STA (RAM4000LoPtr),Y
    DEY
    BNE CopyLoop
    INC RAM4000HiPtr
    INC RAM0835HiPtr
    DEX
    BNE CopyLoop
    JMP InitializeProgram
```

Listing 3.2: Some strictly unnecessary code copying

**Lines 46-50.** Sys2064: Yay, let's do some admin like starting the program! All C64 programs begin execution at position \$0801, it's the law. So every C64 program has a little section like this that the C64 executes first and which tells the C64 what to do next. Here we tell it to start execution at another location \$0810. Since we're currently at location \$0801, that's almost 16 bytes ahead and contains the code we've labelled CopyCodeToRAM which we jump to next.

The command is given in a slightly strange way. Not strange for the time, but an oddly convoluted convention in its own right. The SYS command is followed by a series of PETSCII values forming the decimal number 2064. In hexadecimal this is \$0810 which is the actual location to start executing. So SYS 2064 as given by '\$9E, \$32, \$30, \$36,\$34' means 'start executing whatever is at memory location \$0810'. Which in this case is the little routine CopyCodeToRAM.

The 'Arbitrary Line Number', is just that: completely arbitrary. Here we have '0010', but in the commercial release of Psychedelia the value chosen is \$C1,\$07 - translated to decimal this is '1985', the year of the game's release.

**Lines 56-75.** CopyCodeToRam: Little Commodore 64 programs sure do like copying things around. This routine copies the entire code of the program to a completely new position in memory at location \$4000 and then jumps to the routine InitializeProgram that will actually start running 'Psychedelia'.

I think this was some sort of cheap and dirty copy protection, designed to prevent people from casually disassembling the program if they felt minded to do so.

#### Friendly Blue Learning Box

PETSCII (UNSHIFTED):																
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
20	!	"	#	\$	%	&	'	(	)	*	,	-	.	/		
30	0	1	2	3	4	5	6	7	8	9	:	<	=	>	?	
40	€	¤	฿	₵	₵	₵	₵	₵	₵	₵	₵	₵	₵	₵	₵	₵
50	₱	₵	₵	₵	₵	₵	₵	₵	₵	₵	₵	₵	₵	₵	₵	₵
60	-	+	!	-	-	-	-	-	-	-	!	+	!	+	!	
70	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	
80	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	
90	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	
B0	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	
B0	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	ػ	
	READY.															

PETSCII is the Commodore 64's flavour of the ASCII character set. It is simply a convention that attributes certain byte values to certain characters and numbers. For example, the value \$41 signifies the A character, \$42 signifies the B character. Likewise \$31 signifies the numeral 1, \$32 the numeral 2, and so on. These values are arbitrary, so purely conventional. It is possible for the programmer to use customized values and map them to any character set they please.

let's pretend we can read code

---

### Lines 82-113. InitializeProgram

```
NUM_COLS    = $28
NUM_ROWS    = $18
COLOR_RAM   = $D800
; -----
; InitializeProgram
; -----
InitializeProgram
    LDA #BLACK
    STA $D020      ; Border Color
    STA $D021      ; Background Color 0

PrepareHiLoPtrs
    LDA #>COLOR_RAM
    STA colorRamHiPtr
    LDA #<COLOR_RAM
    STA colorRamLoPtr

    LDX #$00
FillLinePointerLoop
    LDA colorRamHiPtr
    STA colorRAMLineTableHiPtrArray,X
    LDA colorRamLoPtr
    STA colorRAMLineTableLoPtrArray,X
    CLC
    ADC #NUM_COLS
    STA colorRamLoPtr
    LDA colorRamHiPtr
    ADC #$00
    STA colorRamHiPtr
    INX
    CPX #NUM_ROWS+1
    BNE FillLinePointerLoop

    JSR InitializeScreenAndText
    JMP LaunchPsychedelia
```

Listing 3.3: The initialization routine - visited only once at the very start of execution

**Lines 82-113.** InitializeProgram: More admin! First we set the background and border color to black by loading \$00 to the memory locations \$D020 for the border and \$D021 for the background color.

**Lines 90-93.** PrepareHiLoPtrs: COLOR\_RAM (\$D800) is the address at which Color RAM starts. Color RAM is a region of memory 1000 bytes long that contains the color value for each of the 25 rows of 40 characters that get displayed on the screen.

In order to be able to work with this address we need to have a way of working with values that are two bytes long even though our 6502 instruction set only accommodates single byte values. To do this we have to split the address in two and store each half in a separate variable. LDA #>COLOR\_RAM is an assembly language convention for referencing the first byte in the two byte address COLOR\_RAM, i.e \$D800. So another way of writing it is simply: LDA #\$D8. Likewise LDA #<COLOR\_RAM is another way of writing LDA #\$00.

We're storing each of these values in `colorRamHiPtr` and `colorRamLoPtr` respectively as we're going to use them to populate a pair of arrays with each half of the addresses of each of the 25 rows on the screen.

**Lines 101-113.** FillLinePointerLoop: Here we set up the pair of arrays that Psychedelia will make heavy use of: `colorRAMLineTableHiPtrArray` and `colorRAMLineTableLoPtrArray`. Together these arrays will function as a map on to the screen for drawing our pixels. Since the Commodore 64 screen is 25 rows high we make each array 25 elements long and together each element in each array will combine to give the address on the screen for the start of each row.

Element	colorRAMLineTable HiPtrArray	colorRAMLineTable LoPtrArray	Address
0	\$D8	\$00	\$D800
1	\$D8	\$28	\$D828
.	.	.	.
6	\$D8	\$F0	\$D8F0
.	.	.	.
23	\$DB	\$70	\$DB70
24	\$DB	\$98	\$DB98

Figure 3.2: Our two arrays and their contents - each combining to give us an address for the start of each row in Color RAM.

So in practice what this gives us is a 'single array' with which we can address each row on the screen and write color values to the pixel position of our choice.

## let's pretend we can read code

---

\$D800	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
\$D828	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
\$D850	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
\$D878	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
\$D8A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
\$D8C8	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
\$D8F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	02	00	02	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
\$D918	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	02	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
\$D940	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
\$D968	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	03	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
\$D990	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	02	07	07	00	00	02	00	00	00	00	00	00	00	00	00	00	00	00	00
\$D9B8	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	02	05	03	07	07	03	05	02	00	00	00	00	00	00	00	00	00	00	00
\$D9E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	02	00	00	00	00	02	00	00	00	00	00	00	00	00	00	00	00	00	00
\$DA08	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	03	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
\$DA30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	05	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
\$DA58	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	02	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
\$DA80	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	02	00	02	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
\$DAA0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
\$DAF8	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
\$DB20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
\$DB48	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
\$DB70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
\$DB98	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Figure 3.3: Color RAM for the screen with the starting address for each row on the left.

For example, to make the pixel on row 7 and column 15 red above, we have written a value of 02 to the address \$D8FF. This address is formed from adding \$D8F0 (the 7th element in our array) + 15, i.e.  $\$D8F0 + \$0F = \$D8FF$ . To actually do this we would do something like the following:

```
RED = $02  
; Load the 7th element from each array.  
LDX #$06  
; This stores $F0 in currentLineInColorRamLoPtr  
LDA colorRAMLineTableLoPtrArray,X  
STA currentLineInColorRamLoPtr  
; This stores $D8 in currentLineInColorRamHiPtr  
LDA colorRAMLineTableHiPtrArray,X  
STA currentLineInColorRamHiPtr  
; Load $02, i.e. RED, to the Accumulator.  
LDA #RED  
; Y will be our index so give it the offset into the  
; line, i.e. 15.  
LDY #$0F  
; Write $02 to address $DBFF, painting it red.  
STA (currentLineInColorRamLoPtr),Y
```

This gives us an idea of how the array will be used, but let's remind ourselves that all we are doing here is setting it up in the first place. This consists of iterating from 0 to 24 and populating each element in our two arrays with the appropriate value for the line on the screen each element represents. So at each iteration we store the current values into the current position in the array given by X:

```
FillLinePointerLoop  
    LDA colorRamHiPtr  
    STA colorRAMLineTableHiPtrArray,X  
    LDA colorRamLoPtr  
    STA colorRAMLineTableLoPtrArray,X
```

Then we add NUM\_COLS (i.e. 40) to colorRamLoPtr:

```
CLC  
ADC #NUM_COLS  
STA colorRamLoPtr
```

Finally we use ADC \$00 to ensure any carry from the previous addition is added to colorRamHiPtr.

```
LDA colorRamHiPtr  
ADC #$00  
STA colorRamHiPtr
```

This successfully increments both colorRamLoPtr and colorRamHiPtr for storage in the next position in the array. Finally the loop will stop when X eventually reaches NUM\_ROWS + 1, that is, once we have populated an element in the array for every row:

```
INX  
CPX #NUM_ROWS+1  
BNE FillLinePointerLoop
```

let's pretend we can read code

---

#### Lines 715-725. InitializeScreenAndText

```
bannerText
    .TEXT $00,"PSYCHEDELIA...A FORETASTE BY JEFF MINTER"

; -----
; InitializeScreenAndText
; -----
InitializeScreenAndText
    JSR InitializeScreen

    LDX #NUM_COLS
Loop    LDA bannerText,X
        STA SCREEN_RAM + $03BF,X
        LDA #WHITE
        STA COLOR_RAM + $03BF,X
        DEX
        BNE Loop
RTS
```

Listing 3.4: A routine that fills the screen with black and the title text.

#### Lines 118-132. InitializeScreen

```
; -----
; InitializeScreen
; -----
InitializeScreen
    LDX #$00
Loop    LDA #$CF
        STA SCREEN_RAM + $0000,X
        STA SCREEN_RAM + $0100,X
        STA SCREEN_RAM + $0200,X
        STA SCREEN_RAM + $0300,X
        LDA #BLACK
        STA COLOR_RAM + $0000,X
        STA COLOR_RAM + $0100,X
        STA COLOR_RAM + $0200,X
        STA COLOR_RAM + $0300,X
        DEX
        BNE Loop
RTS
```

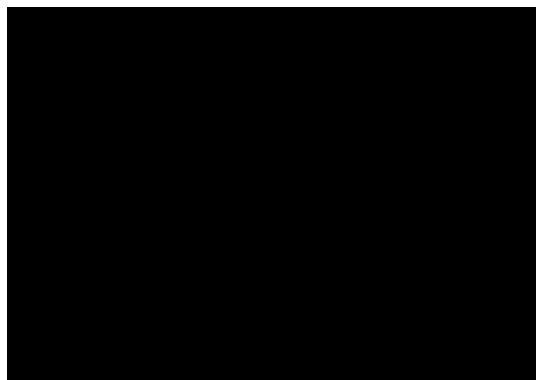
Listing 3.5: Fills the screen with black

**Lines 715-725.** InitializeScreenAndText: Populates the screen with the banner text. The same loop writes the text (contained in bannerText) to the Screen RAM and sets the color of each character to white by writing the value \$0C (White) to the Color RAM.



The screen after InitializeScreenAndText has worked its magic.

**Lines 118-132.** InitializeScreen: This fills the screen with black boxes. These are the boxes we will color in each time we paint a pixel. As mentioned before there are two components to writing a character to a screen. The first is the character itself, the second is the color(s) of the character. Psychedelia works by drawing the same character at every position but painting it black if its empty or filling it with color if it should be displayed. So when we initialize the screen we paint a box at every position and paint it black.



The screen after InitializeScreen has painted it black.

let's pretend we can read code

---

### Lines 300-313. starOneXPosArray

```
starOneXPosArray
    .BYTE $00,$01,$01,$01,$00,$FF,$FF,$FF,$55      ;      5
    .BYTE $00,$02,$00,$FE,$55                         ;
    .BYTE $00,$03,$00,$FD,$55                         ;      4 4
    .BYTE $00,$04,$00,$FC,$55                         ;      3
    .BYTE $FF,$01,$05,$05,$01,$FF,$FB,$FB,$55       ;      2
    .BYTE $00,$07,$00,$F9,$55                         ;      1
    .BYTE $55                                         ;      4 000 4
starOneYPosArray
    .BYTE $FF,$FF,$00,$01,$01,$01,$00,$FF,$55      ; 5 3210 0123 5
    .BYTE $FE,$00,$02,$00,$55                         ;      4 000 4
    .BYTE $FD,$00,$03,$00,$55                         ;      1
    .BYTE $FC,$00,$04,$00,$55                         ;      2
    .BYTE $FB,$FB,$FF,$01,$05,$05,$01,$FF,$55       ;      3
    .BYTE $F9,$00,$07,$00,$55                         ;      4 4
    .BYTE $55                                         ;      5
```

Listing 3.6: Hopefully this looks familiar.

### Lines 321-327. PutRandomByteInAccumulator

```
; -----
; PutRandomByteInAccumulator
; -----
PutRandomByteInAccumulator
randomByteAddress=**+$01
    LDA $E199,X
    INC randomByteAddress
RTS
```

Listing 3.7: Random unused feels like a metaphor.

### Lines 434-445. ReinitializeSequences

```
; -----
; ReinitializeSequences
; -----
ReinitializeSequences
    LDX #$00
    TXA
ReinitializeLoop
    STA pixelXPositionArray,X
    STA pixelYPositionArray,X
    STA currentColorIndexArray,X
    STA initialSmoothingDelayForStep,X
    STA smoothingDelayForStep,X
    INX
    CPX #$40
    BNE ReinitializeLoop
RTS
```

Listing 3.8: Fill our pixel arrays with zeros.

**Lines 300-313.** `starOneXPosArray`: We've encountered this data structure in our previous chapters. It encodes the pattern used in the listing as you hopefully will recall. The commercial edition of *'Psychedelia'* has a whole menagerie of these structures going on and we explore them in detail in '[all the pretty patterns](#)'. So flick ahead there if you are impatient to find out more about them.

**Lines 321-327.** `PutRandomByteInAccumulator`: This routine is unused in the listing but it is used by the commercial version of '*'Psychedelia'*': for feeding random joystick movements to the game's attract/demo mode. A bit of random left-over like this is not surprising when taking the commercial source code and reducing it down for a magazine listing.

I'm obliged to point out that the routine does something very cute. The first time it is called it loads whatever is in the address \$E199 to the accumulator. That's a randomish value that can now be used by the caller. To ensure it produces a different number the next time it's called, it increments the \$E199 to \$E19A before returning. Notice that `randomByteAddress` is pointing to the \$99 part of \$E199.

A fancier way of saying all this is that the routine is self-mutating code! Each time it's called it increments the piece of its code that initially referenced \$E199 by calling `INC randomByteAddress`. Thus ensuring that it always stores a new pseudo-random value in the accumulator(`A`) register.

**Lines 434-445.** `ReinitializeSequences`: This is a loop that fills the 5 arrays defined on the next page with zeroes. `TXA` loads the value of \$00 previously loaded to `X` to the `A` register, so it could as easily be `LDA $00` however it has the dubious advantage of being one byte shorter as an instruction.

Notice that `X` gets incremented on each pass of the loop until it reaches \$40 - filling all 64 elements of each array with \$00.

### Lines 385-429. pixelXPositionArray

```
countToMatchCurrentIndex      .BYTE $01
MAX_INDEX_VALUE              .BYTE $1F
pixelXPositionArray
    .BYTE $0F,$0E,$0D,$0C,$0B,$0A,$09,$04
    .BYTE $05,$06,$07,$08,$09,$0A,$0B,$0C
    .BYTE $0D,$0E,$0F,$10,$11,$12,$13,$14
    .BYTE $15,$16,$17,$14,$13,$12,$11,$10
    .BYTE $00,$00,$00,$00,$00,$00,$00,$00
    .BYTE $00,$00,$00,$00,$00,$00,$00,$00
    .BYTE $00,$00,$00,$00,$00,$00,$00,$00
    .BYTE $00,$00,$00,$00,$00,$00,$00,$00
pixelYPositionArray
    .BYTE $0C,$0D,$0E,$0F,$0F,$0E,$04
    .BYTE $04,$04,$04,$04,$04,$04,$05
    .BYTE $06,$07,$08,$09,$0A,$0B,$0C,$0D
    .BYTE $0D,$0D,$0D,$07,$09,$09,$0A,$0B
    .BYTE $00,$00,$00,$00,$00,$00,$00,$00
    .BYTE $00,$00,$00,$00,$00,$00,$00,$00
    .BYTE $00,$00,$00,$00,$00,$00,$00,$00
    .BYTE $00,$00,$00,$00,$00,$00,$00,$00
currentColorIndexArray
    .BYTE $FF,$FF,$FF,$FF,$FF,$FF,$FF
    .BYTE $FF,$FF,$FF,$FF,$FF,$FF,$FF
    .BYTE $FF,$FF,$FF,$FF,$FF,$FF,$FF
    .BYTE $FF,$FF,$FF,$FF,$FF,$FF,$FF
    .BYTE $FF,$FF,$FF,$FF,$FF,$FF,$FF
    .BYTE $00,$00,$00,$00,$00,$00,$00,$00
    .BYTE $00,$00,$00,$00,$00,$00,$00,$00
    .BYTE $00,$00,$00,$00,$00,$00,$00,$00
    .BYTE $00,$00,$00,$00,$00,$00,$00,$00
initialSmoothingDelayForStep
    .BYTE $0C,$0C,$0C,$0C,$0C,$0C,$0C,$0C
    .BYTE $0C,$0C,$0C,$0C,$0C,$0C,$0C,$0C
    .BYTE $0C,$0C,$0C,$0C,$0C,$0C,$0C,$0C
    .BYTE $0C,$0C,$0C,$0C,$0C,$0C,$0C,$0C
    .BYTE $00,$00,$00,$00,$00,$00,$00,$00
    .BYTE $00,$00,$00,$00,$00,$00,$00,$00
    .BYTE $00,$00,$00,$00,$00,$00,$00,$00
    .BYTE $00,$00,$00,$00,$00,$00,$00,$00
smoothingDelayForStep
    .BYTE $04,$07,$01,$02,$03,$06,$07,$06
    .BYTE $0C,$02,$03,$06,$07,$01,$02,$02
    .BYTE $04,$04,$07,$01,$02,$03,$06,$07
    .BYTE $0C,$02,$03,$02,$03,$07,$01,$02
    .BYTE $00,$00,$00,$00,$00,$00,$00,$00
    .BYTE $00,$00,$00,$00,$00,$00,$00,$00
    .BYTE $00,$00,$00,$00,$00,$00,$00,$00
    .BYTE $00,$00,$00,$00,$00,$00,$00,$00
```

Listing 3.9: The pixel buffers each 64 bytes long though only 32 bytes are used in this version thanks to MAX\_INDEX\_VALUE being set to \$1F (32).

**Lines 385-429.** `pixelXPositionArray`: Five arrays of 64 bytes each, though only the first 32 in each are actually used. This fact is determined by the value of \$1F (32) in `MAX_INDEX_VALUE`. We see this being enforced on the next page in `MainPaintLoop`. Defining 64 bytes, but only using 32 in the end, suggests Minter had to play around with the performance limitations of the C64. Since each byte represents a pattern that must be drawn and painted at a given position in the screen, using all 64 by setting allocated `MAX_INDEX_VALUE` to \$3F was not necessarily prohibitive but certainly resulted in a more sluggish effect.

We've seen the first three arrays in use already when we looked at the core pattern painting routine. Each byte in the array refers to a distinct step in the paint sequence. `pixelXPositionArray` and `pixelYPositionArray` together define a location on the screen that is treated as the origin for painting and drawing an instance of the current pattern. So for example \$0F,\$0C given by the first byte in each represents column 16 (\$0F) and row 14 (\$0C).

The value in the corresponding position of `currentColorIndexArray` represents the color to 'start from'.

We'll cover the behaviour defined by the two 'smoothing delay' arrays in [dials buttons switches](#). But if we look at the effect of setting this parameter to its lowest and highest possible value below, the difference in behaviour is not extreme. It's almost quite subtle. With a higher setting the pattern evolves in a more drawn out fashion - as though a higher value lets Psychedelia linger a little longer on the current state of the pattern rather than immediately evolving it to the next state.

But we're getting ahead of ourselves, we'll come back to this behaviour in a later chapter.

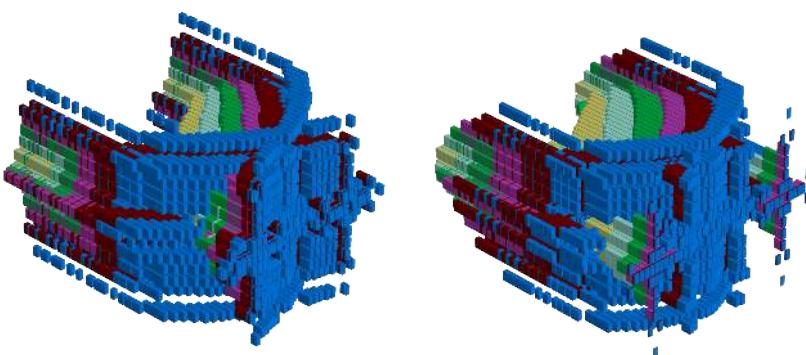


Figure 3.4: Effect of low and high values for Smoothing Delay

let's pretend we can read code

---

### Lines 450-482. LaunchPsychedelia

```
; -----
; LaunchPsychedelia
; -----
LaunchPsychedelia
    JSR ReinitializeSequences
    JSR SetUpInterruptHandler

MainPaintLoop
    ; Part 1: Check if it's time to paint.
    INC currentPositionInArrays
    LDA currentPositionInArrays
    AND MAX_INDEX_VALUE
    STA currentPositionInArrays
    TAX
    DEC smoothingDelayForStep,X
    BNE GoBackToStartOfLoop

    ; Part 2: Check if there's a color to paint.
    LDA initialSmoothingDelayForStep,X
    STA smoothingDelayForStep,X

    LDA currentColorIndexArray,X
    CMP #$FF
    BEQ GoBackToStartOfLoop

    ; Part 3: Actually do a paint.
ActuallyDoAPaint
    STA currentValueInColorIndexArray
    LDA pixelXPositionArray,X
    STA pixelXPosition
    LDA pixelYPositionArray,X
    STA pixelYPosition

    JSR PaintStructureAtCursorPosition

    LDX currentPositionInArrays
    DEC currentColorIndexArray,X

GoBackToStartOfLoop
    JMP MainPaintLoop
```

Listing 3.10: The game's main loop.

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
\$20	0	0	1	0	0	0	0	0
\$1F	0	0	0	1	1	1	1	1
Result	0	0	0	0	0	0	0	0

AND'ing \$20 and \$1F gives \$00, ensuring we always loop between 0 and 32.

**Lines 450-482.** LaunchPsychedelia: Finally a routine that resembles getting down to business. We start by initializing the arrays we saw on the previous page to all zeros in `ReinitializeSequences`. Next we set up the interrupt handler, which will be responsible for processing user input. We'll take a look at this routine in more detail later on.

For now, what follows is the game's core loop. This runs forever, round and round, repeating all the steps between the `MainPaintLoop` and `GoBackToStartOfLoop` labels.

There are three parts to the code inside this loop.

The first is visited on every journey through the loop and increments the value stored in `currentPositionInArrays`. It keeps this value looping between 0 and 32 by always AND'ing it with `MAX_INDEX_VALUE`.

```
LDA currentPositionInArrays  
AND MAX_INDEX_VALUE  
STA currentPositionInArrays
```

This AND operation ensures that once it reaches 33, for example, the result goes back to zero. (See opposite page.)

Now that it has a new value in `currentPositionInArrays` it transfers it to the X register and uses the X as an index into the `smoothingDelayForStep` array (and all the other arrays) which we saw defined on the previous page. It decrements the byte at this index in the `smoothingDelayForStep` array and if it has reached zero proceeds to Part 2 of the loop, otherwise it jumps execution to `GoBackToStartOfLoop`, which itself jumps execution back to the top again at `MainPaintLoop`. Back on the merry-go-round we go.

If we get to Part 2 this time round we first reset the value in the `framesRemainingToNextPaintForStep` array with an initial value for it kept in the corresponding location of `initialFramesRemainingToNextPaintForStep`. We next check if we've run out of colors to paint for this step in the sequence: if the value at the index position in `currentColorIndexArray` has already reached \$FF (because it was decremented below zero and so cycled back to \$FF) then there's nothing to paint and we jump back to the top again via `GoBackToStartOfLoop`. No colors means no painting after all!

If there actually, finally, is something to do for this position in the sequence we can pass to Part 3. Here we load up the X and Y position to use from our `pixelXPositionArray` and `pixelYPositionArray` and do the actual painting work we covered in our previous chapter, all of which happens in `PaintStructureAtCurrentPosition`. As we saw this routine is the top of the tree for the pixel painting routines in Psychedelia, and all it needed from us here was an X and Y position as well as a color loaded from the `currentColorIndexArray`. Equipped with that it can paint anything.

**Lines 217-285.** PaintStructureAtCurrentPosition

```
NUM_ARRAYS = $07
;-----
; PaintStructureAtCurrentPosition
;-----
PaintStructureAtCurrentPosition
    JSR PaintPixelForCurrentSymmetry

    LDY #$00
    LDA currentValueInColorIndexArray
    CMP #NUM_ARRAYS
    BNE CanLoopAndPaint
    RTS

CanLoopAndPaint
    LDA #NUM_ARRAYS
    STA countToMatchCurrentIndex

    LDA pixelXPosition
    STA initialPixelXPosition
    LDA pixelYPosition
    STA initialPixelYPosition

LineReadLoop
    ; See next page for the contents of this loop.
    JMP LineReadLoop

RestorePositionsAndReturn
    LDA initialPixelXPosition
    STA pixelXPosition
    LDA initialPixelYPosition
    STA pixelYPosition
    RTS
```

Listing 3.11: The routine responsible for orchestrating the pattern painting.

**Lines 217-285.** PaintStructureAtCursorPosition: To understand how this routine and the ones it calls on the subsequent pages hang together you should refer back to the previous chapter '[soul of a small light machine](#)'. Here we will limit ourselves to explaining some of the detailed points of the code.

The meat of this routine occurs in the LineReadLoop section on the next page but here we can see the gatekeeping and bookkeeping activities of the code before and after that central section.

The first call to PaintPixelForCurrentSymmetry paints a pixel at the current position given by pixelXPosition and pixelYPosition. The -CurrentSymmetry part ensures that extra pixels are projected along the axes given by the current symmetry setting. In our listing this is the X-axis, ensuring we always paint two patterns. We'll look at this in a little more detail later.

Next we figure out if we should just return early from this routine and do nothing further. This will be the case if currentValueInColorIndexArray is still at the same value as NUM\_ARRAYS (7). If it is we return from the routine straight away (RTS) and MainPaintLoop will decrement it the next time around for this position in the sequence and we can paint an actual color.

You cans see that before and after the LineReadLoop section we're storing the values in pixelX/YPosition in initialPixelX/YPosition and then restoring them when the loop is over. This is because LineReadLoop will frequently overwrite the values as it iterates through the positions given in the pattern array starOneX/YPosArray.

**Friendly Blue Learning Box: CMP**

CMP compares the value given as an argument with the value in the A and returns a 1 if they match and a zero if they do not. So if we have loaded \$02 to A and do CMP #NUM\_ARRAYS (where NUM\_ARRAYS = \$07) it will return a zero, as they do not match. Our subsequent statement BNE CanLoopAndPaint inspects this result: if they are not equal execution will jump to CanLoopAndPaint otherwise it will continue on to RTS and return from the routine without doing anything further.

let's pretend we can read code

---

### Lines 239-278. LineReadLoop

```
LineReadLoop
    LDA initialPixelXPosition
    CLC
    ADC starOneXPosArray,Y
    STA pixelXPosition

    LDA initialPixelYPosition
    CLC
    ADC starOneYPosArray,Y
    STA pixelYPosition

    TYA
    PHA

    JSR PaintPixelForCurrentSymmetry

    PLA
    TAY
    INY

    LDA starOneXPosArray,Y
    CMP #$55
    BNE LineReadLoop

    DEC countToMatchCurrentIndex
    LDA countToMatchCurrentIndex
    CMP currentValueInColorIndexArray
    BEQ RestorePositionsAndReturn
    CMP #$01
    BEQ RestorePositionsAndReturn

    INY
    JMP LineReadLoop
```

Listing 3.12: The core pattern-painting loop.

```
starOneXPosArray
    .BYTE $00,$01,$01,$01,$00,$FF,$FF,$FF,$55      ;      5
    .BYTE $00,$02,$00,$FE,$55                          ;
    .BYTE $00,$03,$00,$FD,$55                          ;      4 4
    .BYTE $00,$04,$00,$FC,$55                          ;      3
    .BYTE $FF,$01,$05,$05,$01,$FF,$FB,$FB,$55        ;      2
    .BYTE $00,$07,$00,$F9,$55                          ;      1
    .BYTE $55                                         ;      4 000 4
starOneYPosArray
    .BYTE $FF,$FF,$00,$01,$01,$01,$00,$FF,$55        ;      4 000 4
    .BYTE $FE,$00,$02,$00,$55                          ;      1
    .BYTE $FD,$00,$03,$00,$55                          ;      2
    .BYTE $FC,$00,$04,$00,$55                          ;      3
    .BYTE $FB,$FB,$FF,$01,$05,$05,$01,$FF,$55        ;      4 4
    .BYTE $F9,$00,$07,$00,$55                          ;      5
    .BYTE $55                                         ;      5
```

**Lines 239-278.** LineReadLoop: A better name for this loop might be ThisIsWhereThe-PsychedeliaHappens. As I keep saying (if only you would listen), we unpicked the mechanics of the PaintStructureAtCurrentPosition routine that this loop is a part of in the previous chapter '[soul of a small light machine](#)'. This little routine reads off lines from our pattern data structure and paints them to the screen by calling PaintPixelForCurrentSymmetry. It doesn't do them all at once of course. The number of lines it reads off and converts to stages of the pattern on the screen is controlled by currentValueInColorIndexArray. So let's take a look at how it works.

Our index into the pattern array is stored in Y, this is what will enable us to read in the data in each line and detect the segment breaks in the pattern given by \$55 . Y starts out at zero and gets incremented at each pass through the loop. So the first paragraph adds the value at index 0 in starOneXPosArray in this pass to initialPixelXPosition and stores it in pixelXPosition. That value is \$00, on the next it will be \$01, on the pass after that it will be \$01 again, as these are the values it is reading from the array. You can hopefully see how the LDA, ADC, and STA instructions are doing that but you might wonder what the CLC is about. This clears the CPU's 'Carry Bit' so that the addition performed by ADC doesn't inadvertently include any carries from previous addition operations in its result.

The next paragraph does the same for pixelYPosition using the starOneXPosArray array. But the paragraph after that does something that requires explanation. It transfers the current value in the Y register to the A register using the instruction TYA. Then it pushes the A register on to a thing called the 'Stack' using the instruction PHA. The 'Stack' is a stack of values that resembles a stack of plates: you add things to the top of it and take things from the top of it. So when you take something from the stack using the complementary instruction PLA you are taking what was last put there (and not pulled off it before you).

So the TYA, PHA, PLA, TAY sequence of instructions that surrounds our call to PaintPixelForCurrentSymmetry has the simple effect of ensuring that we keep our current value of Y unmolested by whatever PaintPixelForCurrentSymmetry gets up to, because that can (and does) include populating and using the Y register for other stuff. As you can imagine this is a common technique in a CPU where there are only three free variables to play with (X,Y,A) and you have more than three things to worry about!

Next, after incrementing Y (INY) we check the next value in starOneXPosArray. If it's \$55 then we've reached the end of a line the pattern and must check whether we've read all the lines in the pattern we need to for this round. If we have, then we return by calling RestorePositionsAndReturn, otherwise we increment Y again to read in the first value of the next line and continue looping.

let's pretend we can read code

---

**Lines 332-382.** PaintPixelForCurrentSymmetry

```
currentSymmetrySettingForStep    .BYTE $01

PaintPixelForCurrentSymmetry
; First paint the normal pattern without any
; symmetry.
LDA pixelXPosition
PHA
LDA pixelYPosition
PHA
JSR PaintPixel

LDA currentSymmetrySettingForStep
BNE HasSymmetry

CleanUpAndReturnFromSymmetry
PLA
STA pixelYPosition
PLA
STA pixelXPosition
RTS

HasSymmetry
; Has a pattern to paint on the X axis
; symmetry so prepare for that.
LDA #NUM_COLS
SEC
SBC pixelXPosition
STA pixelXPosition

JSR PaintPixel

LDA currentSymmetrySettingForStep
CMP #$01
BEQ CleanUpAndReturnFromSymmetry

LDA #NUM_ROWS
SEC
SBC pixelYPosition
STA pixelYPosition
JSR PaintPixel

PLA
TAY
PLA
STA pixelXPosition
TYA
PHA
JSR PaintPixel
PLA
STA pixelYPosition
RTS
```

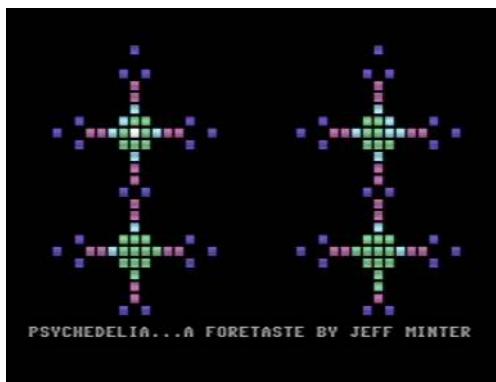
Listing 3.13: Choosing what to paint based on the current symmetry setting.

**Lines 332-382.** `PaintPixelForCurrentSymmetry`: This routine is concerned entirely with setting up the values of `pixelXPosition` and `pixelYPosition` correctly for calling the routine `PaintPixel`. The reason this is necessary is that the commercial version of Psychedelia has up to 4 different arrangements or 'symmetries' for painting the chosen pattern. The simplest arrangement is just to paint one copy of the pattern and be done. The first paragraph here takes care of that, no matter what arrangement we're painting we're always painting the pattern at least once.

While the commercial version has at least three others to worry about, and offers the user a way of switching between each, the listing edition hard-codes a single behaviour of always painting the pattern twice, reflected on the X-axis in a kind of mirror arrangement. This behaviour is defined by the value in `currentSymmetrySettingForStep` and as we can see opposite it is hard-coded to \$01. Because it is set to a non-zero value, `BNE HasSymmetry` always returns true and execution jumps to `HasSymmetry`.

`HasSymmetry` calculates the reflected position on the X-axis for the pattern by simply subtracting the current value of `pixelXPosition` from the total number of columns in the screen (which is 40). This has the neat effect of ensuring the reflected position is always exactly opposite to the original pattern.

The paragraph after `PaintPixel` always returns from the routine because as we saw `currentSymmetrySettingForStep` is hardcoded to \$01. So the code after it is never executed and is redundant here. What it does is draw a full four-pattern arrangement with reflections along the X and Y axes. If we set `currentSymmetrySettingForStep` to \$02 and run the listing we can see this in action.



The display with `currentSymmetrySettingForStep` set to \$02

let's pretend we can read code

---

#### Lines 164-214. PaintPixel

```
presetColorValuesArray
;          0   6   2   4   5   3   7   1
; .BYTE BLACK ,BLUE ,RED ,PURPLE ,GREEN ,CYAN ,YELLOW ,WHITE
;-----
; PaintPixel
;-----
PaintPixel
    LDA pixelXPosition
    AND #$80 ; Detect if has moved off left of screen
    BNE ReturnEarly
    LDA pixelXPosition
    CMP #NUM_COLS
    BPL ReturnEarly
    LDA pixelYPosition
    AND #$80 ; Detect if has moved off top of screen.
    BNE ReturnEarly
    LDA pixelYPosition
    CMP #NUM_ROWS
    BPL ReturnEarly

    JSR LoadXAndYPosition
    LDA (currentLineForPixelInColorRamLoPtr),Y
    AND #COLOR_MAX

    LDX #$00
CheckPresetsLoop
    CMP presetColorValuesArray,X
    BEQ MaybePaintPixel
    INX
    CPX #COLOR_MAX + 1
    BNE CheckPresetsLoop

MaybePaintPixel
    TXA
    STA indexOfCurrentColor
    LDX currentValueInColorIndexArray
    INX
    CPX indexOfCurrentColor
    BEQ ActuallyPaintPixel
    BPL ActuallyPaintPixel
    RTS

ActuallyPaintPixel
    LDX currentValueInColorIndexArray
    LDA presetColorValuesArray,X
    STA (currentLineForPixelInColorRamLoPtr),Y
    RTS
```

Listing 3.14: Where the painting is actually done.

**Lines 164-214.** PaintPixel: The first paragraph searches for a reason to ReturnEarly. The first check, whether the pixel has moved beyond the leftmost position on the screen, is a good example of how assembly language involves logic that might be unintuitive at first. In layman's terms we just want to ask is pixelXPosition less than zero. One neat way of doing this in our situation is to check if decrementing past \$00 has resulted in a value of \$FF (Decrementing zero 'wraps around' to \$FF). Since the rightmost position never exceeds \$40 this means we can just check if the leftmost bit on the value is set or not: if it is then we've gone below zero. So an AND \$80 check will do the job - if it returns true then we've gone below zero:

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
\$FF	1	1	1	1	1	1	1	1
\$80	1	0	0	0	0	0	0	0
Result	1	0	0	0	0	0	0	0

AND'ing \$FF and \$80 gives \$80, a non-zero value - so the AND returns true.

If we get past these checks we can prepare ourselves for writing to the screen by calling LoadXAndYPosition. As we can see below this loads the Y position to the X register and the X position to the Y register. There is a reason that this is what seems like the wrong way round - the use we intend to make of each register is constrained by the rules of assembly language.

```
LoadXAndYPosition
LDX pixelYPosition
LDA colorRAMLineTableLoPtrArray,X
STA currentLineForPixelInColorRamLoPtr
LDA colorRAMLineTableHiPtrArray,X
STA currentLineForPixelInColorRamHiPtr
LDY pixelXPosition
RTS
```

Both X and Y can be used as offsets when referencing addresses in memory, but the specific use we need to make of each means that we cannot store our X position in X and our Y position in Y, which is a shame!

**Lines 210-214.** ActuallyPaintPixel: This is the rubber hitting the road. We're writing data to the screen. Recall that our colorRAMLineTableLoPtrArray array is an array of 25 screen addresses each one pointing to the start of a row. In LoadXAndYPosition we have set up currentLineForPixelInColorRamLoPtr and currentLineForPixelInColorRamHiPtr to contain the address of the pixel we want to paint. So for example if the address of Color RAM we want to populate is \$D810 then currentLineForPixelInColorRamLoPtr will contain \$10 and currentLineForPixelInColorRamHiPtr will contain \$D8.

let's pretend we can read code

---

#### Lines 491-506. SetUpInterruptHandler

```
SetUpIntterruptHandler
    SEI
    LDA #<MainInterruptHandler
    STA $0314      ;IRQ
    LDA #>MainInterruptHandler
    STA $0315      ;IRQ

    LDA #$0A
    STA cursorXPosition
    STA cursorYPosition

    LDA #$01
    STA $D015      ;Sprite display Enable
    STA $D027      ;Sprite 0 Color
    CLI
    RTS
```

Listing 3.15: Setting up our Interrupt Handler

**Lines 491-506.** `SetUpInterruptHandler`: What is an Interrupt Handler when it's at home and why are we setting one up?

Recall that this routine was called immediately before we entered the main loop. It is only ever called once:

```
LaunchPsychedelia
    JSR ReinitializeSequences
    JSR SetUpInterruptHandler
```

In plain English, an 'interrupt' is when the CPU periodically stops what ever its doing and does some other pre-defined amount of work instead. There are a few such interrupts supported by the 6502 CPU but the one we make use of here is called the IRQ or 'Hardware Interrupt'. This interrupt runs 60 times a second (that's right, 60 times a second) and already has some work assigned to it by the Commodore 64, a set of tasks such as checking keyboard and joystick input that is defined at address \$EA31. This doesn't have a proper name of it's own so we will call it `DEFAULT_IRQ_HANDLER`. (The routine called by an interrupt is often referred to as an 'interrupt vector'.)

What we can do is tag some work of our own onto this, we still need to call the routine at \$EA31 but we can do some of our own stuff before it. The way to do this is to load the address of our own routine, which we call `MainInterruptHandler`, to the address that the 6502 CPU consults each time it interrupts its work. We replace \$EA31 at that address with the address of `MainInterruptHandler`.

```
SEI
LDA #<MainInterruptHandler
STA $0314      ;IRQ
LDA #>MainInterruptHandler
STA $0315      ;IRQ
...
CLI
```

Notice that because the address is two bytes long we must split it out, like we did for our Colour RAM address earlier, into two bytes and store each one separate in the consecutive address locations \$0314 and \$0315, which store the second and first byte of the interrupt-handler address respectively.

The `SEI` instruction at the beginning of this routine and the `CLI` instruction at the end tell the CPU that we are updating the interrupt vector. The `SEI` ensures that no further interrupts happen while we are doing this, and the `CLI` gives the CPU the all-clear to start honouring interrupts again now that we've updated the routine it should call when one happens.

let's pretend we can read code

---

#### Lines 491-506. MainInterruptHandler

```
MainInterruptHandler
    DEC countStepsBeforeCheckingJoystickInput
    BEQ PaintCursorAndCheckJoystickInput
    JMP DEFAULT_IRQ_HANDLER

PaintCursorAndCheckJoystickInput
    LDA #$02
    STA countStepsBeforeCheckingJoystickInput
    LDA #$00
    STA currentColorToPaint
    JSR PaintCursorPosition

CheckIfJoystickMovedUpOrDown
    LDA $DC00      ; CIA1: Data Port Register A
    AND #$03
    CMP #$03
    BEQ CheckIfPlayerPressedLeftOrRight

    CMP #$02
    BEQ PlayerHasPressedDown

PlayerHasPressedUp
    ; Player has pressed up. Incrementeent up two lines
    ; so that when we decrement down one, we're still
    ; one up!
    INC cursorYPosition
    INC cursorYPosition

PlayerHasPressedDown
    DEC cursorYPosition
    LDA cursorYPosition
    CMP #$FF
    BNE CheckIfCursorAtBottom

WrapCursorToBottom
    ; Cursor has reached the top of the screen, so loop
    ; around to bottom.
    LDA #NUM_ROWS - 1
    STA cursorYPosition
    JMP CheckIfPlayerPressedLeftOrRight

CheckIfCursorAtBottom
    CMP #NUM_ROWS
    BNE CheckIfPlayerPressedLeftOrRight
    ; Cursor has reached the bottom of the screen, so loop
    ; around to top
    LDA #$00
    STA cursorYPosition
```

Listing 3.16: This is our interrupt handler it runs 60 times a second so has to be fast.

**Lines 491-506.** MainInterruptHandler: You don't have to look too hard at our interrupt handler to divine what it is mainly concerned with: have you moved the joystick or pressed fire? The routine is long enough that we need to spread it over 3 pages here. The section on the opposite page gets as far as reacting to any up or down movement in the joystick and the beginnings of the left/right checks.

Joystick input is read from the CIA1 Data Port Register. If the player interacts with the joystick at all it will show up in the byte at address \$DC00. Only 5 of the 8 bits are used, 4 for each direction and 1 for the fire button. Counter-intuitively, the bits are set to 1 by default, so you can tell if a button has been pressed or the joystick moved if one of the appropriate bits is 0 rather than 1.

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
				Fire	Right	Left	Down	Up
\$FD	1	1	1	1	1	1	0	1

The contents of \$DC00 when the player has pressed down on the joystick.

**Lines 491-506.** CheckIfJoystickMovedUpOrDown: If we AND the value in \$DC00 (for example, \$FE) with \$03 it will tell us if up or down have been pressed if the result is not equal to \$03. This is because we expect both bits to be 1 if up or down have not been pressed, and the AND ensures we are only comparing those two bits when we perform CMP #\$03.

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
				Fire	Right	Left	Down	Up
\$FE	1	1	1	1	1	1	1	0
\$03	0	0	0	0	0	0	1	1
Result	0	0	0	0	0	0	1	0

AND'ing \$FE and \$03 gives \$02, telling us that 'Up' has been pressed.

Once we have established this we can increment and decrement cursorYPosition as appropriate and also check if we've reached the top or bottom of the screen.

**Lines 300-313.** CheckIfPlayerPressedLeftOrRight

```
CheckIfPlayerPressedLeftOrRight
    LDA $DC00      ; CIA1: Data Port Register A
    AND #$0C
    CMP #$0C
    BEQ CheckIfPlayerPressedFire

    CMP #$08
    BEQ PlayerHasPressedLeft

PlayerHasPressedRight
; Player has pressed right.
    INC cursorXPosition
    INC cursorXPosition

PlayerHasPressedLeft
    DEC cursorXPosition
    LDA cursorXPosition
    CMP #$FF
    BNE CheckIfCursorAtExtremeRight

WrapCursorToExtremeRight
    LDA #NUM_COLS - 1
    STA cursorXPosition
    JMP CheckIfPlayerPressedFire

CheckIfCursorAtExtremeRight
    CMP #NUM_COLS
    BNE CheckIfPlayerPressedFire
    LDA #$00
    STA cursorXPosition
```

Listing 3.17: Second part of the Interrupt Handler.

**Lines 300-313.** CheckIfPlayerPressedLeftOrRight: As in the previous page, which was checking for up/down movement, here we're checking for left/right movement in the joystick and updating the cursorXPosition if required.

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
				Fire	Right	Left	Down	Up
\$F7	1	1	1	1	0	1	1	1
\$0C	0	0	0	0	1	1	0	0
Result	0	0	0	0	0	1	0	0

AND'ing \$F7 and \$0C gives \$04, telling us that 'Left' has been pressed.

let's pretend we can read code

---

### Lines 300-313. CheckIfPlayerPressedFire

```
CheckIfPlayerPressedFire
    LDA $DC00      ;CIA1: Data Port Register A
    AND #$10
    BEQ PlayerHasPressedFire

PlayerHasntPressedFire
    LDA #$00
    STA stepsSincePressedFire
    JMP DrawCursorAndReturnFromInterrupt

PlayerHasPressedFire
MaybeUpdatePixelBuffers
    LDA unusedBurstMode
    BEQ UpdateArrayIndex

    LDA stepsSincePressedFire
    BNE DrawCursorAndReturnFromInterrupt
    INC stepsSincePressedFire

UpdateArrayIndex
    INC indexIntoArrays
    LDA indexIntoArrays
    AND MAX_INDEX_VALUE
    STA indexIntoArrays

CheckCurrentIndexInBuffers
    TAX
    LDA currentColorIndexArray,X
    CMP #$FF
    BNE DrawCursorAndReturnFromInterrupt

UpdateBuffersWithCursorPosition
    LDA cursorXPosition
    STA pixelXPositionArray,X
    LDA cursorYPosition
    STA pixelYPositionArray,X
    LDA #COLOR_MAX
    STA currentColorIndexArray,X

    LDA smoothingDelay
    STA initialSmoothingDelayForStep,X
    STA smoothingDelayForStep,X
```

Listing 3.18: Third part of the Interrupt Handler.

**Lines 300-313.** CheckIfPlayerPressedFire: This section of the routine could be much shorter. This is because neither burstModeEnabled nor stepsSincePressedFire are actually used. The code that refers to them seems to be concerned with supporting an unused feature toggled on/off by the variable burstModeEnabled and since it has been hardcoded to \$00 the section could be stripped back to this instead:

```

CheckIfPlayerPressedFire
    LDA $DC00      ;CIA1: Data Port Register A
    AND #$10
    BNE DrawCursorAndReturnFromInterrupt

PlayerHasPressedFire
    INC indexIntoArrays
    LDA indexIntoArrays
    AND MAX_INDEX_VALUE
    STA indexIntoArrays

CheckCurrentIndexInBuffers
    TAX
    LDA currentColorIndexArray,X
    CMP #$FF
    BNE DrawCursorAndReturnFromInterrupt

UpdateBuffersWithCursorPosition
    LDA cursorXPosition
    STA pixelXPositionArray,X
    LDA cursorYPosition
    STA pixelYPositionArray,X
    LDA #COLOR_MAX
    STA currentColorIndexArray,X

    LDA smoothingDelay
    STA initialSmoothingDelayForStep,X
    STA smoothingDelayForStep,X

```

This refactored version makes it much easier to see what's going on: if the player has pressed fire then we increment our index into the pixel arrays and load the cursor's position to the index's position in pixelXPositionArray/pixelYPositionArray and initial values to the arrays managing pixel color and smoothing delay.

If the player hasn't pressed fire or if our current index to the arrays points to a position that is already being used then we skip to just painting the cursor and exiting. Note that our heuristic for deciding whether the current position in the arrays is in use is to make sure that the value at indexIntoArrays in currentColorIndexArray is \$FF - after all, this will only be the case if we haven't already used it (or used it up).

let's pretend we can read code

---

#### Lines 300-313. DrawCursorAndReturnFromInterrupt

```
DrawCursorAndReturnFromInterrupt
    JSR LoadXAndYOfCursorPosition
    LDA (currentLineForCursorInColorRamLoPtr),Y
    AND #COLOR_MAX
    STA lastColorPainted
    LDA #WHITE
    STA currentColorToPaint
    JSR PaintCursorAtCursorPosition
    JMP RETURN_FROM_INTERRUPT
```

Listing 3.19: Paint the cursor and call the system default interrupt handler RETURN\_FROM\_INTERRUPT

#### Lines 300-313. LoadXAndYOfCursorPosition

```
for painting the cursor.]
LoadXAndYOfCursorPosition
    LDX cursorYPosition
    LDA colorRAMLineTableLoPtrArray,X
    STA currentLineForCursorInColorRamLoPtr
    LDA colorRAMLineTableHiPtrArray,X
    STA currentLineForCursorInColorRamHiPtr
    LDY cursorXPosition
    RTS
```

#### Lines 300-313. PaintCursorAtCursorPosition

```
PaintCursorAtCursorPosition
    JSR LoadXAndYOfCursorPosition
    LDA currentColorToPaint
    STA (currentLineForCursorInColorRamLoPtr),Y
    RTS
```

Listing 3.20: Paint the cursor.

**Lines 300-313.** `DrawCursorAndReturnFromInterrupt`: Drawing the cursor should be a relatively simple matter of converting the cursor position stored in `cursorXPosition` and `cursorYPosition` to a position on the screen. Again we have some redundant code to distract us: there is no good reason to determine the `lastColorPainted` since we don't use it for anything (and this code is not present in the commercial edition). So our simplified version of `DrawCursorAndReturnFromInterrupt` looks as follows:

```
DrawCursorAndReturnFromInterrupt
    JSR LoadXAndYOfCursorPosition
    LDA #WHITE
    STA currentColorToPaint
    JSR PaintCursorAtCursorPosition
    JMP RETURN_FROM_INTERRUPT
```

The real work is clearly done by our two helper routines `LoadXAndYOfCursorPosition` and `PaintCursorAtCursorPosition`.

Let's imagine our cursor has an `cursorYPosition` of 5 and an `cursorXPosition` of 10. This means in `LoadXAndYOfCursorPosition` we will retrieve the elements with index 5 in `colorRAMLineTableHiPtrArray` and `colorRAMLineTableLoPtrArray`. So consulting our table below we see the values we end up with in each highlighted in red:

Element	colorRAMLineTable HiPtrArray	colorRAMLineTable LoPtrArray	Address
0	\$D8	\$00	\$D800
1	\$D8	\$28	\$D828
2	\$D8	\$50	\$D850
3	\$D8	\$78	\$D878
4	\$D8	\$A0	\$D8A0
5	\$D8	\$C8	\$D8C8

With an address of \$D8C8 we now know where in Color RAM our line starts. We can now load our `cursorXPosition` to the Y register:

```
LDY cursorXPosition
```

And use that to paint the cursor position white. The address we want to paint is `(currentLineForCursorInColorRamLoPtr) + Y`, i.e.  $\$D8C8 + \$0A = \$D8D2$ . We express this in assembly as follows:

```
LDA currentColorToPaint
STA (currentLineForCursorInColorRamLoPtr),Y
```



# increasing the dose

## Jeffrey Says



I felt that something so basic and lovely deserved more than just being another thing to be sold and profited from. I actually gave that first algorithm away in listing form to a computer magazine. But my parents argued successfully that there was no shame in making some money from it, and so in due course I created a somewhat expanded version with more patterns and control options, and that is what was released as Psychedelia.

The magazine listing was compact enough to reduce to two pages. That is not the case with the commercial edition of Psychedelia which runs to almost sixteen thousand lines of code and creates a binary of 10,000 or so bytes. This is larger, but still relatively small. Since the rest of this book will anatomize the interesting features of the commercial game it makes sense for us to begin with a similar attempt at providing an atlas to the code base as a whole.

As before, we're going to set out the full listing so that we get an initial sense of the size of the program and an understanding of the order in which the features of the game and the data that supports them are implemented.

# increasing the dose

```

; == $0801
;-----
; Start program at InitializeProgram
; SYS 2064 ($0810)
;-----
; .BYTE $0B,$0B
; .BYTE $C1,$07
; .BYTE $9E
; .BYTE $32,$30,$36,$34
; .BYTE $00
; .BYTE $00,$00
; .BYTE $F9,$02,$F9

;-----
; InitializeProgram
;-----
InitializeProgram
LDA #$00
STA $D020
STA $D021

JSR MovePresetDataIntoPosition

STA previousIndexToPixelBuffers

LDA #COLOR_RAM
STA colorRamHiPtr
LDA #COLOR_RAM
STA colorRamLoPtr

LDX #$00
InitColorRamTableLoop
LDA colorRamHiPtr
STA colorRAMLineTableHiPtrArray,X
LDA colorRamLoPtr
STA colorRAMLineTableLoPtrArray,X
CLC
ADC #NUM_COLS
STA colorRamLoPtr
LDA colorRamHiPtr
ADC #$00
STA colorRamHiPtr
IMX
CPX #NUM_ROWS + 1
BNE InitColorRamTableLoop

LDA #$80
STA $D021

LDA #$15
STA $D018

JSR ROM_10INIT
JSR InitializeDynamicStorage
JMP launchPchedelis

colorRAMLineTableLoPtrArray
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
colorRAMLineTableHiPtrArray
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00

;-----
; InitializeScreenWithInitCharacter
;-----
InitializeScreenWithInitCharacter
LDX $000

currentPixel = **$01
InitScreenLoop
LDA #SCF
STA SCREEN_RAM + $0000,X
STA SCREEN_RAM + $0100,X
STA SCREEN_RAM + $0200,X
STA SCREEN_RAM + $0200,X
LDA #$00
STA COLOR_RAM + $0000,X
STA COLOR_RAM + $0100,X
STA COLOR_RAM + $0200,X
STA COLOR_RAM + $0200,X
DEX
BNE InitScreenLoop
RTS

presKeyCodes
.BYTE KEY_LEFT,,KEY_1,KEY_2,KEY_3
.BYTE KEY_4,KEY_5,KEY_6,KEY_7
.BYTE KEY_8,KEY_9,KEY_0,KEY_PLUS
.BYTE KEY_MINUS,KEY_FOUND
.BYTE KEY_CLR_HOME,KEY_INST_DEL

;-----
; LoadXAndYPosition
;-----
LoadXAndYPosition
LDX pixelYPosition
LDA colorRAMLineTableLoPtrArray,X
STA currentLineInColorRamLoPtr2
LDA colorRAMLineTableHiPtrArray,X
STA currentLineInColorRamHiPtr2
LDY pixelXPosition
RTS

ReturnEarlyFromroutine
RTS

; == $0802
;-----
; PaintPixel
;-----
PaintPixel
LDA pixelXPosition
AND #$80
BNE ReturnEarlyFromroutine
LDA pixelXPosition
CMP #NUM_COLS
BPL ReturnEarlyFromroutine
LDA pixelYPosition
AND #$80
BNE ReturnEarlyFromroutine
LDA pixelYPosition
CMP #NUM_ROWS
BPL ReturnEarlyFromroutine

JSR LoadAndYPosition
LDA skipPixel
BNE ActuallyPaintPixel

LDA (currentLineInColorRamLoPtr2),Y
AND #COLOR_MAX

LDY #$00
GetIndexInPresetsLoop
CMP presetColorValuesArray,X
BEO FoundMatchingIndex
INX
CPX #COLOR_VALUES_ARRAY_LEN
BNE GetIndexInPresetsLoop

FoundMatchingIndex
TIA
STA indexOfCurrentColor
LDX currentValueInColorIndexArray
INX
CPX indexOfCurrentColor
BEO ActuallyPaintPixel
BPL ActuallyPaintPixel
RTS

ActuallyPaintPixel
LDX currentValueInColorIndexArray
LDA presetColorValuesArray,X
STA (currentLineInColorRamLoPtr2),Y
RTS

;-----
; PaintStructureAtCurrentPosition
;-----
PaintStructureAtCurrentPosition
JSR PaintPixelForCurrentSymmetry
LDY #$00
LDA currentValueInColorIndexArray
CMP #NUM_ARRAYS
BNE GamLoopAndPaint
RTS

GamLoopAndPaint
LDA #NUM_ARRAYS
STA countToMatchCurrentIndex

LDX pixelXPosition
STA previousPixelXPosition
LDX pixelYPosition
STA previousPixelYPosition

LDX patternIndex
LDA pixelXPositionLoPtrArray,X
STA xPosLoPtr
LDA pixelXPositionHiPtrArray,X
STA xPosHiPtr
LDA pixelYPositionLoPtrArray,X
STA yPosLoPtr
LDA pixelYPositionHiPtrArray,X
STA yPosHiPtr

PaintPaintLoop
LDA previousPixelXPosition
CLC
ADC (xPosLoPtr),Y
STA pixelXPosition
LDA previousPixelYPosition
CLC
ADC (yPosLoPtr),Y
STA pixelYPosition

TYA
PHA
JSR PaintPixelForCurrentSymmetry
PLA
STA pixelYPosition
PLA
STA pixelXPosition
RTS

HasSymmetry
CMP #X_AXIS_SYMMETRY
BEQ XAxisSymmetry

LDA #NUM_COLS - 1
SEC
SBC pixelXPosition
STA pixelXPosition

LDY currentSymmetrySettingForStep
CPY #X_Y_SYMMETRY
BEQ XYSymmetry

JSR PaintPixel

LDA currentSymmetrySettingForStep
CMP #Y_AXIS_SYMMETRY
BEQ CleanUpAndReturnFromSymmetry

LDA #NUM_ROWS - 1
SEC
SBC pixelYPosition
STA pixelYPosition
JSR PaintPixel

PaintXAxisPixelForSymmetry
PLA
TAY
PLA
STA PixelXPosition
TYA
PHA
JSR PaintPixel
PLA
STA PixelYPosition
RTS

; -----
; RestorePositionsAndReturn
; -----
RestorePositionsAndReturn
LDA previousPixelXPosition
STA pixelXPosition
LDA previousPixelYPosition
STA pixelYPosition
RTS

; -----
; 5
; -----
; 4 4
; 3
; 2
; 1
; 4 000 4
; 5 3210 0123 5
; 4 000 4
; 1
; 2
; 3
; 4 4
; 5
; -----
starOneXPosArray
.BYTE $00,$01,$01,$01,$00
.BYTE $FF,$FF,$FF,$FF,$55
.BYTE $00,$02,$00,$0F,$55
.BYTE $00,$03,$00,$0E,$55
.BYTE $00,$04,$00,$0C,$55
.BYTE $FF,$01,$05,$05,$01
.BYTE $FF,$0B,$0F,$05
.BYTE $00,$07,$00,$09,$55
.BYTE $00,$08,$00,$05,$55
.BYTE $55

starOneYPosArray
.BYTE $FF,$FF,$00,$01,$01
.BYTE $01,$00,$0F,$55
.BYTE $FF,$00,$02,$00,$05
.BYTE $FD,$00,$03,$00,$05
.BYTE $FF,$01,$03,$00,$05
.BYTE $FF,$02,$0F,$01,$05
.BYTE $00,$01,$FF,$00,$05
.BYTE $FF,$00,$00,$05,$55
.BYTE $55

countToMatchcurrentIndex .BYTE $00
;-----
; PutRandomByteInAccumulator
;-----
PutRandomByteInAccumulator
randomByteAddress ***$01
LDA $E199,X
INC randomByteAddress
RTS

;-----
; PaintPixelForCurrentSymmetry
;-----
PaintPixelForCurrentSymmetry
LDY pixelXPosition
PHA
LDA pixelYPosition
PHA
JSR PaintPixel

LDA currentSymmetrySettingForStep
BNE HasSymmetry

CleanUpAndReturnFromSymmetry
PLA
STA pixelYPosition
PLA
STA pixelXPosition
RTS

HasSymmetry
CMP #X_AXIS_SYMMETRY
BEQ XAxisSymmetry

LDA #NUM_COLS - 1
SEC
SBC pixelXPosition
STA pixelXPosition

LDY currentSymmetrySettingForStep
CPY #X_Y_SYMMETRY
BEQ XYSymmetry

JSR PaintPixel

LDA currentSymmetrySettingForStep
CMP #Y_AXIS_SYMMETRY
BEQ CleanUpAndReturnFromSymmetry

LDA #NUM_ROWS - 1
SEC
SBC pixelYPosition
STA pixelYPosition
JSR PaintPixel

PaintXAxisPixelForSymmetry
PLA
TAY
PLA
STA PixelXPosition
TYA
PHA
JSR PaintPixel
PLA
STA PixelYPosition
RTS

```

**Lines 1 - 710.** This first section is almost exactly the same as its equivalent in the listing. On this page and the following two we have the core of the game from which the version we saw in 'let's pretend we can read code' was carved out.

What this tells us, I think, is that what we have here and in the following two pages represents the very initial state of Psychedelia when it was first developed:

**Jeffrey Says**



.. one day I was out running and an just appeared unbidden in my head. I still remember the exact bit of road I was on when it happened. It was a simple algorithm, just seeding patterns along a path; the patterns were to expand and change shape and colour over time. I got back from my run and coded up the algo - it fit in about 1K of 6502 assembler code. I ran the code and a white cursor appeared on the screen. i picked up the joystick, moved the cursor, and pressed down the fire button.

In that instant my life changed.

So this three pages of code is the product of an evening's work - the nucleus of the game as a whole. The fact that among the code we have a single data structure for a single pattern reinforces this notion: a simple demo needed just one pattern to work with, so there it is, propped between `PaintStructureAtCurrentPosition` and `PaintPixelForCurrentSymmetry`.

Once the basic demo was working, it was time to add some more patterns and we can see these inserted after the interrupt handler code in the final column on the following pages: 'The Twist', 'La Llamita', and 'Star Two' make their appearance.

The writing of a complete game was underway.

## increasing the dose

```

AxisSymmetry
LDA #NUM_ROWS - 1
SEC
SBC pixelYPosition
STA pixelYPosition
JSR PaintAxisPixelForSymmetry
;XSymmetry
LDA #NUM_ROWS - 1
SEC
SBC pixelYPosition
STA pixelYPosition
JSR PaintPixel
PLA
STA pixelYPosition
PLA
STA pixelXPosition
RTS

pixelXPositionArray
.BYTE $00,$00,$FF,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$FF,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
pixelYPositionArray
.BYTE $00,$00,$FF,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
currentColorIndexArray
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
initialSmoothingDelayArray
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
smoothingDelayArray
.BYTE $FF,$FF,$FF,$FF,$FF,$FF,$FF,$FF
patternIndexArray
.BYTE $FF,$FF,$FF,$FF,$FF,$FF,$FF,$FF
symmetrySettingForStepCount
.BYTE $FF,$FF,$FF,$FF,$FF,$FF,$FF,$FF
; ReinitializeSequences
;-----

ReinitializeSequences
LDX #$00
TXA
Loop
STA pixelXPositionArray,X
STA pixelYPositionArray,X
LDA #$FF
STA currentColorIndexArray,X
LDA #$00
STA initialSmoothingDelayArray,X
STA smoothingDelayArray,X
STA patternIndexArray,X
STA symmetrySettingForStepCount,X
INX
CPX #PIXEL_BUFFER_LENGTH
BNE .Loop
STA timerBetweenKeyStrokes
STA currentPatternElement
STA previousIndexToPixelBuffers
STA skipPixel
LDA #$_AXIS_SYMOMETRY
STA currentSymmetrySetting
RTS

;-----[LaunchPsychedelia]-----;
LaunchPsychedelia
JSR SetUpInterruptHandlers
LDX #$10
Loop TXA
STA SetUpInterruptHandlers,X
DEX
BNE .Loop
JSR ReinitializeScreen
JSR ReinitializeSequences
JSR ClearLastLineOfScreen
;-----[MainPaintLoop]-----;
MainPaintLoop
INC currentIndexToPixelBuffers
LDA lastKeyPressed
CMP #KEY_CRSR_LEFT_RIGHT
BNE HandleAnyCurrentModes
Loop LDA lastKeyPressed
CMP #NO_KEY_PRESSED
BNE .Loop
Loop2 LDA lastKeyPressed
CMP #KEY_CRSR_LEFT_RIGHT
BNE .Loop2
Loop3 LDA lastKeyPressed
CMP #NO_KEY_PRESSED
BNE .Loop3
HandleAnyCurrentModes
LDA currentModeActive
BEQ DoNormalPaint
CMP #CUSTOM_PRESET_MODE_ACTIVE
BNE IsMainInSavePromptMode
JMP HandleCustomPreset
MaybeIsSavePromptMode
CMP #SAVE_PROMPT_MODE_ACTIVE
BNE InitializeScreenAndPaint
JMP DisplaySavePromptScreen
InitializeScreenAndPaint
JSR ReinitializeScreen
DoANormalPaint
LDA currentIndexToPixelBuffers
CMP bufferLength
BNE CheckCurrentBuffer
LDA #$00
STA currentIndexToPixelBuffers
CheckCurrentBuffer
LDX currentIndexToPixelBuffers
LDA currentColorIndexArray,X
CMP #$FF
BNE ShouldDoAPaint
STX previousIndexToPixelBuffers
JMP MainPaintLoop
ShouldDoAPaint
STA currentValueInColorIndexArray
DEC smoothingDelayArray,X
BNE GoBackToStartOfLoop

LDA initialSmoothingDelayArray,X
STA smoothingDelayArray,X
LDA pixelXPositionArray,X
STA pixelYPositionArray,X
STA patternIndexArray,X
LDA symmetrySettingForStepCount,X
STA currentSymmetrySettingForStep
LDA currentValueInColorIndexArray
AND #SLINE_MODE_ACTIVE
BNE PaintLineModeAndLoop
TXA
PHA
JSR PaintStructureAtCurrentPosition
PLA
TAX
DEC currentColorIndexArray,X
GoBackToStartOfLoop
JMP MainPaintLoop
currentIndexToPixelBuffers .BYTE $00
PaintlineModeAndLoop
JMP PaintLineMode

;-----[SetUpInterruptHandlers]-----;
SetUpInterruptHandlers
SEI
LDA #$_MAIN_INTERRUPT_HANDLER
STA $0014
LDA #$_MAIN_INTERRUPT_HANDLER
STA $0015
LDA #$0A
STA cursorXPosition
STA cursorYPosition
LDA #$01
STA $0015
STA $0027
LDA #$_NMI_INTERRUPT_HANDLER
STA $0018
LDA #$_NMI_INTERRUPT_HANDLER
STA $0019
CLI
RTS
countStepsBeforeCheckingJoystick
.BYTE $02,$00
;-----[MainInterruptHandler]-----;
MainInterruptHandler
LDA stepsRemainingInSequencerSequence
BEQ SequencerNotActiveCheckJoystick
DEC stepsRemainingInSequencerSequence
BNE SequencerNotActiveCheckJoystick
LDA sequencerSpeed
STA stepsRemainingInSequencerSequence
CalledFromNMI
JSR LoadDataForSequencer
SequencerNotActiveCheckJoystick
DEC countStepsBeforeCheckingJoystick
BEQ CanUpdatePixelBuffers
JMP CheckKeyboardAndExitInterrupt
CanUpdatePixelBuffers
LDA #BLACK
STA currentColorToPaint
LDA cursorSpeed
STA countStepsBeforeCheckingJoystick
JSR PaintCursorAtCursorPosition
JSR GetJoystickInput
LDA lastJoystickInput
AND #JOYSTICK_UP | JOYSTICK_DOWN
CMP #JOYSTICK_UP | JOYSTICK_DOWN
BEQ CheckIfCursorMovedLeftOrRight
CMP #JOYSTICK_DOWN
BEQ PlayerHasPressedDown
INC cursorYPosition
INC cursorYPosition
PlayerHasPressedDown
DEC cursorYPosition
LDA cursorYPosition
CMP # BELOW_ZERO
BNE CheckIfCursorAtBottom
LDA #NUM_ROWS - 1
STA cursorYPosition
JMP CheckIfCursorMovedLeftOrRight
CheckIfCursorAtBottom
CMP #NUM_ROWS
BNE CheckIfCursorMovedLeftOrRight
LDA #$00
STA cursorYPosition
CheckIfCursorMovedLeftOrRight
LDA lastJoystickInput
AND #JOYSTICK_RIGHT | JOYSTICK_LEFT
CMP #JOYSTICK_RIGHT | JOYSTICK_LEFT
BEQ CheckIfPlayerPressedFire
CMP #JOYSTICK_RIGHT
BEQ CursorMovedLeft
INC cursorXPosition
INC cursorXPosition
CursorMovedLeft
DEC cursorXPosition
LDA cursorXPosition
CMP # BELOW_ZERO
BNE CheckIfCursorAtExtremeRight
LDA #NUM_COLS - 1
STA cursorXPosition
JMP CheckIfPlayerPressedFire
CheckIfCursorAtExtremeRight
CMP #NUM_COLS
BNE CheckIfPlayerPressedFire

```

```

LDA #$00
STA cursorXPosition
CheckIfPlayerPressedFire
LDA lastJoystickInput
AND #$10
BEQ PlayerHasPressedFire

LDA #$00
STA stepsSincePressedFire
JMP DrawCursorAndReturnFromInterrupt

PlayerHasPressedFire
LDA stepsExceeded255
BEQ DecrementPulseWidthCounter
LDA stepsSincePressedFire
BEQ IncrementStepsSincePressedFire
JMP DrawCursorAndReturnFromInterrupt

IncrementStepsSincePressedFire
INC stepsSincePressedFire

DecrementPulseWidthCounter
LDA currentPulseWidth
BEQ DecrementPulseSpeedCounter
DEC currentPulseWidth
BEQ DecrementPulseSpeedCounter
JMP UpdatePixelBuffersForPattern

DecrementPulseSpeedCounter
DEC currentPulseSpeedCounter
BEQ RefreshPulseSpeed
JMP DrawCursorAndReturnFromInterrupt

RefreshPulseSpeed
LDA pulseSpeed
STA currentPulseSpeedCounter
LDA pulseWidth
STA currentPulseWidth

UpdatePixelBuffersForPattern
INC currentStepCount
LDA currentStepCount
CMP bufferLength
BNE UpdateBaseLevelArray

LDA #$00
STA currentStepCount
UpdateBaseLevelArray
TAX
LDA currentColorIndexArray,X
CMP #$FF
BEQ UpdatePositionArrays

LDA previousIndexToPixelBuffers
AND trackingActivated
BEQ DrawCursorAndReturnFromInterrupt

TAX
LDA currentColorIndexArray,X
CMP #$FF
BNE DrawCursorAndReturnFromInterrupt

STX currentStepCount
UpdatePositionArrays
LDA cursorXPosition
STA pixelXPositionArray,X
LDA cursorYPosition
STA pixelYPositionArray,X
LDA lineModeActivated
BEQ LineModeNotActive

LDA #NUM_ROWS + 1
SEC
SBC cursorYPosition
ORA #$80
STA currentColorIndexArray,X
JMP ApplySmoothingDelay

LineModeNotActive
LDA baselevel
STA currentColorIndexArray,X
LDA currentPatternElement
STA patternIndexArray,X

ApplySmoothingDelay
LDA smoothingDelay
STA smoothingDelayArray,X
STA smoothingDelayArray,X
LDA currentSymmetrySetting
STA symmetrySettingForStepCount,X

DrawCursorAndReturnFromInterrupt
LDA #WHITE
STA currentColorToPaint
JSR PaintCursorAtCursorPosition
;-----;
; CheckKeyboardAndExitInterrupt
;-----;
CheckKeyboardAndExitInterrupt
JSR CheckKeyboardInput
JMP RETURN_INTERRUPT

;-----;
; LoadXAndYOfCursorPosition
;-----;
LoadXAndYOfCursorPosition
LDX cursorXPosition
LDA colorRAMLineTableLoPtrArray,X
STA currentLineInColorRamLoPtr

LDA colorRAMLineTableHiPtrArray,X
STA currentLineInColorRamHiPtr

LDY cursorYPosition
ReturnEarlyFromCursorPaint
RTS

;-----;
; PaintCursorAtCursorPosition
;-----;
PaintCursorAtCursorPosition
LDA displaySavePromptActive
BNE ReturnEarlyFromCursorPaint

JSR LoadXAndYOfCursorPosition
LDA currentColorToPaint
STA (currentLineInColorRamLoPtr),Y

RTS

cursorXPosition .BYTE $0A
cursorYPosition .BYTE $0A
currentStepCount .BYTE $00
stepsSincePressedFire .BYTE $00
stepsExceeded255 .BYTE $00

presetValueArray
unusedPresetByte .BYTE $00
smoothingDelay .BYTE $0C
currentX .BYTE $0E
bufferLength .BYTE $1F
pulseSpeed .BYTE $01
indexForColorBarDisplay .BYTE $01
lineWidth .BYTE $07
sequencerSpeed .BYTE $04
pulseWidth .BYTE $01
baseLevel .BYTE $07

theTwistYPosArray
.BYTE $00,$55
.BYTE $01,$02,$55
.BYTE $01,$02,$03,$55
.BYTE $01,$02,$03,$04,$55
.BYTE $00,$00,$00,$55
.BYTE $FF,$FF,$55
.BYTE $FF,$55
.BYTE $55

theTwistXPosArray
.BYTE $FF,$55
.BYTE $FF,$FF,$55
.BYTE $00,$00,$00,$55
.BYTE $01,$02,$03,$55
.BYTE $01,$02,$55
.BYTE $00,$55
.BYTE $55

pixelPositionLoPtrArray
.BYTE <starOneXPosArray,>
theTwistXPosArray,<
lalamitaXPosArray
.BYTE <starTwoXPosArray,>
deltoidXPosArray,<
diffusedXPosArray
.BYTE <multicrossXPosArray,>
pulsarXPosArray

customPatternLoPtrArray
.BYTE <customPattern0XPosArray,>
customPattern1XPosArray
.BYTE <customPattern2XPosArray,>
customPattern3XPosArray
.BYTE <customPattern4XPosArray,>
customPattern5XPosArray
.BYTE <customPattern6XPosArray,>
customPattern7XPosArray
lalamitaXPosArray
.BYTE $00,$00,$00,$55
.BYTE $00,$00,$55
.BYTE $01,$02,$03,$00,$01,$02,$03,$55
.BYTE $04,$05,$06,$04,$00,$01,$02,$02,$55
.BYTE $04,$00,$04,$00,$00,$04,$55
.BYTE $FF,$03,$55
.BYTE $00,$55

lalamitaYPosArray
.BYTE $FF,$00,$01,$55
.BYTE $00,$00,$55
.BYTE $03,$03,$03,$04,$04,$04,$04,$04,$55
.BYTE $02,$02,$03,$04,$05,$05,$05,$05,$55
.BYTE $05,$06,$06,$07,$07,$07,$07,$07,$55
.BYTE $07,$07,$55
.BYTE $00,$55

;-----;
; starOneYPosArray
;-----;
starOneYPosArray
.BYTE $00,$55
.BYTE $01,$55
.BYTE $02,$55
.BYTE $03,$55
.BYTE $04,$55
.BYTE $05,$55
.BYTE $06,$55
.BYTE $07,$55
.BYTE $08,$55
.BYTE $09,$55
.BYTE $0A,$55
.BYTE $0B,$55
.BYTE $0C,$55
.BYTE $0D,$55
.BYTE $0E,$55
.BYTE $0F,$55
.BYTE $00,$55

;-----;
; starTwoYPosArray
;-----;
starTwoYPosArray
.BYTE $FF,$55
.BYTE $FF,$55
.BYTE $FF,$55
.BYTE $FF,$55
.BYTE $02,$55
.BYTE $01,$55
.BYTE $FC,$55
.BYTE $00,$55

```

## increasing the dose

---

```

;      5
;      4
;      3
;      2
;     202
;    20602
;      3   3
;      4   4

;      5      5
deltoidXPosArray
.BYTE $00,$01,$FF,$55
.BYTE $00,$55
.BYTE $00,$01,$02,$FE,$FF,$55
.BYTE $00,$03,$0D,$55
.BYTE $00,$04,$PC,$55
.BYTE $00,$06,$FA,$55
.BYTE $00,$55

deltoidYPosArray
.BYTE $FF,$00,$00,$55
.BYTE $00,$55
.BYTE $FE,$FF,$00,$00,$FF,$55
.BYTE $FD,$01,$01,$55
.BYTE $FC,$02,$02,$55
.BYTE $FA,$04,$04,$55
.BYTE $00,$55

;      4
;      3
;      2
;      5   1   5
;      0   3
;      6   0   3
;      3   1   5
;      2   3
;      3
;      4
;      5
diffusedXPosArray
.BYTE $FF,$01,$55
.BYTE $FE,$02,$55
.BYTE $FF,$03,$55
.BYTE $FC,$04,$PC,$FC,$04,$04,$55
.BYTE $FB,$05,$55
.BYTE $FA,$06,$FA,$06,$06,$55
.BYTE $00,$55

diffusedYPosArray
.BYTE $01,$FF,$55
.BYTE $FE,$02,$55
.BYTE $03,$FD,$55
.BYTE $FC,$04,$FF,$01,$FF,$01,$55
.BYTE $05,$FB,$55
.BYTE $FA,$06,$FE,$02,$FE,$02,$55
.BYTE $00,$55

-----;
; CheckKeyboardInput
; -----
CheckKeyboardInput
LDA currentVariableMode
BEQ CheckForGeneralKeyStrokes
JMP CheckKeyboardInputForMode

CheckForGeneralKeyStrokes
LDA timerBetweenKeyStrokes
BEQ CheckForKeyStroke
DEC timerBetweenKeyStrokes
BNE ReturnFromKeyboardCheck
RTS

CheckForKeyStroke
LDA lastKeyPressed
CMP #NO_KEY_PRESSED
BNE ProcessKeyStroke

LDA $#00
STA timerBetweenKeyStrokes
JSR MaybeDisplayDemoModeMessage
ReturnFromKeyboardCheck
RTS

ProcessKeyStroke
LDI initialTimeBetweenKeyStrokes
STY timerBetweenKeyStrokes
LDY shiftKey
STY shiftKeyPressed

CMP #KEY_SPACE
BNE MaybeSPressed

INC currentPatternElement
LDA currentPatternElement
AND #$0F
STA currentPatternElement

AND #$08
BEQ UpdateCurrentPattern
JMP GetCustomPatternElement

UpdateCurrentPattern
JSR ClearLastLineOfScreen
LDA currentPatternElement
ASL
ASL
ASL
TAY

LDX #$00
txtPresetLoop
LDA txtPresetPatternNames,Y
STA statusLineBuffer,X

INY
INX
CPX #DISPLAY_LINE_LENGTH
BNE txtPresetLoop
JMP WriteLastLineBufferToScreen

MaybeSPressed
CMP #KEY_S
BNE MaybeLPressed

LDA shiftPressed
AND #SHIFT_PRESSED
BEQ JustSPressed
JMP PromptToSave

LDA tapeSavingInProgress
BNE JustSPressed
JMP PromptToSave

JustSPressed
INC currentSymmetrySetting
LDA currentSymmetrySetting
CMP #QUAD_SYMMETRY + 1
BNE DisplaySymmetry

LDA #$00
STA currentSymmetrySetting
DisplaySymmetry
ASL
ASL
ASL
TAY
JSR ClearLastLineOfScreen

LDX #$00
txtSymmloop
LDA txtSymmetrySettingDescriptions,Y
STA statusLineBuffer,X
INY
INX
CPX #DISPLAY_LINE_LENGTH
BNE txtSymmloop
JMP WriteLastLineBufferToScreen
RTS

CheckForKeyLoop
LDX #$00
presetKeyLoop
CPX presetKeyCode,X
BEQ UpdateDisplayedPreset
INX
CPX #DISPLAY_LINE_LENGTH
BNE presetKeyLoop

JMP MaybeWPressed

UpdateDisplayedPreset
JMP DisplayPresetMessage

MaybeWPressed
CMP #KEY_W
BNE MaybeFunctionKeysPressed

LDA #LINE_WIDTH
STA currentVariableMode
RTS

MaybeFunctionKeysPressed
LDX #$00
FnKeyLoop
CPX functionKeys,X
BEQ FunctionKeyWasPressed
INX
CPX #$04
BNE FnKeyLoop
JMP MaybePressed

FunctionKeyWasPressed
STX functionKeyIndex
LDI initialTimeBetweenKeyStrokes
BNE MaybeDPressed
LDA #SEQUENCER_OR_BURST_ACTIVE
STA currentVariableMode
JSR LoadOrProgramBurstGenerator
RTS

MaybeQPressed
CMP #KEY_Q
BNE MaybeVPressed

LDA sequencerActive
BNE TurnSequenceOff
LDA #SEQUENCER_OR_BURST_ACTIVE
STA currentVariableMode
JMP ActivateSequencer

;      5
;      4
;      3
;      2
;      5   1   5
;      0   3
;      6   0   3
;      3   1   5
;      2   3
;      3
;      4
;      5
diffusedXPosArray
.BYTE $FF,$01,$55
.BYTE $FE,$02,$55
.BYTE $FF,$03,$55
.BYTE $FC,$04,$PC,$FC,$04,$04,$55
.BYTE $FB,$05,$55
.BYTE $FA,$06,$FA,$06,$06,$55
.BYTE $00,$55

diffusedYPosArray
.BYTE $01,$FF,$55
.BYTE $FE,$02,$55
.BYTE $03,$FD,$55
.BYTE $FC,$04,$FF,$01,$FF,$01,$55
.BYTE $05,$FB,$55
.BYTE $FA,$06,$FE,$02,$FE,$02,$55
.BYTE $00,$55

-----;
; CheckKeyboardInput
; -----
CheckKeyboardInput
LDA currentVariableMode
BEQ CheckForGeneralKeyStrokes
JMP CheckKeyboardInputForMode

CheckForGeneralKeyStrokes
LDA timerBetweenKeyStrokes
BEQ CheckForKeyStroke
DEC timerBetweenKeyStrokes
BNE ReturnFromKeyboardCheck
RTS

CheckForKeyStroke
LDA lastKeyPressed
CMP #NO_KEY_PRESSED
BNE ProcessKeyStroke

LDA $#00
STA timerBetweenKeyStrokes
JSR MaybeDisplayDemoModeMessage
ReturnFromKeyboardCheck
RTS

ProcessKeyStroke
LDI initialTimeBetweenKeyStrokes
STY timerBetweenKeyStrokes
LDY shiftKey
STY shiftKeyPressed

CMP #KEY_SPACE
BNE MaybeSPressed

INC currentPatternElement
LDA currentPatternElement
AND #$0F
STA currentPatternElement

AND #$08
BEQ UpdateCurrentPattern
JMP GetCustomPatternElement

UpdateCurrentPattern
JSR ClearLastLineOfScreen
LDA currentPatternElement
ASL
ASL
ASL
TAY

LDX #$00
txtPresetLoop
LDA txtPresetPatternNames,Y
STA statusLineBuffer,X

INY
INX
CPX #DISPLAY_LINE_LENGTH
BNE txtPresetLoop
JMP WriteLastLineBufferToScreen

MaybeSPressed
CMP #KEY_S
BNE MaybeLPressed

LDA shiftPressed
AND #SHIFT_PRESSED
BEQ JustSPressed
JMP PromptToSave

LDA tapeSavingInProgress
BNE JustSPressed
JMP PromptToSave

JustSPressed
INC currentSymmetrySetting
LDA currentSymmetrySetting
CMP #QUAD_SYMMETRY + 1
BNE DisplaySymmetry

LDA #$00
STA currentSymmetrySetting
DisplaySymmetry
ASL
ASL
ASL
TAY
JSR ClearLastLineOfScreen

LDX #$00
txtTrackingLoop
LDA txtTrackingOnOff,Y
STA statusLineBuffer,X

INY
INX
CPX #DISPLAY_LINE_LENGTH
BNE txtTrackingLoop
JMP WriteLastLineBufferToScreen
RTS

CheckIfPresetKeysPressed
LDX #$00
presetKeyLoop
CPX presetKeyCode,X
BEQ UpdateDisplayedPreset
INX
CPX #DISPLAY_LINE_LENGTH
BNE presetKeyLoop

JMP MaybeWPressed

UpdateDisplayedPreset
JMP DisplayPresetMessage

MaybeWPressed
CMP #KEY_W
BNE MaybeFunctionKeysPressed

LDA #LINE_WIDTH
STA currentVariableMode
RTS

MaybeFunctionKeysPressed
LDX #$00
FnKeyLoop
CPX functionKeys,X
BEQ FunctionKeyWasPressed
INX
CPX #$04
BNE FnKeyLoop
JMP MaybePressed

FunctionKeyWasPressed
STX functionKeyIndex
LDI initialTimeBetweenKeyStrokes
BNE MaybeDPressed
LDA #SEQUENCER_OR_BURST_ACTIVE
STA currentVariableMode
JSR LoadOrProgramBurstGenerator
RTS

MaybeQPressed
CMP #KEY_Q
BNE MaybeVPressed

LDA sequencerActive
BNE TurnSequenceOff
LDA #SEQUENCER_OR_BURST_ACTIVE
STA currentVariableMode
JMP ActivateSequencer

```

**Lines 710 - 1643.** With the main business of the game taken care of, it is time to add features and settings.

That means creating a lot of configuration options and allowing the player to set them on or off. This and the following pages concern themselves entirely with handling user input and acting on it. The monster function starting opposite, `CheckKeyboardInput`, is run from the interrupt handler and checks for all potential keyboard input from the player, one key at a time. It runs all the way to the following page and is followed by a couple of new pattern definitions, belying the incremental nature of development: add a new feature, then add a few more patterns to keep things interesting.

The code that follows implements some of the complicated keyboard interactions such as increasing/decreasing various modes and color settings. It also looks after programming and loading the burst, sequencer, and preset data which we will cover in the '[beatific bursts](#)', '[sensitive sequencer](#)', and '[particular presets](#)' chapters.

The sheer volume of code required to manage this overhead is an object lesson that holds true for most forms of software development: much of our time is spent managing detail ancillary to the main purpose of the program. The core that delivers the magic is always relatively small and self-contained, making the thing presentable is what generates the bloat and most of the hard labour.

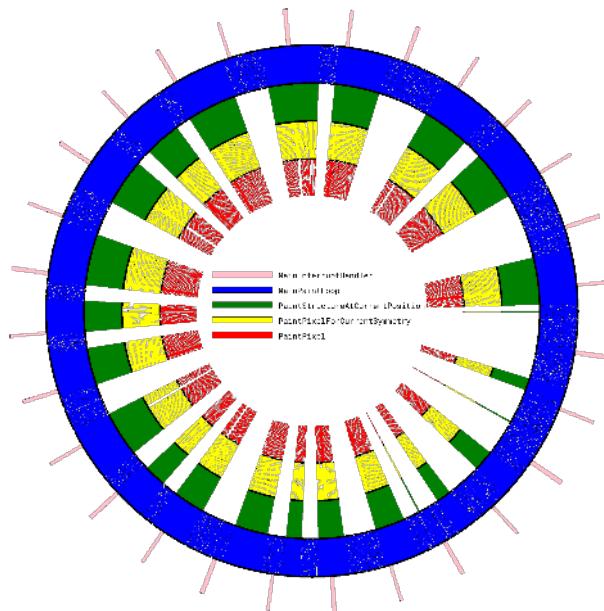


Figure 4.1: The execution map of a full pattern evolution in the commercial edition of Psychedelia.

## increasing the dose

```

TurnSequenceOff
LDA #$000
STA sequencerActive
STA stepsRemainingInSequencerSequence
JMP DisplaySequencerState

MaybeOPressed
CMP #KEY_V
BNE MaybeOPressed

LDA #SEQUENCER_SPEED
STA currentVariableMode
RTS

MaybeOPressed
CMP #KEY_O
BNE MaybeAsteriskPressed

LDA #PULSE_WIDTH
STA currentVariableMode
RTS

MaybeAsteriskPressed
CMP #KEY_ASTERISK
BNE MaybeAsteriskPressed

LDA #BASE_LEVEL
STA currentVariableMode
RTS

MaybeRPressed
CMP #KEY_R
BNE MaybeUpArrowPressed
JMP StopOrStartRecording

MaybeUpArrowPressed
CMP #KEY_UP
BNE MaybeUpArrowPressed
INC pixelShapeIndex
LDA pixelShapeIndex
AND #$0F
TAY
LDA pixelShapeArray,Y

LDX #$000
.Loop STA SCREEN_RAM + $0000,X
STA SCREEN_RAM + $0100,X
STA SCREEN_RAM + $0200,X
STA SCREEN_RAM + $0200,X
DEX
BNE _Loop
STA currentPixel
RTS

MaybeAPressed
CMP #KEY_A
BNE FinalReturnFromKeyboardCheck

LDA demoModeActive
EOR #$01
STA demoModeActive
RTS

FinalReturnFromKeyboardCheck
RTS

initialTimeBetweenKeyStrokes
.BYTE $10

;

; 5   5
; 4   4
; 5 3 2 2 3 5
; 1   1
; 2 0 0 2
;       6
; 2 0 0 2
;       1
; 5 3 2 2 3 5
; 4   4
;       5
multiCrossPosArray
.BYTE $01,$01,$FF,$FF,$55
.BYTE $02,$02,$FF,$FF,$55
.BYTE $01,$03,$03,$01,$FF
.BYTE $FD,$FD,$FF,$FF,$55
.BYTE $03,$03,$FD,$FD,$55
.BYTE $04,$04,$FC,$FC,$55
.BYTE $03,$05,$05,$09,$FD
.BYTE $01,$01,$01,$01,$55
.BYTE $00,$55
multiCrossYPosArray
.BYTE $FF,$01,$01,$FF,$55
.BYTE $FE,$02,$02,$FE,$55
.BYTE $FF,$02,$02,$FE,$55
.BYTE $01,$FF,$FD,$55
.BYTE $FD,$03,$03,$FD,$55
.BYTE $FC,$04,$04,$FC,$55
.BYTE $FB,$FD,$03,$05,$05
.BYTE $03,$FD,$FB,$55
.BYTE $00,$55

```

```

;

; 5
; 4
; 3
; 2
; 1
; 0
; 5432106012345
; 0
; 1
; 2
; 3
; 4
; 5

pulsarXPosArray
.BYTE $00,$01,$00,$FF,$55
.BYTE $00,$02,$00,$FF,$55
.BYTE $00,$03,$00,$FD,$55
.BYTE $00,$04,$00,$FC,$55
.BYTE $00,$05,$00,$FB,$55
.BYTE $00,$06,$00,$FA,$55
.BYTE $00,$55

pulsarYPosArray
.BYTE $FF,$00,$01,$00,$55
.BYTE $FF,$00,$02,$00,$55
.BYTE $FF,$00,$03,$00,$55
.BYTE $FC,$00,$04,$00,$55
.BYTE $FB,$00,$05,$00,$55
.BYTE $FA,$00,$06,$00,$55
.BYTE $00,$55

statusLineBuffer
.BYTE $FF,$FF,$FF,$FF,$FF,$FF
dataFreeDigitOne .BYTE $FF
dataFreeDigitTwo .BYTE $FF
dataFreeDigitThree .BYTE $FF,$FF,$FF,$FF
customPatternValueBufferPtr
.BYTE $FF,$FF,$FF
customPatternValueBufferMessage
.BYTE $FF,$FF,$FF,$FF,$FF,$FF
.BYTE $FF,$FF,$FF,$FF,$FF,$FF
.BYTE $00,$00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00,$00

;

; ClearlastLineOfScreen
;-----
ClearlastLineOfScreen
LDX #NUM_COLS
loop
LDA #$20
STA statusLineBuffer - $01,X
STA SCREEN_RAM + $03BF,X
DEX
BNE _Loop
RTS

;

; WriteLastLineBufferToScreen
;-----
WriteLastLineBufferToScreen
LDX #NUM_COLS
loop
LDA statusLineBuffer - $01,X
AND #$3F
STA SCREEN_RAM + $03BF,X
LDA #$0C
STA COLOR_RAM + $03BF,X
DEX
BNE _Loop
RTS

;

; WriteLastLineBufferToScreen
;-----
WriteLastLineBufferToScreen
LDX #NUM_COLS
loop
LDA statusLineBuffer - $01,X
AND #$3F
STA SCREEN_RAM + $03BF,X
LDA #$0C
STA COLOR_RAM + $03BF,X
DEX
BNE _Loop
RTS

txtPresetPatternNames
.TEXT 'STAR ONE' ,
.TEXT 'THE TWIST' ,
.TEXT 'LA LLANITA' ,
.TEXT 'MIRAGE' ,
.TEXT 'DELTOIDS' ,
.TEXT 'DIFFUSED' ,
.TEXT 'MULTICROSS' ,
.TEXT 'PULSAR' ,
txtSymmetrySettingDescriptions
.TEXT 'NO SYMMETRY' ,
.TEXT 'Y-AXIS SYMMETRY' ,
.TEXT 'X- Y SYMMETRY' ,
.TEXT 'X- AXIS SYMMETRY' ,
.TEXT 'QUAD SYMMETRY' ,

```

```

; -----
; PaintLineMode
; -----
PaintLineMode
LDA currentValueInColorIndexArray
AND #$7F
STA offsetForYPos

LDA #NUM_ROWS + 1
SEC
SBC offsetForYPos
STA pixelyPosition
DEC pixelyPosition

LDA #$800
STA currentValueInColorIndexArray

LDA #ACTIVE
STA skipPixel

JSR PaintPixelForCurrentSymmetry
INC pixelyPosition
LDA #NOT_ACTIVE
STA skipPixel

LDA lineWidth
BDR #$00
STA currentValueInColorIndexArray
LineModeLoop
JSR PaintPixelForCurrentSymmetry
INC pixelyPosition
INC currentValueInColorIndexArray
LDA currentValueInColorIndexArray
CMP #$08
BNE ResetLineModeColorValue
JMP CleanUpAndExitLineModePaint

INC currentValueInColorIndexArray
ResetLineModeColorValue
STA currentValueInColorIndexArray
LDA pixelyPosition
CMP #NUM_ROWS + 1
BNE LineModeLoop

CleanUpAndExitLineModePaint
LDX currentIndexToPixelBuffers
DOR currentValueInColorIndexArray,X
LDA currentValueInColorIndexArray,X
CMP #$800
BEQ ResetIndexAndExitLineModePaint
JMP MainPaintLoop

ResetIndexAndExitLineModePaint
LDA #$FF
STA currentValueInColorIndexArray,X
STX previousIndexToPixelBuffers
JMP MainPaintLoop

ResetIndexAndExitLineModePaint
LDA #$FF
STA currentValueInColorIndexArray,X
STX previousIndexToPixelBuffers
JMP MainPaintLoop

lineModeSettingDescriptions
.TEXT 'LINE MODE: OFF' ,
.TEXT 'LINE MODE: ON' ,
; -----
; DrawColorValueBar
; -----
DrawColorValueBar
LDA colorBarScreenRamHiPtr
PHA
CLC
ADC #$04
STA colorBarScreenRamHiPtr

LDY #$800
_Loop LDA colorBarValues,Y
STA (colorBarScreenRamLoPtr),Y
INY
CPY #$10
BNE _Loop

PLA
STA colorBarScreenRamHiPtr
LDA #$800
STA currentNodeInColorBar
STA currentCountInColorBar
STA offsetToColorBar
LDA maxToDrawOnColorBar
BEQ ReturnEarlyFromColorBar

ColorBarLoop
LDA offsetToColorBar
CLC
ADC currentNodeInColorBarOffset
STA offsetToColorBar
LDX offsetToColorBar
LDY currentNodeInColorBar
LDA colorBarCharacterArray,X
STA (colorBarScreenRamLoPtr),Y
CPX #$08
BNE GoToNextCell

LDA #$800
STA offsetToColorBar
INC currentNodeInColorBar

```

```

GoToNextCell
INC currentCountInColorBar
LDA currentCountInColorBar
CMP maxToDrawnColorBar
BNE ColorBarLoop

ReturnEarlyFromColorBar
RTS

currentColorBarOffset .BYTE $FF
currentNodeInColorBar .BYTE $FF
maxToDrawnColorBar .BYTE $FF
currentCountInColorBar .BYTE $FF
offsetToColorBar .BYTE $FF

colorBarCharacterArray
.BYTE SPACE,_LEFT_BAR,_ONE_FIFTH
.BYTE LEFT_BAR,_TWO_FIFTHS
.BYTE LEFT_BAR,_TWO_FIFTHS2
.BYTE LEFT_BAR,_THREE_FIFTHS
.BYTE RIGHT_BAR,_ONE_FIFTHS
.BYTE RIGHT_BAR,_TWO_FIFTHS
.BYTE RIGHT_BAR,_TWO_FIFTHS2
.BYTE SPACE,_MAYBE

ResetSelectedVariableAndReturn
LDA #NOT_ACTIVE
STA currentVariableMode
RTS

; CheckKeyboardInputForMode
;-----
CheckKeyboardInputForMode
AND #$80
BEQ SlidingScaleActive
JMP CheckKeyboardWhilePromptActive

SlidingScaleActive
LDA timerBetweenKeyStrokes
BEQ MaybeDisplayVariableSelection
DEC timerBetweenKeyStrokes
JMP DisplayVariableSelection

MaybeDisplayVariableSelection
LDA lastKeyPressed
CMP #NO_KEY_PRESSED
BNE MaybeUpdateVariable
JMP DisplayVariableSelection

MaybeUpdateVariable
LDA #$04
STA timerBetweenKeyStrokes

LDA currentVariableMode
CMP #COLOR_CHANGE
BEQ UpdateColorChange
CNP #$BUFFER_LENGTH
BNE UpdateVariableDisplay

UpdateColorChange
LDX #$00
.Loop LDA currentColorIndexArray,X
CMP #$FF
BNE ResetSelectedVariableAndReturn

INX
CPX bufferLength
BNE _Loop

LDA stepRemainingInSequencerSequence
BNE ResetSelectedVariableAndReturn

LDA playbackOrRecordActive
CMP #PLAYING_BACK
BEQ ResetSelectedVariableAndReturn

LDA demoModeActive
BNE ResetSelectedVariableAndReturn

LDA #GENERIC_ACTIVE
STA currentModeActive
LDA #$00
STA currentStepCount

UpdateVariableDisplay
LDA #>SCREEN_RAM + $03D0
STA colorBarScreenRamHlPtr
LDA #<SCREEN_RAM + $03D0
STA colorBarScreenRamLoPtr

LDX currentVariableMode
LDA lastKeyPressed
CMP #KEY_LT
BNE MaybeLeftArrowPressed

INC presetValueArray,X
LDA presetValueArray,X
CMP maxValueForPresetValueArray,X
BNE MaybeInColorMode

DEC presetValueArray,X
JMP MaybeInColorMode

MaybeLeftArrowPressed
CMP #KEY_TRBR
BNE MaybeInColorMode

DEC presetValueArray,X
CMP minValueForPresetValueArray,X
BNE MaybeInColorMode
INC presetValueArray,X

MaybeInColorMode
CPX #$05
BNE MaybeEnterPressed

LDX indexForColorBarDisplay
LDY currentColorSet
LDA colorValuesPtr,X
STA presetColorValuesArray,Y

MaybeEnterPressed
JSR DisplayVariableSelection
JMP CheckIfEnterPressed

; -----
; DisplayVariableSelection
; -----
DisplayVariableSelection
LDA #>SCREEN_RAM + $03D0
STA colorBarScreenRamHlPtr
LDA #<SCREEN_RAM + $03D0
STA colorBarScreenRamLoPtr

LDX currentVariableMode
CPX #COLOR_CHANGE
BNE VariableModeIsNotColorChange

VariableModeIsColorChange
LDX currentColorSet
LDA presetColorValuesArray,X

LDY #$00
.Loop
CMP colorValuesPtr,Y
BEG FoundColorMatch
INY
CPY #DISPLAY_LINE_LENGTH
BNE _Loop

FoundColorMatch
STY indexForColorBarDisplay
LDX currentVariableMode

VariableModeIsNotColorChange
LDA increaseOffsetForPresetValueArray
,X
STA currentColorBarOffset

LDA presetValueArray,X
STA maxToDrawnColorBar

TXA
PHA

LDA enterWasPressed
BNE UpdateVariableLabel

LDA #$01
STA enterWasPressed
JSR ClearLastLineOfScreen

UpdateVariableLabel
PLA
ASL
ASL
ASL
ASL
TAY

LDY #$00
.Loop2
LDA txtVariableLabels,Y
STA statusLineBuffer,X
INY
INX
CPX #DISPLAY_LINE_LENGTH
BNE _Loop2

LDA currentVariableMode
CPX #COLOR_CHANGE
BNE UpdateBarForOtherMode

UpdateBarForColorMode
LDA #$30
CLC
ADC currentColorSet
STA dataFreeDigitTwo
UpdateBarForOtherMode
JSR WriteLastLineBufferToScreen
JMP DrawColorValueBar

```

```

; -----
; CheckIfEnterPressed
; -----
CheckIfEnterPressed
LDA lastKeyPressed
CMP #KEY_RETURN
BEQ EnterHasBeenPressed
RTS

EnterHasBeenPressed
LDA currentVariableMode
CMP #COLOR_CHANGE
BNE ReachedLastColor

INC currentColorSet
LDA currentColorSet
CMP #$08
BEQ ReachedLastColor
RTS

ReachedLastColor
LDA #$00
STA currentVariableMode
STA enterWasPressed
RTS

maxValueForPresetValueArray
.BYTE $00,$40,$00,$40,$10,$08
.BYTE $20,$10,$08
minValueForPresetValueArray
.BYTE $00,$00,$00,$00,$00
.BYTE $00,$00,$00,$00,$00
increaseOffsetForPresetValueArray
.BYTE $00,$01,$08
.BYTE $01,$04,$08,$08,$02,$04,$08
currentVariableMode
.BYTE $00
currentPulseSpeedCounter
.BYTE $01

txtVariableLabels
.TEXT ,
.TEXT 'SMOOTHING DELAY: ',
.TEXT 'CURSOR SPEED :',
.TEXT 'BUFFER LENGTH :'
.TEXT 'PULSE SPEED :'
.TEXT 'PULSE PCT :'
.TEXT 'WIDTH OF LINE :'
.TEXT 'SEQUENCER SPEED: '
.TEXT 'PULSE WIDTH :'
.TEXT 'BASE LEVEL :'

colorValuesPtr
.BYTE $00

colorBarValues
.BYTE BLUE,RED,PURPLE,GREEN
.BYTE CYAN,YELLOW,WHITE,ORANGE
.BYTE BROWN,LTRED,GRAY1,GRAY2
.BYTE LTGREEN,LTYLUE,GRAY3

txtTrackingOnOff
.TEXT 'TRACKING: OFF '
.TEXT 'TRACKING: ON '

; -----
; DisplayPresetMessage
; -----
DisplayPresetMessage
LDA shiftPressed
AND #$04
BEQ SelectNewPreset
JMP MaybeEditCustomPattern

SelectNewPreset
TXA
PHA
JSR ClearLastLineOfScreen
LDY #$00
Loop LDA txtPreset,X
STA statusLineBuffer,X
INX
CPX #DISPLAY_LINE_LENGTH
BNE _Loop

PLA
PHA
TAX
BEQ JumpToUpdateCurrentActivePreset

DataFreeDisplayLoop
INC dataFreeDigitThree
LDA dataFreeDigitThree
CMP #COLON
BNE GoToNextDigit
LDA #'0'
STA dataFreeDigitThree
INC dataFreeDigitTwo
GoToNextDigit
DEX
BNE DataFreeDisplayLoop

```

## increasing the dose

---

```

JumpToUpdateCurrentActivePreset
JMP UpdateCurrentActivePreset

WriteLastLineBufferAndReturn
JSR WriteLastLineBufferToScreen
RTS

txtPreset
.TEXT 'PRESET 00 :'

txtPresetActivatedStored
.TEXT ' ACTIVATED '
.TEXT 'DATA STORED '

shiftPressed
.BYTE $00

;-----
; UpdateCurrentActivePreset
;-----
UpdateCurrentActivePreset
LDA shiftPressed
AND #$SHIFT_PRESSED
ASL
ASL
ASL
ASL
TAY

DisplayActivatedOrStored
LDX #$00
.Loop
LDA txtPresetActivatedStored,Y
STA customPatternValueBufferMessage,X

INY
INX
CPX #DISPLAY_LINE_LENGTH
BNE _Loop

LDA shiftPressed
AND #$SHIFT_PRESSED
BNE StoreCurrentValueAsPreset

JMP RefreshPresetData

StoreCurrentValueAsPreset
PLA
TAX
JSR GetPresetPointersUsingXRegister

LDY #$00
LDX #$00
.Loop
LDA presetValueArray,X
STA (presetSequenceDataLoPtr),Y

INY
INX
CPX #$15
BNE _Loop

LDA currentPatternElement
STA (presetSequenceDataLoPtr),Y

INY

LDA currentSymmetrySetting
STA (presetSequenceDataLoPtr),Y

JMP WriteLastLineBufferAndReturn
;-----
; RefreshPresetData
;-----
RefreshPresetData
PLA
TAX
JSR GetPresetPointersUsingXRegister

LDY #BUFFER_LENGTH
LDA (presetSequenceDataLoPtr),Y
CMP bufferLength
BEQ MaybeReloadPresetData

JSR ResetCurrentActiveMode
JMP LoadSelectedPresetSequence

MaybeReloadPresetData
LDX #$00
LDY #$SEQUENCER_SPEED
.Loop
LDA (presetSequenceDataLoPtr),Y
CMP presetColorValueArray,X
BNE LoadSelectedPresetSequence
INY
INX
CPX #len(presetColorValueArray)
BNE _Loop

JMP LoadSelectedPresetSequence

;-----
; LoadSelectedPresetSequence
;-----
LoadSelectedPresetSequence
LDA #GENERIC_ACTIVE
STA currentModeActive

```

```

LDY #COLOR_BAR_CURRENT
_Loop
LDA (presetSequenceDataLoPtr),Y
STA presetValueArray,Y
INY
CPY #$15
BNE _Loop

LDA (presetSequenceDataLoPtr),Y
STA currentPatternElement
INY
LDA (presetSequenceDataLoPtr),Y
STA currentSymmetrySetting
JMP WriteLastLineBufferAndReturn

;-----
; GetPresetPointersUsingXRegister
;-----
GetPresetPointersUsingXRegister
LDA #>presetSequenceDataHiPtr
STA presetSequenceDataHiPtr
LDA #<presetSequenceData
STA presetSequenceDataLoPtr
TXA
BEQ ReturnFromPresetPointers

_Loop LDA presetSequenceDataLoPtr
CLC
ADC #$20
STA presetSequenceDataLoPtr
LDA presetSequenceDataHiPtr
ADC #$00
STA presetSequenceDataHiPtr
DEX
BNE _Loop
ReturnFromPresetPointers
RTS

;-----
; ResetCurrentActiveMode
;-----
ResetCurrentActiveMode
LDA #GENERIC_ACTIVE
STA currentModeActive
LDA #$00
STA currentStepCount
RTS

currentModeActive .BYTE $00
;-----
; ReinitializeScreen
;-----
ReinitializeScreen
LDA #$00
STA currentIndexToPixelBuffers
STA previousIndexToPixelBuffers

LDX #$00
LDY #$FF
.Loop STA currentColorIndexArray,X
INY
CPX #PIXEL_BUFFER_LENGTH
BNE _Loop

LDA #$00
STA currentModeActive
JMP InitializeScreenWithInitCharacter

enterWasPressed .BYTE $00
functionKeyIndex .BYTE $00
;-----
; LoadOrProgramBurstGenerator
;-----
LoadOrProgramBurstGenerator
JSR ClearLastLineOfScreen
LDA shiftPressed
AND #$SHIFT_PRESSED
BEQ PointToBurstData

LDX #$00
Loop LDA txtDataFree,X
STA statusLineBuffer,X
INX
CPX #DISPLAY_LINE_LENGTH
BNE _Loop
JSR WriteLastLineBufferToScreen

PointToBurstData
LDA #>burstGeneratorF1
STA currentSequencePtrHi
LDY functionKeyIndex
LDA functionKeyToSequenceArray,X
STA currentSequencePtrLo

LDA shiftPressed
AND #$SHIFT_PRESSED
BEQ LoadBurstDataInstead

LDA #$10
STA currentDataFree

LDY #$00
LDA currentSymmetrySetting
STA (currentSequencePtrLo),Y
LDA smoothingDelay
INY
STA (currentSequencePtrLo),Y
RTS

LoadBurstDataInstead
LDA #GENERIC_ACTIVE
STA sequencerActive
JMP LoadBurstData

functionKeyToSequenceArray
.BYTE <burstGeneratorF1,<
.burstGeneratorF2
.BYTE <burstGeneratorF3,<
.burstGeneratorF4

txtDataFree
.TEXT 'DATA: 000 FREE :'

functionKeys
.BYTE $04,$05,$06,$03

currentDataFree .BYTE $FF,$60
;-----
; CheckKeyboardWhilePromptActive
;-----
CheckKeyboardWhilePromptActive
LDA currentVariableMode
CMP #CUSTOM_PRESET_ACTIVE
BNE MaybeSavingActive
JMP CheckInputForCustomPresets

MaybeSavingActive
CMP #SAVING_ACTIVE
BNE MaybeLoadingActive
JMP CheckInputWhileSavePromptActive

MaybeLoadingActive
CMP #LOADING_ACTIVE
BNE SequencerOrBurstActive
JMP CheckInputWhileLoadAbortActive

SequencerOrBurstActive
LDA #0'
STA dataFreeDigitOne
STA dataFreeDigitTwo
STA dataFreeDigitThree
LDX currentDataFree
BNE UpdateDataFreeLoop
JMP ReturnPressed

UpdateDataFreeLoop
INC dataFreeDigitThree
LDA dataFreeDigitThree
CMP #$3A
BNE DecrementDataFreeCounterAndLoop
LDA #0'
STA dataFreeDigitThree
INC dataFreeDigitTwo
LDA dataFreeDigitTwo
CMP #$3A
BNE DecrementDataFreeCounterAndLoop
LDA #0'
STA dataFreeDigitTwo
INC dataFreeDigitOne
DecrementDataFreeCounterAndLoop
DEX
BNE UpdateDataFreeLoop
JSR UpdateDataFreeDisplay

LDA customPromptsActive
BEQ CheckForInputDuringPrompt

LDA lastKeyPressed
CMP #NO_KEY_PRESSED
BEQ ResetPromptAndReturn
RTS

ResetPromptAndReturn
LDA #NOT_ACTIVE
STA customPromptsActive
ReturnFromPromptRoutine
RTS

CheckForInputDuringPrompt
LDA lastKeyPressed
CMP #NO_KEY_PRESSED
BEQ ReturnFromPromptRoutine

LDX #ACTIVE
STX customPromptsActive

CMP #KEY_LEFT
BEQ LeftKeyPressedDuringPrompt

CMP #KEY_RETURN
BEQ ReturnPressed

CMP #KEY_SPACE
BNE ReturnFromUpdateDataFree

JSR UpdateDataFreeDisplay

LDA currentDataFree
STA dataFreeForSequencer

LDA currentSequencePtrLo
STA preSequencePtrLo
LDA currentSequencePtrHi
STA preSequencePtrHi

```

```

LDA #$00
STA currentVariableMode
STA customPromptsActive
STA sequencerActive

LDY #$02
LDA #$FF
STA (<currentSequencePtrLo>),Y

ReturnFromUpdateDataFree
RTS

LeftKeyPressedDuringPrompt
LDY #$02
LDA shiftKey
AND #$01
BEQ ShiftAndLeftPressed

LDA #$C0
JMP StoreInSequenceData

ShiftAndLeftPressed
LDA cursorXPosition

StoreInSequenceData
STA (<currentSequencePtrLo>),Y

LDA cursorYPosition
INY
STA (<currentSequencePtrLo>),Y

LDA currentPatternElement
INY
STA (<currentSequencePtrLo>),Y

LDA currentSequencePtrLo
CLC
ADC #OFFSET_TO_NEXT_BURST
STA currentSequencePtrLo

LDA currentSequencePtrHi
ADC #$00
STA currentSequencePtrHi

DEC currentDataFree
RTS

; -----
; ReturnPressed
; -----
ReturnPressed
JSR UpdateDataFreeDisplay

LDA #$FF
LDY #$02
STA (<currentSequencePtrLo>),Y

LDA #NOT_ACTIVE
STA currentVariableMode
STA customPromptsActive
STA dataFreeForSequencer
STA sequencerActive

RTS

customPromptsActive .BYTE $00
; -----
; UpdateDataFreeDisplay
; -----
UpdateDataFreeDisplay
LDA dataFreeDigitOne
STA SCREEN_RAM + $03C6

LDA dataFreeDigitTwo
STA SCREEN_RAM + $03C7

LDA dataFreeDigitThree
STA SCREEN_RAM + $03C8
RTS

; -----
; LoadBurstData
; -----
LoadBurstData
LDA #NOT_ACTIVE
STA currentVariableMode
TAY
LDA (<currentSequencePtrLo>),Y
STA prevSymmetrySetting
INY
LDA (<currentSequencePtrLo>),Y
STA burstSmoothingDelay

LoadNextBurstPosition
LDY #$02
INC currentStepCount
LDA currentStepCount
CMP bufferLength
BNE DontResetStepCountToZero

LDA #$00
STA currentStepCount

DontResetStepCountToZero
LDX currentStepCount
LDA currentColorIndexArray,X
CMP #$FF
BEQ LoadBurstToBuffers

LDA previousIndexToPixelBuffers
AND trackingActivated
BEQ MoveToNextBurstPosition

STA currentStepCount
TAX
LDA currentColorIndexArray,X
CMP #$FF
BNE MoveToNextBurstPosition

LoadBurstToBuffers
LDA baseLevel
STA currentColorIndexArray,X
LDA pixelYPositionArray,X
INY
LDA (<currentSequencePtrLo>),Y
STA pixelYPositionArray,X
INY
LDA (<currentSequencePtrLo>),Y
STA patternIndexArray,X
LDA burstSmoothingDelay
STA initialSmoothingDelayArray,X
STA smoothingDelayArray,X
STA prevSymmetrySetting
STA symmetrySettingForStepCount,X

MoveToNextBurstPosition
LDA currentSequencePtrLo
CLC
ADC #$03
STA currentSequencePtrLo
LDA currentSequencePtrHi
ADC #$00
STA currentSequencePtrHi
LDY #$02
LDA (<currentSequencePtrLo>),Y
CMP #$FF
BEQ FinishedLoadingBurstData
JMP LoadNextBurstPosition

FinishedLoadingBurstData
LDA #$00
STA sequencerActive
RTS

; -----
; ActivateSequencer
; -----
ActivateSequencer
LDA >>startOfSequencerData
STA currentSequencePtrHi
LDA <>startOfSequencerData
STA currentSequencePtrLo
LDA #GENERIC_ACTIVE
STA sequencerActive
LDA shiftPressed
AND #SHFL1_PRESSED
BNE ShiftPressedDoProgramSequencer

LDA sequencerSpeed
STA stepsRemainingInSequencerSequence
LDA #NOT_ACTIVE
STA currentVariableMode
JSR DisplaySequencerState
RTS

ShiftPressedDoProgramSequencer
LDA dataFreeForSequencer
BEQ SetUpNewSequencer
LDA dataFreeForSequencer
STA currentDataFree
LDA prevSequencePtrLo
STA currentSequencePtrLo
LDA prevSequencePtrHi
STA currentSequencePtrHi
JMP DisplaySeqFree

SetUpNewSequencer
LDA #$FF
STA currentDataFree
LDA currentSymmetrySetting
LDY #$00
STA (<currentSequencePtrLo>),Y
LDA smoothingDelay
INY
STA (<currentSequencePtrLo>),Y

DisplaySeqFree
JSR ClearLastLineOfScreen
LDX #$00
Loop LDA txtSeqFree,X
STA statusLineBuffer,X
INX
CPX #DISPLAY_LINE_LENGTH
BNE _Loop
JMP WriteLastLineBufferToScreen

txtSeqFree
.TEXT 'SEQU: 000 FREE '
; -----
; DisplaySequencerState
; -----
DisplaySequencerState
LDA sequencerActive
AND #$01
ASL
ASL
ASL
TAY
JSR ClearLastLineOfScreen
LDX #$00
Loop LDA txtSeqFree,X
STA statusLineBuffer,X
INX
CPX #DISPLAY_LINE_LENGTH
BNE _Loop
JMP WriteLastLineBufferToScreen

txtSequencer
.TEXT 'SEQUENCER OFF '
.TEXT 'SEQUENCER ON '

```

## increasing the dose

---

```

dataFreeForSequencer .BYTE $00
prevSequencePtrLo .BYTE $00
prevSequencePtrHi .BYTE $00
currentPulseWidth .BYTE $00

;-----
; StopOrStartRecording
;-----
StopOrStartRecording
LDA #dynamicStorage
STA recordingStorageHiPtr

LDA #<dynamicStorage
STA recordingStorageLoPtr

LDA #$01
STA recordingOffset

LDA shiftPressed
AND #$SHIFT_PRESSED
STA shiftPressed

LDA playbackOrRecordActive
ORA shiftPressed
EOR #$02
STA playbackOrRecordActive

AND #PLAYING_BACK
BNE UpdateRecordingDisplay

JMP DisplayStoppedRecording

UpdateRecordingDisplay
LDA playbackOrRecordActive
AND #$01
ASL
ASL
ASL
TAY

JSR ClearLastLineOfScreen

LDX #$00
_Loop LDA txtPlayBackRecord,Y
STA statusLineBuffer,X
INY
INX
CPX #DISPLAY_LINE_LENGTH
BNE _Loop

JSR WriteLastLineBufferToScreen

LDA playbackOrRecordActive
CMP #RECORDING
BNE ResetStateAndReturn

;-----
; InitializeDynamicStorage
;-----
InitializeDynamicStorage
LDA #dynamicStorage
STA dynamicStorageLoPtr

LDA #dynamicStorage
STA dynamicStorageHiPtr

LDY #$00
TYA

LDX #$50
DynamicStorageInitLoop
STA (dynamicStorageLoPtr),Y
DEY
BNE DynamicStorageInitLoop

INC dynamicStorageHiPtr
DEX
BNE DynamicStorageInitLoop

LDA #$FF
STA dynamicStorage
LDA #$01
STA dynamicStorage + $01
LDA cursorXPosition
STA previousCursorPosition
LDA cursorYPosition
STA previousCursorPosition
RTS

ResetStateAndReturn
LDY #BLACK
STA currentColorToPaint
JSR PaintCursorAtCursorPosition
LDA previousCursorPosition
STA cursorXPosition
LDA previousCursorPosition
STA cursorYPosition
LDA #GENERIC_ACTIVE
STA displaySavePromptActive
RTS

txtPlayBackRecord
.TEXT 'PLAYING BACK','$AE,$AE,$AE,$AE',
      '$BYE RECORDING','$AE,$AE,$AE,$AE,
      '$AE,$AE'

```

```

;-----
; DisplayStoppedRecording
;-----
DisplayStoppedRecording
LDA #NOT_ACTIVE
STA playbackOrRecordActive
STA #$D020
STA displaySavePromptActive
TAY
JSR ClearLastLineOfScreen
Loop LDA txtStopped,Y
STA statusLineBuffer,Y
INY
CPY #DISPLAY_LINE_LENGTH
BNE _Loop
JMP WriteLastLineBufferToScreen

.ENC "petscii"
txtStopped
.TEXT 'STOPPED'
.ENC "none"
playbackOrRecordActive
.BYTE $00

;-----
; RecordJoystickMovements
;-----
RecordJoystickMovements
LDA $D000
STA lastJoystickInput
LDY #$00
CMP (recordingStorageLoPtr),Y
BEQ StoreJoystickMovement

MoveStoragePointer
LDA recordingStorageLoPtr
CLC
ADC #$02
STA recordingStorageLoPtr
LDA recordingStorageHiPtr
ADC #$00
STA recordingStorageHiPtr
CMP #$80
BNE ResetStoragePointer
LDA #$00
STA storageOfSomeKind
JMP DisplayStoppedRecording

ResetStoragePointer
LDY #$01
TYA
STA (recordingStorageLoPtr),Y
LDA $D000
DEY
STA (recordingStorageLoPtr),Y
SEC
SEC #$30
CLC
RDR
CLC
ROR
CLC
ROR
TAX
LDA colorBarValues,X
STA #$D020
RTS

StoreJoystickMovement
INY
LDA (recordingStorageLoPtr),Y
CLC
ADC #$01
STA (recordingStorageLoPtr),Y
CMP #$FF
BEQ MoveStoragePointer
RTS

;-----
; GetJoystickInput
;-----
GetJoystickInput
LDA playbackOrRecordActive
BEQ MaybeInDemoMode
CMP #RECORDING
BNE PlayBackInputs
JMP RecordJoystickMovements

PlayBackInputs
JMP PlaybackRecordedJoystickInputs

MaybeInDemoMode
LDA demoModeActive
BEQ GetInputFromJoystick

InDemoMode
JMP MaybePerformRandomMovement

GetInputFromJoystick
LDA $D000
STA lastJoystickInput
RTS

```

```

PlaybackRecordedJoystickInputs
DEC recordingOffset
BEQ GetRecordedByte

LDY #$00
LDA (recordingStorageLoPtr),Y
STA lastJoystickInput
RTS

GetRecordedByte
LDA recordingStorageLoPtr
CLC
ADC #$02
STA recordingStorageLoPtr

LDA recordingStorageHiPtr
ADC #$00
STA recordingStorageHiPtr
CMP #$80
BEQ NoMoreBytes
LDY #$01
LDA (recordingStorageLoPtr),Y
BEQ NoMoreBytes

STA recordingOffset
DEY
LDA (recordingStorageLoPtr),Y
STA lastJoystickInput
RTS

NoMoreBytes
LDA #dynamicStorage
STA recordingStorageHiPtr
LDA #dynamicStorage
STA recordingStorageLoPtr
LDA #$01
STA recordingOffset
LDY #$00
STA currentColorToPaint
JSR PaintCursorAtCursorPosition
LDA previousCursorPosition
STA cursorXPosition
LDA previousCursorPosition
STA cursorYPosition
RTS

recordingOffset .BYTE $00
previousCursorPosition .BYTE $00
previousCursorPosition .BYTE $00
customPatternIndex .BYTE $00
displaySavePromptActive .BYTE $00
txtDefineAllLevelPixels

.TEXT 'DEFINE ALL LEVEL PIXELS'
;-----
; MaybeEditCustomPattern
;-----
MaybeEditCustomPattern
TIA
AND #$08
BEQ EditCustomPattern
RTS

EditCustomPattern
LDA #CUSTOM_PRESET_ACTIVE
STA currentVariableMode

LDA #$00
STA customPatternLoPtr
STA displaySavePromptActive
LDA customPatternHiPtrArray,X
STA customPatternHiPtr
TIA
CLC
ADC #$08
STA customPatternIndex
JSR ClearLastLineOfScreen

LDX #$00
_Loop
LDA txtDefineAllLevelPixels,X
STA statusLineBuffer,X
INY
CPX #NUM_ROWS + 1
BNE _Loop

JSR WriteLastLineBufferToScreen
LDA #$06
STA initialBaseLevelForCustomPresets
LDY #$00
TYA
STA (customPatternLoPtr),Y
INY
LDA #$55
STA (customPatternLoPtr),Y
LDY #$81
STA (customPatternLoPtr),Y
DEY
LDA #$00
STA (customPatternLoPtr),Y
LDY #$01
STA currentIndexToColorValues
LDA #$01
STA currentIndexToPresetValue
LDA #CUSTOM_PRESET_MODE_ACTIVE
STA currentModeActive
RTS

```

**Lines 1643-2300.** This section on the opposite and following two pages deals with the mechanics of allowing the player to record and load play sessions, edit custom patterns, and program a new preset.

In this regard it is largely uninteresting boilerplate. The entire last page for example deals solely with the details of saving a session to tape storage and loading a previously saved one.

We won't explore this code in any detail in later chapters.

**Jeffrey Says**



Shift-R to start recording, R to playback or stop: PSYCHEDELIA can record about half-an-hours' worth of joystick input in its memory. Start recording and play as normal—you get a coloured border whilst recording. When you've done enough press R to stop. Pressing R again starts playback. Try playing back under different parameters, different presets etc. Adjust parameters while you're playing back a display to see what happens. PSY drops out of Record automatically, if it runs out of memory. During playback it repeats the stored performance until R is pressed to halt it.

In addition to recording, this section of the code also deals with storing the configuration you've come up with to tape (should you be so minded). Can I suggest you take a moment to pause and reflect on the dedication and patience required to store and load your *Psychedelia* configurations to and from cassette tape? Not only that, but storing the parameters on a separate side of the tape from the motion configurations. Imagine yourself carefully labelling each side of the tape and storing it away in its plastic cassette box with something helpful scrawled in marker along the spine such as 'PSYCH CONFIG 4'. The past is another country: they futz with cassette tape there.

**Jeffrey Says**



You can SAVE your favorite presets, sequencer runs, bursts etc or stored joystick moves onto tape for later re-loading and use. To start the SAVE: press shift-S. You get the option if saving Parameters or Motion. Selecting PARAMETERS saves all presets, burst gens and sequencer, plus all user-defined shapes. Selecting MOTION saves the stored sequence of joystick moves used in the Record option. (Long performances take a little while!). The parameters are saved as GOATS and the motion as SHEEP (— suggest you use opposite sides of a short cassette to store these on). To LOAD in data you saved already: press shift-L. Prepare the tape and tell the program when you're ready to LOAD. The loader automatically sorts out the SHEEP from the GOATS.

# increasing the dose

---

```

;-----
; HandleCustomPreset
;-----
HandleCustomPreset
LDA #$13
STA cursorXPosition
LDA #$0C
STA cursorYPosition
JSR ReinitializeScreen

.Loop
LDA customPatternIndex
STA patternIndex

LDA initialBaseLevelForCustomPresets
STA currentValueInColorIndexArray

LDA #$00
STA currentSymmetrySettingForStep

LDA #$_13
STA pixelXPosition

LDA #$0C
STA pixelYPosition

JSR PaintStructureAtCurrentPosition

LDA initialBaseLevelForCustomPresets
BNE _Loop

JSR ReinitializeScreen

LDA #NOT_ACTIVE
STA currentModeActive
JMP MainPaintLoop

;-----
; CheckInputForCustomPresets
;-----
CheckInputForCustomPresets
LDA customPromptsActive
BEQ
    CheckForKeyPressDuringCustomPrompt

LDA lastKeyPressed
CMP #$_KEY_PRESSED
BEQ ResetCustomPromptsAndReturn
RTS

ResetCustomPromptsAndReturn
LDA #$00
STA customPromptsActive
ReturnFromOtherPrompts
RTS

CheckForKeyPressDuringCustomPrompt
LDA lastKeyPressed
CMP #$_KEY_PRESSED
BEQ ReturnFromOtherPrompts

LDA #$_GENERIC_ACTIVE
STA customPromptsActive

LDA lastKeyPressed
CMP #$_KEY_RETURN
BEQ EnterPressed

JMP MaybeLeftArrowPressed2

EnterPressed
INC currentIndexToPresetValue
LDA #$00
LDY currentIndexToPresetValue
STA (customPatternLoPtr),Y
PHA
TYA
CLC
ADC #$80
TAX
PLA
STA (customPatternLoPtr),Y
INY
LDA #$55
STA (customPatternLoPtr),Y
LDY currentIndexToPresetValue
STY currentIndexToPresetValue
INY
LDA (customPatternLoPtr),Y
SEC
SBC #$_13
STA (customPatternLoPtr),Y
INY
LDA #$55
STA (customPatternLoPtr),Y
STY currentIndexToPresetValue
INY
LDA #$55
STA (customPatternLoPtr),Y
DEC minIndexToColorValues
BEQ PressEnter
RTS

PressEnter
JMP EnterPressed

;-----
; GetCustomPatternElement
;-----
GetCustomPatternElement
JSR ClearLastLineOfScreen

LDX #$00
txtPatternLoop
LDA txtCustomPatterns,X
STA statusLineBuffer,X
INX
CPX #$0E
BNE txtPatternLoop

LDA currentPatternElement
AND #$07
CLC
ADC #$30
STA customPatternValueBufferPtr
JSR WriteLastLineBufferToScreen

.ENC "petaci"
txtCustomPatterns .TEXT 'USER SHAPE _0
,
.ENC "none"
pixelShapeIndex .BYTE $00
pixelShapePreset
    .BYTE BLOCK,CIRCLE,HEART,DIAMOND
    .BYTE CROSS,TOP_RIGHT,TRIANGLE,DONUT
    .BYTE CHECKER,ANDREWS,CROSS,LEFT_HALF
    .BYTE TOP_LEFT,BRACKET,FULL_CHECKER
    .BYTE BOTTOM_RIGHT,SQUARE
    .BYTE BOTTOM_RIGHT_SQUAREZ,
    SPACE,MAYBE,ASTERISK
    .BYTE $47,$4F,$41,$54,$53,$53,$48,$45
    .BYTE $45,$50

;-----
; DisplaySavePromptScreen
;-----
DisplaySavePromptScreen
LDA #$_13
JSR PRINT
LDA #$_GENERIC_ACTIVE
STA displaySavePromptActive
JSR InitializeScreenWithInitCharacter

.Loop
LDA tapeSavingInProgress
BEQ _Loop

MaybeSaveParameters
CMP #$_SAVE_PARAMETERS
BNE MaybeSaveMotions

SaveParameters
LDA #$01
LDX #$01
LDY #$01
JSR ROM_SETLFS

LDA #$05
LDX #$59
LDY #$1D
JSR ROM_SETNAM

LDA #$01
STA CURRENT_CHAR_COLOR
LDA #>presetSequenceData
STA presetHiPtr
LDA #<presetSequenceData
STA presetLoPtr

LDX #$FF
LDY #$CF
LDA #$FE
JSR ROM_SAVE
JMP ResetSavingStateAndReturn
RTS

ResetVarModeAndReturn
LDA #$00
STA currentVariableMode
JSR ClearLastLineOfScreen
ReturnFromLeftArrow
RTS

```

```

MaybeLeftArrowPressed2
CMP #$_KEY_LEFT
BNE ReturnFromLeftArrow

LDY currentIndexToPresetValue
LDA cursorXPosition
SEC
SBC #$_13
STA (customPatternLoPtr),Y
INY
LDA #$55
STA (customPatternLoPtr),Y
STY currentIndexToPresetValue
TYA
CLC
ADC #$7F
TAX
LDA cursorYPosition
SEC
SBC #$0C
STA (customPatternLoPtr),Y
INY
LDA #$55
STA (customPatternLoPtr),Y
DEC minIndexToColorValues
BEO PressEnter
RTS

PressEnter
JMP EnterPressed

;-----
; GetCustomPatternElement
;-----
GetCustomPatternElement
JSR ClearLastLineOfScreen

LDX #$00
txtPatternLoop
LDA txtCustomPatterns,X
STA statusLineBuffer,X
INX
CPX #$0E
BNE txtPatternLoop

LDA currentPatternElement
AND #$07
CLC
ADC #$30
STA customPatternValueBufferPtr
JSR WriteLastLineBufferToScreen

.ENC "petaci"
txtCustomPatterns .TEXT 'USER SHAPE _0
,
.ENC "none"
pixelShapeIndex .BYTE $00
pixelShapePreset
    .BYTE BLOCK,CIRCLE,HEART,DIAMOND
    .BYTE CROSS,TOP_RIGHT,TRIANGLE,DONUT
    .BYTE CHECKER,ANDREWS,CROSS,LEFT_HALF
    .BYTE TOP_LEFT,BRACKET,FULL_CHECKER
    .BYTE BOTTOM_RIGHT,SQUARE
    .BYTE BOTTOM_RIGHT_SQUAREZ,
    SPACE,MAYBE,ASTERISK
    .BYTE $47,$4F,$41,$54,$53,$53,$48,$45
    .BYTE $45,$50

;-----
; DisplaySavePromptScreen
;-----
DisplaySavePromptScreen
LDA #$_13
JSR PRINT
LDA #$_GENERIC_ACTIVE
STA displaySavePromptActive
JSR InitializeScreenWithInitCharacter

.Loop
LDA tapeSavingInProgress
BEQ _Loop

MaybeSaveParameters
CMP #$_SAVE_PARAMETERS
BNE MaybeSaveMotions

SaveParameters
LDA #$01
LDX #$01
LDY #$01
JSR ROM_SETLFS

LDA #$05
LDX #$59
LDY #$1D
JSR ROM_SETNAM

LDA #$01
STA CURRENT_CHAR_COLOR
LDA #>presetSequenceData
STA presetHiPtr
LDA #<presetSequenceData
STA presetLoPtr

LDX #$FF
LDY #$CF
LDA #$FE
JSR ROM_SAVE
JMP ResetSavingStateAndReturn
RTS

ReturnFromLeftArrow
JSR ROM_SETLFS

SaveMotions
LDA #$01
LDX #$01
LDY #$01
JSR ROM_SETNAM

LDA #$05
LDX #$5E
LDY #$1D
JSR ROM_SETLFS

LDA #WHITE
STA CURRENT_CHAR_COLOR

LDA #$80
STA presetHiPtr
STA presetTempHiPtr

LDA #$80
STA presetLoPtr
STA presetTempLoPtr

LDY #$00
Loop2 LDA (presetTempLoPtr),Y
BEQ ExitSaveLoop
INC presetTempLoPtr
BNE Loop2
INC presetTempHiPtr
JMP _Loop2

ExitSaveLoop
LDX presetTempLoPtr
LDY presetTempHiPtr

LDA #$FE
JSR ROM_SAVE
JMP ResetSavingStateAndReturn

ContinueSave
LDA #$01
LDX #$01
LDY #$01
JSR ROM_SETLFS

LDA #$00
JSR ROM_SETNAM

LDA #WHITE
STA CURRENT_CHAR_COLOR
LDA #$80
JSR ROM_LOAD

JSR ROM_READST
AND #$10
BEQ ResetSavingStateAndReturn
JSR DisplayLoadOrAbort

ResetSavingStateAndReturn
LDA #NOT_ACTIVE
STA currentModeActive
STA displaySavePromptActive
STA tapeSavingInProgress

JSR ROM_CLALL

JSR ReinitializeScreen

JMP MainPaintLoop

RTS

;-----
; PromptToSave
;-----
PromptToSave
LDA stepsRemainingInSequencerSequence
BNE ReturnFromPromptToSave

LDA playbackOrRecordActive
CMP #$_PLAYING_BACK
BEQ ReturnFromPromptToSave

LDA #$_SAVING_ACTIVE
STA currentVariableMode

LDX #$00
Loop LDA txtSavePrompt,X
STA statusLineBuffer,X
INX
CPX #NUM_COLS
BNE _Loop

LDA #NOT_ACTIVE
STA tapeSavingInProgress

JSR WriteLastLineBufferToScreen
ReturnFromPromptToSave
RTS

txtSavePrompt
.TEXT " SAVE (P)ARAMETERS, (M)OTION, (A)BORT? "

```

```

;-----
; CheckInputWhileSavePromptActive
;-----
CheckInputWhileSavePromptActive
LDA currentVariableMode
CMP #$SAVING_ACTIVE
BEQ MaybeAbort
BNE RTS

MaybeAbort
LDA lastKeyPressed
CMP #KEY_A
BNE MaybeM_Pressed

LDA #NOT_ACTIVE
STA currentModeActive

ResetStateAndClearPrompt
LDA #NOT_ACTIVE
STA currentVariableMode

JMP ClearLastLineOfScreen

MaybeM_Pressed
CMP #KEY_M
BNE MaybeP_Pressed

LDA #SAVE_MOTIONS
STA tapeSavingInProgress

LDA #$SAVE_PROMPT_MODE_ACTIVE
STA currentModeActive

JMP ResetStateAndClearPrompt

MaybeP_Pressed
CMP #KEY_P
BNE ReturnFromSave

LDA #SAVE_PARAMETERS
STA tapeSavingInProgress

LDA #$SAVE_PROMPT_MODE_ACTIVE
STA currentModeActive

JMP ResetStateAndClearPrompt

ReturnFromSave
RTS

tapeSavingInProgress .BYTE $00
;-----
; DisplayLoadOrAbort
;-----
DisplayLoadOrAbort
LDA stepsRemainingInSequencerSequence
BNE ReturnFromSave

LDA playbackOrRecordActive
CMP #PLAYING_BACK
BEQ ReturnFromSave

LDA #LOADING_ACTIVE
STA currentVariableMode

LDX #$00
DisplayLoadAbortLoop
LDA txtContinueLoadOrAbort,X
STA statusLineBuffer,X
IMX
CPX #NUM_COLS
BNE DisplayLoadAbortLoop

LDA #NOT_ACTIVE
STA tapeSavingInProgress
JMP WriteLastLineBufferToScreen

;-----
; CheckInputWhileLoadAbortActive
;-----
CheckInputWhileLoadAbortActive
LDA lastKeyPressed
CMP #KEY_A
BNE

    MaybeCPressedWhileLoadAbortActive

LDA #NOT_ACTIVE
STA currentVariableMode
STA tapeSavingInProgress
STA currentModeActive
JMP ClearLastLineOfScreen

MaybeCPressedWhileLoadAbortActive
CMP #KEY_C
BNE

    ReturnFromInputWhileLoadAbortActive

LDA #CONTINUE_SAVE
STA tapeSavingInProgress
LDA #NOT_ACTIVE
STA currentVariableMode
LDA #$SAVE_PROMPT_MODE_ACTIVE
STA currentModeActive
JMP ClearLastLineOfScreen

ReturnFromInputWhileLoadAbortActive
RTS

```

```

txtContinueLoadOrAbort
.TEXT '(C)ONTINUE LOADE OR (A)BORT?'
        '
demoModeActive .BYTE $00
joystickInputDebounce .BYTE $01
joystickInputRandomizer .BYTE $10
        '
; MaybePerformRandomMovement
;-----
MaybePerformRandomMovement
DEC joystickInputDebounce
BEQ PerformRandomJoystickMovement
RTS

PerformRandomJoystickMovement
JSR PutRandomByteInAccumulator
AND #$0F
ORA #$01
STA joystickInputDebounce

LDA joystickInputRandomizer
EOR #$10
STA joystickInputRandomizer

JSR PutRandomByteInAccumulator
AND #$0F
ORA joystickInputRandomizer
EOR #$0F
STA lastJoystickInput
DEC demoModeCountDownToChangePreset
BEO SelectRandomPreset
RTS

SelectRandomPreset
JSR PutRandomByteInAccumulator
AND #$07
ADC #$20
STA demoModeCountDownToChangePreset

JSR PutRandomByteInAccumulator
AND #$0F
TAX

LDA #$00
STA shiftPressed
JMP SelectNewPreset

;-----
; MaybeDisplayDemoModeMessage
;-----
MaybeDisplayDemoModeMessage
LDA demoModeActive
BNE DemoModeMessage
JMP ClearLastLineOfScreen

DisplayDemoModeMessage
LDX #$00
DisplayDemoMsgLoop
LDA demoMessage,X
STA statusLineBuffer,X
INX
CPX #NUM_COLS
BNE DisplayDemoMsgLoop
JMP WriteLastLineBufferToScreen

demoMessage
.TEXT "PSYCHEDELIA BY JEFF MINTER"
* = $1FA9
demoModeCountDownToChangePreset .BYTE
$20
        '
; NMIInterruptHandler
;-----
NMIInterruptHandler
LDX <CalledFromNMI
TXS
LDA #>CalledFromNMI
PHA
LDA #$30
PHA
LDA #$23
PHA
RTI
        '
; MovePresetDataIntoPosition
;-----
MovePresetDataIntoPosition
LDY #$00
TYA
STA copyFromLoPtr
STA copyToLoPtr
LDA #>presetSequenceDataSource
STA copyFromHiPtr
LDA #>presetSequenceData
STA copyToHiPtr

LDY #$10
Loop LDA (copyFromLoPtr),Y
STA (copyToLoPtr),Y
DEY
BNE Loop
RTS

```

## increasing the dose

---

```

preset1
.BYTE $00 : unusedPresetByte
.BYTE $0C : smoothingDelay
.BYTE $02 : cursorSpeed
.BYTE $28 : bufferLength
.BYTE $04 : pulseSpeed
.BYTE $0E : indexForColorBarDisplay
.BYTE $07 : lineWidth
.BYTE $08 : sequencerSpeed
.BYTE $01 : pulseWidth
.BYTE $07 : baseLevel
; presetColorValuesArray:
.BYTE BLACK,BROWN,RED,ORANGE,PURPLE,
LTRED,BLUE,LTBLUE
.BYTE $FF : trackingActivated
.BYTE $00 : lineModeActivated
.BYTE $01 : presetIndex
.BYTE $01 : currentPatternElement
.BYTE $04 : currentSymmetrySetting

preset2
.BYTE $00 : unusedPresetByte
.BYTE $0B : smoothingDelay
.BYTE $02 : cursorSpeed
.BYTE $28 : bufferLength
.BYTE $01 : pulseSpeed
.BYTE $01 : indexForColorBarDisplay
.BYTE $07 : lineWidth
.BYTE $08 : sequencerSpeed
.BYTE $01 : pulseWidth
.BYTE $07 : baseLevel
; presetColorValuesArray:
.BYTE BLACK,BLUE,LTBLUE,CYAN,LTGREEN,
GREEN,LTBLUE,BLUE
.BYTE $FF : trackingActivated
.BYTE $00 : lineModeActivated
.BYTE $05 : presetIndex
.BYTE $05 : currentPatternElement
.BYTE $01 : currentSymmetrySetting

preset3
.BYTE $00 : unusedPresetByte
.BYTE $04 : smoothingDelay
.BYTE $02 : cursorSpeed
.BYTE $26 : bufferLength
.BYTE $01 : pulseSpeed
.BYTE $01 : indexForColorBarDisplay
.BYTE $07 : lineWidth
.BYTE $08 : sequencerSpeed
.BYTE $01 : pulseWidth
.BYTE $07 : baseLevel
; presetColorValuesArray:
.BYTE BLACK,RED,PURPLE,LTRED,LTGREEN,
CYAN,LTBLUE,BLUE
.BYTE $00 : trackingActivated
.BYTE $00 : lineModeActivated
.BYTE $0E : presetIndex
.BYTE $01 : currentPatternElement
.BYTE $02 : currentSymmetrySetting

preset4
.BYTE $00 : unusedPresetByte
.BYTE $0C : smoothingDelay
.BYTE $01 : cursorSpeed
.BYTE $2B : bufferLength
.BYTE $01 : pulseSpeed
.BYTE $07 : indexForColorBarDisplay
.BYTE $05 : lineWidth
.BYTE $0A : sequencerSpeed
.BYTE $01 : pulseWidth
.BYTE $07 : baseLevel
; presetColorValuesArray:
.BYTE BLACK,GRAV1,BLUE,GRAZ2,PURPLE,
GRAY3,CYAN,WHITE
.BYTE $00 : trackingActivated
.BYTE $00 : lineModeActivated
.BYTE $01 : presetIndex
.BYTE $01 : currentPatternElement
.BYTE $01 : currentSymmetrySetting

preset5
.BYTE $00 : unusedPresetByte
.BYTE $0C : smoothingDelay
.BYTE $02 : cursorSpeed
.BYTE $2B : bufferLength
.BYTE $01 : pulseSpeed
.BYTE $07 : indexForColorBarDisplay
.BYTE $07 : lineWidth
.BYTE $0C : sequencerSpeed
.BYTE $06 : pulseWidth
.BYTE $07 : baseLevel
; presetColorValuesArray:
.BYTE BLACK,GRAV1,BLUE,GRAZ2,PURPLE,
GRAY3,CYAN,WHITE
.BYTE $00 : trackingActivated
.BYTE $00 : lineModeActivated
.BYTE $06 : presetIndex
.BYTE $06 : currentPatternElement
.BYTE $03 : currentSymmetrySetting

```

```

preset6
.BYTE $00 : unusedPresetByte
.BYTE $0F : smoothingDelay
.BYTE $02 : cursorSpeed
.BYTE $30 : bufferLength
.BYTE $04 : pulseSpeed
.BYTE $07 : indexForColorBarDisplay
.BYTE $07 : lineWidth
.BYTE $0F : sequencerSpeed
.BYTE $01 : pulseWidth
.BYTE $07 : baseLevel
; presetColorValuesArray:
.BYTE BLACK,BLUE,RED,PURPLE,GREEN,CYAN
YELLOW,WHITE
.BYTE $FF : trackingActivated
.BYTE $00 : lineModeActivated
.BYTE $03 : presetIndex
.BYTE $03 : currentPatternElement
.BYTE $04 : currentSymmetrySetting

preset7
.BYTE $00 : unusedPresetByte
.BYTE $0B : smoothingDelay
.BYTE $01 : cursorSpeed
.BYTE $1C : bufferLength
.BYTE $02 : pulseSpeed
.BYTE $07 : indexForColorBarDisplay
.BYTE $07 : lineWidth
.BYTE $09 : sequencerSpeed
.BYTE $01 : pulseWidth
.BYTE $07 : baseLevel
; presetColorValuesArray:
.BYTE BLACK,YELLOW,CYAN,LTBLUE,BLUE,
RED,PURPLE,LTRED
.BYTE $00 : trackingActivated
.BYTE $00 : lineModeActivated
.BYTE $07 : presetIndex
.BYTE $07 : currentPatternElement
.BYTE $01 : currentSymmetrySetting

preset8
.BYTE $00 : unusedPresetByte
.BYTE $04 : smoothingDelay
.BYTE $01 : cursorSpeed
.BYTE $28 : bufferLength
.BYTE $02 : pulseSpeed
.BYTE $01 : indexForColorBarDisplay
.BYTE $07 : lineWidth
.BYTE $0A : sequencerSpeed
.BYTE $01 : pulseWidth
.BYTE $07 : baseLevel
; presetColorValuesArray:
.BYTE BLACK,ORANGE,BROWN,GREEN,CYAN,
LTGREEN,LTBLUE,BLUE
.BYTE $FF : trackingActivated
.BYTE $00 : lineModeActivated
.BYTE $0E : presetIndex
.BYTE $01 : currentPatternElement
.BYTE $03 : currentSymmetrySetting

preset9
.BYTE $00 : unusedPresetByte
.BYTE $11 : smoothingDelay
.BYTE $01 : cursorSpeed
.BYTE $2D : bufferLength
.BYTE $07 : pulseSpeed
.BYTE $01 : indexForColorBarDisplay
.BYTE $07 : lineWidth
.BYTE $0C : sequencerSpeed
.BYTE $01 : pulseWidth
.BYTE $07 : baseLevel
; presetColorValuesArray:
.BYTE BLACK,BLUE,CYAN,BLUE,CYAN,BLUE,
CYAN,BLUE
.BYTE $FF : trackingActivated
.BYTE $00 : lineModeActivated
.BYTE $07 : presetIndex
.BYTE $07 : currentPatternElement
.BYTE $04 : currentSymmetrySetting

preset10
.BYTE $00 : unusedPresetByte
.BYTE $01 : smoothingDelay
.BYTE $02 : cursorSpeed
.BYTE $1F : bufferLength
.BYTE $02 : pulseSpeed
.BYTE $04 : lineWidth
.BYTE $09 : indexForColorBarDisplay
.BYTE $08 : sequencerSpeed
.BYTE $01 : pulseWidth
.BYTE $07 : baseLevel
; presetColorValuesArray:
.BYTE BLACK,BLUE,RED,RED,PURPLE,LTRED,
ORANGE,BROWN
.BYTE $FF : trackingActivated
.BYTE $01 : lineModeActivated
.BYTE $00 : presetIndex
.BYTE $00 : currentPatternElement
.BYTE $00 : currentSymmetrySetting

```

```

preset11
.BYTE $00 : unusedPresetByte
.BYTE $01 : smoothingDelay
.BYTE $01 : cursorSpeed
.BYTE $13 : bufferLength
.BYTE $06 : pulseSpeed
.BYTE $01 : indexForColorBarDisplay
.BYTE $07 : lineWidth
.BYTE $05 : sequencerSpeed
.BYTE $05 : pulseWidth
.BYTE $07 : baseLevel
; presetColorValuesArray:
.BYTE BLACK,RED,PURPLE,GREEN,CYAN
YELLOW,WHITE
.BYTE $FF : trackingActivated
.BYTE $00 : lineModeActivated
.BYTE $0F : presetIndex
.BYTE $0F : currentPatternElement
.BYTE $04 : currentSymmetrySetting

preset12
.BYTE $00 : unusedPresetByte
.BYTE $0C : smoothingDelay
.BYTE $02 : cursorSpeed
.BYTE $28 : bufferLength
.BYTE $01 : pulseSpeed
.BYTE $02 : indexForColorBarDisplay
.BYTE $07 : lineWidth
.BYTE $09 : sequencerSpeed
.BYTE $01 : pulseWidth
.BYTE $07 : baseLevel
; presetColorValuesArray:
.BYTE BLACK,BLUE,LTBLUE,CYAN,LTGREEN,
YELLOW,PURPLE,RED
.BYTE $00 : trackingActivated
.BYTE $00 : lineModeActivated
.BYTE $04 : presetIndex
.BYTE $04 : currentPatternElement
.BYTE $01 : currentSymmetrySetting

preset13
.BYTE $00 : unusedPresetByte
.BYTE $0B : smoothingDelay
.BYTE $01 : cursorSpeed
.BYTE $1C : bufferLength
.BYTE $02 : pulseSpeed
.BYTE $04 : indexForColorBarDisplay
.BYTE $07 : lineWidth
.BYTE $0B : sequencerSpeed
.BYTE $01 : pulseWidth
.BYTE $07 : baseLevel
; presetColorValuesArray:
.BYTE BLACK,BLUE,LTBLUE,BLUE,
CYAN,LTGREEN,LTBLUE,BLUE
.BYTE $00 : trackingActivated
.BYTE $00 : lineModeActivated
.BYTE $00 : presetIndex
.BYTE $03 : currentPatternElement
.BYTE $04 : currentSymmetrySetting

preset14
.BYTE $00 : unusedPresetByte
.BYTE $0C : smoothingDelay
.BYTE $01 : cursorSpeed
.BYTE $2D : bufferLength
.BYTE $01 : pulseSpeed
.BYTE $04 : indexForColorBarDisplay
.BYTE $07 : lineWidth
.BYTE $0D : sequencerSpeed
.BYTE $01 : pulseWidth
.BYTE $07 : baseLevel
; presetColorValuesArray:
.BYTE BLACK,RED,BROWN,ORANGE,PURPLE,
CYAN,LTBLUE,BLUE
.BYTE $FF : trackingActivated
.BYTE $00 : lineModeActivated
.BYTE $04 : presetIndex
.BYTE $04 : currentPatternElement
.BYTE $02 : currentSymmetrySetting

preset15
.BYTE $00 : unusedPresetByte
.BYTE $03 : smoothingDelay
.BYTE $01 : cursorSpeed
.BYTE $20 : bufferLength
.BYTE $06 : pulseSpeed
.BYTE $01 : indexForColorBarDisplay
.BYTE $07 : lineWidth
.BYTE $00 : sequencerSpeed
.BYTE $01 : pulseWidth
.BYTE $07 : baseLevel
; presetColorValuesArray:
.BYTE BLACK,BLUE,RED,PURPLE,GREEN,CYAN
YELLOW,WHITE
.BYTE $FF : trackingActivated
.BYTE $01 : lineModeActivated
.BYTE $04 : presetIndex
.BYTE $04 : currentPatternElement
.BYTE $04 : currentSymmetrySetting

```

**Lines 2300-2800.** With the exception of the pattern definitions sprinkled through the code base Psychedelia's data is stored contiguously at the end of the code base. It starts with the section of memory reserved for defining the 16 available preset. We go into the operation of these presets in detail in the chapter '[particular presets](#)'.

Each preset (numbered from 0 to 15 opposite) consists of a bundle of different settings that can be applied together to produce a certain effect. This enables the player to define and save a combination of settings that give a pleasing effect. These can then be invoked with a single key-stroke.

There are 16 different settings bundled together in each preset. Note that the first byte is ignored. This is because the starting index for reading data out of the preset starts at 1 rather than 0, so it's purely for programming convenience that this is the case:

```
preset15
.BYTE $00 ; unusedPresetByte
.BYTE $03 ; smoothingDelay
.BYTE $01 ; cursorSpeed
.BYTE $1F ; bufferLength
.BYTE $06 ; pulseSpeed
.BYTE $01 ; indexForColorBarDisplay
.BYTE $07 ; lineWidth
.BYTE $00 ; sequencerSpeed
.BYTE $01 ; pulseWidth
.BYTE $07 ; baseLevel
; presetColorValuesArray:
.BYTE BLACK,BLUE,RED,PURPLE,GREEN,CYAN,YELLOW,WHITE
.BYTE $FF ; trackingActivated
.BYTE $00 ; lineModeActivated
.BYTE $04 ; presetIndex
.BYTE $04 ; currentPatternElement
.BYTE $04 ; currentSymmetrySetting
```

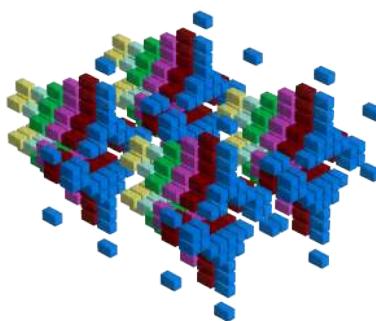


Figure 4.2: Preset 15 come to life.

## increasing the dose

```

burstGeneratorF1
; currentSymmetrySetting
.BYTE _Y_AXIS_SYMMETRY
; smoothingDelay
.BYTE $0C
; Burst Position 1
.BYTE $06,$06
; Index to pattern
.BYTE PULSTAR
; Burst Position 2
.BYTE $11,$0D
; Index to pattern
.BYTE PULSTAR
; Burst Position 3
.BYTE $06,$11
; Index to pattern
.BYTE PULSTAR
; Burst Position 4
.BYTE $FF,$0B
; Index to pattern
.BYTE PULSTAR
; Burst Position 5
.BYTE $FF,$00
; Index to pattern
.BYTE $FF
; Burst Position 6
.BYTE $21,$06
; Index to pattern
.BYTE $00
; Burst Position 7
.BYTE $06,$01
; Index to pattern
.BYTE $06
; Burst Position 8
.BYTE $41,$FF
; Index to pattern
.BYTE $00
; Burst Position 9
.BYTE $01,$01
; Index to pattern
.BYTE $06
; Burst Position 10
.BYTE $01,$06
; Index to pattern
.BYTE $00

```

```

burstGeneratorF2
; currentSymmetrySetting
.BYTE _Y_AXIS_SYMMETRY
; smoothingDelay
.BYTE $0C
; Burst Position 1
.BYTE $13,$08
; Index to pattern
.BYTE STARONE
; Burst Position 2
.BYTE $07,$0F
; Index to pattern
.BYTE SHOOTME
; Burst Position 3
.BYTE $FF,$00
; Index to pattern
.BYTE MULTICROSS
; Burst Position 4
.BYTE $01,$2A
; Index to pattern
.BYTE $41
; Burst Position 5
.BYTE $02,$00
; Index to pattern
.BYTE $00
; Burst Position 6
.BYTE $62,$FF
; Index to pattern
.BYTE $41
; Burst Position 7
.BYTE $06,$40
; Index to pattern
.BYTE $00
; Burst Position 8
.BYTE $6B,$04
; Index to pattern
.BYTE $41
; Burst Position 9
.BYTE $FF,$00
; Index to pattern
.BYTE $FF
; Burst Position 10
.BYTE $00,$FF
; Index to pattern
.BYTE $00

```

```

burstGeneratorF3
; currentSymmetrySetting
.BYTE QUAD_SYMMETRY
; smoothingDelay
.BYTE $01
; Burst Position 1
.BYTE $06,$01
; Index to pattern
.BYTE LALLAMITA
; Burst Position 2
.BYTE $FF,$01
; Index to pattern
.BYTE LALLAMITA
; Burst Position 3
.BYTE $08,$01
; Index to pattern
.BYTE $01
; Burst Position 4
.BYTE $08,$01
; Index to pattern
.BYTE $02
; Burst Position 5
.BYTE $08,$01
; Index to pattern
.BYTE $02
; Burst Position 6
.BYTE $08,$01
; Index to pattern
.BYTE $02
; Burst Position 7
.BYTE $08,$01
; Index to pattern
.BYTE $02
; Burst Position 8
.BYTE $08,$01
; Index to pattern
.BYTE $02
; Burst Position 9
.BYTE $08,$03
; Index to pattern
.BYTE $02
; Burst Position 10
.BYTE $08,$03
; Index to pattern
.BYTE $02

```

```

burstGeneratorF4
; currentSymmetrySetting
.BYTE _Y_AXIS_SYMMETRY
; smoothingDelay
.BYTE $11
; Burst Position 1
.BYTE $12,$09
; Index to pattern
.BYTE CUSTOMPATTERN0
; Burst Position 2
.BYTE $FF,$08
; Index to pattern
.BYTE STTWNO
; Burst Position 3
.BYTE $02,$08
; Index to pattern
.BYTE $03
; Burst Position 4
.BYTE $02,$08
; Index to pattern
.BYTE $03
; Burst Position 5
.BYTE $02,$FF
; Index to pattern
.BYTE $00
; Burst Position 6
.BYTE $00,$00
; Index to pattern
.BYTE $00
; Burst Position 7
.BYTE $01,$24
; Index to pattern
.BYTE $00
; Burst Position 8
.BYTE $00,$01
; Index to pattern
.BYTE $00
; Burst Position 9
.BYTE $00,$00
; Index to pattern
.BYTE $00
; Burst Position 10
.BYTE $00,$BD
; Index to pattern
.BYTE $00

```

```

; -----+
; Unused Data
; -----
.BYTE $00,$BD,$00,$BD,$00,$BD,$00,$BD
.BYTE $81,$BD,$81,$FF,$00,$BD,$F1,$FF
.BYTE $00,$2B,$81,$FF,$81,$AE,$82,$AE
.BYTE $00,$FF,$81,$EE,$81,$AC,$C1,$BD
.BYTE $C1,$24,$81,$FF,$C1,$FF,$00,$EE
.BYTE $81,$BF,$85,$AE,$81,$EC,$E1,$BF
.BYTE $83,$37,$00,$EE,$81,$BF,$C3,$2E
.BYTE $81,$2E,$00,$FF,$00,$FF,$00,$FD
.BYTE $81,$8C,$81,$FF,$00,$BD,$F0,$DC
.BYTE $00,$6C,$81,$EC,$00,$EE,$81,$EE
.BYTE $85,$6C,$81,$EE,$01,$EC,$87,$EE
.BYTE $85,$FD,$83,$CD,$42,$EF,$00,$FF
.BYTE $00,$2B,$02,$EE,$81,$BD,$85,$BF
.BYTE $81,$FF,$85,$EE,$00,$BF,$87,$BF
.BYTE $00,$EE,$87,$FF,$81,$FF,$A7,$FF
.BYTE $01,$FF,$80,$EE,$FD,$FF,$FF,$FF

```

**Lines 2800-3200.** This is the data used for generating 'bursts', the subject of our chapter '[beatific bursts](#)'.

Only four are defined, each bound to one of the 'Function' keys on the C64 keyboard. Each burst consists of a header specifying the symmetry and smoothing delay used for the burst:

```
; currentSymmetrySetting
.BYTE NO_SYMMETRY
; smoothingDelay:
.BYTE $11
```

.. followed by a sequence of 'cells' of three bytes each. Each cell gives the X and Y co-ordinates to place a pattern and a reference to the pattern to draw:

```
; Burst Position 1
.BYTE $12,$09
; Index to pattern
.BYTE CUSTOMPATTERNO
```

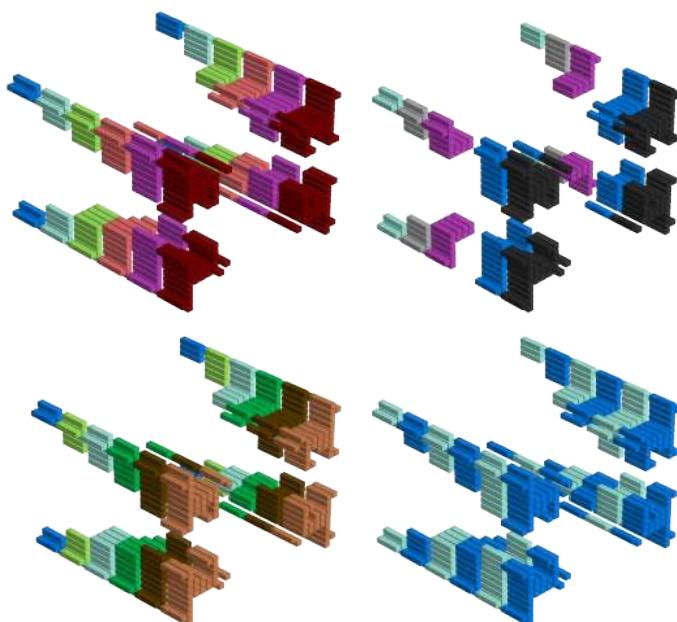


Figure 4.3: The 'F2' burst in different color schemes



**Lines 2300-2800.** This data supports Psychedelia's sequencer which we cover in '[sensitive sequencer](#)'. Like the burst data it consists of a header, again defining the symmetry and smoothing delay:

```
; currentSymmetrySetting
.BYTE $01
; smoothingDelay
.BYTE $0B
```

Again like the burst data, the rest of the data consists of cells of three bytes defining X and Y co-ordinates and a pattern:

```
; Sequencer Position 1
.BYTE $04,$04 ; X/Y Co-ordinates
.BYTE PULSAR ; Index to pattern
```

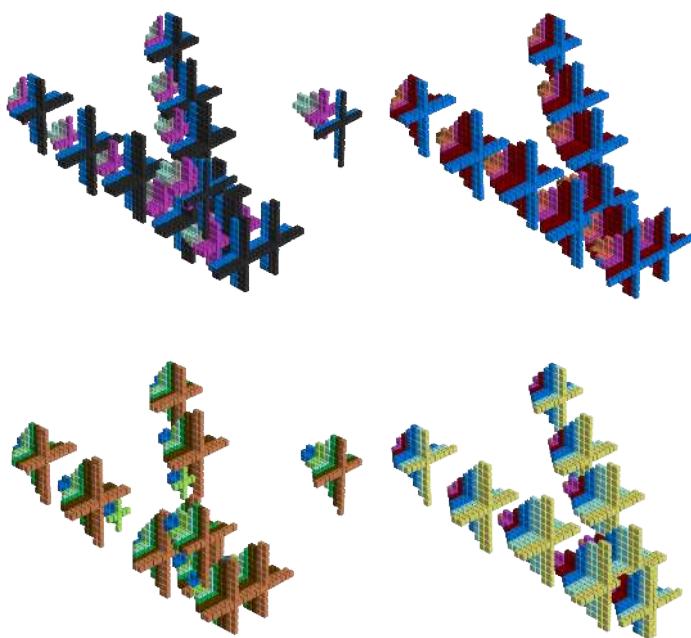


Figure 4.4: The sequencer in different color schemes







# all the pretty patterns

## Jeffrey Says



Choose a pattern you like and get ready to experiment. Press S to change the Symmetry. (The pattern gets reflected in various planes, or not at all according to the setting). Press SPACE to alter the pattern element. There are eight permanent ones, and eight you can define for yourself (more on this later!). The latter eight are all set up when you load, so you can always choose from 16 shapes.

We deserve a rest from reading so much code. So let's take a brief intermission where we admire all the different patterns that Psychedelia is packaged with. When presenting a pattern for your consideration I give a 3D rendering of each pattern's evolution according to the '[Sunday Run Algorithm](#)'. along with the two data arrays used to generate the pattern and a step by step visualisation of the pattern's evolution.

The bad news is that there is some extra code for us to cover from the commercial edition, not much, but you will find it sprinkled among the pretty pictures and it will hopefully shed some light on how the commercial edition of Psychedelia managed the large number of patterns available to the player.

In order to enjoy the pretty pictures it might help to understand what they mean. We're already familiar with the data structure that underlies the patterns. Here is the structure for the Star Pattern again, a pair of arrays: the first giving the X co-ordinates, the second giving the Y co-ordinates.

## all the pretty patterns

---

```
starOneXPosArray
    .BYTE $00,$01,$01,$01,$00,$FF,$FF,$FF,$55      ;      5
    .BYTE $00,$02,$00,$FE,$55                         ;
    .BYTE $00,$03,$00,$FD,$55                         ;      4 4
    .BYTE $00,$04,$00,$FC,$55                         ;      3
    .BYTE $FF,$01,$05,$05,$01,$FF,$FB,$FB,$55       ;      2
    .BYTE $00,$07,$00,$F9,$55                         ;      1
    .BYTE $55                                         ;      4 000 4
starOneYPosArray
    .BYTE $FF,$FF,$00,$01,$01,$01,$00,$FF,$55      ;      4 000 4
    .BYTE $FE,$00,$02,$00,$55                         ;      1
    .BYTE $FD,$00,$03,$00,$55                         ;      2
    .BYTE $FC,$00,$04,$00,$55                         ;      3
    .BYTE $FB,$FB,$FF,$01,$05,$05,$01,$FF,$55       ;      4 4
    .BYTE $F9,$00,$07,$00,$55                         ;
    .BYTE $55                                         ;      5
```

Listing 5.1: Source code for the Star.

For each of the patterns I attempt to visualize its evolution using a table such as this one for the Star One pattern:

.BYTE \$00,\$01,\$01,\$01,\$00,\$FF,\$FF,\$FF	
.BYTE \$FF,\$FF,\$00,\$01,\$01,\$00,\$FF,\$FF	
.BYTE \$00,\$02,\$00,\$FE	
.BYTE \$FE,\$00,\$02,\$00	
.BYTE \$00,\$03,\$00,\$FD	
.BYTE \$FD,\$00,\$03,\$00	
.BYTE \$00,\$04,\$00,\$FC	
.BYTE \$FC,\$00,\$04,\$00	
.BYTE \$FF,\$01,\$05,\$05,\$01,\$FF,\$FB,\$FB	
.BYTE \$FB,\$FB,\$FF,\$01,\$05,\$05,\$01,\$FF	
.BYTE \$00,\$07,\$00,\$F9	
.BYTE \$F9,\$00,\$07,\$00	

In order to understand what this table is telling us, let's look at the first entry more closely.

```
.BYTE $00,$01,$01,$01,$00,$FF,$FF,$FF
.BYTE $FF,$FF,$00,$01,$01,$01,$00,$FF
```



This is the first line in `starOneXPosArray` and the first line in `starOneYPosArray` listed together. It's followed by how these pairs of co-ordinates are plotted onto the screen. If you're paying close attention you'll notice that the \$55 at the end of each line has been dropped. That's because the \$55 is a 'sentinel' value that tells the 'Sunday Run' algorithm to treat this as a kind of line-break and draw the preceding co-ordinates a single iteration of the paint loop.

The true puzzle is how values like \$FF and \$00 can constitute co-ordinates! The answer to this is illustrated in the table below where we show how this first pair of lines from `starOneXPosArray` and `starOneYPosArray` is plotted. The numbers given in the middle of the table represent the order that the pairs appear in each array.

	\$F8	\$F9	\$FA	\$FB	\$FC	\$FD	\$FE	\$FF	\$00	\$01	\$02	\$03	\$04	\$05	\$06	\$07
\$F9	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
\$FA	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
\$FB	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
\$FC	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
\$FD	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
\$FE	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
\$FF	.	.	.	.	.	.	.	8	1	2	.	.	.	.	.	.
\$00	.	.	.	.	.	.	.	7	.	3	.	.	.	.	.	.
\$01	.	.	.	.	.	.	.	6	5	4	.	.	.	.	.	.
\$02	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
\$03	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
\$04	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
\$05	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
\$06	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
\$07	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

So for example, the first pair from each array (\$00 and \$FF) is given as 1 above. The second pair (\$01 and \$FF) is given as 2. So as you can hopefully infer: \$FF is our way of saying '-1' in this co-ordinate system, \$FE our way of saying '-2' and so on. This is the conventional way of representing or treating single byte values when we want to represent both positive and negative numbers. Conventionally, since a single byte can represent 256 different values, if we want it to contain both positive and negative numbers we can use a single byte to represent anything between -128 and +128.

If we look at the second row in our table we've moved on to the co-ordinates given by the second line in each of `starOneXPosArray` and `starOneYPosArray`:

```
.BYTE $00,$02,$00,$FE  
.BYTE $FE,$00,$02,$00
```



Here we plot in the four pixels given by the lines. We've added them in gray below.

\$F8	\$F9	\$FA	\$FB	\$FC	\$FD	\$FE	\$FF	\$00	\$01	\$02	\$03	\$04	\$05	\$06	\$07
\$F9	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
\$FA	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
\$FB	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
\$FC	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
\$FD	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
\$FE	.	.	.	.	.	.	.	.	9	.	.	.	.	.	.
\$FF	.	.	.	.	.	.	8	1	2	.	.	.	.	.	.
\$00	.	.	.	.	.	12	7	.	3	10	.	.	.	.	.
\$01	.	.	.	.	.	.	6	5	4	.	.	.	.	.	.
\$02	.	.	.	.	.	.	.	11	.	.	.	.	.	.	.
\$03	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
\$04	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
\$05	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
\$06	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
\$07	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

The two pictures on the right hand side of the table above show us layering in this extra set of pixels. But they also show us something else: the way Psychedelia is cycling the colors of the pixels while painting the structure at this stage of its evolution.

Finally let's look at all the entries in our two arrays plotted on the same graph. In this instance we use numbers in the plot to represent the line in the array they come from (and give each a different color too to aid identification). So for example the points plotted by the first line in `starOneXPosArray` and `starOneYPosArray` are shown in green with the value 0. These colors and values are just to aid apprehensions - they have no other meaning.

As we can see each 'line' the arrays builds up the evolution of the pattern. The purpose of the tables on the following pages is to help you understand how the pattern is built up in each case as well as to show how the Sunday Run algorithm colours the pixels during each painting iteration.

\$F8	\$F9	\$FA	\$FB	\$FC	\$FD	\$FE	\$FF	\$00	\$01	\$02	\$03	\$04	\$05	\$06	\$07
\$F9	.	.	.	.	.	.	.	.	5	.	.	.	.	.	.
\$FA	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
\$FB	.	.	.	.	.	.	.	4	.	4	.	.	.	.	.
\$FC	.	.	.	.	.	.	.	3	.	.	.	.	.	.	.
\$FD	.	.	.	.	.	.	.	2	.	.	.	.	.	.	.
\$FE	.	.	.	.	.	.	.	1	.	.	.	.	.	.	.
\$FF	.	.	.	4	.	.	.	0	0	0	.	.	4	.	.
\$00	.	5	.	.	3	2	1	0	0	0	1	2	3	.	5
\$01	.	.	.	4	.	.	.	0	0	0	.	.	4	.	.
\$02	.	.	.	.	.	.	.	1	.	.	.	.	.	.	.
\$03	.	.	.	.	.	.	.	2	.	.	.	.	.	.	.
\$04	.	.	.	.	.	.	.	3	.	.	.	.	.	.	.
\$05	.	.	.	.	.	.	.	4	.	4	.	.	.	.	.
\$06	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
\$07	.	.	.	.	.	.	.	5	.	.	.	.	.	.	.

In addition to tabulating the pattern evolution in this way, we also visualize the evolution of each pattern in three dimensions. Hopefully it is obvious how these pictures represent the evolution of the pattern over time. In each case we view the evolution from both the 'front' and the 'back'. They show how the coloring of the pixels for each portion of the pattern changes over time as well as how the pattern emerges while it is constructed.

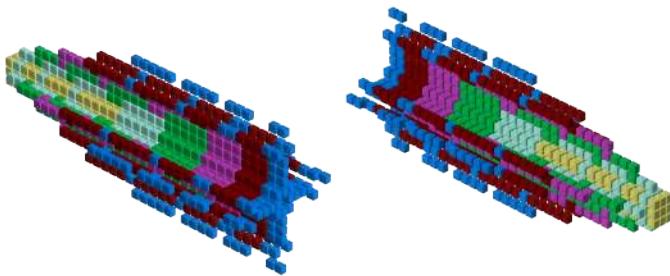
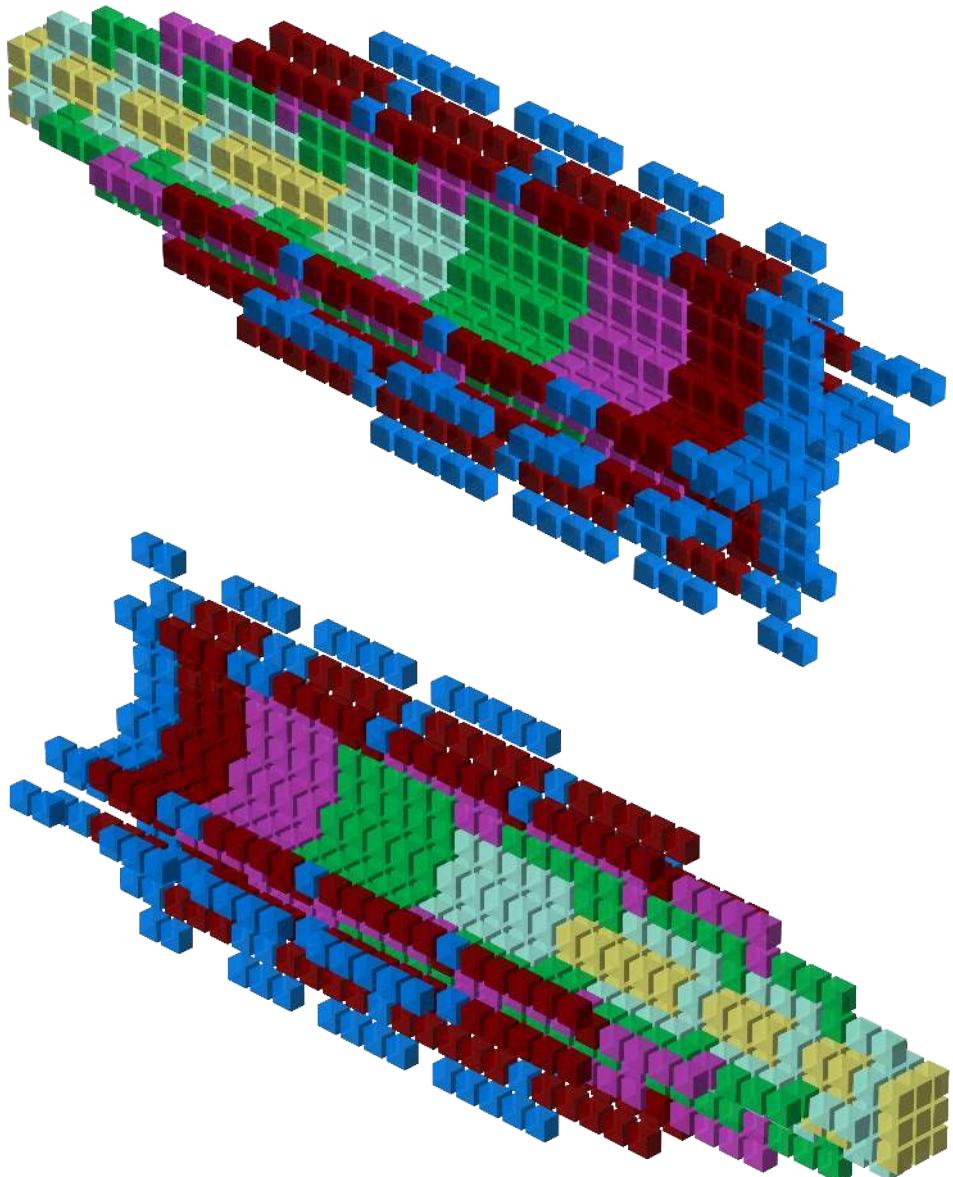


Figure 5.1: Evolution of the 'Star One' pattern.

As we said earlier there's some code and commentary sprinkled across the following pages that is relevant to the drawing of these patterns. Hopefully it will continue to shed more light on the operation of Psychedelia.



```

starOneXPosArray
    .BYTE $00,$01,$01,$01,$00,$FF,$FF,$FF,$55      ;      5
    .BYTE $00,$02,$00,$FE,$55                         ;
    .BYTE $00,$03,$00,$FD,$55                         ;      4 4
    .BYTE $00,$04,$00,$FC,$55                         ;      3
    .BYTE $FF,$01,$05,$05,$01,$FF,$FB,$FB,$55       ;      2
    .BYTE $00,$07,$00,$F9,$55                         ;      1
    .BYTE $55                                         ;      4 000 4
starOneYPosArray
    .BYTE $FF,$FF,$00,$01,$01,$01,$00,$FF,$55       ;      4 000 4
    .BYTE $FE,$00,$02,$00,$55                         ;      1
    .BYTE $FD,$00,$03,$00,$55                         ;      2
    .BYTE $FC,$00,$04,$00,$55                         ;      3
    .BYTE $FB,$FB,$FF,$01,$05,$05,$01,$FF,$55       ;      4 4
    .BYTE $F9,$00,$07,$00,$55                         ;
    .BYTE $55                                         ;      5

```

Listing 5.2: Source code for the Star.

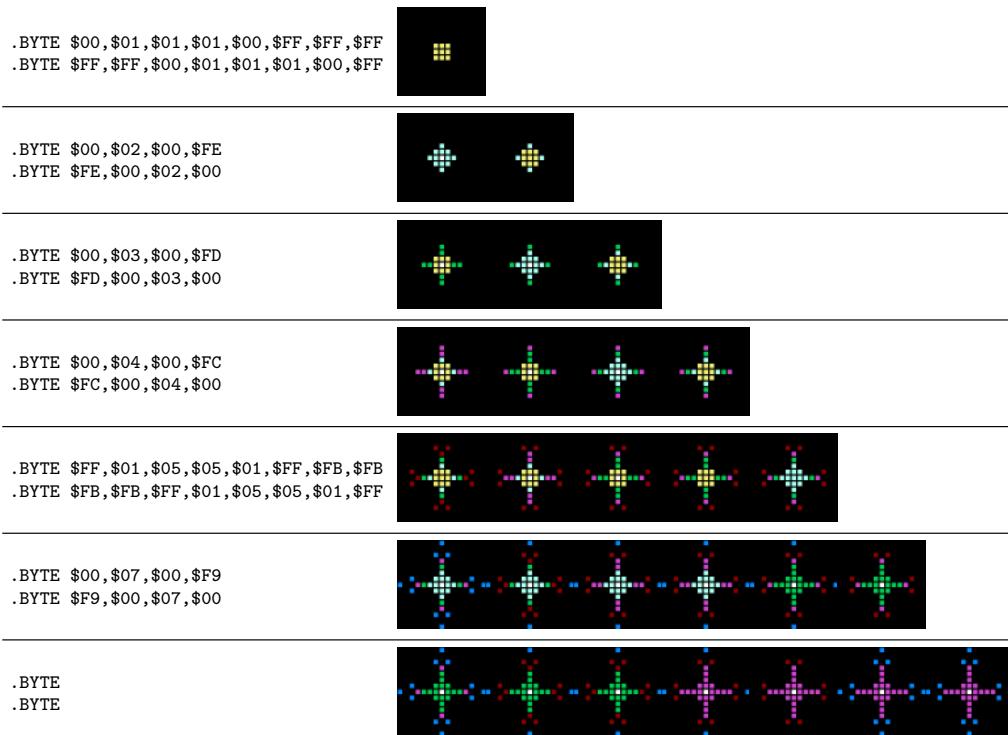


Figure 5.3: Pattern Progression for 'Star One'

**Lines 1017-1052.** pixelXPositionLoPtrArray, pixelYPositionLoPtrArray

```
; A pair of arrays together constituting a list of pointers
; to positions in memory containing X position data.
; (i.e. $097C, $0E93,$0EC3, $0F07, $0F23, $0F57, $1161, $11B1)
pixelXPositionLoPtrArray
    .BYTE <starOneXPosArray,<theTwistXPosArray,<laLlamitaXPosArray
    .BYTE <starTwoXPosArray,<deltoidXPosArray,<diffusedXPosArray
    .BYTE <multicrossXPosArray,<pulsarXPosArray
    .BYTE <customPattern0XPosArray,<customPattern1XPosArray
    .BYTE <customPattern2XPosArray,<customPattern3XPosArray
    .BYTE <customPattern4XPosArray,<customPattern5XPosArray
    .BYTE <customPattern6XPosArray,<customPattern7XPosArray

pixelXPositionHiPtrArray
    .BYTE >starOneXPosArray,>theTwistXPosArray,>laLlamitaXPosArray
    .BYTE >starTwoXPosArray,>deltoidXPosArray,>diffusedXPosArray
    .BYTE >multicrossXPosArray,>pulsarXPosArray
    .BYTE >customPattern0XPosArray,>customPattern1XPosArray
    .BYTE >customPattern2XPosArray,>customPattern3XPosArray
    .BYTE >customPattern4XPosArray,>customPattern5XPosArray
    .BYTE >customPattern6XPosArray,>customPattern7XPosArray

; A pair of arrays together constituting a list of pointers
; to positions in memory containing Y position data.
; (i.e. $097C, $0E93,$0EC3, $0F07, $0F23, $0F57, $1161, $11B1)
pixelYPositionLoPtrArray
    .BYTE <starOneYPosArray,<theTwistYPosArray,<laLlamitaYPosArray
    .BYTE <starTwoYPosArray,<deltoidYPosArray,<diffusedYPosArray
    .BYTE <multicrossYPosArray,<pulsarYPosArray
    .BYTE <customPattern0YPosArray,<customPattern1YPosArray
    .BYTE <customPattern2YPosArray,<customPattern3YPosArray
    .BYTE <customPattern4YPosArray,<customPattern5YPosArray
    .BYTE <customPattern6YPosArray,<customPattern7YPosArray

pixelYPositionHiPtrArray
    .BYTE >starOneYPosArray,>theTwistYPosArray,>laLlamitaYPosArray
    .BYTE >starTwoYPosArray,>deltoidYPosArray,>diffusedYPosArray
    .BYTE >multicrossYPosArray,>pulsarYPosArray
    .BYTE >customPattern0YPosArray,>customPattern1YPosArray
    .BYTE >customPattern2YPosArray,>customPattern3YPosArray
    .BYTE >customPattern4YPosArray,>customPattern5YPosArray
    .BYTE >customPattern6YPosArray,>customPattern7YPosArray
```

Listing 5.3: All the pattern data structures in Psychedelia organized into a set of arrays. We use these arrays to choose the correct user-selected pattern at painting time.

**Lines 1017-1033.** `pixelXPositionLoPtrArray/pixelXPositionHiPtrArray`: Psychedelia offers 16 different pretty patterns to choose from, so requires some way of managing them, particularly when it comes time to painting them on the screen. The four arrays on the opposite page do this job. They allow us to reference each pattern with an index. For example, index 0 will reference the X and Y Position data structures for the 'Star One' pattern in `starOneXPosArray` and `starOneYPosArray`, index 1 will allow us to reference the data structures for 'The Twist' pattern, and so on.

On the following pages we'll see how we make practical use of these arrays, but for now we only really need to make clear that each one contains one byte of the two-byte address at which each data structure is stored. The use of < and > in the listing is a convention that tells the assembler we're looking at the first byte (>) or the second byte (<). Hopefully this table makes this explicit to the reader:

Element	pixelXPosition HiPtrArray	pixelXPosition LoPtrArray	Address	Name
0	\$09	\$7C	\$097C	<code>starOneXPosArray</code>
1	\$0E	\$93	\$0E93	<code>theTwistXPosArray</code>
2	\$0E	\$C3	\$0EC3	<code>lallamitaXPosArray</code>
3	\$0F	\$07	\$0F07	<code>starTwoXPosArray</code>
4	\$0F	\$23	\$0F23	<code>deltoidXPosArray</code>
5	\$0F	\$57	\$0F57	<code>diffusedXPosArray</code>
.	.	.	.	.
15	\$CE	\$00	\$CE00	<code>customPattern6XPosArray</code>
16	\$CF	\$00	\$CF00	<code>customPattern7XPosArray</code>

Figure 5.4: Our two arrays and their contents - each combining to give us an address for the X Position data structure for each pattern. The line with ellipses indicates that we've left out some elements for the sake of brevity.

**Lines 1033-1052.** `pixelYPositionLoPtrArray/pixelYPositionHiPtrArray`: As for the X position array, so for the Y position array.

Element	pixelYPosition HiPtrArray	pixelYPosition LoPtrArray	Address	Name
0	\$09	\$A3	\$09A3	<code>starOneYPosArray</code>
1	\$0E	\$AB	\$0EAB	<code>theTwistYPosArray</code>
2	\$0E	\$E5	\$0EE5	<code>lallamitaYPosArray</code>
.	.	.	.	.
15	\$CE	\$00	\$CE00	<code>customPattern6YPosArray</code>
16	\$CF	\$00	\$CF00	<code>customPattern7YPosArray</code>

Figure 5.5: Our two arrays and their contents - each combining to give us an address for the Y Position data structure for each pattern.

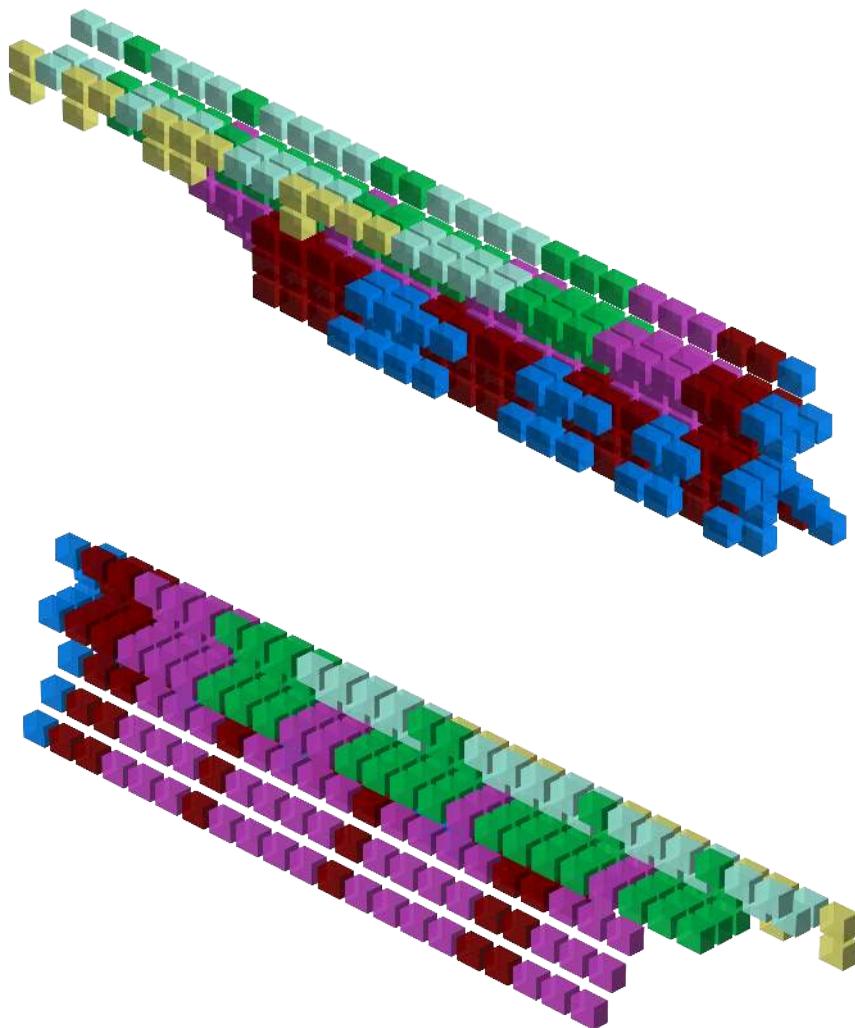


Figure 5.6: The 'Twist'.

```

theTwistXPosArray .BYTE $00,$55 ;      1
                   .BYTE $01,$02,$55 ; 01
                   .BYTE $01,$02,$03,$55 ; 6 222
                   .BYTE $01,$02,$03,$04,$55 ; 543
                   .BYTE $00,$00,$00,$55 ; 5 4 3
                   .BYTE $FF,$FE,$55 ; 4 3
                   .BYTE $55 ; 3
theTwistYPosArray .BYTE $FF,$55
                   .BYTE $FF,$FE,$55
                   .BYTE $00,$00,$00,$55
                   .BYTE $01,$02,$03,$04,$55
                   .BYTE $01,$02,$03,$55
                   .BYTE $01,$02,$55
                   .BYTE $00,$55
                   .BYTE $55

```

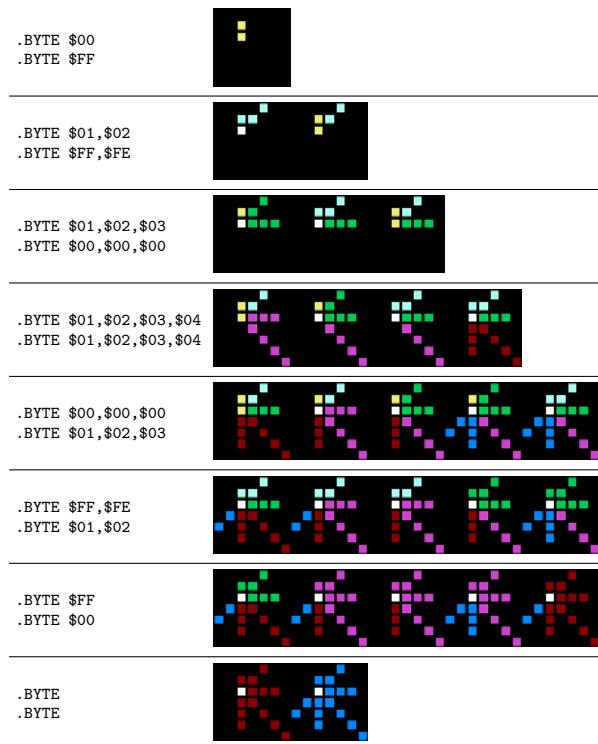


Figure 5.7: Pattern Progression for 'The Twister'

**Lines 251-321.** PaintStructureAtCursorPosition

```
xPosLoPtr = $0D
xPosHiPtr = $0E
yPosLoPtr = $10
yPosHiPtr = $11
;-----
; PaintStructureAtCursorPosition
;-----
PaintStructureAtCursorPosition
    JSR PaintPixelForCurrentSymmetry
    LDY #$00
    LDA baseLevelForCurrentPixel
    CMP #$07
    BNE CanLoopAndPaint
    RTS

CanLoopAndPaint
    LDA #$07
    STA countToMatchCurrentIndex

    LDA pixelXPosition
    STA previousCursorXPosition
    LDA pixelYPosition
    STA previousPixelYPosition

    LDX patternIndex
    LDA pixelXPositionLoPtrArray ,X
    STA xPosLoPtr
    LDA pixelXPositionHiPtrArray ,X
    STA xPosHiPtr
    LDA pixelYPositionLoPtrArray ,X
    STA yPosLoPtr
    LDA pixelYPositionHiPtrArray ,X
    STA yPosHiPtr

    ; Paint pixels in the sequence until hitting a break
    ; at $55
PixelPaintLoop
    ; Get the next byte from the pattern's X pos array.
    LDA previousCursorXPosition
    CLC
    ADC (xPosLoPtr),Y
    STA pixelXPosition

    ; Get the next byte from the pattern's Y pos array.
    LDA previousPixelYPosition
    CLC
    ADC (yPosLoPtr),Y
    STA pixelYPosition

    ; Push Y to the stack.
    TYA
    PHA

    JSR PaintPixelForCurrentSymmetry

    ; Pull Y back from the stack and increment
    PLA
    TAY
   INY
```

Listing 5.4: The routine responsible for painting patterns.

**Lines 251-321.** PaintStructureAtCursorPosition: We've already encountered this routine in [our walk through of the listing..](#) This is the version that shipped with the commercial edition of Psychedelia so has a necessary extra complication to deal with the fact that we are going to paint one of up to 16 possible patterns. What we want to figure out here is the X and Y position we should paint for each element in the pattern's data structure.

The pattern the user has selected is stored as an index value in patternIndex. We use it to fetch the element from each array:

```
LDX patternIndex
LDA pixelXPositionLoPtrArray ,X
STA xPosLoPtr
LDA pixelXPositionHiPtrArray ,X
STA xPosHiPtr
LDA pixelYPositionLoPtrArray ,X
STA yPosLoPtr
LDA pixelYPositionHiPtrArray ,X
STA yPosHiPtr
```

Imagine our patternIndex is 1. This will give us the following values:

xPosHiPtr	xPosLoPtr	Address	Name
\$0E	\$93	\$0E93	theTwistXPosArray

Now, using this as our basis we can read each byte from this address onwards (from each array) and use it to calculate the X and Y position to paint a pixel at. Recall that the way we read the theTwistXPosArray and theTwistYPosArray data structures is to treat each element as an offset from the cursor's current X/Y position. So that means taking previousCursorXPosition and adding the value we read from the array to it. The way we read in a byte from the array using xPosLoPtr and add it to get the new position is as follows:

```
LDA previousCursorXPosition
CLC
ADC (xPosLoPtr),Y
STA pixelXPosition
```

(xPosLoPtr) in the above does something very useful: it points to the address you get from combining xPosLoPtr (\$93) and xPosHiPtr (\$0E). It can do this because xPosLoPtr and xPosHiPtr are adjacent to each other in memory.

With Y as an offset (Y is the counter in our loop, incremented at each iteration) ADC is able to point to the next byte in the theTwistXPosArray and add it to the accumulator, so that we can store our result in pixelXPosition.

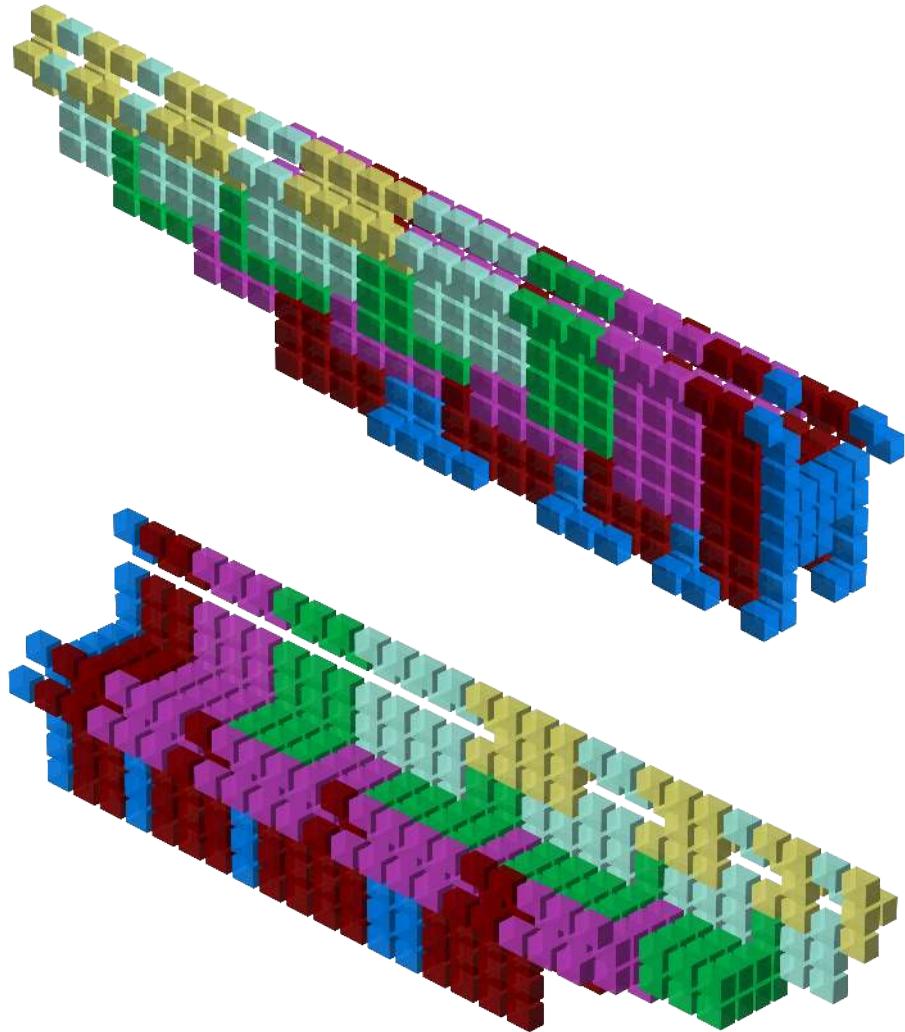


Figure 5.8: 'La Llamita'.

```

laLlamitaXPosArray    .BYTE $00,$FF,$00,$55      ;  0
                      .BYTE $00,$00,$55      ;  06
                      .BYTE $01,$02,$03,$00,$01,$02,$03,$55  ;  0
                      .BYTE $04,$05,$06,$04,$00,$01,$02,$55  ;  1   3
                      .BYTE $04,$00,$04,$00,$04,$55      ; 12223 3
                      .BYTE $FF,$03,$55      ; 22223
                      .BYTE $00,$55      ; 333 4
laLlamitaYPosArray    .BYTE $FF,$00,$01,$55      ; 4   4
                      .BYTE $02,$03,$55      ; 54   54
                      .BYTE $03,$03,$03,$04,$04,$04,$04,$55
                      .BYTE $03,$02,$03,$04,$05,$05,$05,$55
                      .BYTE $05,$06,$06,$07,$07,$07,$55
                      .BYTE $07,$07,$55
                      .BYTE $00,$55

```

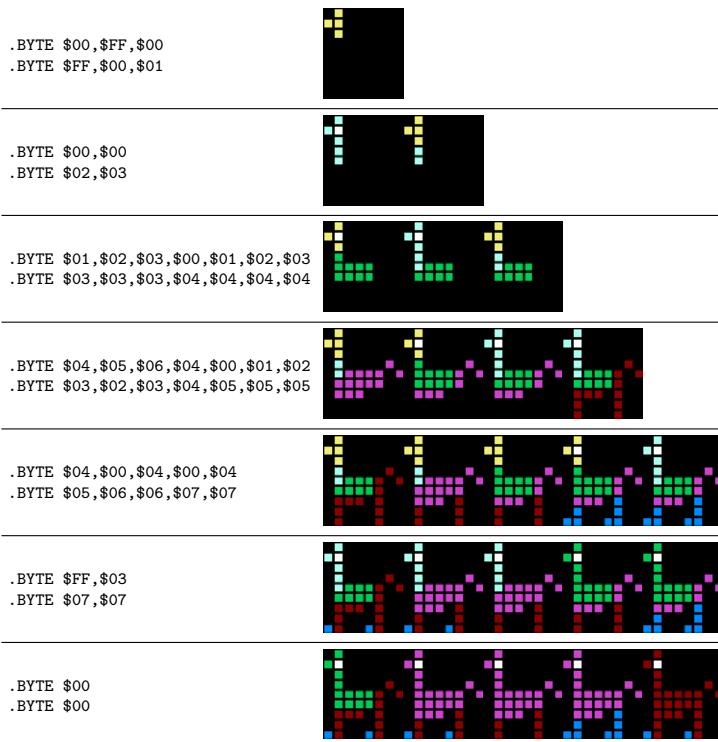


Figure 5.9: Pattern Progression for 'La Llamita'

**Lines 1144-1207.** CheckKeyboardInput

```
; -----
; CheckKeyboardInput
; -----
CheckKeyboardInput
    ...
CheckForKeyStroke
    LDA lastKeyPressed
    CMP #$40
    BNE ProcessKeyStroke

    ; No key was pressed. Return early.
    LDA #$00
    STA timerBetweenKeyStrokes
    JSR DisplayDemoModeMessage
ReturnFromKeyboardCheck
    RTS

    ; A key was pressed. Figure out which one.
ProcessKeyStroke
    LDY initialTimeBetweenKeyStrokes
    STY timerBetweenKeyStrokes
    LDY shiftKey
    STY shiftPressed

    CMP #KEY_SPACE ; Space pressed?
    BNE MaybeSPressed

SpacePressed
    ; Space pressed. Selects a new pattern element. "There are
    ; eight permanent ones, and eight you can define for yourself
    ; (more on this later!). The latter eight are all set up when
    ; you load, so you can always choose from 16 shapes."
    INC currentPatternElement
    LDA currentPatternElement
    AND #$0F
    STA currentPatternElement
    AND #$08
    BEQ UpdateCurrentPattern
    ; The first 8 patterns are standard, the rest are custom.
    JMP GetCustomPatternElement
UpdateCurrentPattern
    JSR ClearLastLineOfScreen
    LDA currentPatternElement
    ASL
    ASL
    ASL
    ASL
    TAY

    LDX #$00
WritePatternDescription
    LDA txtPresetPatternNames,Y
    STA lastLineBufferPtr,X
    INY
    INX
    CPX #$10
    BNE WritePatternDescription
    JMP WriteLastLineBufferToScreen
    ; Returns
```

**Lines 1181-1188.** SpacePressed: This is the routine that detects when the player has selected a new pattern by pressing the 'Space' key. It is part of the much larger routine CheckKeyboardInput which periodically checks for keyboard input by polling the byte at address \$00C5 (which we label lastKeyPressed). This address always contains the value of the most recently pressed key on the keyboard.

Otherwise we increment currentPatternElement, which is the variable we use for storing the currently selected pattern. Each press of the 'Space' key increments the value of currentPatternElement until it reaches 15. Once that happens, we clamp it back to zero by using an AND #\$0F.

**Lines 1190-1207.** UpdateCurrentPattern: What is the series of ASL instructions doing? A tricky piece of business of course. We have loaded currentPatternElement to the A register (LDA currentPatternElement) and it has a value between 0 and 8. ASL performs a leftward bit-shift on the A register. A single left-shift has an interesting property - it doubles the value of the byte. A second left-shift will double it again, and so on. So our four ASL instructions have the effect of turning 1 into 16, 2 into 32, 3 into 48, and 4 into 64. This has the very useful effect of giving us an index into the description of each pattern!

```
txtPresetPatternNames
.TEXT    'STAR ONE      ,
.TEXT    'THE TWIST     ,
.TEXT    'LA LLAMITA   ,
.TEXT    'STAR TWO      ,
.TEXT    'DETOIDS        ,
.TEXT    'DIFFUSED       ,
.TEXT    'MULTICROSS    ,
.TEXT    'PULSAR         ,
```

So when we take our resulting value and load it into the Y register all our WritePatternDescription needs to do is start at the index given by Y and write out the next 16 bytes to the screen, displaying the selected pattern briefly to the player.

## all the pretty patterns

---

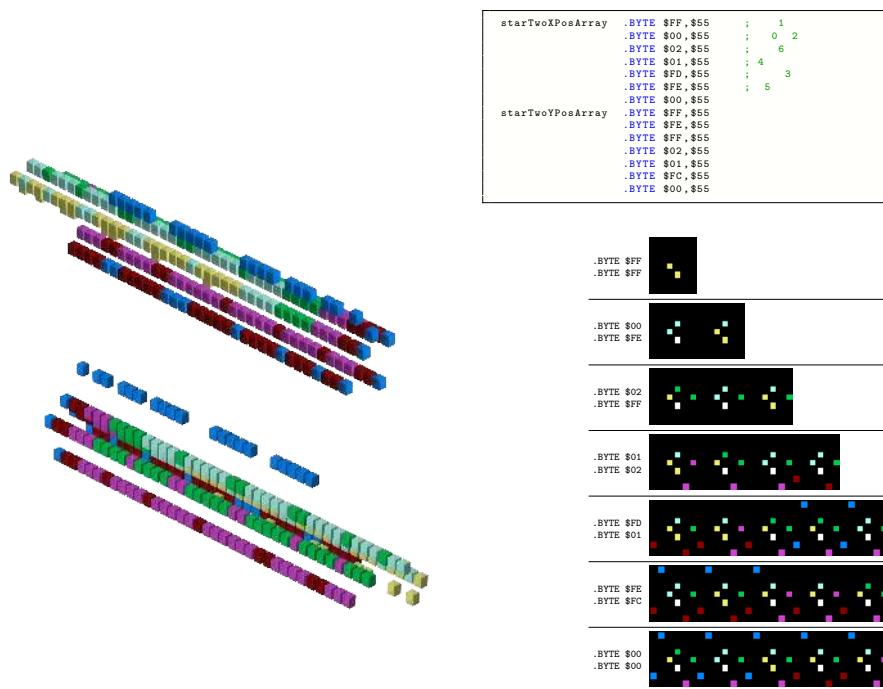


Figure 5.10: 'Star Two'.

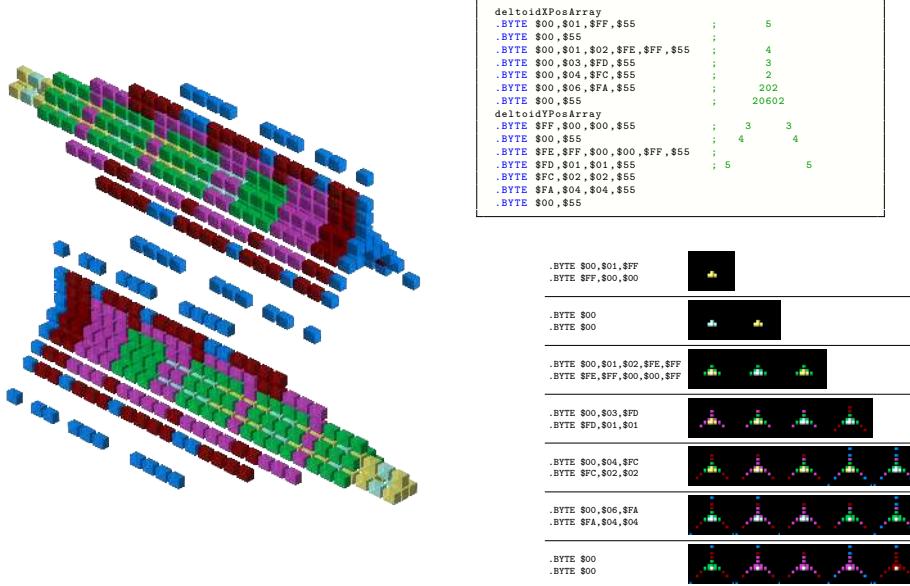


Figure 5.11: 'Deltoid'.

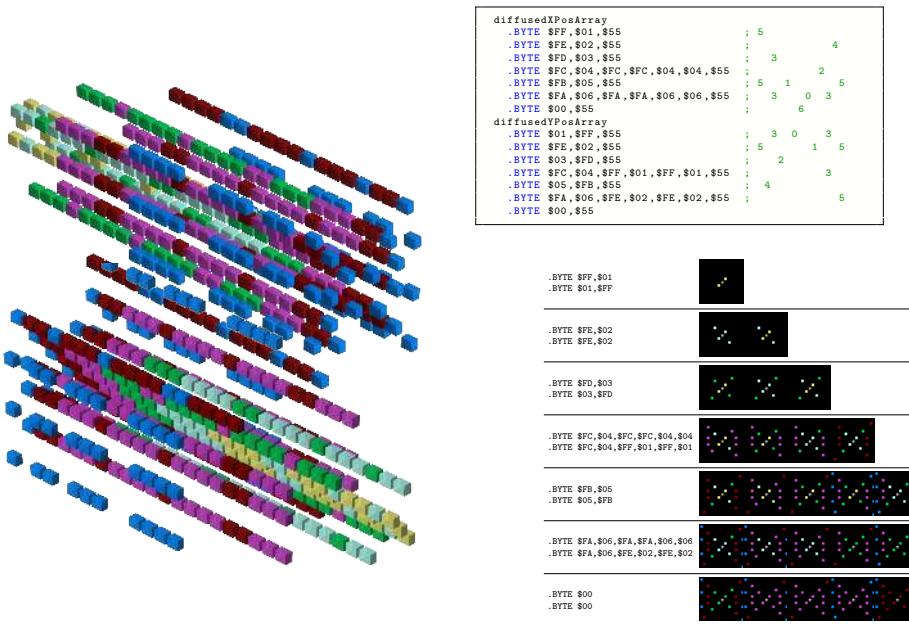


Figure 5.12: 'Diffused'.

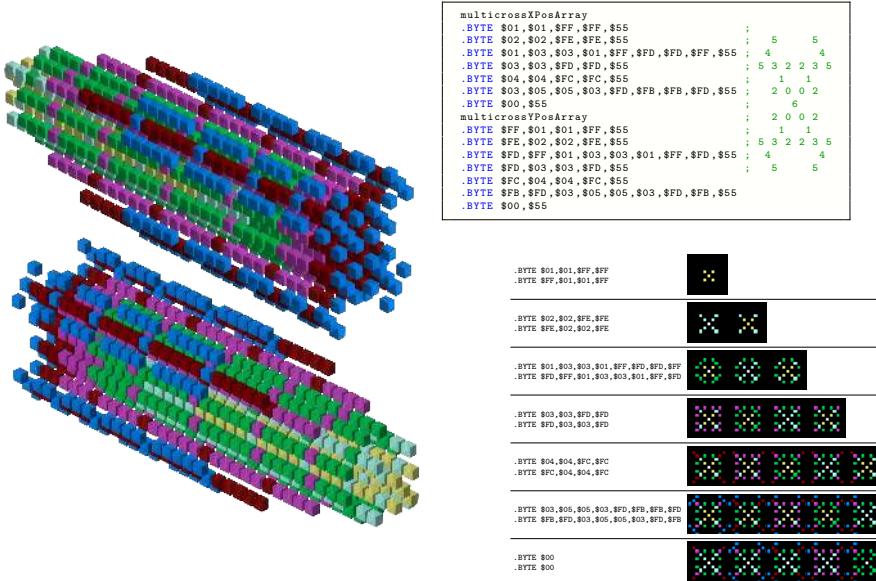


Figure 5.13: 'Multi-Cross'.

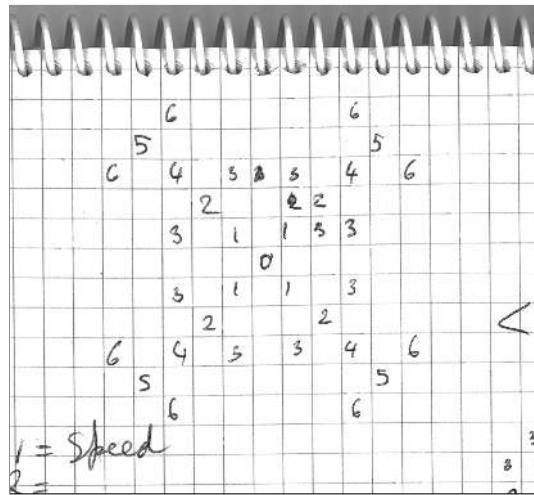


Figure 5.14: A sketch from Minter's development notes of the multi-cross pattern. Compared with the final form on the previous page a slight difference in the paint order is noticeable: the centre pixel is painted first in the sketch but last in Psychedelia. This is because the sketch is for the version of this pattern that was used in Colourspace, not Psychedelia. It's interesting that the reuse of the pattern was not a copy/paste exercise - instead it was redesigned from a blank page.

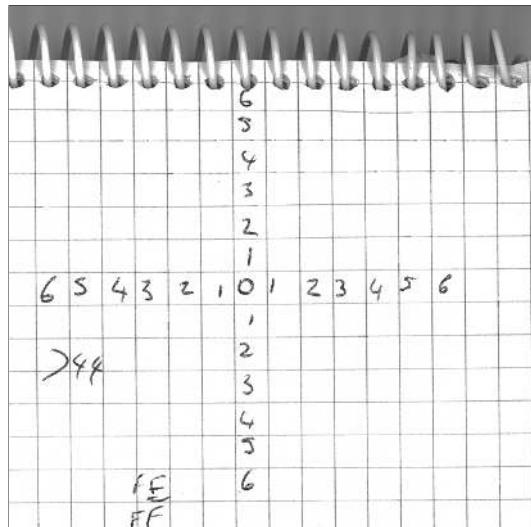


Figure 5.15: From the same page in the notebook, the Colourspace 'version' of the 'Pulsar' pattern was again redrawn from scratch. Like the Multi-Cross the centre pixel is the first one to be drawn in Colourspace rather than the last as here in Psychedelia.

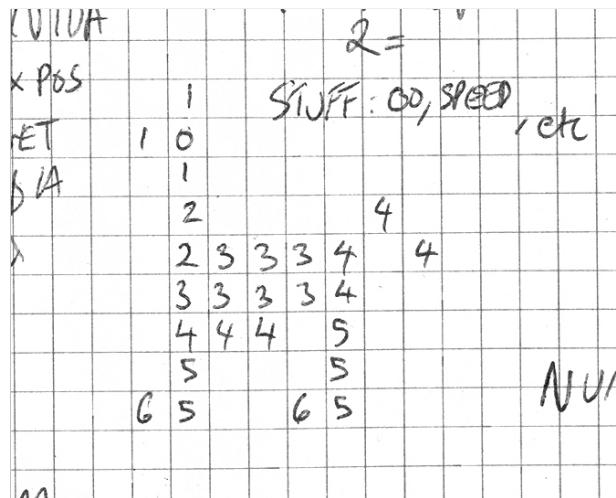


Figure 5.16: Again from the same page in the notebook, the 'Colourspace' version of La Llamita sketched out. The order of pixels is substantially different from that used in Psychedelia, below.

```

laLlamitaXPosArray    .BYTE $00,$FF,$00,$55          ;  0
                      .BYTE $00,$00,$55          ;  06
                      .BYTE $01,$02,$03,$00,$01,$02,$03,$55   ;  0
                      .BYTE $04,$05,$06,$04,$00,$01,$02,$55   ;  1   3
                      .BYTE $04,$00,$04,$00,$04,$55          ;  12223 3
                      .BYTE $FF,$03,$55          ;  22223
                      .BYTE $00,$55          ;  333 4
laLlamitaYPosArray    .BYTE $FF,$00,$01,$55          ;  4   4
                      .BYTE $02,$03,$55          ;  54  54
                      .BYTE $03,$03,$03,$04,$04,$04,$04,$55
                      .BYTE $03,$02,$03,$04,$05,$05,$05,$55
                      .BYTE $05,$06,$06,$07,$07,$07,$55
                      .BYTE $07,$07,$55
                      .BYTE $00,$55

```

Listing 5.5: The paint order of pixels for 'La Llamita' in Psychedelia.

## all the pretty patterns

---

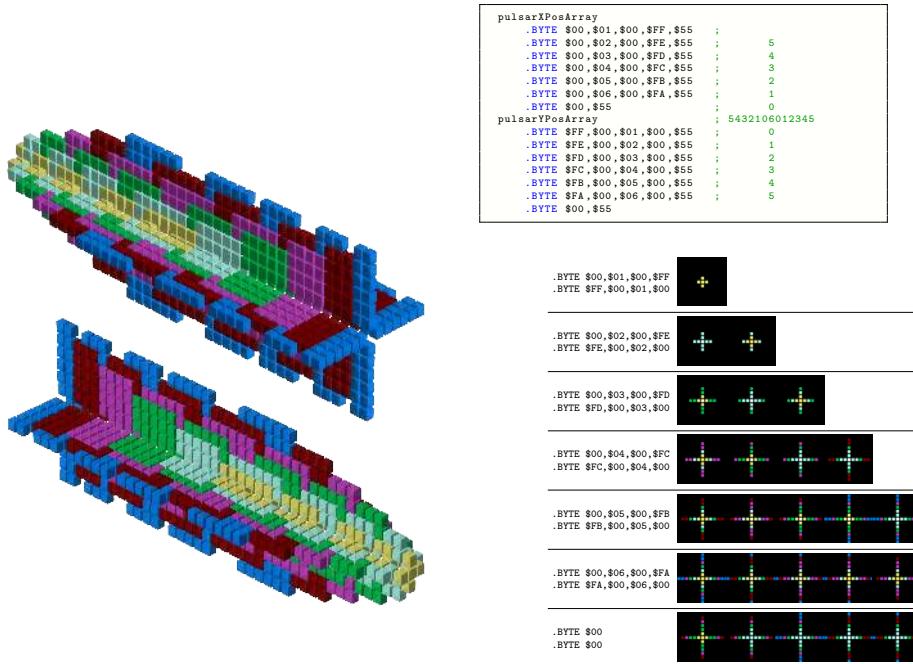


Figure 5.17: 'Pulsar'.

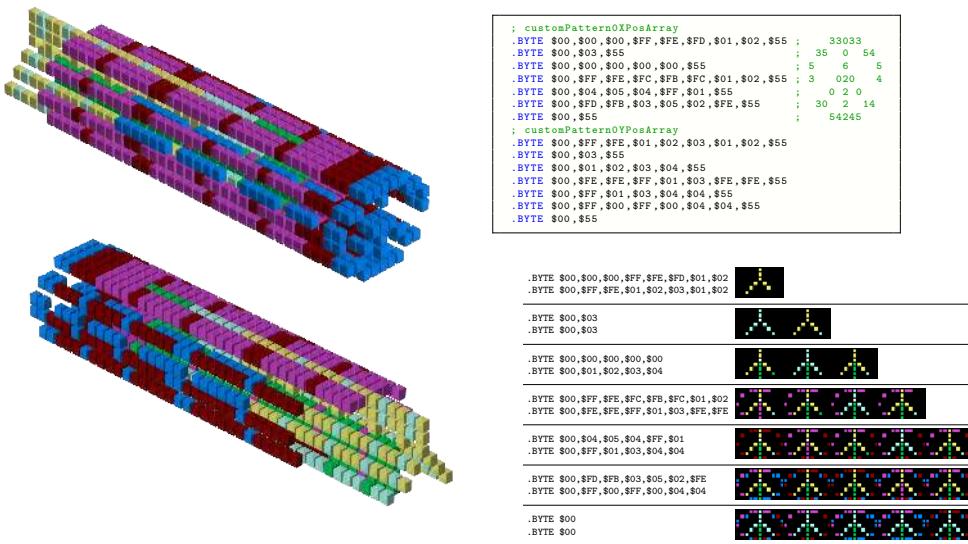
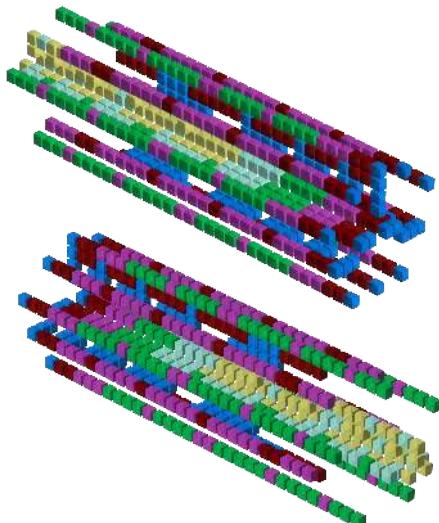


Figure 5.18: 'Custom Pattern 0'.



```
; customPattern1XPosArray : 3
.BYTE $00,$00,$FF,$01,$55 ; 4 5 4
.BYTE $00,$FF,$02,$55
.BYTE $00,$00,$FA,$06,$03,$FD,$55 ; 3 1 3
.BYTE $00,$00,$03,$FB,$05,$55 ; 7
.BYTE $00,$00,$00,$55 ; 21 12
.BYTE $00,$00,$FC,$04,$03,$FD,$55 ; 466 664
.BYTE $00,$55
; customPattern1YPosArray : 3 5 3
.BYTE $00,$FF,$01,$01,$55
.BYTE $00,$01,$01,$55
.BYTE $00,$FC,$FF,$FF,$05,$05,$55
.BYTE $00,$FD,$FD,$02,$02,$55
.BYTE $00,$05,$FD,$55
.BYTE $00,$FE,$02,$02,$02,$02,$55
.BYTE $00,$55
```

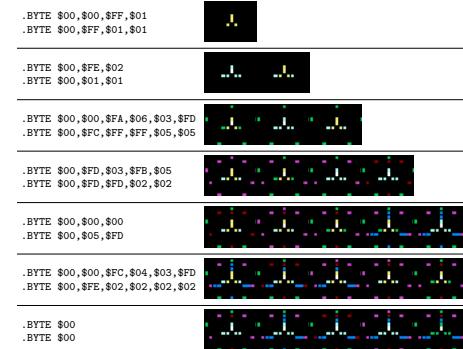
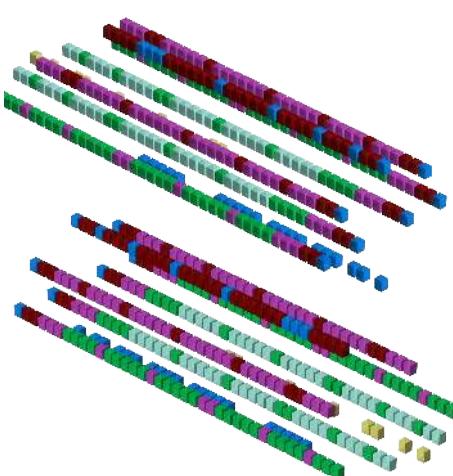


Figure 5.19: 'Custom Pattern 1'.



```
; customPattern2XPosArray : 5
.BYTE $00,$55
.BYTE $00,$FD,$03,$55 ; 4 8 8
.BYTE $00,$F9,$07,$55
.BYTE $00,$FB,$05,$55
.BYTE $00,$00,$55 ; 3 2 9 2 3
.BYTE $00,$00,$55
.BYTE $00,$55
.BYTE $FE,$02,$55
.BYTE $00,$55
; customPattern2YPosArray : 6
.BYTE $00,$55
.BYTE $00,$00,$00,$55
.BYTE $00,$00,$00,$55
.BYTE $00,$00,$FD,$55
.BYTE $00,$00,$FF,$55
.BYTE $00,$04,$55
.BYTE $00,$05,$55
.BYTE $FC,$FC,$55
.BYTE $00,$55
```

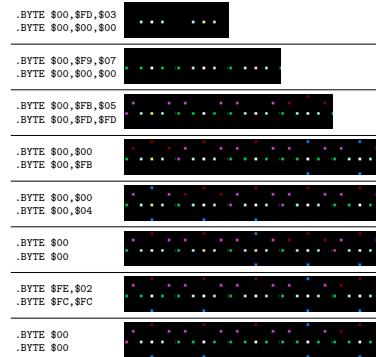


Figure 5.20: 'Custom Pattern 2'.

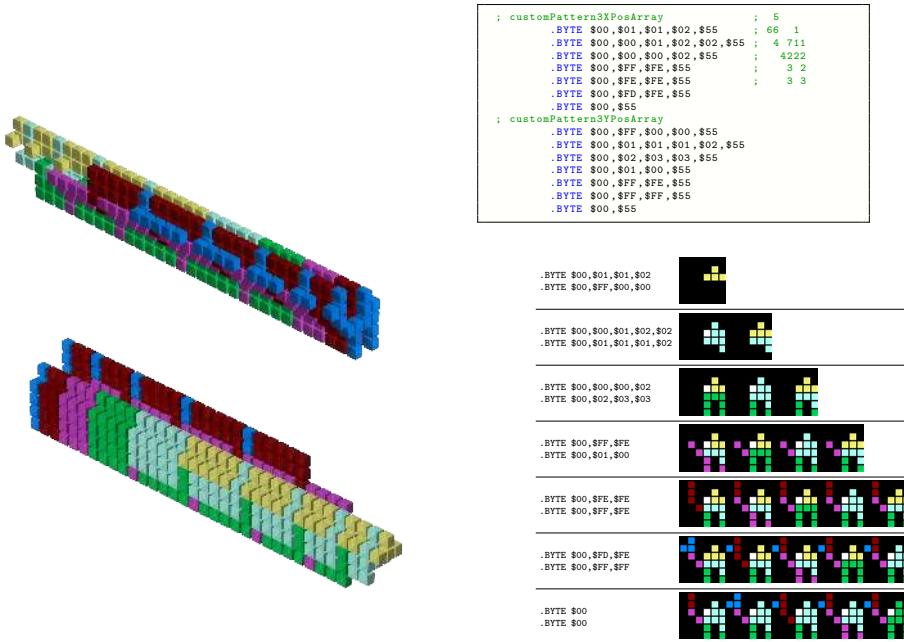


Figure 5.21: 'Custom Pattern 3'.

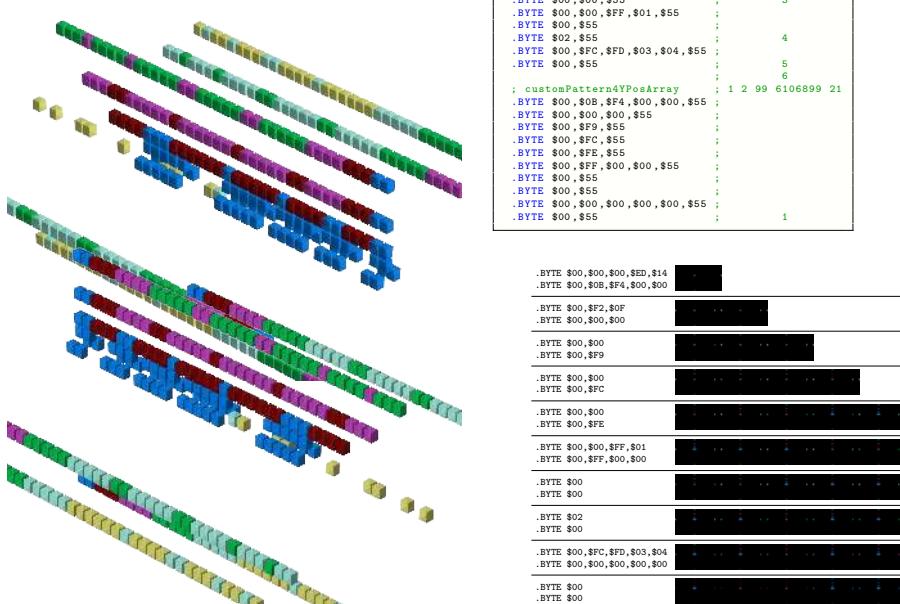


Figure 5.22: 'Custom Pattern 4'.

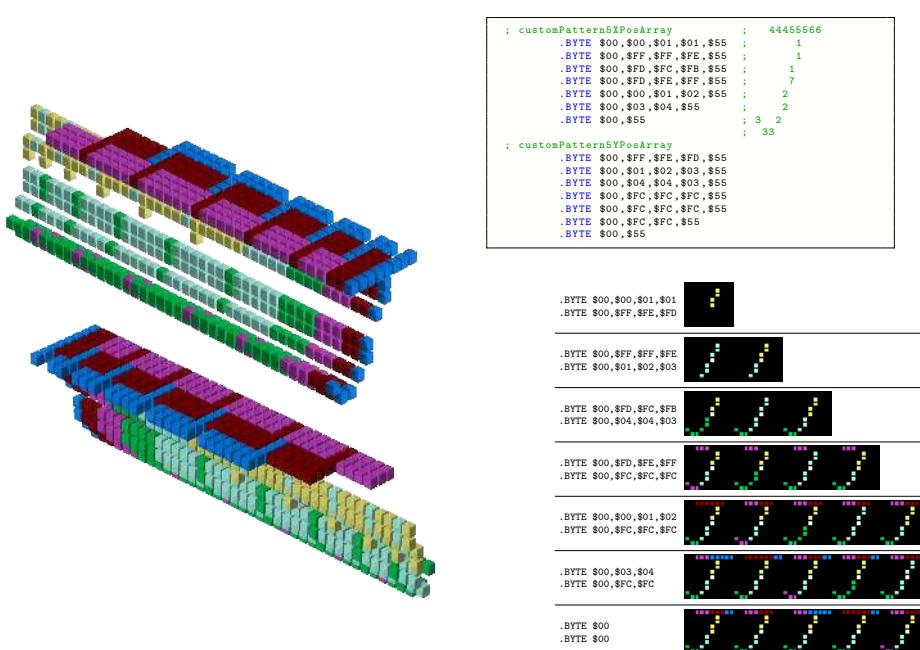


Figure 5.23: 'Custom Pattern 5'.

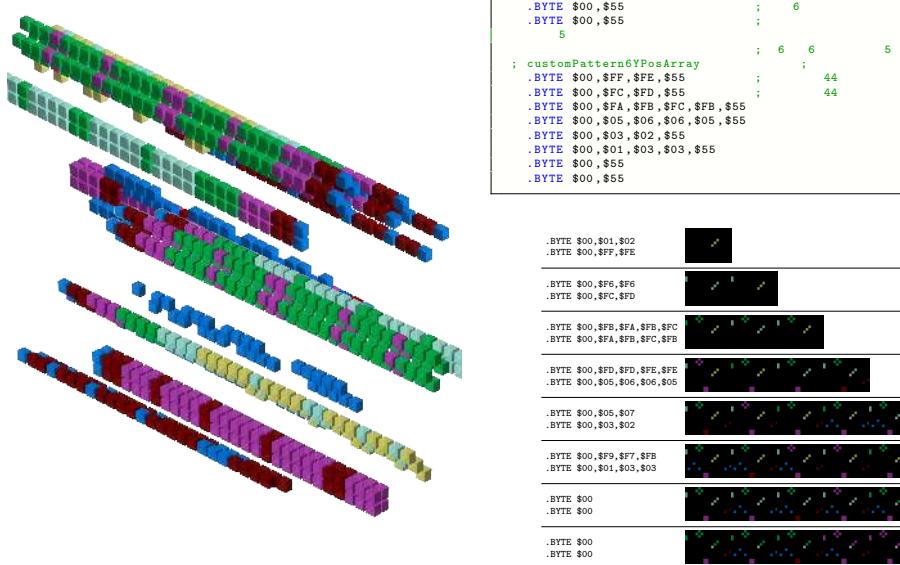


Figure 5.24: 'Custom Pattern 6'.



# fearful symmetries

Jeffrey Says

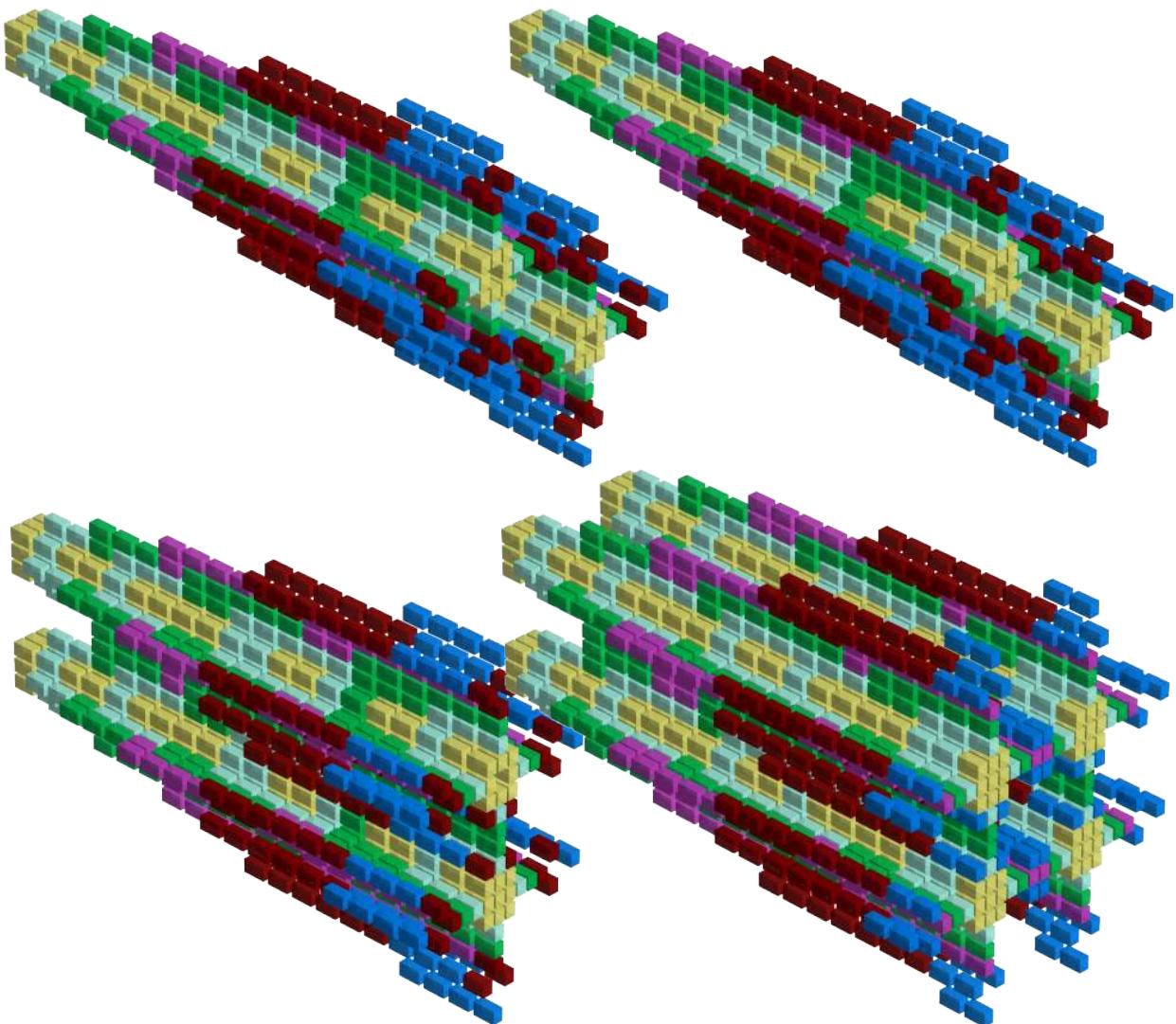


**Press S to change the Symmetry.**

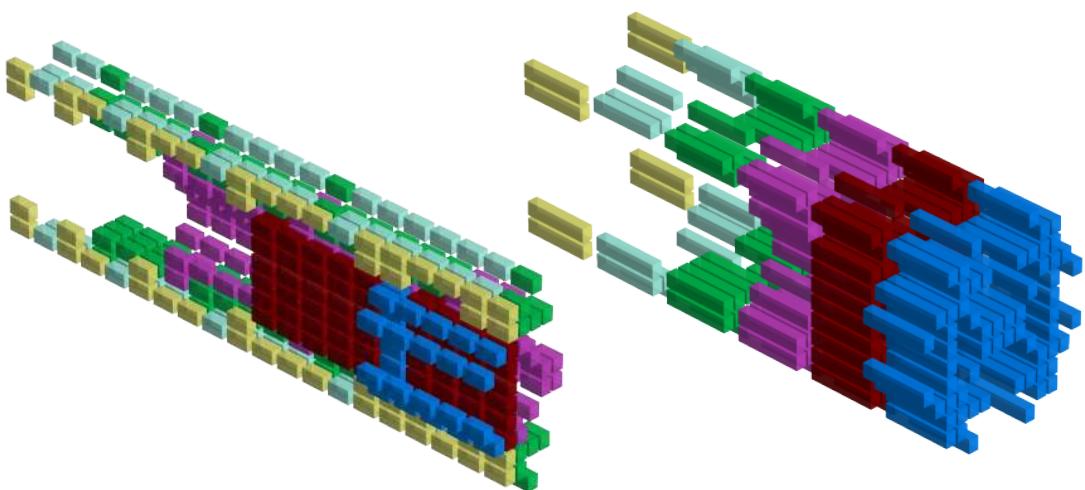
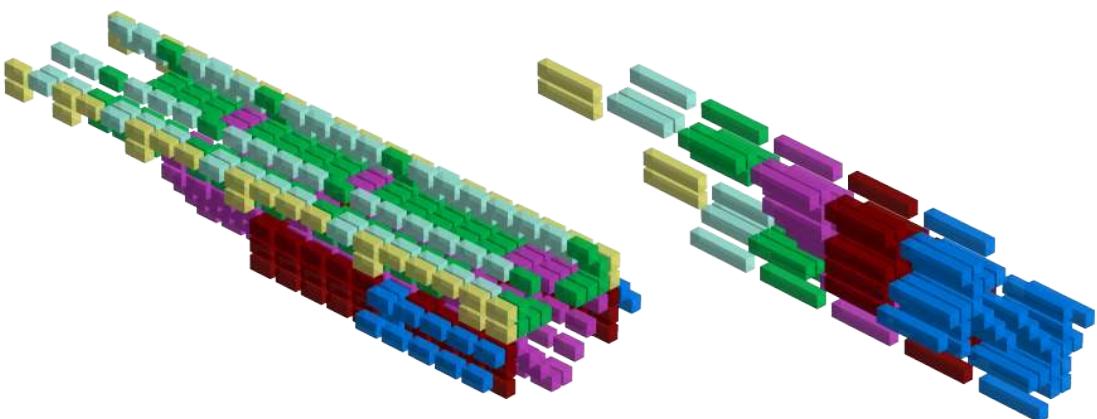
The pattern gets reflected in various planes, or not at all according to the setting.

There are 4 possible symmetries for displaying patterns in Psychedelia. As we saw when reviewing the code for the magazine listing the painting of the symmetries is managed by `PaintPixelForCurrentSymmetry`. There it was hardcoded to use a simple reflection symmetry along the X-axis, but the commercial version of Psychedelia has to go a little further and support reflections along all three possible axes.

Now that the player can change the active symmetry at any time, there is also a little more book-keeping for Psychedelia to do in terms of recognizing the active symmetry for each step in the pattern's evolution. In this chapter we'll take a look at how this *fascinating* bit of accounting works and console ourselves the while with some pretty pictures of the symmetry rendering in operation. First, a couple of pretty pictures. Then some potentially tedious unpacking of code.



Star One: Y-Axis, X-Y Symmetry, X-Axis Symmetry, Quad Symmetry



The Twist: Y-Axis, X-Y Symmetry, X-Axis Symmetry, Quad Symmetry

**Lines 1210-1231.** CheckKeyboardInput

```
; -----
; CheckKeyboardInput
; -----
CheckKeyboardInput
    ...
    LDA lastKeyPressed
    ...
MaybeSPressed
    CMP #KEY_S ; 'S' pressed.
    BNE MaybeLPressed

    ; Check if shift was pressed too.
    LDA shiftPressed
    AND #$01
    BEQ JustSPressed

    LDA tapeSavingInProgress
    BNE JustSPressed
    ; Shift + S pressed: Save.
    JMP PromptToSave
    ; Returns

    ; 'S' pressed.
JustSPressed
    INC currentSymmetrySetting
    LDA currentSymmetrySetting
    CMP #$05
    BNE SetUpYRegisterToGetText
    LDA #$00
    STA currentSymmetrySetting

SetUpYRegisterToGetText
    ASL
    ASL
    ASL
    ASL
    TAY
    JSR ClearLastLineOfScreen

    ; currentSymmetrySetting is in Y
    LDX #$00
WriteSymmetryDescription
    LDA txtSymmetrySettingDescriptions,Y
    STA lastLineBufferPtr,X
    INY
    INX
    CPX #$10
    BNE WriteSymmetryDescription

    JMP WriteLastLineBufferToScreen
    ; Returns
```

**Lines 1210-1231.** MaybeSPressed: This is the routine that detects when the player has selected a new symmetry by pressing the 'S' key. It is part of the much larger routine CheckKeyboardInput which periodically checks for keyboard input by polling the byte at address \$00C5 (which we label lastKeyPressed). This address always contains the value of the most recently pressed key on the keyboard.

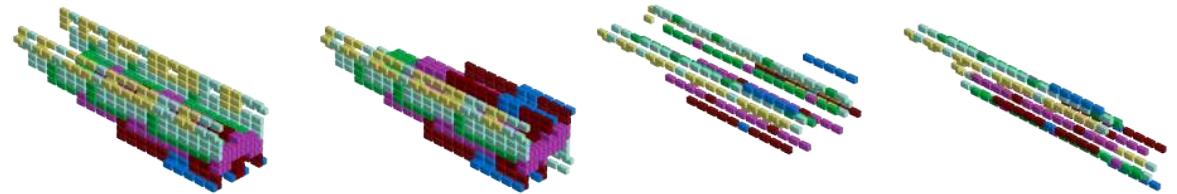
If Shift was pressed the user is actually looking to trigger a save, in which case execution passes to PromptToSave.

Otherwise we increment currentSymmetrySetting, which is the variable we use for storing the currently selected symmetry. Each press of the 'S' key increments the value of currentSymmetrySetting until it reaches 5. Once that happens, we reset it to zero again (LDA #\$00; STA currentSymmetrySetting).

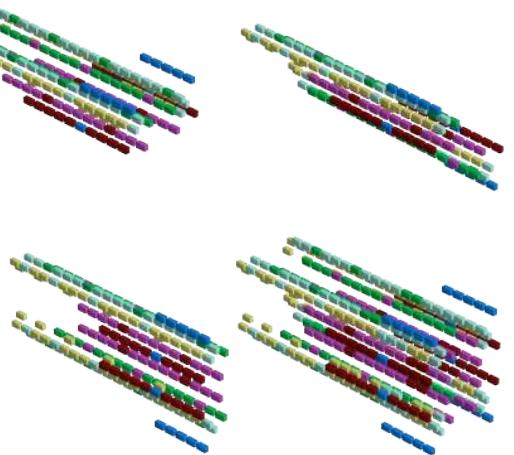
**Lines 1236-1254.** SetUpYRegisterToGetText: What is the series of ASL instructions doing? A tricky piece of business of course. We have loaded currentSymmetrySetting to the A register (LDA currentSymmetrySetting) and it has a value between 0 and 4. ASL performs a leftward bit-shift on the A register. A single left-shift has an interesting property - it doubles the value of the byte. A second left-shift will double it again, and so on. So our four ASL instructions have the effect of turning 1 into 16, 2 into 32, 3 into 48, and 4 into 64. This has the very useful effect of giving us an index into the description of each symmetry!

```
txtSymmetrySettingDescriptions
    .TEXT 'NO SYMMETRY' ; Starts at index 0
    .TEXT 'Y-AXIS SYMMETRY' ; Starts at index 16
    .TEXT 'X-Y SYMMETRY' ; Starts at index 32
    .TEXT 'X-AXIS SYMMETRY' ; Starts at index 48
    .TEXT 'QUAD SYMMETRY' ; Starts at index 64
```

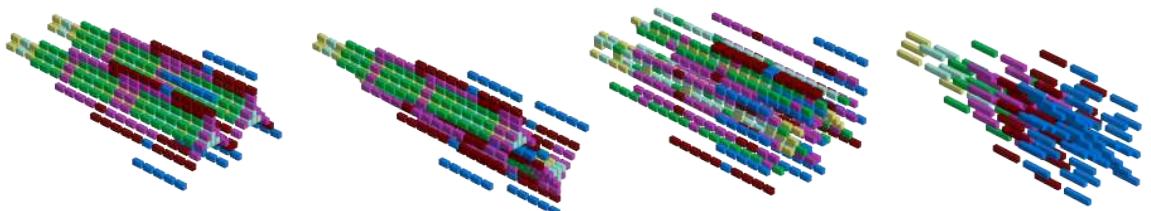
So when we take our resulting value and load it into the Y register all our Write-SymmetryDescription needs to do is start at the index given by Y and write out the next 16 bytes to the screen, displaying the selected symmetry briefly to the player.



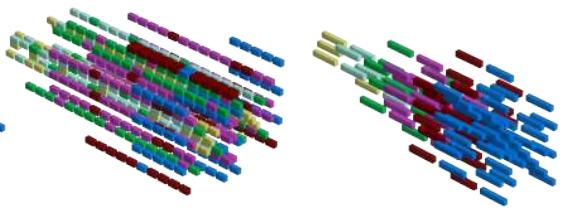
La Llamita



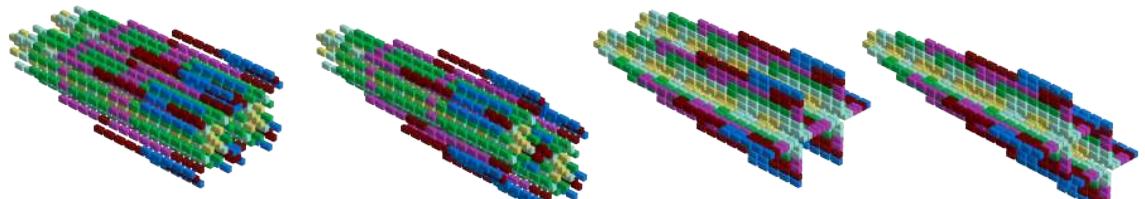
Star Two



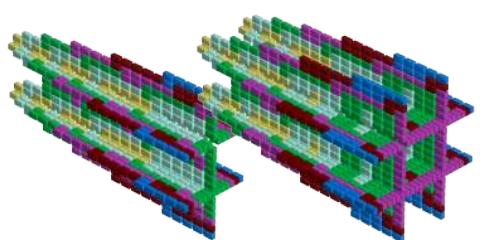
Deltoid



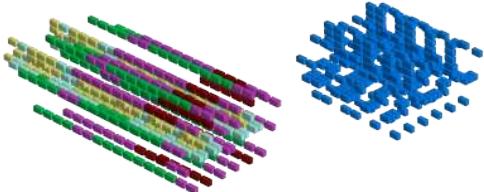
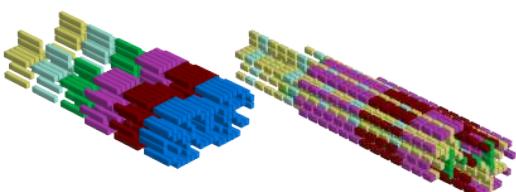
Diffused



Multi-Cross



Pulsar



Custom Pattern 1

Custom Pattern 2

**Lines 368-409.** PaintPixelForCurrentSymmetry

```
;-----
; PaintPixelForCurrentSymmetry
;-----

PaintPixelForCurrentSymmetry
    ; First paint the normal pattern without any
    ; symmetry.
    LDA pixelXPosition
    PHA
    LDA pixelYPosition
    PHA
    JSR PaintPixel

    LDA currentSymmetrySettingForStep
    BNE HasSymmetry

CleanUpAndReturnFromSymmetry
    PLA
    STA pixelYPosition
    PLA
    STA pixelXPosition
    RTS

    RTS

HasSymmetry
    CMP #X_AXIS_SYMMETRY
    BEQ XAxisSymmetry

        ; Has a pattern to paint on the Y axis
        ; symmetry so prepare for that.
        LDA #$27
        SEC
        SBC pixelXPosition
        STA pixelXPosition

        ; If it has X_Y_SYMMETRY then we just
        ; need to paint that, and we're done.
        LDY currentSymmetrySettingForStep
        CPY #X_Y_SYMMETRY
        BEQ XYSymmetry

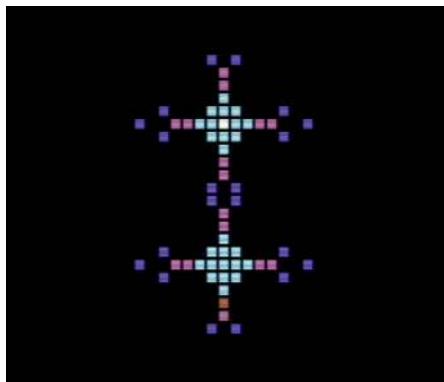
        ; If we're here it's either Y_AXIS_SYMMETRY
        ; or QUAD_SYMMETRY so we can paint a pattern
        ; on the Y axis.
        JSR PaintPixel
```

**Lines 368-409.** PaintPixelForCurrentSymmetry: This routine seems a lot more convoluted than it ought to be. What it needs to do is figure out which of the four symmetries it ought to paint and then paint them. Clarity is a victim of optimization in this instance - making the routine as compact as possible means we need to pick through it to understand it.

**Lines 372,374.** PHA: Notice that we're pushing the values in pixelXPosition and pixelYPosition on to the stack. The reason for this is that we may have to modify one or both of pixelXPosition/pixelYPosition when painting the symmetries, so pushing them onto the stack here means we can later pull them back off the stack (PLA) and restore each to their original values when we're finished.

**Lines 380-387.** CleanUpAndReturnFromSymmetry: The simplest case is easiest. If we get to here the symmetry setting just requires one pattern to be drawn (NO\_SYMMETRY) so after calling PaintPixel we can return early.

**Lines 389-409.** HasSymmetry: We know what this section ought to be doing, but you really do have to stare at it quite a long time to work out how it's doing it. What it ought to be doing is painting a pixel reflected on one or more of the X and Y axes depending on the setting of currentSymmetrySettingForStep. In the process it jumps around a bit, shuffles values on and off the stack and has a couple of points where the routine can return early. None of this really aids the cause of understanding. We'll pick the second simplest case to help give you an idea of how the routine works. This is where the X\_AXIS\_SYMMETRY is active so the code branches to XAxisSymmetry. It's called X-AXIS SYMMETRY but the reflection is on the Y Axis, i.e. on the vertical. Maybe it's called that because the patterns share the same X-coordinate? Either way, it's a confusing name for what it is.



X-AXIS SYMMETRY - despite the name it's along the Y axis.

**Lines 409-458.** PaintPixelForCurrentSymmetry continued.

```
; If it's Y_AXIS_SYMMETRY we're done and can
; return.
LDA currentSymmetrySettingForStep
CMP #Y_AXIS_SYMMETRY
BEQ CleanUpAndReturnFromSymmetry

; Has QUAD_SYMMETRY so the remaining steps are
; to paint two more: one on our X axis and one
; on our Y axis.

; First we do the Y axis.
LDA #NUM_ROWS
SEC
SBC pixelYPosition
STA pixelYPosition
JSR PaintPixel

; Paint one on the X axis.
PaintXAxisPixelForSymmetry
PLA
TAY
PLA
STA pixelXPosition
TYA
PHA
JSR PaintPixel
PLA
STA pixelYPosition
RTS

XAxisSymmetry
LDA #NUM_ROWS
SEC
SBC pixelYPosition
STA pixelYPosition
JMP PaintXAxisPixelForSymmetry

XYSymmetry
LDA #NUM_ROWS
SEC
SBC pixelYPosition
STA pixelYPosition
JSR PaintPixel
PLA
STA pixelYPosition
PLA
STA pixelXPosition
RTS
```

**Lines 441-446.** `XAxisSymmetry`: So what we actually want to do here is reflect the pattern on the Y axis. This means manipulating `pixelYPosition` so that it's offset halfway down the screen. The easiest way to do this is to subtract our current Y position from the total height of the screen. We do this by loading the height (`NUM_ROWS`) into the Accumulator and subtracting our Y position from it (`SBC pixelYPosition` - we then update our Y position with the new value (`STA pixelYPosition`). Next we jump to `PaintXAxisPixelForSymmetry`.

**Lines 429-439.** `PaintXAxisPixelForSymmetry`: Now we can update our X position. We do this by retrieving it from the stack, where we stowed it at the top of `PaintPixelForCurrentSymmetry` and in case we previously modified it elsewhere in the routine. Because we pushed the X position first, and the Y position after it, the first item we pull from the stack will be the Y position. So we pull and stow it in the Y register:

```
PaintXAxisPixelForSymmetry
    PLA
    TAY
```

Now we can pull the X position:

```
PLA
STA pixelXPosition
```

And then push the Y position back onto the stack again:

```
TYA
PHA
```

After calling `PaintPixel` we pull the Y position off the stack again and reload it to `pixelYPosition` so that we leave the routine with both `pixelXPosition/pixelYPosition` by any of the monkey business we got up to in `PaintPixel`.

```
JSR PaintPixel
PLA
STA pixelYPosition
RTS
```

**Lines 518-526.** symmetrySettingForStepCount

```
symmetrySettingForStepCount
    .BYTE $FF,$FF,$FF,$FF,$FF,$FF,$FF
    .BYTE $FF,$FF,$FF,$FF,$FF,$FF,$FF
```

**Lines 644-668.** ShouldDoAPaint

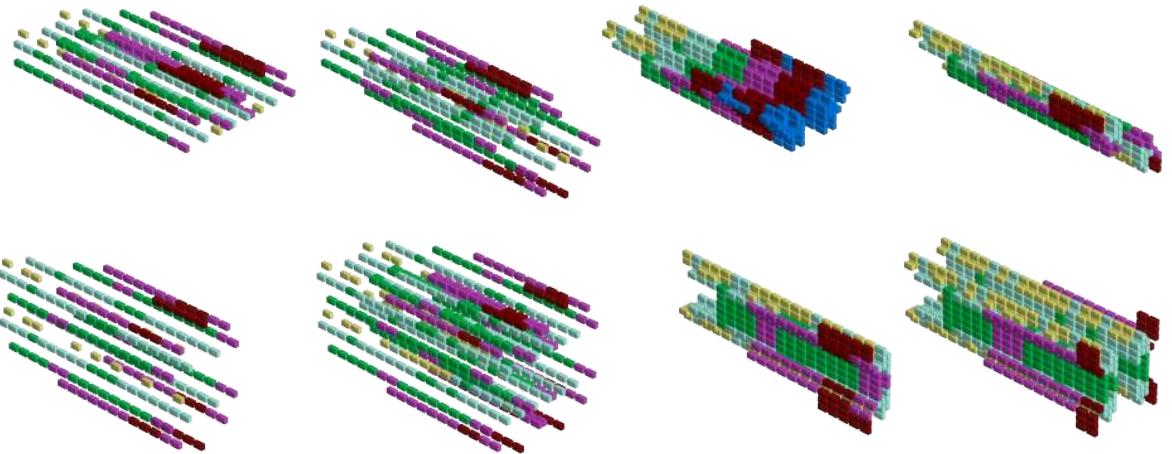
```
ShouldDoAPaint
    ...
; Get the x and y positions for this pixel.
; X is currentIndexToPixelBuffers
LDA pixelXPositionArray,X
STA pixelXPosition
LDA pixelYPositionArray,X
STA pixelYPosition

LDA patternIndexArray,X
STA patternIndex

LDA symmetrySettingForStepCount,X
STA currentSymmetrySettingForStep
```

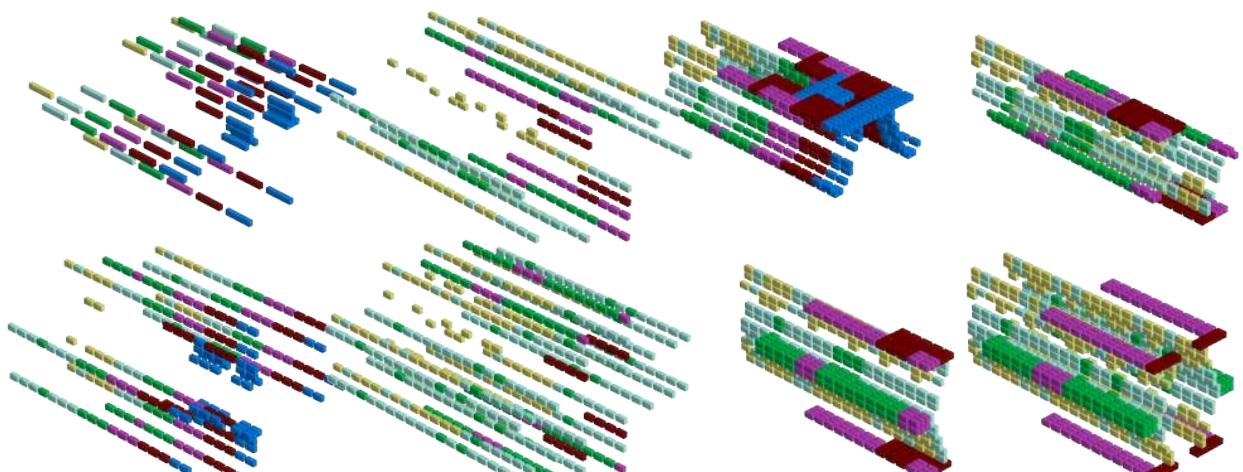
**Lines 518-526.** `symmetrySettingForStepCount`: You probably didn't notice that in `PaintPixelForCurrentSymmetry` we inspected the current symmetry setting in a variable called `currentSymmetrySettingForStep` rather than `currentSymmetrySetting`. Unlike in the listing version, where the symmetry setting was immutable, the user can change the setting at any time so that forces us to manage a condition where different symmetries are in force at different stages in the display. For this reason we have an additional array called `symmetrySettingForStepCount` that tells us what the symmetry used for the current step is.

**Lines 644-668.** `ShouldDoAPaint`: Here we load the symmetry for the current step from our array and store it in `currentSymmetrySettingForStep` for use later in `PaintPixelForCurrentSymmetry`.



Custom Pattern 3

Custom Pattern 4



Custom Pattern 5

Custom Pattern 6

# beatific bursts

## Jeffrey Says



These allow you to preprogram and recall at will instantaneous flashes on the screen. Set up symmetry and smoothing delay as required, then press SHIFT plus the fkey to which you want to assign your FX. Move the cursor to where you want a burst then press Left-Arrow to enter that point. Do this up to 16 times. Press RETURN when done. Pressing the fkey thus assigned stuffs all the points you defined into the buffer instantaneously. Don't worry about it - try the ones I've defined!

Bursts are short pre-programmed sequences of patterns that can be invoked using the Function keys. This allows the player to define up to four different burst sequences, one per function key. A burst can consist of up to 16 different patterns displayed in 16 different positions on the screen. Each burst has a defined symmetry and smoothing delay. The greater the smoothing delay, the longer the pattern will stay on screen.

The bursts are programmable, the player can define new ones through the slightly finicky process of pressing Shift and the corresponding Function key, then selecting the different points on the screen for each of up to 16 patterns to appear. You can see the data structure that supports this on the following page. Note that the symmetry and smoothing delay are set for the burst sequence as a whole, but each element of the burst sequence has its own pattern and position co-ordinates.

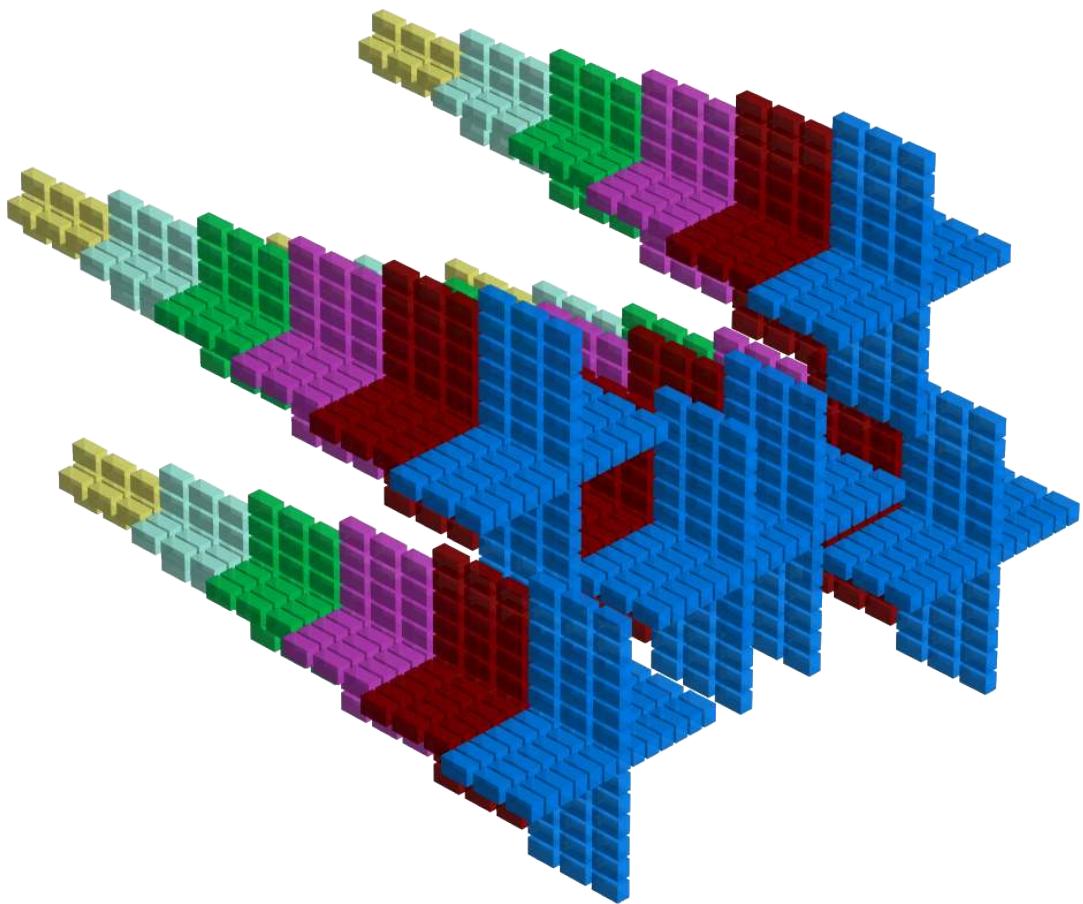


Figure 7.1: Evolution of the default burst at F1.

```

burstGeneratorF1 = $C200
; currentSymmetrySetting: 'Current symmetry setting.'
; Possible values are 0 - 4:
; 'NO SYMMETRY',
; 'Y-AXIS SYMMETRY',
; 'X-Y SYMMETRY',
; 'X-AXIS SYMMETRY',
; 'QUAD SYMMETRY'
.BYTE $01
; smoothingDelay: 'Because of the time taken to draw larger patterns
; speed increase-decrease is not linear. You can adjust the
; compensating delay which often smooths out jerky patterns.
; Can be used just for special FX) though. Suck it and see.'
.BYTE $0C

; Burst Position 1
; X/Y Co-ordinates: X/Y Position relative to cursor to place the
; burst.
.BYTE $07,$06
; Index to pattern in patternIndexArray
.BYTE PULSAR

; Burst Position 2
; X/Y Co-ordinates: X/Y Position relative to cursor to place the
; burst.
.BYTE $11,$0D
; Index to pattern in patternIndexArray
.BYTE PULSAR

; Burst Position 3
; X/Y Co-ordinates: X/Y Position relative to cursor to place the
; burst.
.BYTE $06,$11
; Index to pattern in patternIndexArray
.BYTE PULSAR

; - An $FF in the first byte of 'X/y Co-ordinates' indicates the
; end of the data, e.g. in 'Burst Position 4' below.
; Burst Position 4
; X/Y Co-ordinates: X/Y Position relative to cursor to place the
; burst.
.BYTE $FF,$0B
; Index to pattern in patternIndexArray
.BYTE PULSAR

```

Listing 7.1: Source code for the F1 Burst.

**Lines 1419-1438.** CheckKeyboardInput

```
functionKeys .BYTE $04,$05,$06,$03
;-----
; CheckKeyboardInput
;-----
CheckKeyboardInput
...
CheckForKeyStroke
    LDA lastKeyPressed
    ...

MaybeFunctionKeysPressed
    ; Was one of the function keys pressed?
    LDX #$00
FnKeyLoop
    CMP functionKeys,X
    BEQ FunctionKeyWasPressed ; One of them was pressed!
    INX
    CPX #$04
    BNE FnKeyLoop

    ; Continue checking
    JMP MaybeQPressed

    ; A Function key was pressed, ignore if the
    ; sequencer is active.
FunctionKeyWasPressed
    STX functionKeyIndex

    LDA sequencerActive
    BNE MaybeQPressed

    LDA #SEQUENCER_ACTIVE
    STA currentVariableMode
    JSR LoadOrProgramBurstGenerator
    RTS

    ...
```

**Lines 1419-1438.** MaybeFunctionKeysPressed: This is the routine that detects when the player has selected a new Burst by pressing one of the four 'Function' keys (picured to the right). It is part of the much larger routine CheckKeyboardInput which periodically checks for keyboard input by polling the byte at address \$00C5 (which we label lastKeyPressed). This address always contains the value of the most recently pressed key on the keyboard.



**Lines 1419-1428.** MaybeFunctionKeysPressed: The ellipses before this statement indicated all the code we've left out in the listing opposite. As you might guess, the absent code is checking for lots of other keys you are likely to have pressed but this function checks whether you pressed one of the 4 Function keys we're interested in. Note that it's a loop, checking for each of the 4 function key values stored in functionKeys. If we actually hit one, the loop breaks and calls FunctionKeyWasPressed. Otherwise it reaches JMP MaybeQPressed and calls that function instead - in other words continues on checking for other keys you might have pressed.

**Lines 1431-1438.** FunctionKeyWasPressed: So if we get here a Function key was pressed! Since we broke out of FnKeyLoop when we hit one, the actual function key that we hit is stored in the X register. So first we save that off in a bit of storage called functionKeyIndex for use later. Since the Sequencer (which we will come to in the next chapter) shares functionality with the Burst generator we don't let them both be active at the same time. If the Sequence is not currently active we're free to do some bursting. We set our little state variable called currentVariableMode which keeps track of any fancy business we're doing with the information that we're about to do some burst activation and call the routine that will actually load a burst for us: LoadOrProgramBurstGenerator.

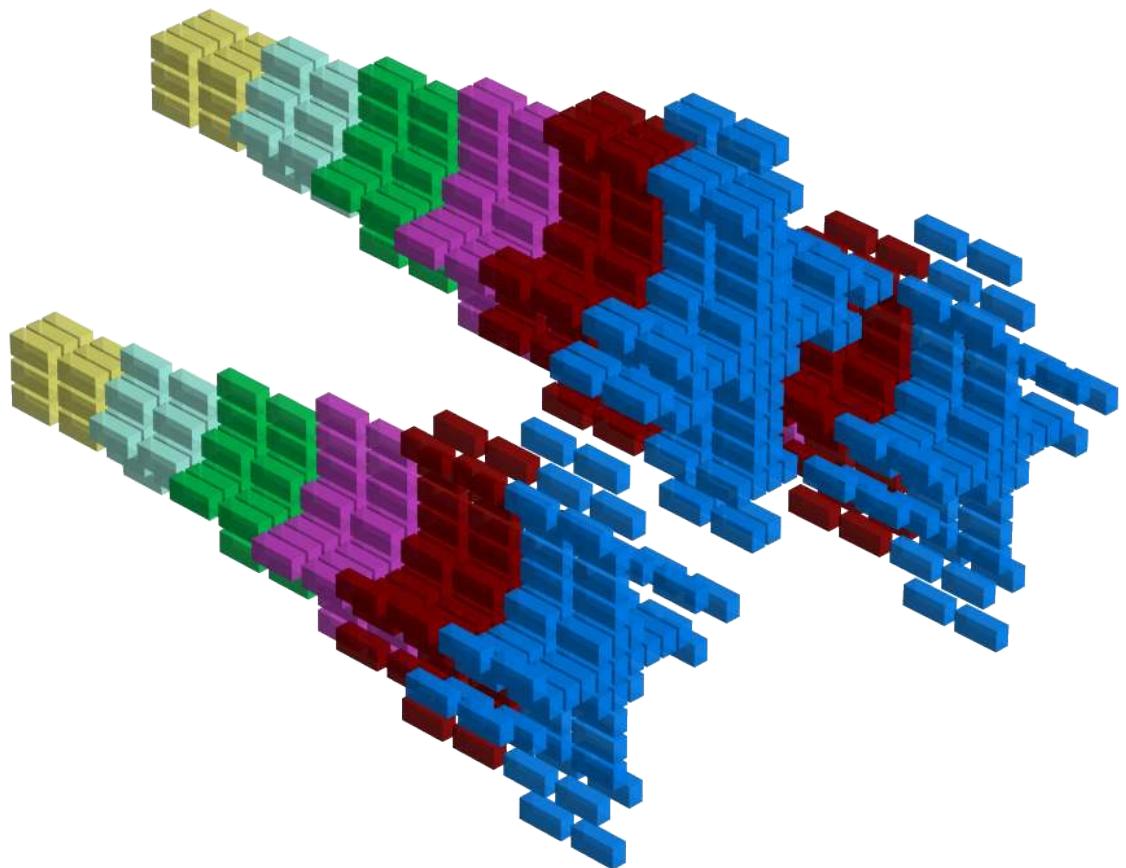


Figure 7.2: Evolution of the default burst at F3.

```
burstGeneratorF3 = $C220
; currentSymmetrySetting: 'Current symmetry setting.'
; Possible values are 0 - 4:
; 'NO SYMMETRY',
; 'Y-AXIS SYMMETRY',
; 'X-Y SYMMETRY',
; 'X-AXIS SYMMETRY',
; 'QUAD SYMMETRY'
.BYTE Y_AXIS_SYMMETRY

; smoothingDelay: 'Because of the time taken to draw larger patterns
; speed increase/decrease is not linear. You can adjust the
; compensating delay which often smooths out jerky patterns.
; Can be used just for special FX) though. Suck it and see.'
.BYTE $0C

; Burst Position 1
; X/Y Co-ordinates: X/Y Position relative to cursor to place burst.
.BYTE $13,$08
; Index to pattern in patternIndexArray
.BYTE STARONE

; Burst Position 2
; X/Y Co-ordinates: X/Y Position relative to cursor to place burst.
.BYTE $07,$0F
; Index to pattern in patternIndexArray
.BYTE STARONE

; Burst Position 3
; X/Y Co-ordinates: X/Y Position relative to cursor to place burst.
.BYTE $FF,$00
; Index to pattern in patternIndexArray
.BYTE MULTICROSS
```

Listing 7.2: Source code for the F3 Burst.

**Lines 2247-2292. LoadOrProgramBurstGenerator**

```
functionKeyToSequenceArray
    .BYTE <burstGeneratorF1,<burstGeneratorF3
    .BYTE <burstGeneratorF5,<burstGeneratorF7

; -----
; LoadOrProgramBurstGenerator
; -----
LoadOrProgramBurstGenerator
    JSR ClearLastLineOfScreen

    LDA shiftPressed
    AND #$01
    BEQ PointToBurstDataForFunctionKey

    ; Display the data free message
    LDX #$00
DisplayDataFreeLoop
    LDA txtDataFree,X
    STA lastLineBufferPtr,X
    INX
    CPX #$10
    BNE DisplayDataFreeLoop
    JSR WriteLastLineBufferToScreen

PointToBurstDataForFunctionKey
    LDA #>burstGeneratorF1
    STA currentSequencePtrHi
    LDX functionKeyIndex
    LDA functionKeyToSequenceArray,X
    STA currentSequencePtrLo

    LDA shiftPressed
    AND #$01
    BEQ LoadBurstDataInstead

ProgramBurstData
    ; Set the current data free to 16
    LDA #$10
    STA currentDataFree

    ; Store the current symmetry setting and smoothing delay
    ; in the storage selected by the function key.
    LDY #$00
    LDA currentSymmetrySetting
    STA (currentSequencePtrLo),Y
    LDA smoothingDelay
    INY
    STA (currentSequencePtrLo),Y
    RTS

LoadBurstDataInstead
    LDA #$FF
    STA sequencerActive
    JMP LoadBurstData
```

**Lines 2247-2292.** `LoadOrProgramBurstGenerator`: OK, we're either loading a burst of programming one: which is it? The way we decide this is by finding out if the Shift key was pressed (`LDA shiftPressed`).. If it was not, then the user wants to load the burst associated with that Function key and we can skip to `PointToBurstDataForFunctionKey`.

**Lines 2264-2273.** `PointToBurstDataForFunctionKey`: This is where we make use of a couple of clever steps of preparation done elsewhere. First of all, the burst data we have (viewable on the pages opposite our pretty pictures) is stored at a number of named locations which we store in an array called `functionKeyToSequenceArray`:

```
functionKeyToSequenceArray
    .BYTE <burstGeneratorF1,<burstGeneratorF3
    .BYTE <burstGeneratorF5,<burstGeneratorF7
```

This means that we can use the `functionKeyIndex` as an index into this array to get the relevant entry for the key that was pressed.

So for example if F3 was pressed our `functionKeyIndex` has the value 01 and we will here retrieve the value `<burstGeneratorF3`. This translates to \$40, the low byte of `burstGeneratorF3`'s location in memory. The high byte is the same for all four of the burst generator data structures: \$C2. This is because they are all located beside each other in memory.

So we get the high byte from `>burstGeneratorF1 ($C2)` and store it in `currentSequencePtrHi` and store our low byte (\$40) in `currentSequencePtrLo`: we now have a pointer to the data for the burst generator associated with F3 and are ready to load it. We'll see how this is done on the next page.

For now though, we check again whether the Shift key is pressed (it isn't) and jump to `LoadBurstDataInstead` and then `LoadBurstData` so that we can get on with, you guessed it, loading the burst data.

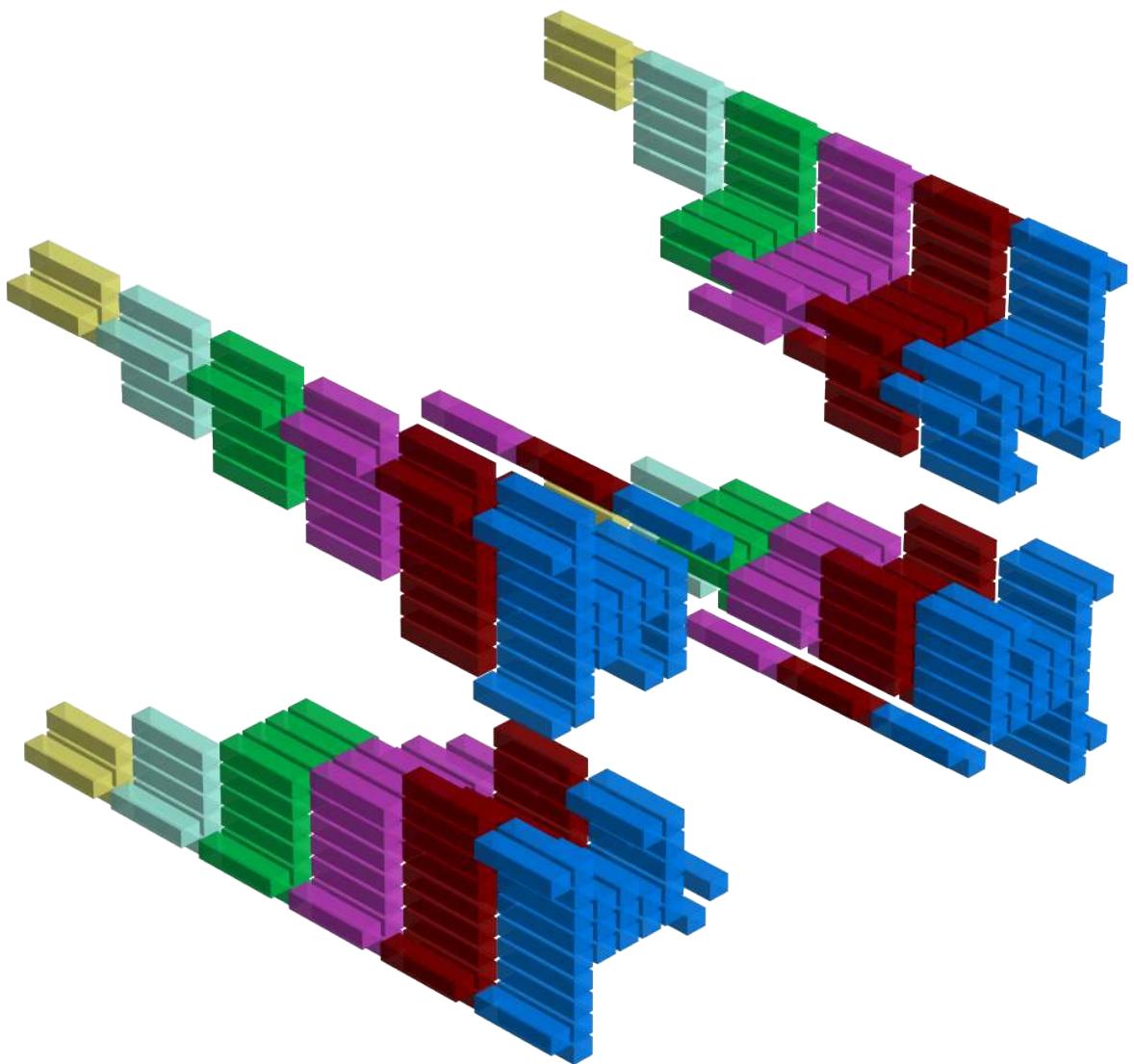


Figure 7.3: Evolution of the default burst at F5.

```
burstGeneratorF5 = $C240
; currentSymmetrySetting: 'Current symmetry setting.'
; Possible values are 0 - 4:
; 'NO SYMMETRY',
; 'Y-AXIS SYMMETRY',
; 'X-Y SYMMETRY',
; 'X-AXIS SYMMETRY',
; 'QUAD SYMMETRY',
.BYTE QUAD_SYMMETRY
; smoothingDelay: 'Because of the time taken to draw larger patterns
; speed increase-decrease is not linear. You can adjust the
; compensating delay which often smooths out jerky patterns.
; Can be used just for special FX) though. Suck it and see.'
.BYTE $01

; Burst Position 1
; X/Y Co-ordinates: X/Y Position relative to cursor to place the
; burst.
.BYTE $08,$01
; Index to pattern in patternIndexArray
.BYTE LALLAMITA

; Burst Position 2
; X/Y Co-ordinates: X/Y Position relative to cursor to place the
; burst.
.BYTE $FF,$01
; Index to pattern in patternIndexArray
.BYTE LALLAMITA
```

Listing 7.3: Source code for the F5 Burst.

**Lines 2477-2511.** LoadBurstData

```
; -----  
; LoadBurstData  
;  
LoadBurstData  
    LDA #\$00  
    STA currentVariableMode  
  
    TAY  
    LDA (currentSequencePtrLo),Y  
    STA symmetrySettingForBurst  
  
    INY  
    LDA (currentSequencePtrLo),Y  
    STA burstSmoothingDelay  
  
LoadNextBurstPosition  
    LDY #$02  
    INC currentStepCount  
    LDA currentStepCount  
    CMP bufferLength  
    BNE DontResetStepCountToZero  
  
    ; If currentStepCount would exceed the length of  
    ; the arrays, reset it to zero.  
    LDA #\$00  
    STA currentStepCount  
  
DontResetStepCountToZero  
    LDX currentStepCount  
    LDA currentColorIndexArray,X  
    CMP #\$FF  
    BEQ LoadBurstToBuffers  
  
    LDA shouldDrawCursor  
    AND trackingActivated  
    BEQ MoveToNextBurstPosition  
  
    STA currentStepCount  
  
    TAX  
    LDA currentColorIndexArray,X  
  
    CMP #\$FF  
    BNE MoveToNextBurstPosition
```

**Lines 2477-2511.** LoadBurstData: OK we're loading burst data this time, honest.

To load the data we have to know where it is, and we established this in the previous section. The player pressed F3 so we stored \$C2 in currentSequencePtrHi and \$40 in currentSequencePtrLo to represent the address \$C240 in memory where the burst data for F3 is stored (see the figure on the previous page also!).

This being the case we can now load the burst data from this address in memory one byte at a time. Let's take as an example the very first byte we load, which we store in the variable symmetrySettingForBurst:

```
LDA (currentSequencePtrLo),Y
STA symmetrySettingForBurst
```

With Y as zero, this will take the first byte at address \$C240 and store it. To load the next byte we increment Y and store the second byte (from address \$C241) in burstSmoothingDelay:

```
INY
LDA (currentSequencePtrLo),Y
STA burstSmoothingDelay
```

Hopefully you get the idea! Loading the rest of the data for the burst follows the same pattern, but has to deal with the complexity that while the two settings we just loaded apply to the visualization as a whole, the remaining burst settings come in groups: each group defines a pattern to draw and a position to draw it at.

**Lines 2487-2492.** LoadNextBurstPosition: This section, which flows on to the following page and forms a loop with the statement JMP LoadNextBurstPosition near the end, is the part that manages loading each group of Burst Positions.

What it has to do is read in the following data as a group and store it in the right place in our pixelXPositionArray/pixelYPositionArray/patternIndexArray.

```
; Burst Position 1
; X/Y Co-ordinates: X/Y Position relative to cursor to place burst.
.BYTE $13,$08
; Index to pattern in patternIndexArray
.BYTE STARONE
```

As you hopefully remember these are the arrays used by Psychedelia to drive the generation of patterns displayed on the screen. It really isn't much of a surprise that in order to display some customized patterns like this, the routine has to ultimately tap the data into the same arrays used to drive the player's interactions with the game.

**Lines 2513-2550.** LoadBurstData continued.

```
; -----  
; LoadBurstData continued.  
;  
LoadBurstToBuffers  
    LDA baseLevel  
    STA currentColorIndexArray,X  
  
    LDA (currentSequencePtrLo),Y  
    CMP #$C0  
    BEQ MoveToNextBurstPosition  
  
    STA pixelXPositionArray,X  
  
    INY  
    LDA (currentSequencePtrLo),Y  
    STA pixelYPositionArray,X  
  
    INY  
    LDA (currentSequencePtrLo),Y  
    STA patternIndexArray,X  
  
    LDA burstSmoothingDelay  
    STA initialSmoothingDelayForStep,X  
    STA smoothingDelayForStep,X  
  
    LDA symmetrySettingForBurst  
    STA symmetrySettingForStepCount,X  
  
MoveToNextBurstPosition  
    LDA currentSequencePtrLo  
    CLC  
    ADC #$03  
    STA currentSequencePtrLo  
  
    LDA currentSequencePtrHi  
    ADC #$00  
    STA currentSequencePtrHi  
  
    LDY #$02  
    LDA (currentSequencePtrLo),Y  
    CMP #$FF  
    BEQ FinishedLoadingBurstData  
    JMP LoadNextBurstPosition  
  
FinishedLoadingBurstData  
    LDA #$00  
    STA sequencerActive  
    RTS
```

**Lines 2513-2531.** LoadBurstToBuffers: Given what we learned on the previous pages, this section might be easy to comprehend.

We're populating our main arrays with the data from the Burst Position. We use Y as our index into the data at the position the user selected (\$C240). We increment Y as we go along to move to the next position in the data and store the byte we retrieve in the appropriate place. When reading in the byte that gives us the X-Coordinate for positioning the burst the appropriate place to store this is of course the pixelXPositionArray.

But where in the array do we store it? The answer to this is given by the variable currentStepCount which we increment at the very start of LoadNextBurstPosition on the previous page. As we increment this we have to ensure it doesn't exceed the length of the arrays, which is why if you flick back and take a look you'll see some code there which detects if it exceeds bufferLength and if it has, resets it back to zero.

**Lines 2533-2545.** MoveToNextBurstPosition: Once we've processed a Burst Position, it's time to move on to the next one. Since each Burst Position is three bytes long, we just have to increment currentSequencePtrLo by 3 bytes:

```
LDA currentSequencePtrLo
CLC ; This ensures we don't add any carry to the result.
ADC #$03
STA currentSequencePtrLo
```

Even though we're not touching the high byte, we also need to reload it, since incrementing currentSequencePtrLo may have inadvertently carried over into it. A strange little dance, but a necessary one:

```
LDA currentSequencePtrHi
ADC #$00
STA currentSequencePtrHi
```

Finally we can check if the first byte in the Burst Position contains \$FF. If it does, that means we have reached the end of the Burst Data and we should bail out. Otherwise we can proceed to load it by looping back to LoadNextBurstPosition.

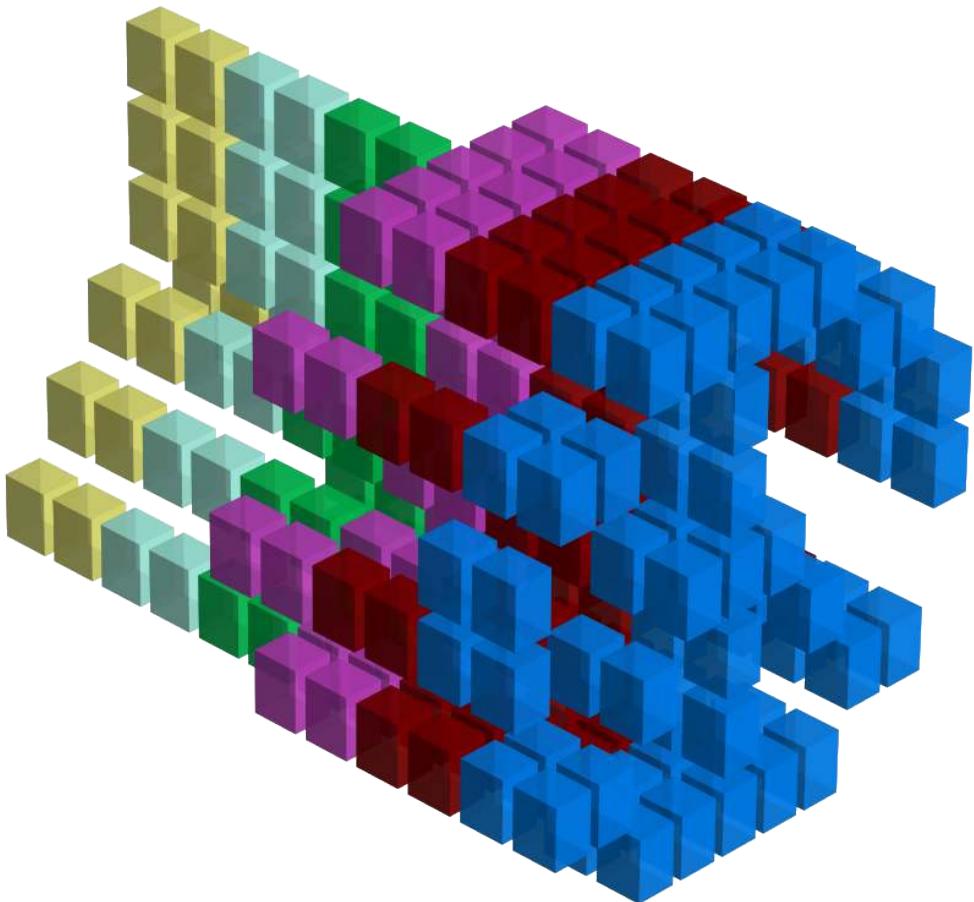


Figure 7.4: Evolution of the default burst at F7.

```
burstGeneratorF7 = $C260
; currentSymmetrySetting: 'Current symmetry setting.'
; Possible values are 0 - 4:
; 'NO SYMMETRY',
; 'Y-AXIS SYMMETRY',
; 'X-Y SYMMETRY',
; 'X-AXIS SYMMETRY',
; 'QUAD SYMMETRY'
.BYTE NO_SYMMETRY
; smoothingDelay: 'Because of the time taken to draw larger patterns
; speed increase-decrease is not linear. You can adjust the
; compensating delay which often smooths out jerky patterns.
; Can be used just for special FX) though. Suck it and see.'
.BYTE $11

; Burst Position 1
; X/Y Co-ordinates: X/Y Position relative to cursor to place burst.
.BYTE $12,$09
; Index to pattern in patternIndexArray
.BYTE CUSTOMPATTERNO

; Burst Position 2
; X/Y Co-ordinates: X/Y Position relative to cursor to place burst.
.BYTE $12,$09
; Index to pattern in patternIndexArray
.BYTE CUSTOMPATTERNO

; Burst Position 3
; X/Y Co-ordinates: X/Y Position relative to cursor to place burst.
.BYTE $FF,$08
; Index to pattern in patternIndexArray
.BYTE STARTTWO
```

Listing 7.4: Source code for the F7 Burst.



# sensitive sequencer

## Jeffrey Says



Programming is as for the Burst Generators, but you have the freedom of 255 steps allowed played back at varying speeds via the Sequencer Speed control. You can leave the program mode in two ways: press SPACE, and next time you go back in with SHIFT-Q the stuff you already defined is not cleared and you add to the end of it, or press RETURN, and next time you go in the sequencer is cleared. Use the SPACE option to change pattern in mid-sequence, for example, or to 'see how it looks so far'.

A sequencer is just a pretend person, gaily twiddling new patterns at different positions on the screen. Instead of painting pixels the boring way with one's bare hands we can program a whole bunch of pretend user input and let Psychedelia play it for us. In order to see how this works in practice, let's look at how the sequencer data is loaded and processed by walking through the sequencer that comes packaged with Psychedelia. Before that, we can take a look at it in all its glory on the next page along with the less glorious looking data structure that is responsible for it.

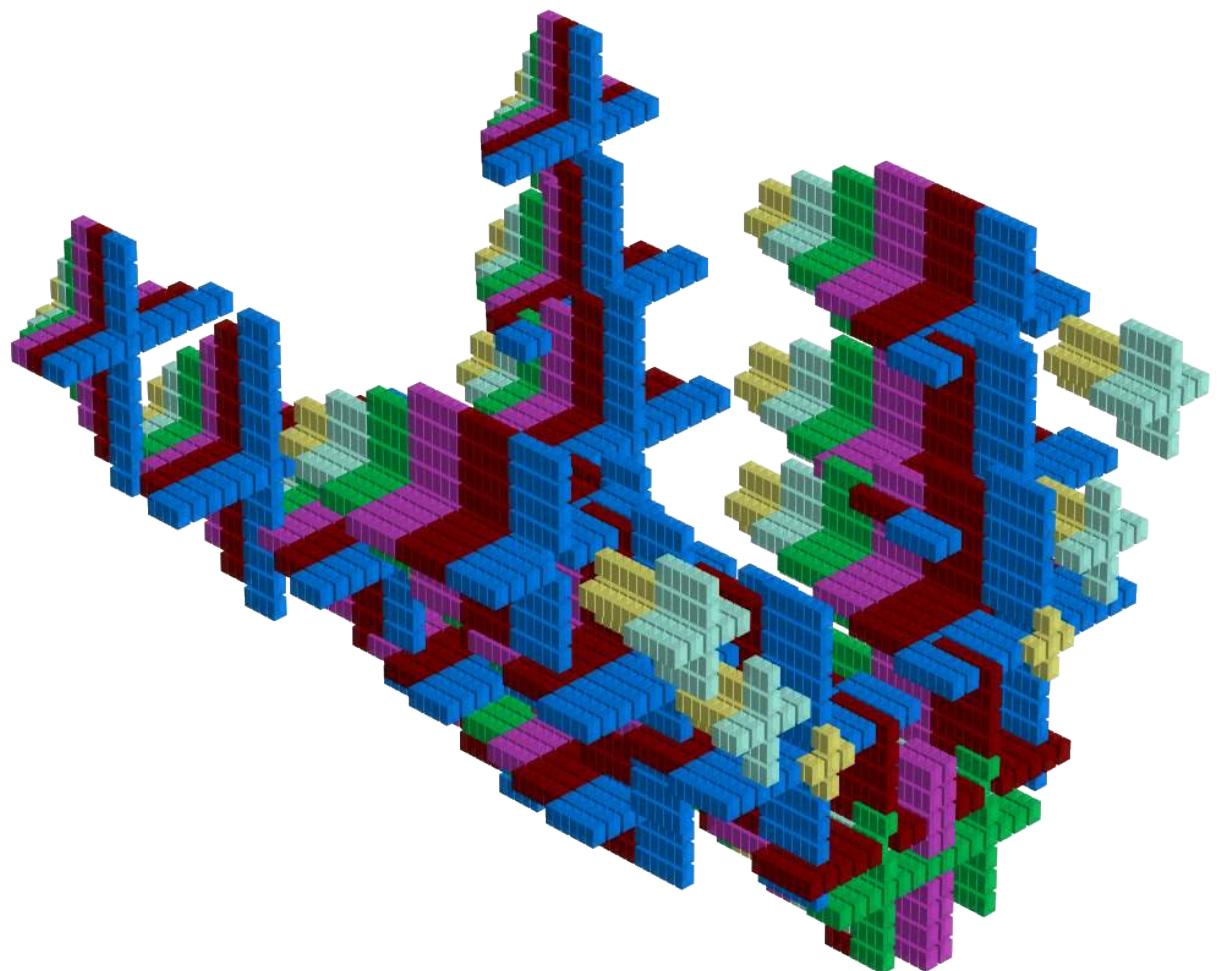


Figure 8.1: Evolution of the Default Sequencer.

```

sequencerDataStorage = $C300
; currentSymmetrySetting: 'Current symmetry setting.'
; Possible values are 0 - 4:
; 'NO SYMMETRY',
; 'Y-AXIS SYMMETRY',
; 'X-Y SYMMETRY',
; 'X-AXIS SYMMETRY',
; 'QUAD SYMMETRY',
.BYTE $01           ; Y-Axis Symmetry

; smoothingDelay: 'Because of the time taken to draw larger patterns speed
; increase/decrease is not linear. You can adjust the compensating delay
; which often smooths out jerky patterns. Can be used just for special FX)
; though. Suck it and see.'
.BYTE $0B

; Sequencer Position 1
.BYTE $04,$04;   ; X/Y Co-ordinates: X/Y Position (absolute position).
.BYTE PULSAR;    ; Index to pattern in pixelXPositionHi/LoPtrArray

; Sequencer Position 2
.BYTE $08,$09;   ; X/Y Co-ordinates: X/Y Position (absolute position).
.BYTE PULSAR;    ; Index to pattern in pixelXPositionHi/LoPtrArray

; Sequencer Position 3
.BYTE $0C,$0C;   ; X/Y Co-ordinates: X/Y Position (absolute position).
.BYTE PULSAR;    ; Index to pattern in pixelXPositionHi/LoPtrArray

; Sequencer Position 4
.BYTE $10,$11;   ; X/Y Co-ordinates: X/Y Position (absolute position).
.BYTE PULSAR;    ; Index to pattern in pixelXPositionHi/LoPtrArray

; Sequencer Position 5
.BYTE $14,$13;   ; X/Y Co-ordinates: X/Y Position (absolute position).
.BYTE PULSAR;    ; Index to pattern in pixelXPositionHi/LoPtrArray

; Sequencer Position 6
.BYTE $17,$13;   ; X/Y Co-ordinates: X/Y Position (absolute position).
.BYTE PULSAR;    ; Index to pattern in pixelXPositionHi/LoPtrArray

; Sequencer Position 7
.BYTE $FF,$01;   ; X/Y Co-ordinates: X/Y Position (absolute position).
.BYTE MULTICROSS ; Index to pattern in pixelXPositionHi/LoPtrArray

; Sequencer Position 8
.BYTE $41,$FF;   ; X/Y Co-ordinates: X/Y Position (absolute position).
.BYTE $00;;      ; Index to pattern in pixelXPositionHi/LoPtrArray

; Sequencer Position 9
.BYTE $06,$01;   ; X/Y Co-ordinates: X/Y Position (absolute position).
.BYTE MULTICROSS ; Index to pattern in pixelXPositionHi/LoPtrArray

...
; Sequencer Position 254
.BYTE $FF,$80;   ; X/Y Co-ordinates: X/Y Position (absolute position).
.BYTE $EE;;      ; Index to pattern in pixelXPositionHi/LoPtrArray

; Sequencer Position 255
.BYTE $FD,$FF;   ; X/Y Co-ordinates: X/Y Position (absolute position).
.BYTE $FF;;      ; Index to pattern in pixelXPositionHi/LoPtrArray

```

Listing 8.1: Sequencer definition in sequencer\_data.asm.

**Lines 1440-1457.** MaybeQPressed

```
MaybeQPressed
    CMP #KEY_Q ; Q pressed?
    BNE MaybeVPressed

    ; Q was pressed. Toggle the sequencer on or off.
    LDA sequencerActive
    BNE TurnSequenceOff

TurnSequenceOn
    LDA #SEQUENCER_ACTIVE
    STA currentVariableMode
    JMP ActivateSequencer
    ; Returns

    ; Turn the sequencer off.
TurnSequenceOff
    LDA #$00
    STA sequencerActive
    STA stepsRemainingInSequencerSequence
    JMP DisplaySequencerState
```

**Lines 2558-2575.** ActivateSequencer

```
ActivateSequencer
    LDA #>sequencerDataStorage
    STA currentSequencePtrHi

    LDA #<sequencerDataStorage
    STA currentSequencePtrLo

    LDA #$FF
    STA sequencerActive

    LDA shiftPressed
    AND #$01
    BNE ShiftPressedSoProgramSequencer

    ; Start Playing the Sequencer
    LDA sequencerSpeed
    STA stepsRemainingInSequencerSequence

    LDA #$00
    STA currentVariableMode

    JSR DisplaySequencerState
    RTS
```

**Lines 1440-1457.** MaybeQPressed: To see the sequencer in action we have to activate it. Pressing Q turns it on or off, depending on whether it is already, um, on or off. This is where we make this important decision so it seems like a good place to start in our effort to understand how the thing works.

sequencerActive is the variable that keeps track of on- and off-ness. Anyway if it's on: we turn it off with TurnSequenceOff. If it's off however: we do something much more exciting. We turn it on with the grandly titled routine ActivateSequencer, which we come to next. Programming is fascinating and complex I find.

**Lines 2558-2575.** ActivateSequencer: Despite its heady name, this routine just sets a couple of variables. Storing \$FF in sequencerActive flags that the sequencer is active, which as we just saw is useful to know if the player wants to turn it off by pressing Q again.

There are a couple of other meaningful things to do in here. One is to set up the interval at which we will feed data into the sequencer. This is managed by stepsRemaining-InSequencerSequence. We'll see how this is used in the very next page. The value we initialize it with is itself a user-definable setting called sequencerSpeed.

The final bit to do is to store the location of the sequencer data in a pair of variables that we will use to read in the sequencer data. We saw this technique used in the previous chapter on Burst data. The high and low bytes of the address sequencerDataStorage (which is \$C300) are loaded to currentSequencePtrHi (which gets \$C3) and currentSequencePtrLo (which gets \$00) respectively:

```
LDA #>sequencerDataStorage  
STA currentSequencePtrHi  
  
LDA #<sequencerDataStorage  
STA currentSequencePtrLo
```

With that, we can consider the Sequencer fully activated. Psychedelia is now ready to use it in a way that you might find slightly convoluted, or even moderately interesting. Read on.

**Lines 728-743.** MainInterruptHandler

```
MainInterruptHandler
; The sequencer is played by the interrupt handler.
; Check if it's active.
LDA stepsRemainingInSequencerSequence
BEQ SequencerNotActiveCheckJoystickInput
DEC stepsRemainingInSequencerSequence
BNE SequencerNotActiveCheckJoystickInput

; If the sequencer is active we'll end up here and
; load the sequencer data so that it can be played.
LDA sequencerSpeed
STA stepsRemainingInSequencerSequence
JSR LoadDataForSequencer
```

**Lines 728-743.** MainInterruptHandler: We already encountered the operation of the MainInterruptHandler [our walk through of the listing](#). What we have here is the first part of the same routine that shipped with the commercial version of Psychedelia. It is pretty much the same in principle: it looks for input from the user on the joystick and moves the cursor around in response, and it also keeps the buffers fed: the buffers being the arrays that tell Psychedelia what to paint and where.

Normally this would be based on the user input, but if we have a sequencer active then this is also our opportunity to feed those buffers with data from the sequencer, over and over. As we mentioned at the head of the chapter, the Sequencer is just like a pre-programmed bit of user input fed repeatedly into the Psychedelia engine, over and over, until you make it stop.

So since MainInterruptHandler is the place where actual user input is processed it makes sense that it is also the place where 'simulated' user input such as that from the sequencer data is also digested.

All we do in this little section is determine whether we want to process some of the sequencer data or not. This is where the variable `stepsRemainingInSequencerSequence` we initialized on the previous page comes in. If it's zero the sequencer isn't active and we can just jump straight to `SequencerNotActiveCheckJoystickInput`. Otherwise we decrement it by one. If it's still not zero, then we also skip loading any sequencer data. However if it has reached the magic number of zero it's time to chew some of the goodness in `sequencerDataStorage`. First we reinitialize `stepsRemainingInSequencerSequence` to the selected sequencer speed so it won't be zero the next time around. That done we are ready to process us some sequencer data by calling `LoadDataForSequencer`. Let's go to the next page and see what does. (I wonder what!)

**Lines 2616-2663.** LoadDataForSequencer

```
LoadDataForSequencer
    INC currentStepCount
    LDA currentStepCount
    CMP bufferLength
    BNE MaybeLoadSequencerDataToCurrentSlot

    LDA #$00
    STA currentStepCount

MaybeLoadSequencerDataToCurrentSlot
    TAX
    LDA currentIndexForCurrentStepArray ,X
    CMP #$FF
    BEQ LoadValuesFromSequencerData

    LDA shouldDrawCursor
    AND trackingActivated
    BEQ MoveToNextPositionInSequencer

    TAX
    LDA currentIndexForCurrentStepArray ,X
    CMP #$FF
    BNE MoveToNextPositionInSequencer

LoadValuesFromSequencerData
    LDY #$02
    LDA (currentSequencePtrLo),Y
    CMP #$C0
    BEQ MoveToNextPositionInSequencer

    LDA baseLevel
    STA currentIndexForCurrentStepArray ,X

    LDA sequencerDataStorage + $01
    STA initialSmoothingDelayForStep ,X
    STA smoothingDelayForStep ,X

    LDA sequencerDataStorage
    STA symmetrySettingForStepCount ,X

    LDY #$02
    LDA (currentSequencePtrLo),Y
    STA pixelXPositionArray ,X

    INY
    LDA (currentSequencePtrLo),Y
    STA pixelYPositionArray ,X

    INY
    LDA (currentSequencePtrLo),Y
    STA patternIndexArray ,X
```

**Lines 2616-2663.** LoadDataForSequencer: We've already seen a version of this movie before. When we loaded the burst data in the previous chapter our routine was parsing the data piece by piece and loading it into the buffers. We're doing the same thing here. The only difference is that instead of doing it just the once (and loading all of it in one go), for a burst, we're calling this routine a few times every second from MainInterruptHandler. The exact frequency was controlled on the previous page by stepsRemainingInSequencerSequence.

Since we're being called at least a few times every second we're not attempting to read all of the sequencer data at once here like we did for the bursts. Instead we just read in the next step in the sequence and add that to the buffers. Once that's done, we advance our pointer (currentSequencePtrLo) a few bytes so that it is in the correct position the next time we're called.

Each step is three bytes long. The first step we load in, the first time we're called, is this guy:

```
; Sequencer Position 1
.BYTE $04,$04; ; X/Y Co-ordinates: X/Y Position (absolute position).
.BYTE PULSAR; ; Index to pattern in pixelXPositionHi/LoPtrArray
```

**Lines 2625-2637.** MaybeLoadSequencerDataToCurrentSlot: Is there a free spot in the buffers for some of our sequence data or not? If yes, great: we can jump to LoadValuesFromSequencerData. Otherwise we just move our pointer currentSequencePtrLo to the next three bytes in the sequencer data and bail out by calling MoveToNextPositionInSequencer.

**Lines 2639-2663.** LoadValuesFromSequencerData: I don't know, maybe we should like, load some data? The first two bytes are taken from the very top of the sequencer data every time:

```
sequencerDataStorage = $C300
.BYTE Y_AXIS_SYMMETRY ; Y-Axis Symmetry
.BYTE $0B ; Smoothing Delay
```

We load them like so the appropriate position in the initialSmoothingDelayForStep and symmetrySettingForStepCount buffers like so:

```
LDA sequencerDataStorage + $01
STA initialSmoothingDelayForStep,X
STA smoothingDelayForStep,X

LDA sequencerDataStorage
STA symmetrySettingForStepCount,X
```

**Lines 2665-2684.** LoadDataForSequencer continued.

```
MoveToNextPositionInSequencer
    LDA currentSequencePtrLo
    CLC
    ADC #$03
    STA currentSequencePtrLo

    LDA currentSequencePtrHi
    ADC #$00
    STA currentSequencePtrHi

    LDY #$02
    LDA (currentSequencePtrLo),Y
    CMP #$FF
    BEQ ResetSequencerToStart
    RTS

ResetSequencerToStart
    LDA #<sequencerDataStorage
    STA currentSequencePtrLo

    LDA #>sequencerDataStorage
    STA currentSequencePtrHi

    RTS
```

Next we load the three bytes specific to this step in the sequence using our pointer `currentSequencePtrLo` to the `pixelXPositionArray`, `pixelYPositionArray`, and `patternIndexArray` respectively. With that we've loaded our little step of sequence data to the buffers and they are ready to paint when the engine finally gets around to it in a few milliseconds or so.

**Lines 2665-2684.** `MoveToNextPositionInSequencer`: Now that we've loaded a step of sequence data we can repoint our pointer and get out of here.

Maybe I didn't mention it but we don't have much time to waste in here, since we're in the middle of an interrupt and there is supposed to be 60 of these every second. If we spend too much time executing code all sorts of nasty things will happen on the screen and Llamasoft will receive letters of complaint.

Skipping ahead to the next step is easy, we increment our pointer by three bytes:

```
LDA currentSequencePtrLo  
CLC  
ADC #$03  
STA currentSequencePtrLo
```

Of course, there's other bookkeeping to do like checking if we've reached the end of the sequencer's data by seeing if we have a `$FF` in the first byte of our new step. If that's the case we need to reset our pointer `currentSequencePtrLo` back to the start of the location where the sequencer data is stored (`$C300` aka `sequencerDataStorage`):

```
LDA #<sequencerDataStorage  
STA currentSequencePtrLo  
  
LDA #>sequencerDataStorage  
STA currentSequencePtrHi
```

And with that, we're out of here. A step of the sequencer has been loaded and will shortly turn up on the screen. All that remains is to do this numerous times per second ad infinitum until the player begs us to stop.



# particular presets

## Jeffrey Says



Variations: Try pressing any of the top row of keys from Left-Arrow up to Insert/Del. This calls in one of the 16 presets, stored Lightsynth parameters which give different effects. Try them all out to see some of the multitude of effects which you can achieve using the system. Some are fast, some slow, some pulse, others swirl. Play with them all, try them to different music.

Psychedelia allows you to save whatever your configuration is at the moment for easy use later on. The 16 available slots come pre-configured with some colorful configurations. We'll take a look at them all here, as well as understanding how the presets are loaded by the player.

On the following two pages we show the first preset configuration in action and on the facing page the data structure that sits behind it, enumerating all the settings a 'preset' contains. Elsewhere in this chapter we illustrate the 15 other presets slightly more concisely, with a diagram and a briefer version of the same data structure.

particular presets

---

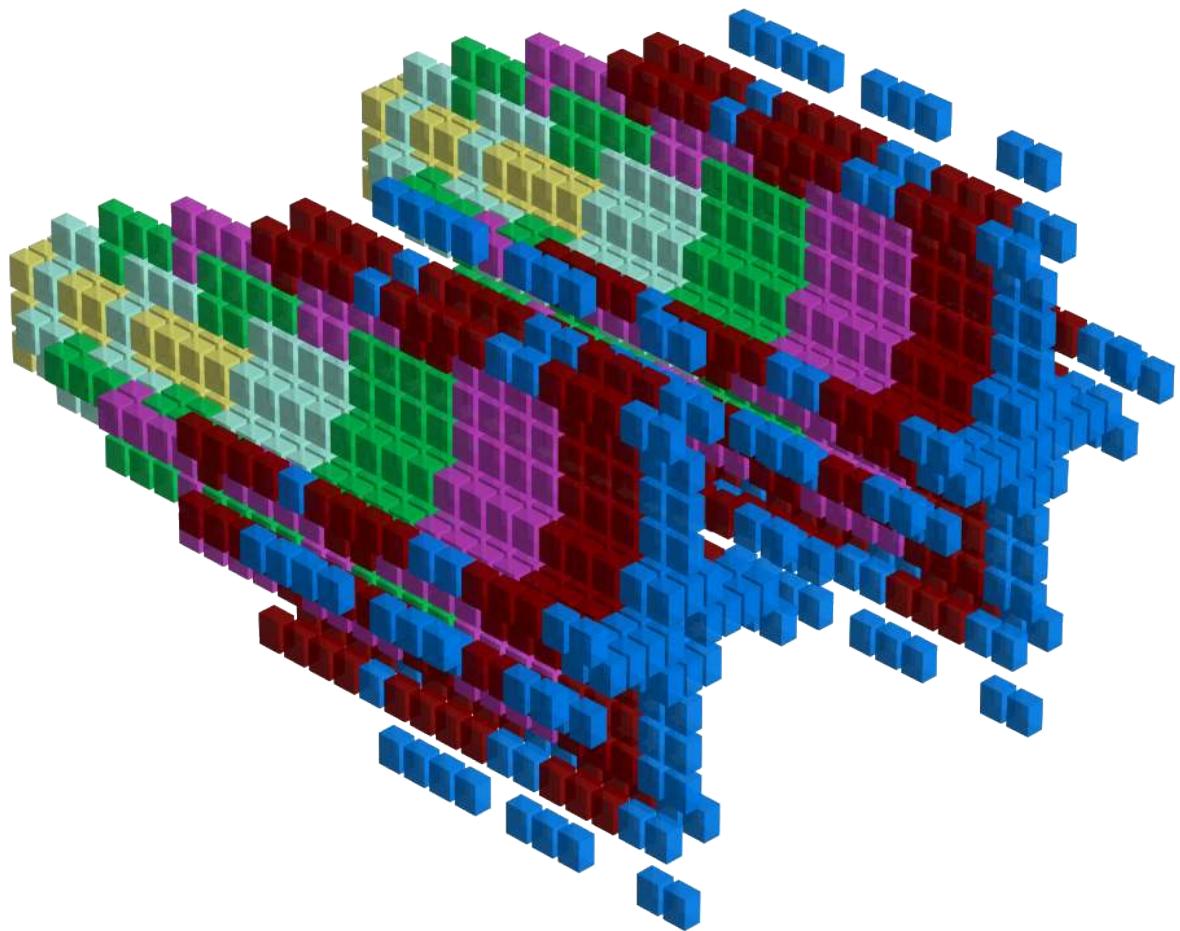


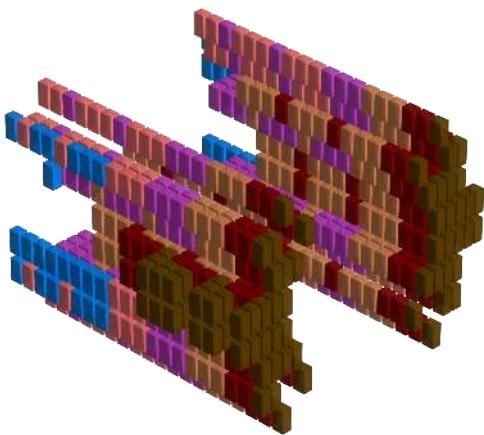
Figure 9.1: Evolution of Preset 0.

```

;preset0
; unusedPresetByte: Unused Byte
.BYTE $00
; smoothingDelay: 'Because of the time taken to draw larger patterns speed
; increase/decrease is not linear. You can adjust the 'compensating delay'
; which often smooths out jerky patterns. Can be used just for special FX),
; though. Suck it and see.'
.BYTE $0C
; cursorSpeed: 'Gives you a slow or fast little cursor, according to setting.'
.BYTE $02
; bufferLength: 'Larger patterns flow more smoothly with a shorter
; Buffer Length - not so many positions are retained so less plotting to do.
; Small patterns with a long Buffer Length are good for 'steamer' effects.
; N.B. Cannot be adjusted whilst patterns are actually onscreen.'
.BYTE $1F
; pulseSpeed: 'Usually if you hold down the button you get a continuous
; stream. Setting the Pulse Speed allows you to generate a pulsed stream, as
; if you were rapidly pressing and releasing the FIRE button.'
.BYTE $01
; indexForColorBarDisplay: 'The initial index for the color displayed
; in the color bar when adjusting the colors for each step.'
.BYTE $01
; lineWidth: 'Sets the width of the lines produced in Line Mode.'
.BYTE $07
; sequencerSpeed: 'Controls the rate at which sequencer feeds in its data. '
.BYTE $04
; pulseWidth: 'Sets the length of the pulses in a pulsed stream output.
; Don't worry about what that means - just get in there and mess with it.'
.BYTE $01
; baseLevel: 'Controls how many 'levels' of pattern are plotted.'
.BYTE $07
; presetColorValuesArray: 'Allows you to set the colour for each of the
; seven pattern steps. Set up the colour you want, press RETURN, and the
; command offers the next colour along, up to no. 7, then ends. Cannot be
; adjusted while patterns being generated.'
.BYTE BLACK,BLUE,RED,PURPLE,GREEN,CYAN,YELLOW,WHITE
; trackingActivated: 'Controls whether logic-seeking is used in the
; buffer or not. The upshot of this for you is a slightly different feel -
; continuous but fragmented when ON, or together-ish bursts when OFF. Try it.'
.BYTE $00
; lineModeActivated: 'A bit like drawing with the Aurora Borealis'
.BYTE $00
; presetIndex: 'This calls in one of the 16 presets, stored Lightsynth
; parameters which give different effects. Try them all out to see some of
; the multitude of effects which you can achieve using the system. Some are
; fast, some slow, some pulse, others swirl. Play with them all, try them to
; different music..'
.BYTE $00
; currentPatternElement: 'Initial pattern used by this preset.'
.BYTE $00
; currentSymmetrySetting: 'Current symmetry setting.'
; Possible values are 0 - 4:
; 'NO SYMMETRY',
; 'Y-AXIS SYMMETRY',
; 'X-Y SYMMETRY',
; 'X-AXIS SYMMETRY',
; 'QUAD SYMMETRY',
.BYTE $01
; Unused Data.
.BYTE $FF,$00,$FF,$FF,$00,$FF,$00,$FF,$00

```

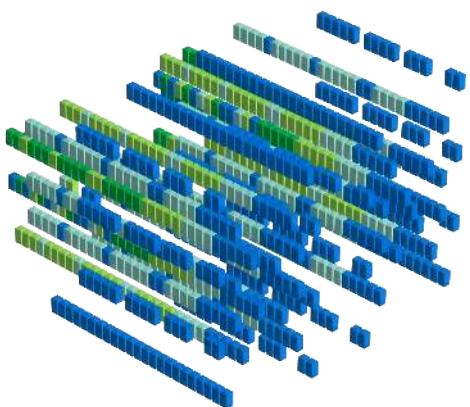
Listing 9.1: Data structure for Preset 0.



```
preset1
.BYTE $00 ; unusedPresetByte
.BYTE $0C ; smoothingDelay
.BYTE $02 ; cursorSpeed
.BYTE $28 ; bufferLength
.BYTE $01 ; pulseSpeed
.BYTE $0E ; indexForColorBarDisplay
.BYTE $07 ; lineWidth
.BYTE $08 ; sequencerSpeed
.BYTE $01 ; pulseWidth
.BYTE $07 ; baseLevel
; presetColorValuesArray:
.BYTE BLACK,BROWN,RED,ORANGE,PURPLE,
    LTRED,BLUE,LTBLUE
.BYTE $FF ; trackingActivated
.BYTE $00 ; lineModeActivated
.BYTE $01 ; presetIndex
.BYTE $01 ; currentPatternElement
.BYTE $04 ; currentSymmetrySetting
```

Listing 9.2: Data structure for Preset 1.

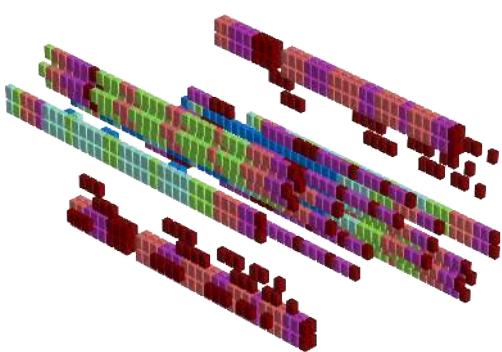
Figure 9.2: Evolution of Preset 1.



```
preset2
.BYTE $00 ; unusedPresetByte
.BYTE $0B ; smoothingDelay
.BYTE $02 ; cursorSpeed
.BYTE $28 ; bufferLength
.BYTE $01 ; pulseSpeed
.BYTE $01 ; indexForColorBarDisplay
.BYTE $07 ; lineWidth
.BYTE $0B ; sequencerSpeed
.BYTE $01 ; pulseWidth
.BYTE $07 ; baseLevel
; presetColorValuesArray:
.BYTE BLACK,BLUE,LTBLUE,CYAN,LTGREEN
    ,GREEN,LTBLUE,BLUE
.BYTE $FF ; trackingActivated
.BYTE $00 ; lineModeActivated
.BYTE $05 ; presetIndex
.BYTE $05 ; currentPatternElement
.BYTE $01 ; currentSymmetrySetting
```

Listing 9.3: Data structure for Preset 2.

Figure 9.3: Evolution of Preset 2.



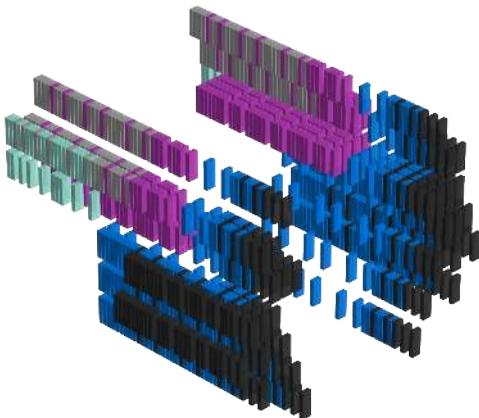
```

preset3
.BYTE $00 ; unusedPresetByte
.BYTE $04 ; smoothingDelay
.BYTE $02 ; cursorSpeed
.BYTE $26 ; bufferLength
.BYTE $01 ; pulseSpeed
.BYTE $01 ; indexForColorBarDisplay
.BYTE $07 ; lineWidth
.BYTE $0A ; sequencerSpeed
.BYTE $01 ; pulseWidth
.BYTE $07 ; baseLevel
; presetColorValuesArray:
.BYTE BLACK,RED,PURPLE,LTRED,LTGREEN
,CYAN,LTBLUE,BLUE
.BYTE $00 ; trackingActivated
.BYTE $00 ; lineModeActivated
.BYTE $0E ; presetIndex
.BYTE $0E ; currentPatternElement
.BYTE $02 ; currentSymmetrySetting

```

Listing 9.4: Data structure for Preset 3.

Figure 9.4: Evolution of Preset 3.



```

preset4
.BYTE $00 ; unusedPresetByte
.BYTE $0C ; smoothingDelay
.BYTE $01 ; cursorSpeed
.BYTE $2B ; bufferLength
.BYTE $01 ; pulseSpeed
.BYTE $07 ; indexForColorBarDisplay
.BYTE $07 ; lineWidth
.BYTE $08 ; sequencerSpeed
.BYTE $01 ; pulseWidth
.BYTE $07 ; baseLevel
; presetColorValuesArray:
.BYTE BLACK,GRAY1,BLUE,GRAY2,PURPLE,
GRAY3,CYAN,WHITE
.BYTE $00 ; trackingActivated
.BYTE $00 ; lineModeActivated
.BYTE $01 ; presetIndex
.BYTE $01 ; currentPatternElement
.BYTE $01 ; currentSymmetrySetting

```

Listing 9.5: Data structure for Preset 4.

Figure 9.5: Evolution of Preset 4.

particular presets

---

#### Lines 980-994. presetValueArray

```
; This is where the presets get loaded to. It represents
; the data structure for the presets.
; currentVariableMode is an index into this data structure
; when the user adjusts settings.
presetValueArray
unusedPresetByte      .BYTE $00
smoothingDelay        .BYTE $0C
cursorSpeed           .BYTE $02
bufferLength          .BYTE $1F
pulseSpeed            .BYTE $01
indexForColorBarDisplay .BYTE $01
lineWidth              .BYTE $07
sequencerSpeed         .BYTE $04
pulseWidth             .BYTE $01
baseLevel              .BYTE $07
presetColorValuesArray .BYTE BLACK ,BLUE ,RED ,PURPLE ,GREEN ,CYAN ,YELLOW ,
                        WHITE
trackingActivated     .BYTE $FF
lineModeActivated     .BYTE $00
patternIndex          .BYTE $05
```

#### Lines 179-182. presetKeyCodes

```
presetKeyCodes
    .BYTE KEY_LEFT ,KEY_1 ,KEY_2 ,KEY_3 ,KEY_4 ,KEY_5 ,KEY_6 ,KEY_7
    .BYTE KEY_8 ,KEY_9 ,KEY_0 ,KEY_PLUS ,KEY_MINUS ,KEY_POUND
    .BYTE KEY_CLR_HOME ,KEY_INST_DEL
```

#### Lines 1393-1405. CheckIfPresetKeysPressed

```
; -----
; CheckKeyboardInput
; -----
CheckKeyboardInput
    ...
    ; Check if one of the presets has been selected.
CheckIfPresetKeysPressed

    LDX #$00
presetKeyLoop
    CMP presetKeyCodes,X
    BEQ WriteTemplateTextForPreset
    INX
    CPX #$10
    BNE presetKeyLoop

    JMP MaybeWPressed

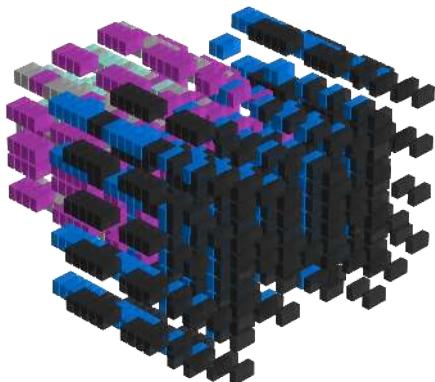
WriteTemplateTextForPreset
    JMP DisplayPresetMessage
```

**Lines 980-994.** `presetValueArray`: Let's start at the end. This is where the values stored in the preset get loaded to, for use by Psychedelia. They are conveniently bunched together in the same order that they are stored in the preset itself. It is possible, as we shall see in '['dials buttons knobs'](#)', to set each of these values individually. We store them together here to make loading and programming resets more convenient for us as the programmer.

**Lines 1393-1405.** `CheckIfPresetKeysPressed`: Yet another entry in the long list of keyboard checks that makes up `CheckKeyboardInput`. This time we're checking if any of the 16 keys associated with a preset has been pressed.

The easiest way to do this is to define an array called `presetKeyCodes` that contains all 16 key codes. Then in our little loop `presetKeyLoop` check each one of them against the key that has actually been pressed. If we get a hit, then it's time to load the preset data by calling `DisplayPresetMessage`.

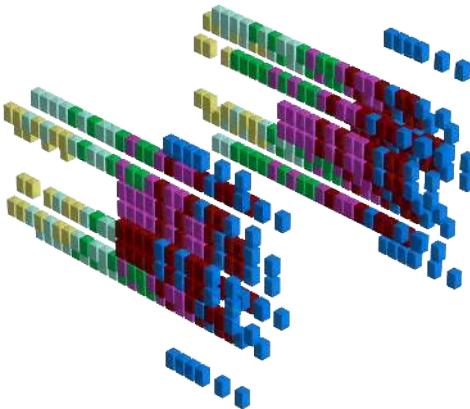
One thing to note here is that we've stored the index associated with the selected preset in `X`. For example, if preset 2 was selected, `X` will contain 2. We'll use this value in `X` in the routines in the following pages.



```
preset5
.BYTE $00 ; unusedPresetByte
.BYTE $0C ; smoothingDelay
.BYTE $02 ; cursorSpeed
.BYTE $2B ; bufferLength
.BYTE $01 ; pulseSpeed
.BYTE $07 ; indexForColorBarDisplay
.BYTE $07 ; lineWidth
.BYTE $0C ; sequencerSpeed
.BYTE $01 ; pulseWidth
.BYTE $07 ; baseLevel
; presetColorValuesArray:
.BYTE BLACK,GRAY1,BLUE,GRAY2,PURPLE,
GRAY3,CYAN,WHITE
.BYTE $00 ; trackingActivated
.BYTE $00 ; lineModeActivated
.BYTE $06 ; presetIndex
.BYTE $06 ; currentPatternElement
.BYTE $03 ; currentSymmetrySetting
```

Listing 9.6: Data structure for Preset 5.

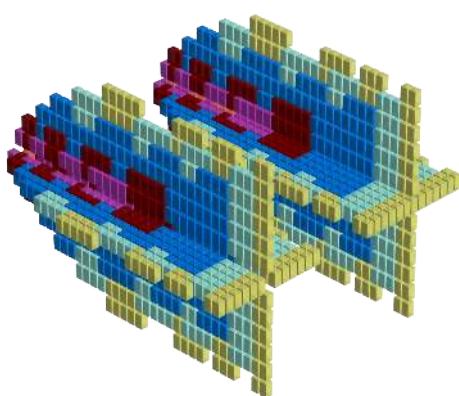
Figure 9.6: Evolution of Preset 5.



```
preset6
.BYTE $00 ; unusedPresetByte
.BYTE $0F ; smoothingDelay
.BYTE $02 ; cursorSpeed
.BYTE $3F ; bufferLength
.BYTE $01 ; pulseSpeed
.BYTE $01 ; indexForColorBarDisplay
.BYTE $07 ; lineWidth
.BYTE $0F ; sequencerSpeed
.BYTE $01 ; pulseWidth
.BYTE $07 ; baseLevel
; presetColorValuesArray:
.BYTE BLACK,BLUE,RED,PURPLE,GREEN,
CYAN,YELLOW,WHITE
.BYTE $FF ; trackingActivated
.BYTE $00 ; lineModeActivated
.BYTE $03 ; presetIndex
.BYTE $03 ; currentPatternElement
.BYTE $04 ; currentSymmetrySetting
```

Listing 9.7: Data structure for Preset 6.

Figure 9.7: Evolution of Preset 6.



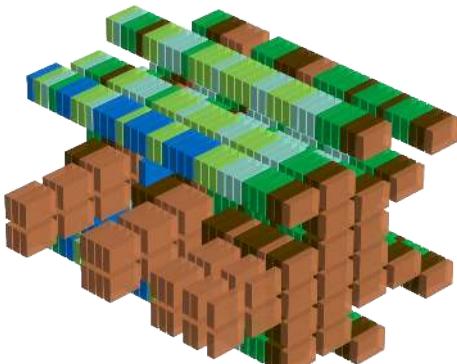
```

preset7
.BYTE $00 ; unusedPresetByte
.BYTE $0B ; smoothingDelay
.BYTE $01 ; cursorSpeed
.BYTE $1C ; bufferLength
.BYTE $02 ; pulseSpeed
.BYTE $0A ; indexForColorBarDisplay
.BYTE $07 ; lineWidth
.BYTE $09 ; sequencerSpeed
.BYTE $01 ; pulseWidth
.BYTE $07 ; baseLevel
; presetColorValuesArray:
.BYTE BLACK,YELLOW,CYAN,LTBLUE,BLUE,
    RED,PURPLE,LTRED
.BYTE $00 ; trackingActivated
.BYTE $00 ; lineModeActivated
.BYTE $07 ; presetIndex
.BYTE $07 ; currentPatternElement
.BYTE $01 ; currentSymmetrySetting

```

Listing 9.8: Data structure for Preset 7.

Figure 9.8: Evolution of Preset 7.



```

preset8
.BYTE $00 ; unusedPresetByte
.BYTE $04 ; smoothingDelay
.BYTE $01 ; cursorSpeed
.BYTE $28 ; bufferLength
.BYTE $02 ; pulseSpeed
.BYTE $01 ; indexForColorBarDisplay
.BYTE $07 ; lineWidth
.BYTE $0A ; sequencerSpeed
.BYTE $01 ; pulseWidth
.BYTE $07 ; baseLevel
; presetColorValuesArray:
.BYTE BLACK,ORANGE,BROWN,GREEN,CYAN,
    LTGREEN,LTBLUE,BLUE
.BYTE $FF ; trackingActivated
.BYTE $00 ; lineModeActivated
.BYTE $01 ; presetIndex
.BYTE $01 ; currentPatternElement
.BYTE $03 ; currentSymmetrySetting

```

Listing 9.9: Data structure for Preset 8.

Figure 9.9: Evolution of Preset 8.

particular presets

---

#### Lines 2030-2070. DisplayPresetMessage

```
; -----
; DisplayPresetMessage
; -----
DisplayPresetMessage
    LDA shiftPressed
    AND #$04
    BEQ WriteTemplateTextForPreset

    JMP EditCustomPattern

WriteTemplateTextForPreset
    TXA
    PHA
    JSR ClearLastLineOfScreen

    LDX #$00
_Loop    LDA txtPreset,X
    STA statusFieldName,X
    INX
    CPX #$10
    BNE _Loop

PrepareToWritePresetValue
    PLA
    PHA
    TAX
    BEQ FinishedUpdatingDisplay

WriteTemplateTextForPresetDigits
    INC presetDigitOne
    LDA presetDigitOne
    CMP # "0"
    BNE GoToNext
    LDA # "0"
    STA presetDigitOne
    INC presetDigitTwo
GoToNext
    DEX
    BNE WriteTemplateTextForPresetDigits

FinishedUpdatingDisplay
    JMP UpdateCurrentActivePreset

WriteLastLineBufferAndReturn
    JSR WriteLastLineBufferToScreen
    RTS
```

**Lines 2030-2070.** `DisplayPresetMessage`: Our objective here is modest enough. Display something like the following on the bottom of the screen to show the player the preset they've selected:



Figure 9.10: Our desired result!

Our data for this is stored as follows:

```
txtPreset
    .TEXT 'PRESET 00      :'
txtPresetActivatedStored
    .TEXT 'ACTIVATED      ,'
    .TEXT 'DATA STORED    ,'
```

`txtPreset` contains the template for displaying the selected value. `txtPresetActivatedStored` contains two templates: one for when we're activating a selected preset and the second for when we're storing currently configured values as a preset.

**Lines 2037-2040.** `WriteTemplateTextForPreset`: We'll split the task in two. First let's get some of the boiler plate text above ('PRESET 00 :) set up. This is what we do in here. (We'll deal with writing 'ACTIVATED' a little later.) With the help of a little loop the contents of `txtPreset` get stored in `statusFieldName`.

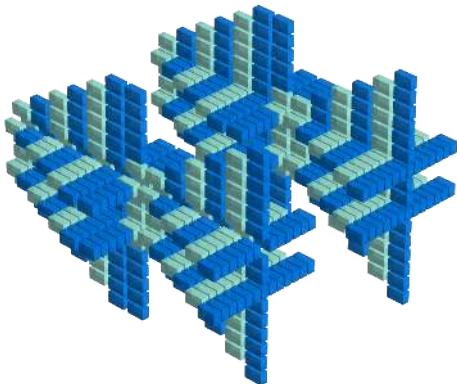
**Lines 2053-2063.** `WriteTemplateTextForPresetDigits`: Now we write the corresponding value for the selected preset. Remember that `X` contains the selected value, so we increment the displayed value in a loop while simultaneously decrementing the value in `X`. Once `X` reaches zero the displayed value will match the selected value.

**Lines 2065-2066.** `FinishedUpdatingDisplay`: When we've finished we'll now have something like this displayed on the screen.



Figure 9.11: Making progress!

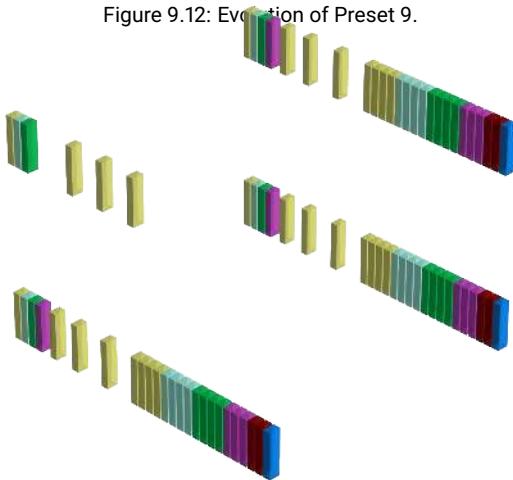
We're ready to fill out the 'ACTIVATED' part and actually load the preset values. So let's `JMP` to `UpdateCurrentActivePreset` and do that!



```
preset9
.BYTE $00 ; unusedPresetByte
.BYTE $11 ; smoothingDelay
.BYTE $01 ; cursorSpeed
.BYTE $0D ; bufferLength
.BYTE $07 ; pulseSpeed
.BYTE $01 ; indexForColorBarDisplay
.BYTE $07 ; lineWidth
.BYTE $0C ; sequencerSpeed
.BYTE $01 ; pulseWidth
.BYTE $07 ; baseLevel
; presetColorValuesArray:
.BYTE BLACK,BLUE,CYAN,BLUE,CYAN,BLUE
,CYAN,BLUE
.BYTE $FF ; trackingActivated
.BYTE $00 ; lineModeActivated
.BYTE $07 ; presetIndex
.BYTE $07 ; currentPatternElement
.BYTE $04 ; currentSymmetrySetting
```

Listing 9.10: Data structure for Preset 9.

Figure 9.12: Evolution of Preset 9.



```
preset10
.BYTE $00 ; unusedPresetByte
.BYTE $01 ; smoothingDelay
.BYTE $02 ; cursorSpeed
.BYTE $1F ; bufferLength
.BYTE $02 ; pulseSpeed
.BYTE $09 ; indexForColorBarDisplay
.BYTE $04 ; lineWidth
.BYTE $08 ; sequencerSpeed
.BYTE $01 ; pulseWidth
.BYTE $07 ; baseLevel
; presetColorValuesArray:
.BYTE BLACK,BLUE,RED,RED,PURPLE,
LTRED,ORANGE,BROWN
.BYTE $FF ; trackingActivated
.BYTE $01 ; lineModeActivated
.BYTE $00 ; presetIndex
.BYTE $00 ; currentPatternElement
.BYTE $04 ; currentSymmetrySetting
```

Listing 9.11: Data structure for Preset 10.

Figure 9.13: Evolution of Preset 10.

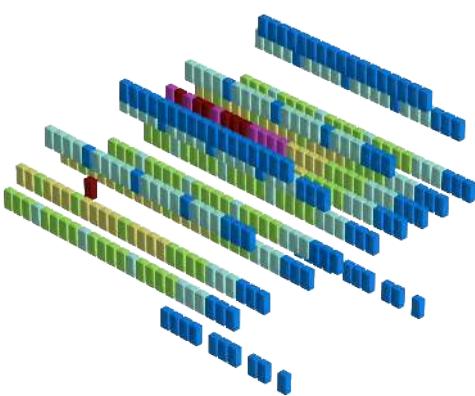


Figure 9.14: Evolution of Preset 11.

```

preset11
.BYTE $00 ; unusedPresetByte
.BYTE $01 ; smoothingDelay
.BYTE $01 ; cursorSpeed
.BYTE $13 ; bufferLength
.BYTE $06 ; pulseSpeed
.BYTE $01 ; indexForColorBarDisplay
.BYTE $07 ; lineWidth
.BYTE $08 ; sequencerSpeed
.BYTE $05 ; pulseWidth
.BYTE $07 ; baseLevel
; presetColorValuesArray:
.BYTE BLACK,BLUE,RED,PURPLE,GREEN,
CYAN,YELLOW,WHITE
.BYTE $FF ; trackingActivated
.BYTE $00 ; lineModeActivated
.BYTE $0F ; presetIndex
.BYTE $0F ; currentPatternElement
.BYTE $04 ; currentSymmetrySetting

```

Listing 9.12: Data structure for Preset 11.

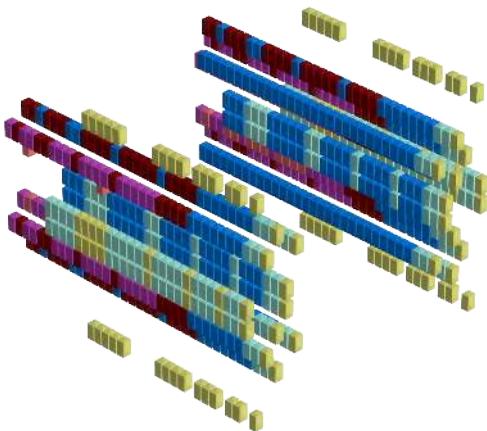


Figure 9.15: Evolution of Preset 12.

```

preset12
.BYTE $00 ; unusedPresetByte
.BYTE $0C ; smoothingDelay
.BYTE $02 ; cursorSpeed
.BYTE $28 ; bufferLength
.BYTE $01 ; pulseSpeed
.BYTE $02 ; indexForColorBarDisplay
.BYTE $07 ; lineWidth
.BYTE $09 ; sequencerSpeed
.BYTE $01 ; pulseWidth
.BYTE $07 ; baseLevel
; presetColorValuesArray:
.BYTE BLACK,BLUE,LBLUE,CYAN,LGREEN
,YELLOW,PURPLE,RED
.BYTE $00 ; trackingActivated
.BYTE $00 ; lineModeActivated
.BYTE $0A ; presetIndex
.BYTE $0A ; currentPatternElement
.BYTE $01 ; currentSymmetrySetting

```

Listing 9.13: Data structure for Preset 12.

particular presets

---

**Lines 2085-2129.** UpdateCurrentActivePreset

```
; -----
; UpdateCurrentActivePreset
; -----
UpdateCurrentActivePreset
    LDA shiftPressed
    AND #$SHIFT_PRESSED
    ASL
    ASL
    ASL
    ASL
    TAY

DisplayActivatedOrStored
    LDX #$00
_Loop    LDA txtPresetActivatedStored,Y
    STA customPatternValueBufferMessage,X
    INY
    INX
    CPX #DISPLAY_LINE_LENGTH
    BNE _Loop

    LDA shiftPressed
    AND #$SHIFT_PRESSED
    BNE StoreCurrentValuesAsPreset

    ; Shift wasn't pressed, so load the selected preset.
    JMP RefreshPresetData

StoreCurrentValuesAsPreset
    PLA
    TAX
    JSR GetPresetPointersUsingXRegister

    LDY #$00
    LDX #$00
_Loop    LDA presetValueArray,X
    STA (presetSequenceDataLoPtr),Y
    INY
    INX
    CPX #$15
    BNE _Loop

    LDA currentPatternElement
    STA (presetSequenceDataLoPtr),Y
    INY
    LDA currentSymmetrySetting
    STA (presetSequenceDataLoPtr),Y
    JMP WriteLastLineBufferAndReturn
```

**Lines 2085-2129.** UpdateCurrentActivePreset: In the current path we're following (the one where we load a selected preset) we only have to concern ourselves with the first half of the routine opposite. This is because the second half is the part that looks after storing currently configured values to a preset, as opposed to loading one.

So the only part that we execute here is DisplayActivatedOrStored which finishes off our status text with the 'ACTIVATED' string. Now we loop through our text in txtPresetActivatedStored:

```
txtPresetActivatedStored
    .TEXT  ' ACTIVATED ',
```

and write it to storage starting at customPatternValueBufferMessage.

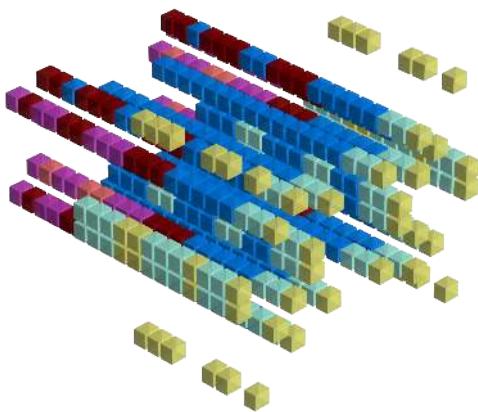
This is the final piece in statusLineBuffer that we need to fill out in order to display our text. We've already filled out the first two parts of this data structure (statusFieldName and presetDigitOne/presetDigitTwo and now have it in the state below:

```
statusLineBuffer
statusFieldName          .TEXT  'PRESET '
presetDigitOne           .TEXT  '0'
presetDigitTwo           .TEXT  '2'
presetDigitThree         .TEXT  ''
customPatternValueBufferPtr .TEXT  ' '
customPatternValueBufferMessage .TEXT  'ACTIVATED ',
```

Later on we will actually write this buffer to the screen by copying all the values in statusLineBuffer together to the last line on the screen:

```
WriteLastLineBufferToScreen
    LDX #NUM_COLS
_Loop
    LDA statusLineBuffer - $01,X
    AND #$3F
    STA SCREEN_RAM + $03BF,X
    LDA #$0C
    STA COLOR_RAM + $03BF,X
    DEX
    BNE _Loop
    RTS
```

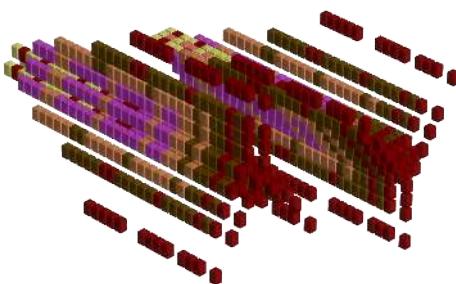
For now though we still have to actually load the preset data. So we JMP to RefreshPresetData after we've finished looping through DisplayActivatedOrStored.



```
preset13
.BYTE $00 ; unusedPresetByte
.BYTE $0B ; smoothingDelay
.BYTE $01 ; cursorSpeed
.BYTE $1C ; bufferLength
.BYTE $02 ; pulseSpeed
.BYTE $0A ; indexForColorBarDisplay
.BYTE $07 ; lineWidth
.BYTE $09 ; sequencerSpeed
.BYTE $01 ; pulseWidth
.BYTE $07 ; baseLevel
; presetColorValuesArray:
.BYTE BLACK,YELLOW,CYAN,LTBLUE,BLUE,
    RED,PURPLE,LTRED
.BYTE $00 ; trackingActivated
.BYTE $00 ; lineModeActivated
.BYTE $03 ; presetIndex
.BYTE $03 ; currentPatternElement
.BYTE $04 ; currentSymmetrySetting
```

Listing 9.14: Data structure for Preset 13.

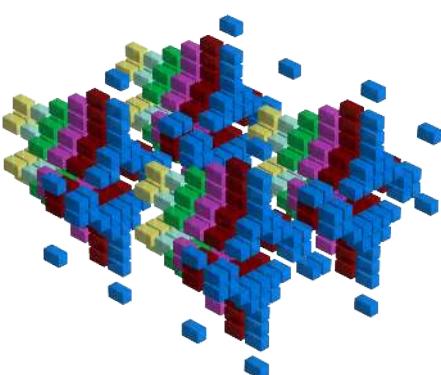
Figure 9.16: Evolution of Preset 13.



```
preset14
.BYTE $00 ; unusedPresetByte
.BYTE $0C ; smoothingDelay
.BYTE $02 ; cursorSpeed
.BYTE $2B ; bufferLength
.BYTE $01 ; pulseSpeed
.BYTE $0A ; indexForColorBarDisplay
.BYTE $07 ; lineWidth
.BYTE $08 ; sequencerSpeed
.BYTE $01 ; pulseWidth
.BYTE $07 ; baseLevel
; presetColorValuesArray:
.BYTE BLACK,RED,BROWN,ORANGE,PURPLE,
    RED,YELLOW,LTRED
.BYTE $FF ; trackingActivated
.BYTE $00 ; lineModeActivated
.BYTE $04 ; presetIndex
.BYTE $04 ; currentPatternElement
.BYTE $02 ; currentSymmetrySetting
```

Listing 9.15: Data structure for Preset 14.

Figure 9.17: Evolution of Preset 14.



```
preset15
.BYTE $00 ; unusedPresetByte
.BYTE $03 ; smoothingDelay
.BYTE $01 ; cursorSpeed
.BYTE $1F ; bufferLength
.BYTE $06 ; pulseSpeed
.BYTE $01 ; indexForColorBarDisplay
.BYTE $07 ; lineWidth
.BYTE $00 ; sequencerSpeed
.BYTE $01 ; pulseWidth
.BYTE $07 ; baseLevel
; presetColorValuesArray:
.BYTE BLACK,BLUE,RED,PURPLE,GREEN,
CYAN,YELLOW,WHITE
.BYTE $FF ; trackingActivated
.BYTE $00 ; lineModeActivated
.BYTE $04 ; presetIndex
.BYTE $04 ; currentPatternElement
.BYTE $04 ; currentSymmetrySetting
```

Listing 9.16: Data structure for Preset 15.

Figure 9.18: Evolution of Preset 15.

particular presets

---

#### Lines 2134-2160. RefreshPresetData

```
RefreshPresetData
    PLA
    TAX
    JSR  GetPresetPointersUsingXRegister

    LDY  #BUFFER_LENGTH
    LDA  (presetSequenceDataLoPtr),Y
    CMP  bufferLength
    BEQ  CheckColorValuesForDifferences

    JSR  ResetCurrentActiveMode
    JMP  LoadSelectedPresetSequence
; Returns

; Check the preset against current data
; and reload if different.
CheckColorValuesForDifferences
    LDX  #$00
    LDY  #$07
_Loop  LDA  (presetSequenceDataLoPtr),Y
    CMP  presetColorValuesArray,X
    BNE  LoadSelectedPresetSequence
    INY
    INX
    CPX  #$08
    BNE  _Loop

    JMP  LoadSelectedPresetSequence
```

**Lines 2134-2160.** RefreshPresetData: Our first order of business here is to get the location of the selected preset in our storage. This is where storing the value of the selected preset in X will come in useful.

We call GetPresetPointersUsingXRegister to get this done. We've encountered this design pattern previously: we use the value in X to iterate from the start of the preset data (given by presetSequenceData and stored in a pair of pointers (presetSequenceDataLoPtr/presetSequenceDataHiPtr. We then iterate from zero until we reach the value of the preset selected by the player, incrementing presetSequenceDataLoPtr by 32 bytes each time. Once we reach the value selected, our two pointers will contain the address of the selected preset data. In the case of preset '2' this means we will start at \$C000 and end up at \$C040.

```

GetPresetPointersUsingXRegister
    LDA #>presetSequenceData
    STA presetSequenceDataHiPtr
    LDA #<presetSequenceData
    STA presetSequenceDataLoPtr
    TXA
    BEQ ReturnFromPointers

_Loop   LDA presetSequenceDataLoPtr
        CLC
        ADC #$20
        STA presetSequenceDataLoPtr
        LDA presetSequenceDataHiPtr
        ADC #$00
        STA presetSequenceDataHiPtr
        DEX
        BNE _Loop

ReturnFromPointers
RTS

```

For some reason not clear to me, the rest of this routine goes to a bit of trouble in checking the data we're about to load and whether it's the same as the preset data we already have loaded. All of these checks end in the same outcome: we call LoadSelectedPresetSequence to actually load the data anyway. So, to me at least, it looks like there was the intention to optimize in some way here but it was abandoned. Despite all of the footling around we just go ahead and load the preset data.

particular presets

---

#### Lines 2165-2183. LoadSelectedPresetSequence

```
LoadSelectedPresetSequence
    LDA #$FF
    STA currentModeActive

    ; Copy the value from the preset sequence into
    ; current storage.
    LDY #0
_Loop    LDA (presetSequenceDataLoPtr),Y
    STA presetValueArray,Y
    INY
    CPY #$15
    BNE _Loop

    LDA (presetSequenceDataLoPtr),Y
    STA currentPatternElement

    INY
    LDA (presetSequenceDataLoPtr),Y
    STA currentSymmetrySetting

    JMP WriteLastLineBufferAndReturn
```

**Lines 2165-2183.** LoadSelectedPresetSequence: After all this trouble we go ahead and load the data using our pointers into presetValueArray. As we saw at the start of the chapter the fact that our preset values are stored contiguously means that a simple loop will do the bulk of the job. However there are two trailing items the currentPatternElement and currentSymmetrySetting that we need to pick out by name.

When we're done we finally write the status line to the screen and return. That completes loading of the preset data. Now we can take a look at some of these individual settings more closely.



# dials buttons switches

## Jeffrey Says



Gradually the idea solidified into the concept of a light-show generator, something interactive, creative but simple enough so that anyone could do it, yet complex enough to produce breathtaking results once learned well. A program to do for light, in fact, what a synthesiser does for sound.

Dials, buttons, switches, and knobs that we can twiddle and adjust are at the heart of the Psychedelia player's experience. Fortunately they are mostly packaged up into presets and bursts, as we have already seen. However it is possible, if you are prepared to clutch the game's manual to your chest at night in bed, to configure many individual settings in all sorts of different ways.

In this penultimate chapter we'll take a look at the more interesting of these settings in some detail, develop a better idea of how they are supposed to work, and figure out how they were incorporated into the core routines we covered in earlier chapters.

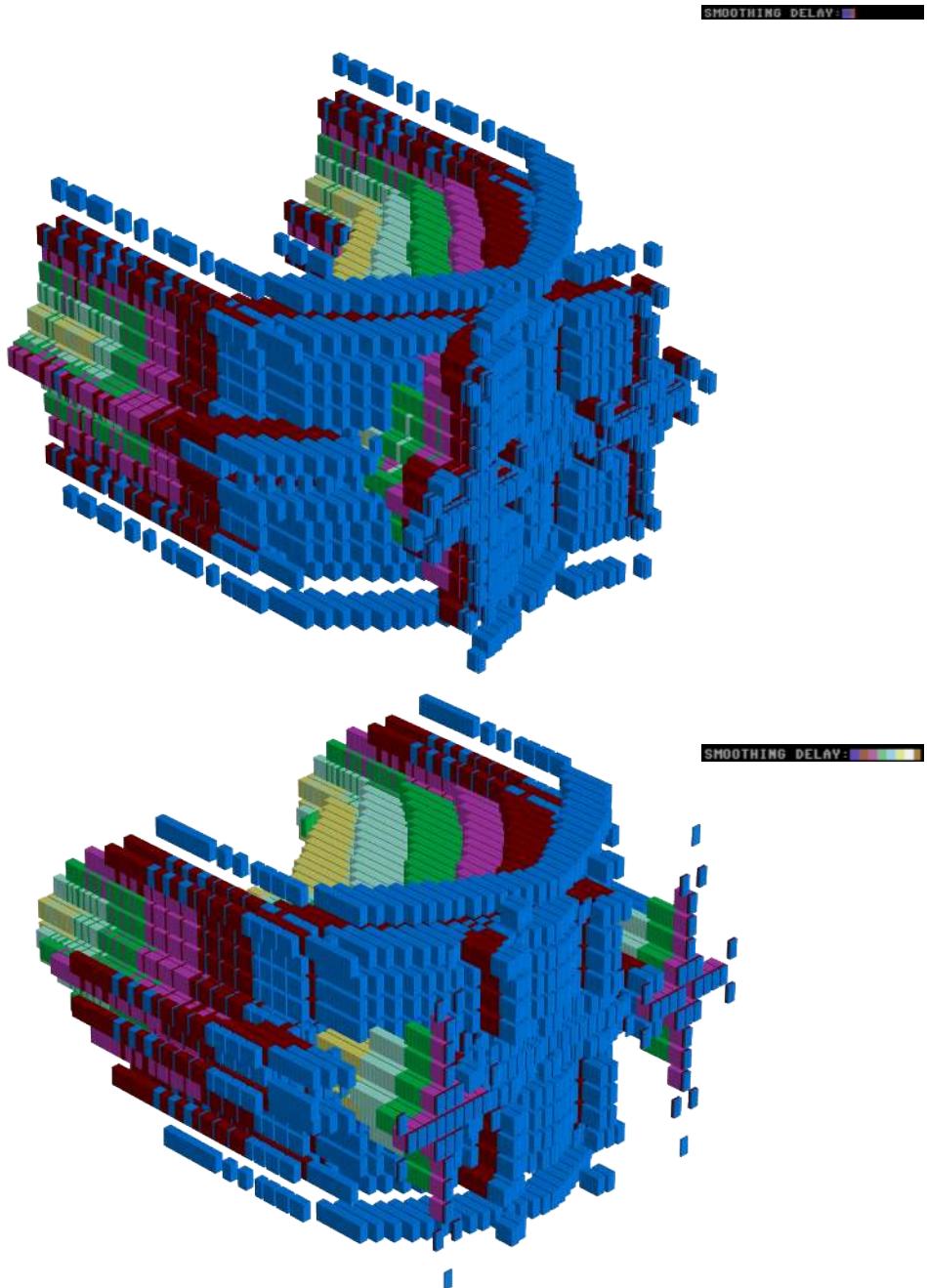


Figure 10.1: Effect of low and high values for Smoothing Delay

## smoothing delay

Let's try to understand 'smoothing delay' better. Let's start by trying to understand what it is. Perhaps Jeff Minter can shed some light:

**Jeffrey Says**



**D to activate:** Because of the time taken to draw larger patterns speed increase/decrease is not linear. You can adjust the 'compensating delay' which often smooths out jerky patterns. Can be used just for special FX, though. Suck it and see.

So that's clear then. Smoothing delay is what you want to use when you need to smooth out the delay. What more explanation could we need?

If we look at the effect of setting this parameter to its lowest and highest possible value opposite, the difference in behaviour is not extreme. It's almost quite subtle. With a higher setting the pattern evolves in a more drawn out fashion - as though a higher value lets Psychedelia linger a little longer on the current state of the pattern rather than immediately evolving it to the next state.

Let's dig into the code and see if this intuition is correct.

First of all though we'll take a look at the code that allows us to twiddle this:

**SMOOTHING DELAY:**

to this:

**SMOOTHING DELAY:**

**Lines 1297-1303.** MaybeDPressed

```
MaybeDPressed
    CMP #KEY_D ; 'D' pressed?
    BNE MaybeCPressed

DPressed
    LDA #$01
    STA currentVariableMode
    RTS
```

Listing 10.1: From CheckKeyboardInput.

**Lines 1838-1868.** UpdateVariableDisplay

```
UpdateVariableDisplay
    ...

    LDX currentVariableMode
    LDA lastKeyPressed
    CMP #KEY_GT ; > pressed?
    BNE MaybeLeftArrowPressed

    RightArrowPressed
        ; > pressed, increase the value bar.
        INC presetValueArray,X
        LDA presetValueArray,X
        ; Make sure we don't exceed the max value.
        CMP maxValueForPresetValueArray,X
        BNE MaybeInColorMode
        DEC presetValueArray,X
        JMP MaybeInColorMode

    MaybeLeftArrowPressed
        CMP #KEY_LT ; < pressed?
        BNE MaybeInColorMode

        ; < pressed, decrease the value bar.
        DEC presetValueArray,X
        LDA presetValueArray,X
        ; Make sure we don't exceed the min value.
        CMP minValueForPresetValueArray,X
        BNE MaybeInColorMode
        INC presetValueArray,X
```

Listing 10.2: From CheckKeyboardInputForActiveVariable.

**Lines 1297-1303.** MaybeDPressed: When D is pressed we don't update a setting there and then as you might expect. Instead we get ready to display the 'Smoothing Delay' control bar, by... loading the value \$01 to currentVariableMode? Okay, we'll go with that for now.

**Lines 1838-1868.** UpdateVariableDisplay: The next time we loop around to CheckKeyboardInput, we hit this little piece of logic at the very top of it:

```
CheckKeyboardInput
    LDA currentVariableMode
    BEQ CheckForGeneralKeystrokes
    JMP CheckKeyboardInputForActiveVariable
```

Well, we just loaded \$01 to currentVariableMode above so it's not zero. It follows that the BEQ check will give a negative result (the check means 'is the value in A equal to zero?'), so instead of forking to CheckForGeneralKeystrokes we'll JMP to CheckKeyboardInputForActiveVariable.

This is where the function we're looking at here lives. As we can see the first thing it does is load currentVariableMode to the X register. This is because we're going to use it as an index into an array we encountered in the previous chapter on Presets. This is the array presetValueArray which contains a lot of the settings we'll be looking at in this chapter huddled together like a gaggle of ducklings, with smoothingDelay near the head at index 1 (index 0 being taken by unusedPresetByte):

```
presetValueArray
unusedPresetByte      .BYTE $00
smoothingDelay       .BYTE $0C
cursorSpeed          .BYTE $02
bufferLength         .BYTE $1F
pulseSpeed           .BYTE $01
...
```

With X set to 1 we can now use it increment the value for smoothingDelay by simply executing:

```
INC presetValueArray,X
```

And decrement it by doing:

```
DEC presetValueArray,X
```

This is handy, and worth remembering as the technique will be reused for adjusting other values that we look at that also live in presetValueArray.

**Lines 923-928.** ApplySmoothingDelay

```
ApplySmoothingDelay
    LDA smoothingDelay
    STA initialSmoothingDelayForStep,X
    STA smoothingDelayForStep,X
```

Listing 10.3: From MainInterruptHandler.

**Lines 644-679.** ShouldDoAPaint

```
ShouldDoAPaint
    ...
    DEC smoothingDelayForStep,X
    BNE GoBackToStartOfLoop

    ; Actually paint some pixels to the screen.
    ; Reset the delay for this step.
    LDA initialSmoothingDelayForStep,X
    STA smoothingDelayForStep,X

    ; Get the x and y positions for this pixel.
    LDA pixelXPositionArray,X
    STA pixelXPositionZP
    LDA pixelYPositionArray,X
    STA pixelYPositionZP

    LDA patternIndexArray,X
    STA presetIndex

    LDA symmetrySettingForStepCount,X
    STA currentSymmetrySettingForStep

    LDA currentIndexToColorValues
    AND #$80
    BNE ResetAndGoBackToStartOfLoop

    TXA
    PHA
    JSR PaintStructureAtCurrentPosition
    PLA
    TAX
```

Listing 10.4: From MainPaintLoop.

**Lines 923-928.** `ApplySmoothingDelay`: With the player's selected value loaded safely to `smoothingDelay` we now have to see how it is applied to the evolution of the patterns.

The immediate answer to this is given here: it is loaded to the current position in the `smoothingDelayForStep` and `initialSmoothingDelayForStep` arrays. Remember that these arrays, along with several others, help track the different phases in the evolution of the pattern.

**Lines 644-679.** `ShouldDoAPaint`: Back to the main paint loop, where the pixel painting is actually done. We looked at this loop in detail in earlier chapters. And it is in here that we see the 'smoothing delay' get applied. At the top of function we've extracted here the value in `smoothingDelayForStep` is used as a countdown counter. This is what our smoothing delay is: a counter that we decrement at each pass. If it's not zero yet (`BNE`), we `GoBackToStartOfLoop`.

If it is zero we can actually paint a pixel. We use the value in `initialSmoothingDelayForStep` to reset our counter and get on with doing a full paint.

So after all that we can see that 'smoothing delay' acts as a brake on the painting loop - the greater the value the longer it will defer the next paint, resulting in a slower and smoother evolution of the pattern being painted.

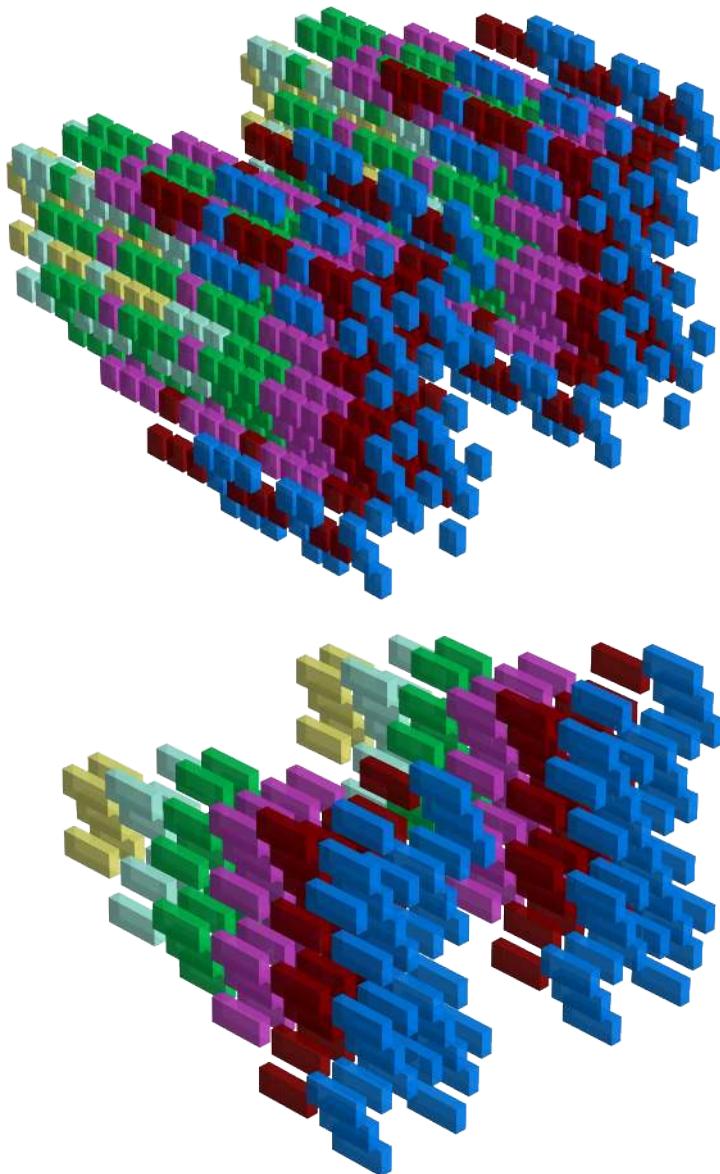


Figure 10.2: Tracking 'on' (top) and 'off' (bottom).

## tentative tracking

### Jeffrey Says



**T to Activate:** Controls whether logic-seeking is used in the buffer or not. The upshot of this for you is a slightly different feel - continuous but fragmented when ON, or together-ish bursts when OFF. Try it.

Tracking is 'on' by default and nearly all off the figures and diagrams we've used to illustrate Psychedelia's behaviour have demonstrated the behaviour of 'tracking'. In essence, tracking means using the values previously painted to a pixel position on screen as partial feedback into the next color value that we will paint into it.

When tracking is enabled the colors of pixels will change much more often, whereas when it is disabled the display becomes much less interesting, with a more consistent and smoother evolution in the colors of the patterns as they evolve.

**Lines 1362-1390.** MaybeTPressed

```
; -----
; CheckKeyboardInput
; -----
CheckKeyboardInput
...
MaybeTPressed
    CMP #KEY_T ; T pressed.
    BNE CheckIfPresetKeysPressed

    ; Toggle tracking on or off.
    LDA trackingActivated
    EOR #$FF
    STA trackingActivated

    ; Use the new setting to get an offset into our screen message.
    ; in txtTrackingOnOff
    AND #$01
    ASL
    ASL
    ASL
    ASL
    ASL
    TAY

    JSR ClearLastLineOfScreen

    ; Display the updated setting of 'tracking'.
    LDX #$00
txtTrackingLoop
    LDA txtTrackingOnOff,Y
    STA lastLineBufferPtr,X
    INY
    INX
    CPX #$10
    BNE txtTrackingLoop

    JMP WriteLastLineBufferToScreen
    RTS
```

**Lines 1362-1390.** MaybeTPressed: Pressing T toggles tracking on or off. Tracking's on/off state is stored in trackingActivated, with \$00 meaning 'off' and \$FF meaning 'on'. This means we can use a neat, economical trick for toggling the value every time the user presses the 'T' key:

```
; Toggle tracking on or off.
LDA trackingActivated
EOR #$FF
STA trackingActivated
```

The EOR statement performs an exclusive-or that has the neat property of turning \$00 into \$FF and \$FF into \$00 - equivalent to switching a value on or off.

This is what the bit by bit operation looks like when 'EOR \$FF' turns trackingActivated 'on':

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
\$00	0	0	0	0	0	0	0	0
\$FF	1	1	1	1	1	1	1	1
Result	1	1	1	1	1	1	1	1

X-OR'ing \$FF and \$00 gives \$FF, the 'on' value for trackingActivated.

And this is what it looks like when EOR \$FF turns trackingActivated 'off' again:

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
\$FF	1	1	1	1	1	1	1	1
\$FF	1	1	1	1	1	1	1	1
Result	0	0	0	0	0	0	0	0

X-OR'ing \$FF and \$FF gives \$00, the 'off' value for trackingActivated.

**Lines 728-943. MainInterruptHandler**

```
; -----
; MainInterruptHandler
; -----
MainInterruptHandler
    ...
    ; Finally, update the pixel buffers with a byte
    ; each for the current pattern.
UpdatePixelBuffersForPattern
    INC currentStepCount
    LDA currentStepCount
    CMP bufferLength
    BNE UpdateBaseLevelArray

    LDA #$00
    STA currentStepCount

UpdateBaseLevelArray
    TAX
    LDA currentColorIndexArray,X
    CMP #$FF
    BEQ UpdatePositionArrays

CheckIfTrackingActivated
    LDA previousIndexToPixelBuffers
    AND trackingActivated
    BEQ DrawCursorAndReturnFromInterrupt

    TAX
    LDA currentColorIndexArray,X
    CMP #$FF
    BNE DrawCursorAndReturnFromInterrupt

    STX currentStepCount

UpdatePositionArrays
    LDA cursorXPosition
    STA pixelXPositionArray,X
    LDA cursorYPosition
    STA pixelYPositionArray,X

    ...
DrawCursorAndReturnFromInterrupt
    LDA #$01
    STA currentColorToPaint
    JSR PaintCursorAtCursorPosition
    ; Falls through
```

**Lines 728-943.** MainInterruptHandler: As you may remember the MainInterruptHandler runs every 1/60th of a second. You may also recall its main job is to fill the pixel buffers (e.g. pixelXPositionArray, pixelYPositionArray and so on) so that the MainPaintLoop can use them to paint the screen.

**Lines 890-896.** CheckIfTrackingActivated: trackingActivated comes into play here. The idea is that if 'tracking' is enabled then we should do a re-paint of the last entry in the buffers that was processed by MainPaintLoop. Naturally, if a pixel is being painted twice over this will create a 'tracking' effect, as though the pattern is trailing a glowing tail after it.

We use the value in trackingActivated to decide whether or not to paint the previously processed position in the buffers by simply AND'ing the two together.

```
LDA previousIndexToPixelBuffers  
AND trackingActivated  
BEQ DrawCursorAndReturnFromInterrupt
```

If it's zero, we bail out completely until the next interrupt by jumping to DrawCursorAndReturnFromInterrupt. But if we get a non-zero value that means tracking is enabled and we should go ahead and refresh the values in the buffers at the index given by the value in previousIndexToPixelBuffers. So we 'fall through' instead to UpdatePositionArrays and a number of other sub-routines that update the values of the previous position in our pixel buffers.

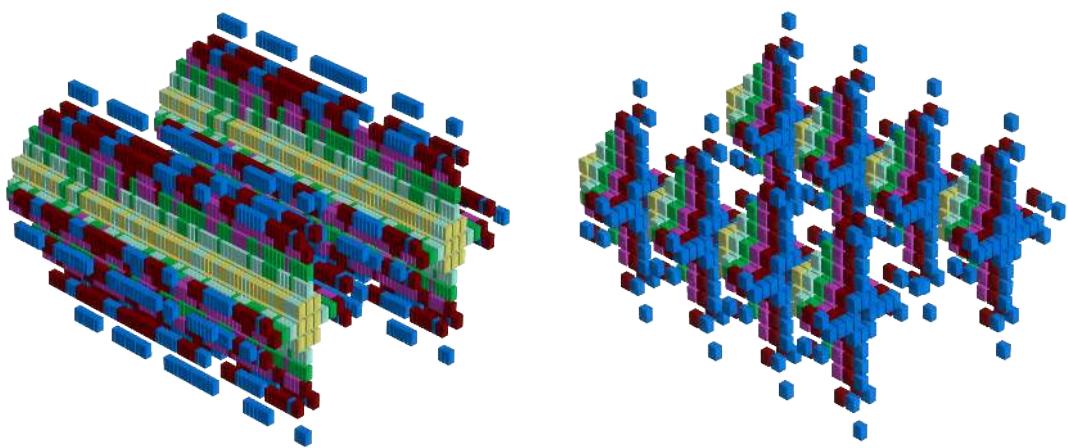


Figure 10.3: Effect of low and high values for Pulse Speed

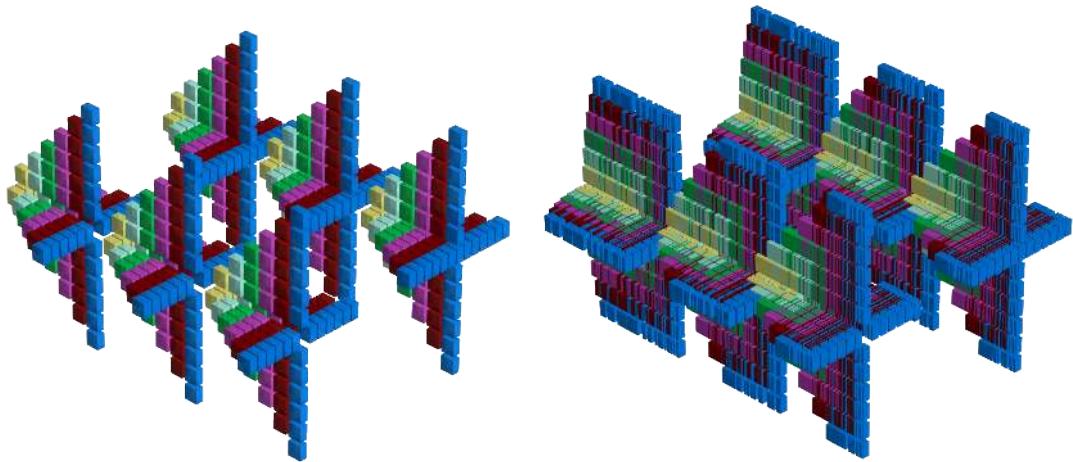


Figure 10.4: Effect of low and high values for Pulse Width

## pulse speed

### Jeffrey Says



**P to activate:** Usually if you hold down the button you get a continuous stream. Setting the Pulse Speed allows you to generate a pulsed stream, as if you were rapidly pressing and releasing the FIRE button.

Pulsing is self explanatory enough, especially when you compare the illustrations opposite. In a way, pulse width behaves a lot like smoothing delay.

## pulse width

### Jeffrey Says



**O to activate:** Sets the length of the pulses in a pulsed stream output. Don't worry about what that means - just get in there and mess with it.

Pulse speed and pulse width are managed together in the code as we shall see.

**Lines 1334-1342.** MaybePPressed

```
MaybePPressed
    CMP #KEY_P ; P pressed
    BNE MaybeHPressed

    ; P pressed.
    LDA #$04
    STA currentVariableMode
    RTS
```

Listing 10.5: From CheckKeyboardInput.

**Lines 1838-1868.** UpdateVariableDisplay

```
decrements the value in presetValueArray pointed to by \icode{X}\, i.e.
\icode{currentVariableMode}.]
UpdateVariableDisplay
    ...

    LDX currentVariableMode
    LDA lastKeyPressed
    CMP #KEY_GT ; > pressed?
    BNE MaybeLeftArrowPressed

    RightArrowPressed
        ; > pressed, increase the value bar.
        INC presetValueArray,X
        LDA presetValueArray,X
        ; Make sure we don't exceed the max value.
        CMP maxValueForPresetValueArray,X
        BNE MaybeInColorMode
        DEC presetValueArray,X
        JMP MaybeInColorMode

    MaybeLeftArrowPressed
        CMP #KEY_LT ; < pressed?
        BNE MaybeInColorMode

        ; < pressed, decrease the value bar.
        DEC presetValueArray,X
        LDA presetValueArray,X
        ; Make sure we don't exceed the min value.
        CMP minValueForPresetValueArray,X
        BNE MaybeInColorMode
        INC presetValueArray,X
```

**Lines 1334-1342.** MaybePPressed: When P is pressed we don't update a setting there and then as you might expect. Instead we get ready to display the 'Smoothing Delay' control bar, by... loading the value \$04 to currentVariableMode? Okay, we'll go with that for now.

**Lines 1838-1868.** UpdateVariableDisplay: The next time we loop around to CheckKeyboardInput, we hit this little piece of logic at the very top of it:

```
CheckKeyboardInput
    LDA currentVariableMode
    BEQ CheckForGeneralKeystrokes
    JMP CheckKeyboardInputForActiveVariable
```

Well, we just loaded \$01 to currentVariableMode above so it's not zero. It follows that the BEQ check will give a negative result (the check means 'is the value in A equal to zero?'), so instead of forking to CheckForGeneralKeystrokes we'll JMP to CheckKeyboardInputForActiveVariable.

This is where the function we're looking at here lives. As we can see the first thing it does is load currentVariableMode to the X register. This is because we're going to use it as an index into an array we encountered in the previous chapter on Presets. This is the array presetValueArray which contains a lot of the settings we'll be looking at in this chapter huddled together like a gaggle of ducklings, with pulseSpeed near at index 4 (index 0 being taken by unusedPresetByte):

```
presetValueArray
unusedPresetByte      .BYTE $00
smoothingDelay       .BYTE $0C
cursorSpeed          .BYTE $02
bufferLength         .BYTE $1F
pulseSpeed           .BYTE $01 ; <-- Index $04
...
```

With X set to 4 we can now use it increment the value for pulseSpeed by simply executing:

```
INC presetValueArray,X
```

And decrement it by doing:

```
DEC presetValueArray,X
```

This is handy, and worth remembering as the technique will be reused for adjusting other values that we look at that also live in presetValueArray.

**Lines 728-943. MainInterruptHandler:**

```
; -----
; MainInterruptHandler
; -----
MainInterruptHandler
    ...
    ; Player has pressed fire.
PlayerHasPressedFire
    LDA stepsExceeded255
    BEQ DecrementPulseWidthCounter
    LDA stepsSincePressedFire
    BEQ IncrementStepsSincePressedFire
    JMP DrawCursorAndReturnFromInterrupt

IncrementStepsSincePressedFire
    INC stepsSincePressedFire

DecrementPulseWidthCounter
    LDA currentPulseWidth
    BEQ DecrementPulseSpeedCounter
    DEC currentPulseWidth
    BEQ DecrementPulseSpeedCounter
    JMP UpdatePixelBuffersForPattern

DecrementPulseSpeedCounter
    DEC currentPulseSpeedCounter
    BEQ RefreshPulseSpeedAndWidth
    JMP DrawCursorAndReturnFromInterrupt

RefreshPulseSpeedAndWidth
    LDA pulseSpeed
    STA currentPulseSpeedCounter
    LDA pulseWidth
    STA currentPulseWidth

    ; Finally, update the pixel buffers with a byte
    ; each for the current pattern.
UpdatePixelBuffersForPattern
    INC currentStepCount
    LDA currentStepCount
    CMP bufferLength
    BNE UpdateBaseLevelArray

    LDA #$00
    STA currentStepCount
```

Listing 10.6: From MainInterruptHandler.

**Lines 728-943.** MainInterruptHandler As you may remember the MainInterruptHandler runs every 1/60th of a second. You may also recall its main job is to fill the pixel buffers (e.g. pixelXPositionArray, pixelYPositionArray and so on) so that the MainPaintLoop can use them to paint the screen.

**Lines 856-861.** DecrementPulseWidthCounter: Pulse speed and width both act as counters, each defining an interval. Pulse speed defines an interval that controls how often the buffers are refreshed, while pulse width effectively acts as a multiplier for pulse speed.

We can see this when we observe that we decrement pulse width first and it is only when `currentPulseWidth` reaches zero that we consider decrementing pulse speed.

Meanwhile when we do decrement pulse speed (effectively every pulse width reaches zero) we will always bail out without updating our pixel buffers until `currentPulseSpeedCounter` reaches zero.

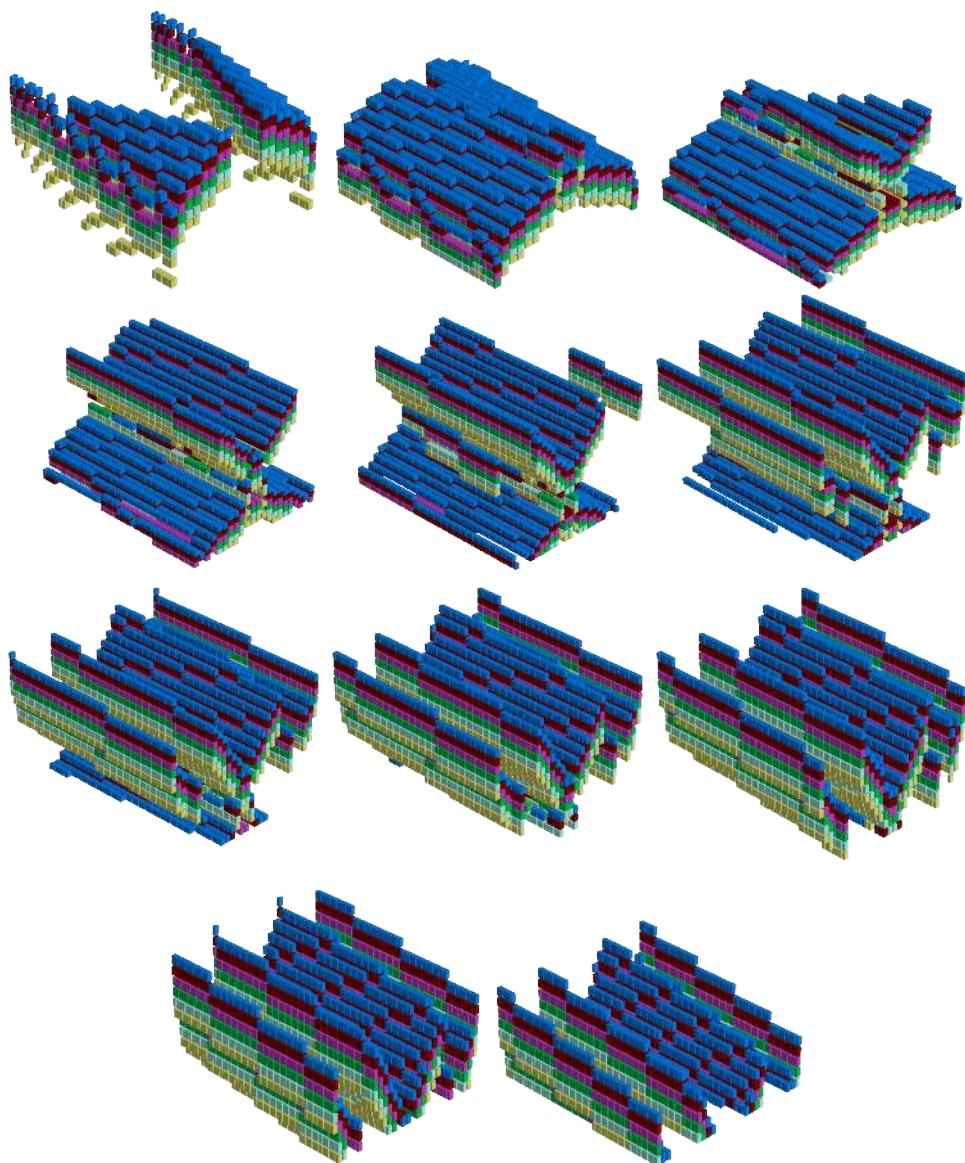


Figure 10.5: Line Mode with Pulse Width at Maximum. These pleasing patterns were created by moving the cursor around a bit during painting.

## line mode

Jeffrey Says



**Press L to turn on and off the Line Mode** - a bit like drawing with the Aurora Borealis.

**Press W to adjust line width:** Sets the width of the lines produced in Line Mode.

'Line Mode' is a completely different mode of painting. It involves no use of patterns at all. In fact it is so completely true to its name that it consists simply of a line of vertical pixels that shoot down the screen from underneath the cursor. The length of the line is determined by 'Line Width', with a maximum 'width' of 7 pixels high and a minimum of 1 pixel high.



Figure 10.6: Line Mode, a line of pixels that shoot down the screen, shown at the 7 different possible settings of 'Line Width'. Here shown with 'X-Symmetry' so that there are two lines instead of one.

Since it is a completely free-standing painting mode it makes no use of the painting algorithm that has dominated our discussion of Psychedelia until now. Let's take a look at how it worked.

**Lines 1275-1294.** JustLPressed

```
JustLPressed
    ; 'L' pressed. Turn line mode on or off.
    LDA lineModeActivated
    EOR #$01
    STA lineModeActivated
    ASL
    ASL
    ASL
    ASL
    TAY

    ; Briefly display the new linemode on the bottom of the screen.
    JSR ClearLastLineOfScreen
    LDX #$00
_Loop    LDA lineModeSettingDescriptions,Y
    STA lastLineBufferPtr,X
    INY
    INX
    CPX #$10
    BNE _Loop
    JMP WriteLastLineBufferToScreen
    ; Returns
```

Listing 10.7: From CheckKeyboardInput.

**Lines 1275-1294.** JustLPressed: Pressing L toggles line mode on or off. Line mode's on/off state is stored in lineModeActivated, with \$00 meaning 'off' and \$01 meaning 'on'. This means we can use a neat, economical trick for toggling the value every time the user presses the 'L' key:

```
LDA lineModeActivated
EOR #$01
STA lineModeActivated
```

The EOR statement performs an exclusive-or that has the neat property of turning \$00 into \$01 and \$01 into \$00 - equivalent to switching a value on or off.

This is what the bit by bit operation looks like when 'EOR \$01' turns trackingActivated 'on':

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
\$00	0	0	0	0	0	0	0	0
\$01	0	0	0	0	0	0	0	1
Result	0	0	0	0	0	0	0	1

X-OR'ing \$01 and \$00 gives \$01, the 'on' value for lineModeActivated.

And this is what it looks like when EOR \$01 turns lineModeActivated 'off' again:

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
\$01	0	0	0	0	0	0	0	1
\$01	0	0	0	0	0	0	0	1
Result	0	0	0	0	0	0	0	0

X-OR'ing \$01 and \$01 gives \$00, the 'off' value for lineModeActivated.

**Lines 906-914.** ApplyLineMode

```
; -----
; MainInterruptHandler
; -----
MainInterruptHandler
    ...
ApplyLineMode
    LDA lineModeActivated
    BEQ LineModeNotActive

        ; Line Mode Active
        LDA #NUM_ROWS + 1
        SEC
        SBC cursorYPosition
        ORA #$80
        STA currentColorIndexArray,X
```

Listing 10.8: From MainInterruptHandler.

**Lines 575-683.** MainPaintLoop

```
; -----
; MainPaintLoop
; -----
MainPaintLoop
    ...
        ; Line Mode sets the top bit of currentValueInColorIndexArray
        LDA currentValueInColorIndexArray
        AND #$80      ; #LINE_MODE_ACTIVE
        BNE PaintLineModeAndLoop
    ...

PaintLineModeAndLoop
    ; Loops back to MainPaintLoop
    JMP PaintLineMode
```

Listing 10.9: From MainPaintLoop.

**Lines 906-914.** `ApplyLineMode`: Here `lineModeActivated` is used to tag one of the pixel buffers to indicate that line mode, rather than the usual painting routine in `PaintStructureAtCursorPosition` should be used for painting. This 'tagging' is done by setting the leftmost bit of the byte stored to the current position in the `currentColorIndexArray`. We do this by 'OR'ing the manipulated Y position of the cursor with \$80 before we store it:

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
\$15	0	0	0	1	0	1	0	1
\$80	1	0	0	0	0	0	0	0
Result	1	0	0	1	0	1	0	1

OR'ing a `cursorYPosition` of \$15 and \$80 gives \$95, a value of \$15 with the leftmost bit set.

This is not the only strange thing happening here, it also appears we storing a Y coordinate in the color index array, an array we typically use for storing color values. Specifically we're taking the 'reflected' vertical position of the cursor and storing it there. Clearly line mode is a different beast from our normal painting routine. We will see what lies behind this a little later.

**Lines 575-683.** `MainPaintLoop`: For now, in the main paint loop, when it comes time to do some painting, we use the 'tag' we set above to detect if we should paint in 'Line Mode'. We do this by AND'ing the value we loaded from `currentColorIndexArray` with \$80 to detect if it has been tagged:

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
\$95	1	0	0	1	0	1	0	1
\$80	1	0	0	0	0	0	0	0
Result	1	0	0	0	0	0	0	0

AND'ing a `currentValueInColorIndexArray` of \$95 and \$80 gives \$80 - this non-zero value tells us Line Mode is active.

If line mode is active, we JMP to `PaintLineMode` which we cover next.

**Lines 1644-1665.** PaintLineMode

```
PaintLineMode
    LDA currentValueInColorIndexArray
    AND #$7F
    STA offsetForYPos

    LDA #NUM_ROWS + 1
    SEC
    SBC offsetForYPos
    STA pixelYPosition

    DEC pixelYPosition

    LDA #$00
    STA currentValueInColorIndexArray
    LDA #ACTIVE
    STA forcePaintPixel

    JSR PaintPixelForCurrentSymmetry

    INC pixelYPosition

    LDA #NOT_ACTIVE
    STA forcePaintPixel

    LDA lineWidth
    EOR #$07
    STA currentValueInColorIndexArray
```

Listing 10.10: From PaintLineMode.

**Lines 1644-1665.** PaintLineMode: 'Line Mode' painting, as we've already mentioned, consists of painting a line of pixels that appear to drop vertically from the cursor. The figure below shows the sequence of paints involved in animating line mode with 'Line Width' set to maximum. We begin by adding pixels below the cursor and once we've added the seven pixels allowed by the line 'width' (which is really more of a 'height' as you can see), we begin moving them down the screen until they reach the edge.

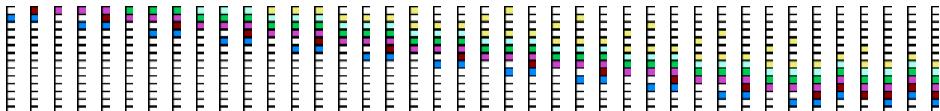


Figure 10.7: Animating the evolution of line width, from left to right. Each line above is rendered by a single visit to PaintLineMode. So rendering the entering sequence requires forty or so visits to this routine.

When we look at the routine PaintLineMode we find it breaks the rendering of a single frame of this animation task into two parts. The first part, shown opposite, is concerned with setting up the initial Y co-ordinate that we will paint from. This takes up the majority of the code in the routine as a whole even though it achieves the least of the outcome. The second, and most productive part, occurs afterwards in LineModeLoop which we cover on the next page.

Notice there's only a single paint performed opposite JSR PaintPixelForCurrentSymmetry. Prior to calling it we figure out the pixelYPosition to draw from and set the color we want to paint as \$00, i.e. black. Notice that we use forcePaintPixel to force the PaintPixel routine to just go ahead and paint the color we've provided - we're not interested in it figuring out the color by itself as it usually does.

With that done we set ourselves up to paint the current frame of the descending line. To avoid a monotonous use of colors we make our color value index depend on our current setting of line width. By making this initial color value a product of an exclusive-or of line width:

```
LDA lineWidth
EOR #$07
STA currentValueInColorIndexArray
```

We can introduce some variety to our palette:



Figure 10.8: The color palette produced X-OR'ing a line width of \$04.

**Lines 1665-1693.** LineModeLoop

```
LineModeLoop
    JSR PaintPixelForCurrentSymmetry

    INC pixelYPosition
    INC currentValueInColorIndexArray

    LDA currentValueInColorIndexArray
    CMP #$08
    BNE IncrementYPositionAndLoop

    JMP CleanUpAndExitLineModePaint

; This line is never reached because the JMP above
; will always exit the routine completely.
    INC currentValueInColorIndexArray

IncrementYPositionAndLoop
    STA currentValueInColorIndexArray
    LDA pixelYPosition
    CMP #NUM_ROWS + 1
    BNE LineModeLoop

CleanUpAndExitLineModePaint
    LDX currentIndexToPixelBuffers
    DEC currentColorIndexArray,X
    LDA currentColorIndexArray,X
    CMP #$80
    BEQ ResetIndexAndExitLineModePaint
    JMP MainPaintLoop

ResetIndexAndExitLineModePaint
    LDA #$FF
    STA currentColorIndexArray,X
    STX previousIndexToPixelBuffers
    JMP MainPaintLoop
```

Listing 10.11: From PaintLineMode.

**Lines 1665-1693.** LineModeLoop: Everything above CleanUpAndExitLineModePaint on the page opposite is what achieves the effect of a line of pixels dropping down the screen. It concerns itself entirely with painting the a pixel at the current position to screen..

```
JSR PaintPixelForCurrentSymmetry
```

.. then incrementing our Y co-ordinate and color value index..

```
INC pixelYPosition  
INC currentValueInColorIndexArray
```

.. and finally, checking if we've reached the maximum value possible for the color index value.

If we have, then we have run out of colors to paint so we jump to CleanUpAndExitLineModePaint and exit the routine.

Otherwise, we go on to check if we've reached the edge of the screen, if we have not, and there's still more room to paint pixels on, we can do another pass of the loop:

```
LDA pixelYPosition  
CMP #NUM_ROWS + 1  
BNE LineModeLoop
```

The end result is a compact routine that paints our little lines down the screen.

Notice that our use of PaintPixelForCurrentSymmetry means that we don't need to worry about accounting for X-Y symmetry or Quad Symmetry here: PaintPixelForCurrentSymmetry will do that for us:

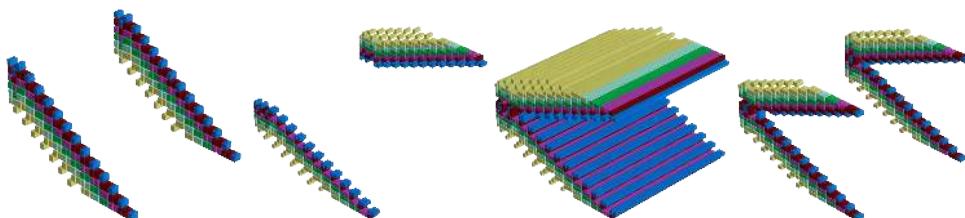


Figure 10.9: The different symmetries of Line Mode.



# after effects

After Psychedelia, Jeff Minter spent the rest of his working life make 'light synthesizers' of one sort or another. We will cover the very next incarnation of the Psychedelia concept in the second half of this book, an exploration of the 1985 title 'Colourspace' written just a few months after the C64 original. This took advantage of the superior graphical capabilities and colour palette of the Atari 800.

But Minter himself didn't stop with the Atari 800. He went on to develop the light synthesiser concept on nearly every generation of video game console released over the next thirty years. Some of these efforts were doomed by the failure of the platform they were written for, such as the 'Virtual Light Machine' on the ill-fated Atari Jaguar and the 'VLM 1' for the now rapidly defunct Nuon DVD platform developed by VM Labs. This chequered history culminated in something of a small triumph, when his PC-generation version of the Virtual Light Machine engine, known as 'Neon', was incorporated into the XBOX 360 when it was released in 2005. For the first time in his career, Minter's light synthesizer went mainstream and was put in the hands of millions of customers.

For us though, we will end this first half of our book in 1985, with Minter's final iteration of his light machine on 8-bit hardware: a psychedelic sub-game in his 1985 title 'Batalyx'. Here we find the idea reduced again to its kernel, back to a version of its beginnings in PC Computer Weekly magazine as a short lump of code that can fit on a few pages. Like the listing we acquainted ourselves at the start of this book, the version of Psychedelia in 'Batalyx' supports only a single lightform (or pattern) and only minimal configuration. Minter's motivation was simple:

## after effects

```

* = $0801
;-----+
; Start program at InitializeProgram (
    SYS 2064) ;SYS 2064 ($0810)
;-----+
.BYTE $0B,$0B
.BYTE $C1,$07
.BYTE $9E
.BYTE $32,$30,$36,$34
.BYTE $00
.BYTE $00,$00
.BYTE $F9,$02,$F9

;-----+
; LaunchPsychedelia
;-----+
JSR InitializeScreenLinePtrArray
JSR ClearScreen
JSR SetUpInterrupts
SEI
JSR InitializePsychedelia
JSR SetUpBackgroundPainting
JSR InitializeColorIndexArray
JSR InitializeStatusDisplayText
JSR UpdateCurrentSettingsDisplay
CLI
PsychedeliaLoop
JSR MaybeUpdateFromBuffersAndPaint
JSR CheckKeyboardInput
JMP PsychedeliaLoop

a40D7 .BYTE $03
a4142 .BYTE $01
;-----+
; SetUpInterrupts
;-----+
SetUpInterrupts
SEI
LDA #$7F
STA $DC0D ;CIA1: CIA Interrupt
    Control Register
LDA #>TitleScreenInterruptHandler
STA $0314 ;IRQ
LDA #>TitleScreenInterruptHandler
STA $0315 ;IRQ
LDA $800
STA currentRasterArrayIndex
JSR UpdateRasterPosition
JSR InitializeRasterJumpTablePtrArray

LDA $801
STA $D01A ;VIC Interrupt Mask
    Registered (IMR)
RTS
;-----+
; UpdateRasterPosition
;-----+
UpdateRasterPosition
LDA $D011 ;VIC Control Register 1
AND #$7F
STA $D011 ;VIC Control Register 1
LDX currentRasterArrayIndex
LDA rasterPositionArray,X
CMP #$FF
BNE b4224

LDA $800
STA currentRasterArrayIndex
LDA rasterPositionArray
b4224 STA $D012 ;Raster Position
LDA $801
STA $D019 ;VIC Interrupt Request
    Register (IRR)
RTS

rasterJumpTableLoPtr2***$01
rasterJumpTableLoPtr3***$02
rasterJumpTableHiPtrArray
.BYTE $55,$55
.BYTE $22,$22,$22,$46,$46,$46,$46
.BYTE $46,$46
rasterJumpTableHiPtr2***$01
rasterJumpTableHiPtr3***$02
rasterJumpTableHiPtrArray
.BYTE $C0,$C0
.BYTE $C3,$C3,$C3,$C3,$42,$42,$42,$42
.BYTE $42,$42
rasterPositionArray
.BYTE $E0,$FF,$C0,$FF,$A0,$C0,$FF

;-----+
; InitializeRasterJumpTablePtrArray
;-----+
InitializeRasterJumpTablePtrArray
LDX #$800
b4574
LDA $846
STA rasterJumpTableLoPtrArray,X
LDA $842
STA rasterJumpTableHiPtrArray,X
INX
CPX #$0C
BNE b4574
RTS

;-----+
; TitleScreenInterruptHandler
;-----+
TitleScreenInterruptHandler
LDA $D019 ;VIC Interrupt
AND #$01
BNE b4237
JMP $EA31
; Returns

; After a delay calculated from the
; IRQ switch to the Zarjas poster
; and back again.
b4237
LDX currentRasterArrayIndex
LDA rasterJumpTableLoPtrArray,X
STA a08

LDA rasterJumpTableHiPtrArray,X
STA a09

JMP ($0008)
;Returns

;-----+
; ClearScreen
;-----+
ClearScreen
LDA #$00
.Loop
LDA #SPACE
STA SCREEN_RAM,X
STA SCREEN_RAM + $100,X
STA SCREEN_RAM + $200,X
STA SCREEN_RAM + $300,X

LDA #WHITE
STA COLOR_RAM + $02F0,X
DEC
BNE _Loop
RTS

;-----+
; InitializeScreenLinePtrArray
;-----+
InitializeScreenLinePtrArray
LDA #>SCREEN_RAM
STA screenLinesHiPtr
LDA #<SCREEN_RAM
STA screenLinesLoPtr

LDX #$800
b14AD
LDA screenLinesLoPtr
STA screenLinesLoPtrArray,X
LDA screenLinesHiPtr
STA screenLinesHiPtrArray,X
LDA screenLinesLoPtr
CLC
ADC #NUM_COLS
STA screenLinesLoPtr

LDA screenLinesHiPtr
ADC #$00
STA screenLinesHiPtr
INX
CPX #NUM_ROWS + 2
BNE b414D
RTS

;-----+
; GetCurrentCharAddress
;-----+
GetCurrentCharAddress
LDX currentCharYPos
LDY currentCharXPos

LDA screenLinesLoPtrArray,X
STA screenLinesLoPtr

LDA screenLinesHiPtrArray,X
STA screenLinesHiPtr
RTS

;-----+
; WriteCurrentCharToScreen
;-----+
WriteCurrentCharToScreen
JSR GetCurrentCharAddress

LDA currentChar
STA (screenLinesLoPtr),Y

LDA screenLinesHiPtr
PHA

; Update color of character
CLC
ADC #OFFSET_TO_COLOR_RAM
STA screenLinesHiPtr

LDA defaultColorValue
STA (screenLinesLoPtr),Y

PLA
STA screenLinesHiPtr
RTS

;-----+
; InitializePsychedelia
;-----+
InitializePsychedelia
LDX #$800
.Loop
LDA #SPACE
STA SCREEN_RAM,X
DEX
BNE _Loop

LDA $D016 ;VIC Control
AND #$F0
ORA $80
STA $D016 ;VIC Control

LDA #$18
STA $D018 ;VIC Memory

LDA #BLACK
STA $D020 ;Border
STA $D021 ;Background
STA $D400 ;Voice 1

LDA #$04
STA $D407 ;Voice 2

LDA #$08
STA $D40E ;Voice 3

LDA #TOP_Y_POSITION - 7
STA currentCharYPos

LDA #$63
STA currentChar

LDA currentValue
STA defaultColorValue

SetUpScreenLoop
LDA $800
STA currentCharXPos

InnerLoop
JSR WriteCurrentCharToScreen

LDA currentCharYPos
PHA
SEC
SBC #TOP_Y_POSITION - 7
STA currentCharYPos

LDA currentChar
PHA
CLC
CLC
ADC #$5D
STA currentChar

JSR WriteCurrentCharToScreen

PLA
STA currentChar

PLA
STA currentCharYPos

INC currentCharXPos

LDA currentCharXPos
CMP #NUM_COLS
BNE InnerLoop

INC currentChar
INC currentCharYPos
LDA currentCharYPos
CMP #TOP_Y_POSITION + 1
BNE SetUpScreenLoop

SetUpSpritesAndVoiceRegisters
LDA $D011 ;VIC Control Register 1
AND #$F8
ORA $80
STA $D011 ;VIC Control Register 1

LDA #$80
STA $D015 ;Sprite display Enable
STA a40D7
STA a4142
STA $D404 ;Voice 1: Control Register
STA $D40B ;Voice 2: Control Register
STA $D412 ;Voice 3: Control Register

DrawTwoMiddleLines
LDX #$800
.Loop
LDA #$42
STA SCREEN_RAM + (NUM_COLS * 8),X

LDA currentValue
STA COLOR_RAM + (NUM_COLS * 8),X

INX
CPX #(2 * NUM_COLS)
BNE _Loop
RTS

```

**Jeffrey Says**



Well I was going to put a PAUSE mode in, but this is much better. When you need to, drop into SUB6 and relax. The timer stops and you can stay in the subgame until you've got your head together enough to play on. The controls are a subset of real PSYCHEDELIA, allowing S=symmetry change and C=cursor speed. You can also use F1 and SHIFT-F1 to change fore- and background colours.

**Design Notes:** Well it's more interesting than freezing the screen.

The feel and appearance of this light synthesizer is familiar enough and as we shall see the overall structure of its internals have not radically changed from the code we analysed in its predecessor. However closer inspection will reveal some subtle changes in the algorithm, some optimization and simplifications, that mean this is not just a desultory lift-and-shift of previous work to fill a gap in a game in need of a pause mode.

Even a cursory glance at the screenshots below shows an overall change in the aesthetic, an attempt within the confines of the C64's limited capabilities to create a less obviously pixellated appearance, an attempt create an impression of blur and flow:

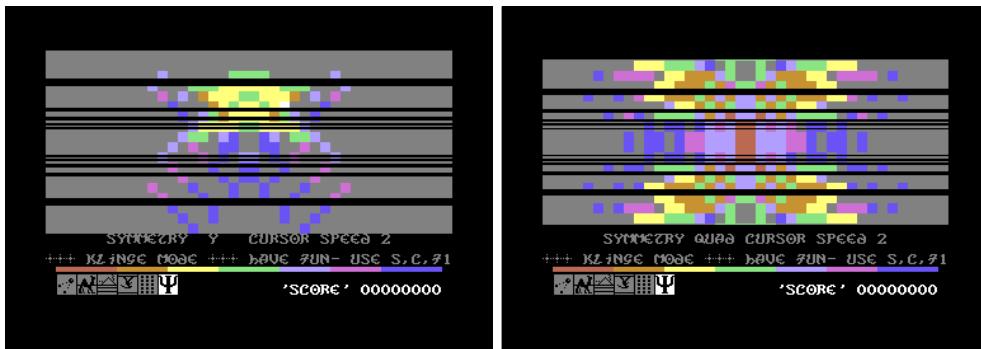


Figure 11.1: The Psychedelia sub-game in Batalyx. 'Y' symmetry on the left, and 'Quad' symmetry on the right.

Before we take a look at the core routines in detail, with a view to teasing out how they have evolved from their equivalents in Psychedelia, we should note how small the code involved has remained. It takes up a total of just three and a half pages here and in those that follow. The only reason it is slightly larger than the listing we reviewed in '[let's pretend we can read code](#)'. is the overhead entailed in adding a scrolling receding background. This effect is achieved with a neat conjunction of interrupt handlers, character set swapping, and a bit of brute force.

## after effects

```

;-----;
; SetUpBackgroundPainting
;-----;
SetUpBackgroundPainting
LDX #$00
b78A4
    LDA psyRasterPositionArray,X
    STA rasterPositionArray,X
    LDA psyRasterJumpTableLoPtrArray,X
    STA rasterJumpTableLoPtrArray,X
    LDA psyRasterJumpTableHiPtrArray,X
    STA rasterJumpTableHiPtrArray,X
    INX
    CPX #$03
    BNE b78A4
    RTS

psyRasterPositionArray
.BYTE $C0,$FF,$FF
psyRasterJumpTableLoPtrArray
.BYTE <PaintBackgroundColor,<
      PaintBackgroundColorI
.BYTE <PaintBackgroundColor
psyRasterJumpTableHiPtrArray
.BYTE >PaintBackgroundColor,>
      PaintBackgroundColor
.BYTE >PaintBackgroundColor

;-----;
; IncrementAndUpdateRaster
;-----;
IncrementAndUpdateRaster
INC currentRasterIndex
JSR UpdateRasterPosition
PLA
TAY
PLA
TAX
PLA
RTI

;-----;
; PaintBackgroundColor
;-----;
PaintBackgroundColor
LDA currentBackgroundColor
AND #$0F
STA $D020 ;Border Color
STA $D021 ;Background Color 0

JSR UpdateBackgroundData
JSR FetchBackgroundData
JSR WriteLinesToScreen
JSR $FF90 - scan keyboard
JMP IncrementAndUpdateRaster

currentColorValue .BYTE $0B
cursorPosition .BYTE $15
cursorPosition .BYTE $0B
colorValueToWrite .BYTE $01

screenRAMLoPtr = $23
screenRAMHiPtr = $24
;-----;
; WriteValueToColorRAM
;-----;
WriteValueToColorRAM
LDY cursorXPosition
LDX cursorYPosition

LDA screenLinesLoPtrArray,X
STA screenRAMLoPtr

LDA screenLinesHiPtrArray,X
CLC
ADC #OFFSET_TO_COLOR_RAM
STA screenRAMHiPtr

LDA colorValueToWrite
STA ($screenRAMLoPtr),Y
RTS

;-----;
; WriteLinesToScreen
;-----;
WriteLinesToScreen
LDA currentColorValue
STA colorValueToWrite

JSR WriteValueToColorRAM
JSR MaybeCheckJoystickInput

LDA #WHITE
STA colorValueToWrite

JSR WriteValueToColorRAM
RTS

;-----;
; MaybeCheckJoystickInput
;-----;
MaybeCheckJoystickInput
DEC prevCursorSpeed
BEQ CheckJoystickInput
RTS

prevCursorSpeed .BYTE $02

;-----;
; CheckJoystickInput
;-----;
CheckJoystickInput
LDA cursorSpeed
STA prevCursorSpeed
;CLAI Data Port Register A
LDA $DC00
STA lastJoystickInput

MaybeUpPressed
AND #JOYSTICK_DOWN
BEQ MaybeDownPressed

UpPressed
DEC cursorPosition
LDA cursorPosition
CMP # BELOW_ZERO
BNE MaybeDownPressed

LDA #TOP_Y_POSITION
STA cursorYPosition

MaybeDownPressed
LDA lastJoystickInput
AND #JOYSTICK_UP
BEQ MaybeLeftPressed

LDA #NUM_COLS-1
STA cursorXPosition

MaybeLeftPressed
LDA lastJoystickInput
AND #JOYSTICK_RIGHT
BEQ MaybeRightPressed

LDA #NUM_COLS-1
STA cursorXPosition

MaybeRightPressed
LDA lastJoystickInput
AND #JOYSTICK_LEFT
BEQ CheckFireButton

LDA #NUM_COLS-1
STA cursorXPosition

MaybeLightPressed
LDA lastJoystickInput
AND #JOYSTICK_UP
BEQ CheckFireButton

RightPressed
INC cursorPosition
LDA cursorPosition
CMP #NUM_COLS
BNE CheckFireButton

LDA #NUM_COLS-1
STA cursorXPosition

CheckFireButton
JSR maybeFirePressed
RTS

colorRAMLoPtr = $25
colorRAMHiPtr = $26
;-----;
; PaintPixel
;-----;
PaintPixel
LDX initialPixelYPosition
LDY initialPixelXPosition
LDA screenLinesLoPtrArray,X
STA colorRAMLoPtr

LDA screenLinesHiPtrArray,X
ADC #OFFSET_TO_COLOR_RAM
STA colorRAMHiPtr

LDA (colorRAMLoPtr),Y
AND #$0F
CMP currentColorValue
BEQ ActuallyPaintPixel

TAX
LDA colorComparisonArray,X
CMP lastColorValue
BEQ ActuallyPaintPixel
BPL ActuallyPaintPixel
RTS

ActuallyPaintPixel
LDA currentColor
STA (colorRAMLoPtr),Y
RTS

colorComparisonArray
.BYTE ORANGE,ORANGE,WHITE
.BYTE ORANGE,BLUE,PURPLE,YELLOW,CYAN
.BYTE RED,ORANGE,ORANGE,ORANGE
.BYTE ORANGE,ORANGE,GREEN,ORANGE
.BYTE ORANGE

;-----;
; PaintPixelForCurrentSymmetry
;-----;
PaintPixelForCurrentSymmetry
LDA initialPixelYPosition
AND #$80
BEQ CanpaintPixelOnThisLine

CleanUpAndReturn
RTS

CanPaintPixelOnThisLine
LDA initialPixelYPosition
CMP #TOP_Y_POSITION+1
BPL CleanUpAndReturn

LDA initialPixelXPosition
AND #$80
BNE CleanUpAndReturn

LDA initialPixelXPosition
CMP #NUM_COLS
BPL CleanUpAndReturn

LDA currentColor
TAX

LDA colorComparisonArray,X
STA lastColorValue
DEC lastColorValue

JSR PaintPixel

LDA currentSymmetrySettingForStep
BNE HasSymmetry

ReturnFromSymmetry
RTS

HasSymmetry
CMP #X_Y_SYMMETRY
BEQ HasXYSymmetry

CMP #X_AXIS_SYMMETRY
BEQ HasAxisSymmetry

LDA #NUM_COLS-1
SEC
SBC initialPixelXPosition
STA initialPixelXPosition

JSR PaintPixel

LDA currentSymmetrySettingForStep
CMP #Y_AXIS_SYMMETRY
BEQ ReturnFromSymmetry

LDA #TOP_Y_POSITION
SEC
SBC initialPixelYPosition
STA initialPixelYPosition

JSR PaintPixel

LDA #NUM_COLS-1
SEC
SBC initialPixelXPosition
STA initialPixelXPosition

JMP PaintPixel

HasXAxisSymmetry
LDA #NUM_COLS-1
SEC
SBC initialPixelYPosition
STA initialPixelYPosition

JMP PaintPixel

HasXYSymmetry
LDA #TOP_Y_POSITION
SEC
SBC initialPixelYPosition
STA initialPixelYPosition

JMP HasXYSymmetry

currentSymmetrySettingForStep
.BYTE $01
presetColorValuesArray
.BYTE RED,ORANGE
.BYTE YELLOW
.BYTE GREEN,LTBLUE
.BYTE PURPLE,BLUE
unusedVariable
.BYTE $0B
currentLineInPattern
.BYTE $07
currentPatternIndex
.BYTE $13
lastJoystickInput
.BYTE $7F

```

The extra overhead for managing the background is in the SetUpInterrupts, UpdateRasterPosition, TitleScreenInterruptHandler, and InitializeRasterJumpTablePtrArray routines on the previous page, and the SetUpBackgroundPainting, IncrementAndUpdateRaster, and PaintBackgroundColor routines on the page opposite. These are all required to put in place the machinery for updating the background as the raster lines moves down the screen 60 or so times each second.

The actual jiggery-pokery that achieves the receding effect is at the end of the listing, in FetchBackgroundData, UpdateBackgroundDataCharacters, and UpdateBackgroundData. The trick at work here is to avoid repainting the entire background manually by instead altering the definition of the character used to paint it in the first place. So for example, UpdateBackgroundData will do the heavy lifting of redefining the character used to paint an entire row, from this:

```
.BYTE $FF ; 11111111 *****
.BYTE $FF ; 11111111 *****
.BYTE $00 ; 00000000
.BYTE $00 ; 00000000
.BYTE $FF ; 11111111 *****
.BYTE $FF ; 11111111 *****
.BYTE $00 ; 00000000
.BYTE $00 ; 00000000
```

to this:

```
.BYTE $FF ; 11111111 *****
.BYTE $FF ; 11111111 *****
.BYTE $00 ; 00000000
.BYTE $00 ; 00000000
.BYTE $00 ; 00000000
.BYTE $FF ; 11111111 *****
.BYTE $FF ; 11111111 *****
.BYTE $FF ; 11111111 *****
```

The next time C64 paints a row of the screen with the contents of character '\$63' it will use this updated definition to do so and create the effect of the bottom row expanding and moving downwards thanks to the extra line of '1's included in the new definition.

When we look at the pattern and color palette used here we also notice some subtle changes. The palette is lighter and less stark than in Psychedelia:

```
colorComparisonArray
.BYTE ORANGE ,ORANGE ,WHITE ,ORANGE ,BLUE ,PURPLE ,YELLOW ,CYAN
.BYTE RED ,ORANGE ,ORANGE ,ORANGE ,ORANGE ,ORANGE ,GREEN ,ORANGE
.BYTE ORANGE
```

We can see this in the illustration of the pattern evolution in Figure 11.2. We also notice that the evolution of the pattern in Figure 11.3 while subject to the same basic constraints is behaving slightly differently.

## after effects

```

;-----+
; PaintStructureAtCurrentPosition
;-----+
PaintStructureAtCurrentPosition
LDA #$00
STA currentPatternIndex
STA currentLineInPattern

LDA currentPixelXPosition
STA initialPixelXPosition

LDA currentPixelYPosition
STA initialPixelYPosition

JSR PaintPixelForCurrentSymmetry

LDA currentColorIndex
BNE PixelPaintloop
RTS

PixelPaintloop
LDX currentPatternIndex
LDA patternXPosArray,X
CMP #$55
BEQ MoveToNextLineInPattern

CLC
ADC currentPixelXPosition
STA initialPixelXPosition

LDA patternYPosArray,X
CLC
ADC currentPixelYPosition
STA initialPixelYPosition

JSR PaintPixelForCurrentSymmetry

INC currentPatternIndex
JMP PixelPaintloop

MoveToNextLineInPattern
INC currentPatternIndex
INC currentLineInPattern

LDA currentLineInPattern
CMP currentColorIndex
BNE PixelPaintloop
RTS

patternXPosArray
.BYTE $FF,$01,$55 ; 6
.BYTE $FE,$02,$55 ; 5
.BYTE $FD,$03,$55 ; 4
.BYTE $FC,$04,$55 ; 3
.BYTE $FB,$05,$55 ; 2
.BYTE $FA,$06,$55 ; 1
.BYTE $55,$55 ; 1

patternYPosArray
.BYTE $01,$02,$55 ; 2
.BYTE $03,$04,$55 ; 3
.BYTE $05,$06,$55 ; 4
.BYTE $07,$08,$55 ; 5
.BYTE $09,$0A,$55 ; 6
.BYTE $FA,$06,$55
.BYTE $55,$55

currentColorIndex .BYTE $07
currentPixelXPosition .BYTE $15
currentPixelYPosition .BYTE $06
symmetrySettingForStep .BYTE $01
    .BYTE $01,$01,$01,$01,$01,$01,$01
    .BYTE $01,$01,$01,$01,$01,$01,$01

pixelXPositionArray
.BYTE $15,$15,$15,$15,$15,$15,$15
.BYTE $15,$15,$15,$15,$15,$15,$15
.BYTE $15,$15,$15,$15,$15,$15,$15
.BYTE $15,$15,$15,$15,$15,$15,$15
.BYTE $15,$15,$15,$15,$15,$15,$15
.BYTE $15,$15,$15,$15,$15,$15,$15
.BYTE $17,$18,$19,$1A,$1B,$1C,$1D,$1E
.BYTE $1D,$1C,$1B,$1A,$19,$18,$17,$16

pixelYPositionArray
.BYTE $02,$03,$04,$05,$06,$07,$08,$09
.BYTE $04,$08,$0C,$0D,$0E,$0F,$10,$11
.BYTE $00,$01,$02,$03,$04,$05,$06,$07
.BYTE $08,$09,$0A,$0B,$0C,$0D,$0E,$0F
.BYTE $10,$11,$00,$01,$02,$03,$04,$05
.BYTE $06,$07,$08,$09,$0A,$0C,$0B,$0A
.BYTE $09,$08,$07,$06,$05,$04,$03,$02
.BYTE $01,$00,$11,$11,$11,$11,$00,$00

smoothingDelayArray
.BYTE $0C,$0C,$0C,$0C,$0C,$0C,$0C
.BYTE $0C,$0C,$0C,$0C,$0C,$0C,$0C
.BYTE $0C,$0C,$0C,$0C,$0C,$0C,$0C
.BYTE $0C,$0C,$0C,$0C,$0C,$0C,$0C
.BYTE $0C,$04,$07,$0A,$02,$0C,$0C,$0C
.BYTE $0C,$0C,$0C,$0C,$0C,$0C,$0C
.BYTE $0C,$0C,$0C,$0C,$0C,$0C,$0C

lastIndexToBuffers .BYTE $01
;-----+
; CheckKeyboardInput
;-----+
CheckKeyboardInput
LDA currentPressedKey
CMP #NU_KEY_PRESSED
BNE KeyboardInputReceived

LDA #$00
STA processingKeyStroke
ReturnFromKeyboardInput
RTS

KeyboardInputReceived
LDY processingKeyStroke
BNE ReturnFromKeyboardInput

MaybeKeyPressed
CMP #KEY_S
BNE MaybeKeyPressed

; 'S' pressed.
LDA $01
STA processingKeyStroke

INC currentSymmetrySetting
LDA currentSymmetrySetting
CMP #$00
BNE UpdateStatusLineAndReturn

LDA #$00
STA currentSymmetrySetting

UpdateStatusLineAndReturn
JSR UpdateCurrentSettingsDisplay
RTS

MaybeCKeyPressed
CMP #KEY_C
BNE MaybeCKeyPressed

; C pressed.
; Cursor Speed C to activate: Just
; that. Gives you a slow or fast
; little
; cursor, according to setting.
LDA $01
STA processingKeyStroke

INC cursorSpeed
LDA cursorSpeed
CMP #$05
BNE UpdateStatusLineAndReturn

LDA #$01
STA cursorSpeed
JMP UpdateStatusLineAndReturn

MaybeF1Pressed
CMP #KEY_F1_F2
BEQ CycleBackgroundColor
RTS

cursorSpeed .BYTE $02
processingKeyStroke .BYTE $00

;-----+
; CycleBackgroundColor
;-----+
CycleBackgroundColor
LDA shiftKey
AND #$01
BEQ ChangeBorderColor

ChangeBackgroundColor
INC currentBackgroundColor
LDA $01
STA processingKeyStroke
RTS

ChangeBorderColor
LDA currentColorValue
STA currentBorderColor

SEI

INC currentColorValue
LDA currentColorValue
AND #$0F
STA currentColorValue

STA unusedVariable

LDX #$00
Loop LDA COLOR_RAM + $0000,X
JSR CheckCurrentBorderColor
STA COLOR_RAM + $0000,X

LDA COLOR_RAM + $0100,X
JSR CheckCurrentBorderColor
STA COLOR_RAM + $0100,X

LDA COLOR_RAM + $0100,X
JSR CheckCurrentBorderColor
STA COLOR_RAM + $0100,X

DEX
BNE _Loop

```

To understand why this is the case we will take a look at the core routines in more detail. As with all previous versions these are present in a familiar form: a routine called `PaintStructureAtCursorPosition` supported by two helpers called `PaintPixelForCurrentSymmetry` and `PaintPixel`.

We'll take a closer look at these after setting out the pattern evolution and data structure underlying it.

<pre> LDA #\$01 STA processingKeyStroke  CLI Return RTS  ;----- ; CheckCurrentBorderColor ;----- CheckCurrentBorderColor AND #\$0F CMN currentBorderColor BNE Return LDA currentColorValue RTS  currentBorderColor .BYTE \$00 ;----- ; InitializeStatusDisplayText ;----- InitializeStatusDisplayText LDX #\$00 .Loop LDA statusLineOne,X AND #\$3F STA SCREEN_RAM + (NUM_COLS * 20),X  LDA #\$0B STA COLOR_RAM + (NUM_COLS * 20),X  INX CPX #NUM_COLS BNE _Loop RTS  statusLineOne .TEXT "*** KLINGE MODE ***" .TEXT "HAVE FUN- USE S,C,F1" statusLineTwo .TEXT "      SYMMETRY . . ." .TEXT "CURSOR SPEED 0  ."  ;----- ; UpdateCurrentSettingsDisplay ;----- UpdateCurrentSettingsDisplay LDY #\$00 .Loop LDA statusLineTwo,X AND #\$3F STA SCREEN_RAM + (NUM_COLS * 18),X  LDA #\$0B STA COLOR_RAM + (NUM_COLS * 18),X  INX CPX #NUM_COLS BNE _Loop  LDA cursorSpeed CLC ADC #\$30 STA SCREEN_RAM + (NUM_COLS * 18) + 33 LDA currentSymmetrySetting ASL ASL TAY  LDX #\$00 .Loop2 LDA symmetrySettingTxt,Y AND #\$3F </pre>	<pre> STA SCREEN_RAM + (NUM_COLS * 18) +     15,X INY INX CPX #\$04 BNE _Loop2 RTS  symmetrySettingTxt .TEXT "NONE Y     X-Y QUAD" currentBackgroundColor .BYTE \$00 currentSymmetrySetting .BYTE \$01,\$DD  LEN_INITIAL_ARRAY = \$1C initialBkgUpdateControlArray .INYTE \$08,\$08,\$08,\$08,\$04,\$04,\$04 .BYTE \$02,\$02,\$02,\$02,\$01,\$01,\$01 .BYTE \$01,\$01,\$01,\$01,\$01,\$01,\$01 .BYTE \$01,\$01,\$01,\$01  numberOfUpdatesToMakeToChars .BYTE \$01,\$01,\$01,\$01,\$02,\$02,\$02 .BYTE \$03,\$03,\$03,\$03,\$04,\$04,\$04 .BYTE \$05,\$05,\$05,\$05,\$06,\$06,\$06 .BYTE \$07,\$07,\$07,\$07,\$08,\$08,\$08  LEN_BG_CTRL_ARRAY = \$10 indexArrayForBackgroundChars .BYTE \$00,\$00,\$00,\$00,\$00,\$00,\$00,\$00 .BYTE \$00,\$00,\$00,\$00,\$00,\$00,\$00,\$00 backgroundUpdateControlArray .INYTE \$00,\$00,\$00,\$00,\$00,\$00,\$00,\$00 .BYTE \$00,\$00,\$00,\$00,\$00,\$00,\$00,\$00  ;----- ; UpdateBackgroundData ;----- UpdateBackgroundData DEC controlByteForUpdatingBackground BNE UpdateChars LDA #LEN_INITIAL_ARRAY STA controlByteForUpdatingBackground  LDX #\$00 .Loop LDA backgroundUpdateControlArray,X BEO ResetArray INX CPX #LEN_BG_CTRL_ARRAY BNE _Loop JMP UpdateChars  ResetArray LDX #\$01 STA backgroundUpdateControlArray,X LDA #\$00 STA indexArrayForBackgroundChars,X  UpdateChars LDX #\$00 .Loop LDA backgroundUpdateControlArray,X BEO _ExitLoop DEC backgroundUpdateControlArray,X BNE _ExitLoop  TXA PHA JSR UpdateBackgroundDataCharacters PLA </pre>	<pre> TAX _EXITLoop INX CPX #LEN_BG_CTRL_ARRAY BNE _Loop RTS  NUM_HORIZON_CHARACTERS = \$09 ;- ; UpdateBackgroundDataCharacters ;- UpdateBackgroundDataCharacters LDA indexArrayForBackgroundChars,X STA indexIntoDataChars  CLC ROR STA indexToBackgroundControlArray INC indexArrayForBackgroundChars,X  LDA #\$FF LDY indexIntoDataChars STA rollingHorizonCharacters,Y  INY TYA  STA indexIntoDataChars CMP #(8 * NUM_HORIZON_CHARACTERS) BNE ResetHorizonCharacter RTS  ResetHorizonCharacter LDY indexToBackgroundControlArray LDA initialBkgUpdateControlArray,Y STA backgroundUpdateControlArray,X  LDX numberOfUpdatesToMakeToChars,Y LDY indexIntoDataChars  LDA #\$00 .Loop INY STA rollingHorizonCharacters,Y DEX BNE _loop RTS  controlByteForUpdatingBackground .BYTE \$01,\$63,\$64,\$65 .BYTE \$66,\$67,\$68,\$69 .BYTE \$6A,\$6B,\$6C,\$6D .BYTE \$6E,\$6F  ;- ; FetchBackgroundData ;- FetchBackgroundData LDX #\$FF LDY #\$00  .Loop LDA rollingHorizonCharacters,Y STA activeBackgroundCharacters,X  DEX INY CPY #\$40 BNE _Loop RTS </pre>
---	---	---

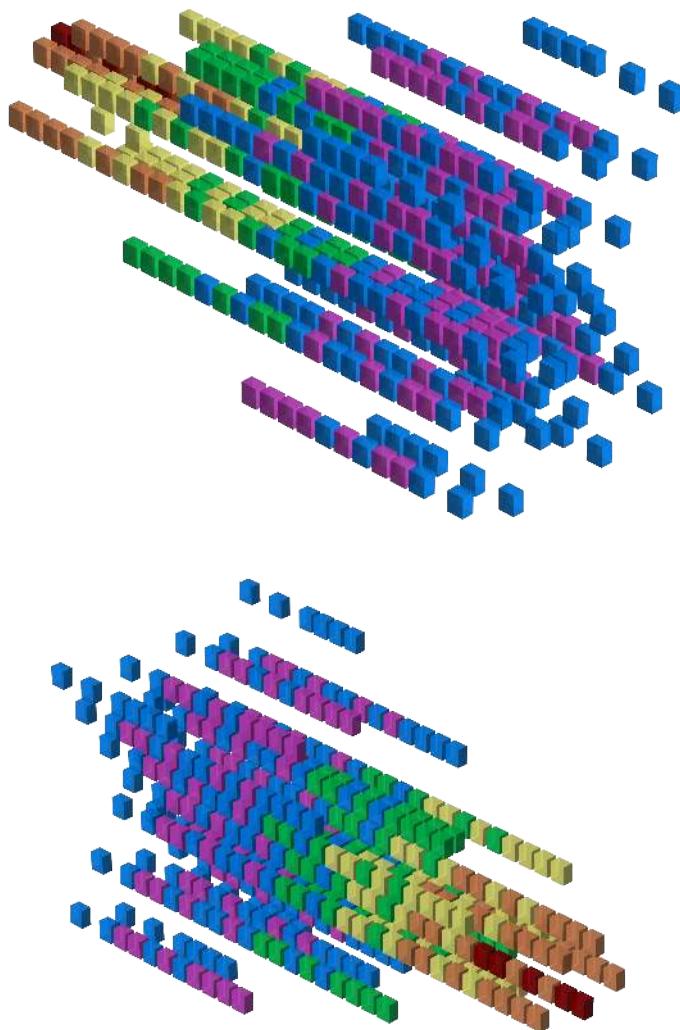


Figure 11.2: Evolution of the 'Batalyx' pattern.

```

patternXPosArray
    .BYTE $FF,$01,$55      ; 6
    .BYTE $FE,$02,$55      ; 5
    .BYTE $FD,$03,$55      ; 4
    .BYTE $FC,$04,$55      ; 3
    .BYTE $FB,$05,$55      ; 2
    .BYTE $FA,$06,$55      ; 1
    .BYTE $55,$55          ;
patternYPosArray
    .BYTE $01,$FF,$55      ; 1
    .BYTE $FE,$02,$55      ; 2
    .BYTE $03,$FD,$55      ; 3
    .BYTE $FC,$04,$55      ; 4
    .BYTE $05,$FB,$55      ; 5
    .BYTE $FA,$06,$55      ; 6
    .BYTE $55,$55          ;

```

Listing 11.1: Source code for the Batalyx pattern..

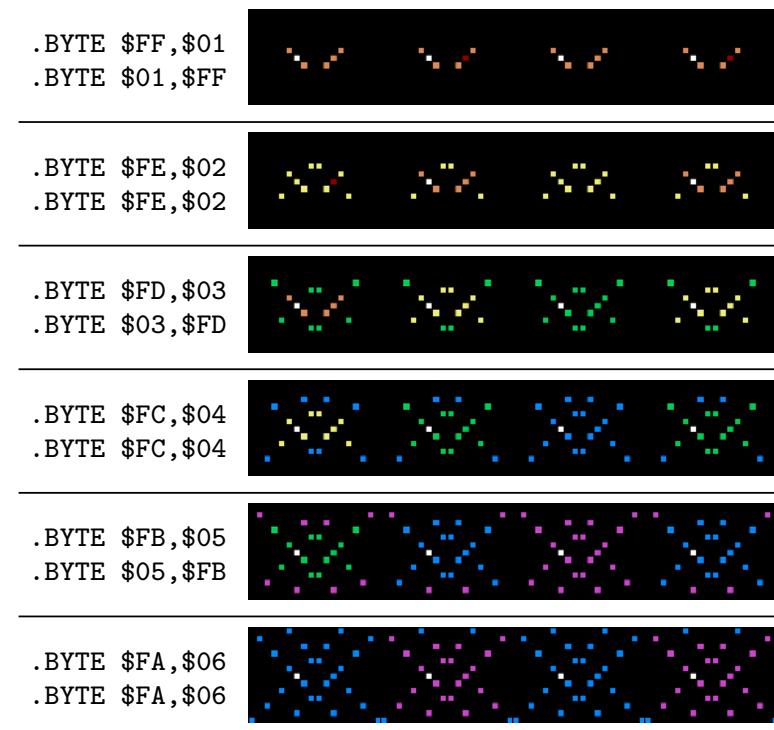


Figure 11.3: Pattern Progression for 'Batalyx'

**Lines 41-55.** LaunchPsychedelia

```
; -----
; LaunchPsychedelia
; -----
LaunchPsychedelia
    SEI
    JSR InitializePsychedelia
    JSR SetUpBackgroundPainting
    JSR InitializeColorIndexArray
    JSR InitializeStatusDisplayText
    JSR UpdateCurrentSettingsDisplay
    CLI
PsychedeliaLoop
    JSR MaybeUpdateFromBuffersAndPaint
    JSR CheckKeyboardInput
    JMP PsychedeliaLoop
```

**Lines 805-841.** MaybeUpdateFromBuffersAndPaint

```
MaybeUpdateFromBuffersAndPaint
    LDX currentIndexToPixelBuffers
    LDA currentColorIndexArray,X
    AND #$08
    BNE BufferUpdateComplete

    DEC smoothingDelayArray,X
    BNE BufferUpdateComplete

    LDA initialSmoothingDelayArray,X
    STA smoothingDelayArray,X

    LDA pixelXPositionArray,X
    STA currentPixelXPosition
    LDA pixelYPositionArray,X
    STA currentPixelYPosition

    LDA currentColorIndexArray,X
    STA currentColorIndex

    TAY
    LDA symmetrySettingForStep,X
    STA currentSymmetrySettingForStep

    LDA presetColorValuesArray,Y
    STA currentColor

    INC currentColorIndexArray,X
    JSR PaintStructureAtCursorPosition

BufferUpdateComplete
    INC currentIndexToPixelBuffers

    LDA currentIndexToPixelBuffers
    AND #$3F
    STA currentIndexToPixelBuffers
    RTS
```

Listing 11.2: The routine responsible for painting patterns.

**Lines 41-55.** LaunchPsychedelia: A nice compact launch routine compared to what we're used to. As an added bonus it also contains the game's main loop in PsychedeliaLoop! The first thing to note here is that, unlike all the previous versions we've looked at, this main loop is also checking keyboard input - something previously managed by the interrupt handler. The interrupt handler, as well as animating the background, still looks after joystick input but the reduced set of keyboard commands and the increased load on the interrupt handler, means that we can fit the keyboard check in here instead.

Of course PsychedeliaLoop's main purpose in life is to repeatedly call MaybeUpdateFromBuffersAndPaint.

**Lines 805-841.** MaybeUpdateFromBuffersAndPaint: The business end of the main loop which co-ordinates all pattern painting. The logic used to determine whether a paint looks superficially similar to Psychedelia, but there is a subtle, yet ultimately critical, difference. Whereas Psychedelia managed the values in currentColorIndexArray by decrementing them and continuing to paint the pixel represented by that position until it was decremented past zero, here we manage currentColorIndexArray by incrementing it and ceasing painting once the value reaches \$08, i.e. when there are no more colors to reference.

Below we see the very different execution profiles of the commercial edition of Psychedelia on the left and the 'Batalyx' edition we are examining here. Rather than occurring in clusters, the painting in this version is much more evenly spread over time.

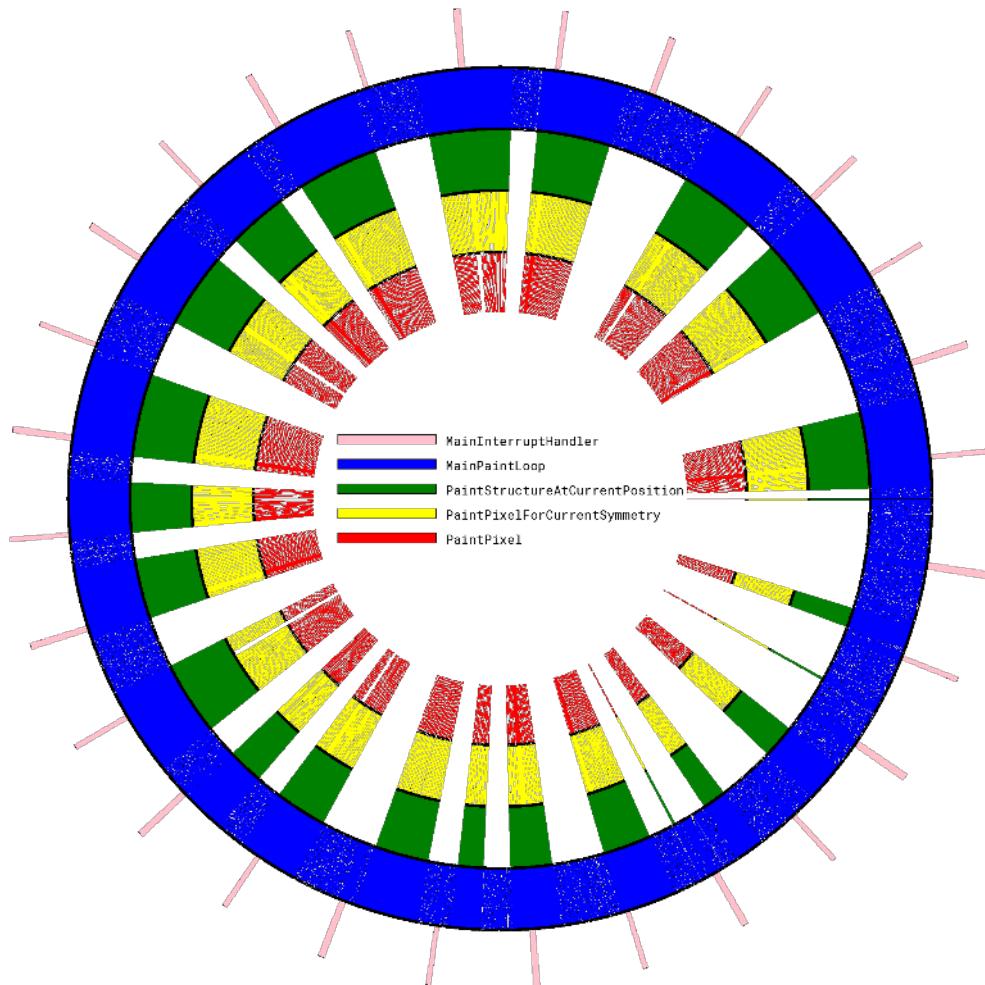


Figure 11.4: The execution map of a full pattern evolution in the original edition of Psychedelia.

As we shall see, there is something new in the logic in `PaintStructureAtCurrentPosition` that accounts for this change in behaviour.

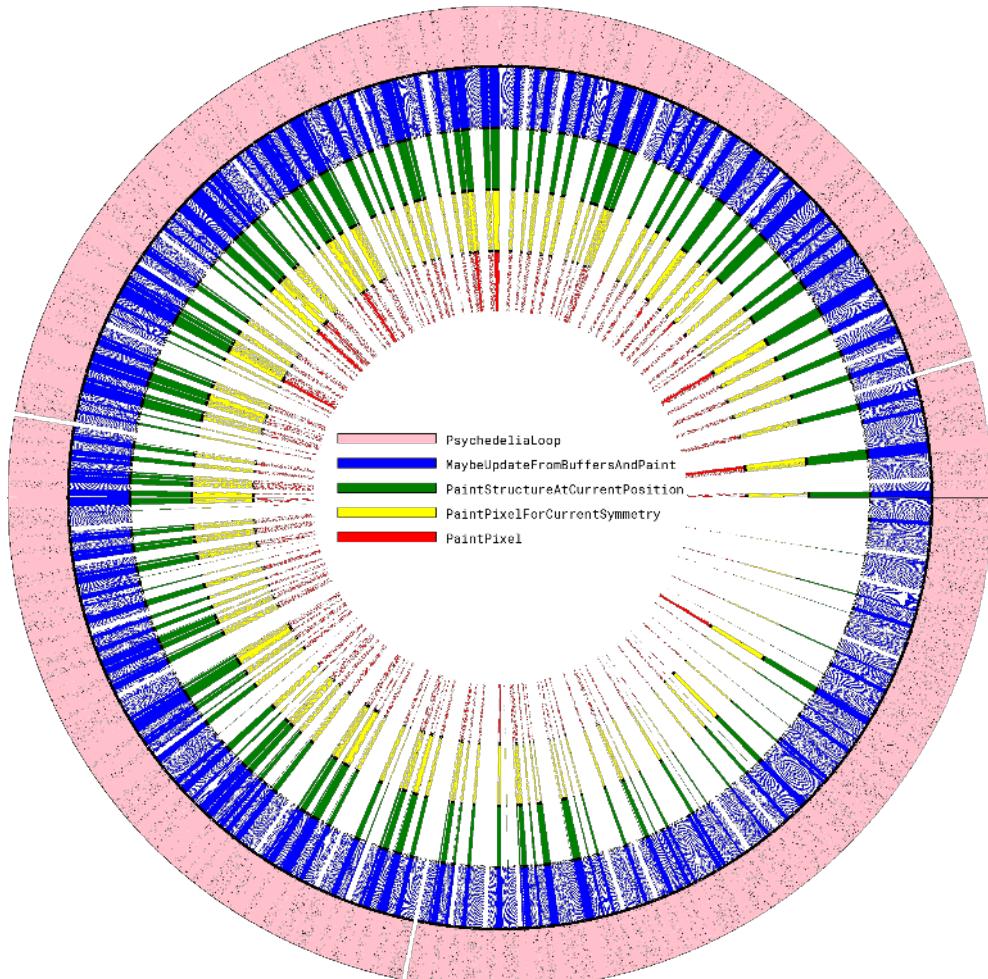


Figure 11.5: The execution map of a full pattern evolution in the Batalyx edition of Psychedelia.

**Lines 635-677.** PaintStructureAtCursorPosition

```
; -----  
; PaintStructureAtCursorPosition  
;  
PaintStructureAtCursorPosition  
    LDA #$00  
    STA currentPatternIndex  
    STA currentLineInPattern  
  
    LDA currentPixelXPosition  
    STA initialPixelXPosition  
    LDA currentPixelYPosition  
    STA initialPixelYPosition  
  
    JSR PaintPixelForCurrentSymmetry  
  
    LDA currentColorIndex  
    BNE PixelPaintLoop  
    RTS  
  
PixelPaintLoop  
    LDX currentPatternIndex  
    LDA patternXPosArray ,X  
    CMP #$55  
    BEQ MoveToNextLineInPattern  
  
    CLC  
    ADC currentPixelXPosition  
    STA initialPixelXPosition  
  
    LDA patternYPosArray ,X  
    CLC  
    ADC currentPixelYPosition  
    STA initialPixelYPosition  
  
    JSR PaintPixelForCurrentSymmetry  
  
    INC currentPatternIndex  
    JMP PixelPaintLoop  
  
MoveToNextLineInPattern  
    INC currentPatternIndex  
    INC currentLineInPattern  
    LDA currentLineInPattern  
    CMP currentColorIndex  
    BNE PixelPaintLoop  
    RTS
```

Listing 11.3: All the pattern data structures in Psychedelia organized into a set of arrays.

**Lines 635-677.** PaintStructureAtCursorPosition: There's a difference in approach here. The logic is much simpler than in the versions of this routine we've seen in earlier editions. In a nutshell, we arrive into this routine with a value between 1 and 8 in currentColorIndex - whatever that value is we will read in that many lines from the pattern's data structure. So if our currentColorIndex is 3 we will read in the first 3 lines of patternXPosArray and patternYPosArray:

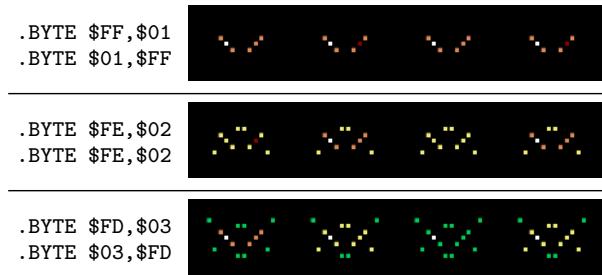
```

patternXPosArray
    .BYTE $FF,$01,$55      ; 6
    .BYTE $FE,$02,$55      ; 5
    .BYTE $FD,$03,$55      ; 4
    ...
    ...

patternYPosArray      ; 1
    .BYTE $01,$FF,$55      ; 2
    .BYTE $FE,$02,$55      ; 3
    .BYTE $03,$FD,$55      ; 4
    ...
    ...

```

This translates into the routine painting the pattern as follows.



As before, once the lines are ready to be processed the heavy lifting passes to PaintPixelForCurrentSymmetry.

**Lines 543-601.** PaintPixelForCurrentSymmetry

```
; -----  
; PaintPixelForCurrentSymmetry  
;  
PaintPixelForCurrentSymmetry  
    LDA initialPixelYPosition  
    AND #$80  
    BEQ CanPaintPixelOnThisLine  
  
CleanUpAndReturn  
    RTS  
  
CanPaintPixelOnThisLine  
    LDA initialPixelYPosition  
    CMP #TOP_Y_POSITION+1  
    BPL CleanUpAndReturn  
  
    LDA initialPixelXPosition  
    AND #$80  
    BNE CleanUpAndReturn  
  
    LDA initialPixelXPosition  
    CMP #NUM_COLS  
    BPL CleanUpAndReturn  
  
    LDA currentColor  
    TAX  
    LDA colorComparisonArray,X  
    STA lastColorValue  
    DEC lastColorValue  
  
    JSR PaintPixel  
  
    LDA currentSymmetrySettingForStep  
    BNE HasSymmetry  
  
ReturnFromSymmetry  
    RTS  
  
HasSymmetry  
    CMP #X_Y_SYMMETRY  
    BEQ HasXYSymmetry  
  
    CMP #X_AXIS_SYMMETRY  
    BEQ HasXAxisSymmetry  
  
    LDA #NUM_COLS-1  
    SEC  
    SBC initialPixelXPosition  
    STA initialPixelXPosition
```

**Lines 601-625.** PaintPixelForCurrentSymmetry continued.

```

JSR PaintPixel

LDA currentSymmetrySettingForStep
CMP #Y_AXIS_SYMMETRY
BEQ ReturnFromSymmetry

LDA #TOP_Y_POSITION
SEC
SBC initialPixelYPosition
STA initialPixelYPosition

JSR PaintPixel

LDA #NUM_COLS -1
SEC
SBC initialPixelXPosition
STA initialPixelXPosition

JMP PaintPixel

HasXYSymmetry
LDA #TOP_Y_POSITION
SEC
SBC initialPixelYPosition
STA initialPixelYPosition

JMP PaintPixel

HasXAxisSymmetry
LDA #NUM_COLS -1
SEC
SBC initialPixelXPosition
STA initialPixelXPosition
JMP HasXYSymmetry

```

Listing 11.4: The routine responsible for handling the symmetry in use for painting pixels.

**Lines 543-625.** PaintPixelForCurrentSymmetry: This routine feels like an improvement on previous iterations. It makes heavy use of 'early returns' (CleanUpAndReturn) and the way in which it is determining what symmetry is in effect and how to decompose that into the painting of individual pixels is much more legible to the reader than in earlier versions of the routine in Psychedelia and the 'listing edition' of Psychedelia.

**Lines 504-528. PaintPixel**

```
; -----
; PaintPixel
; -----
PaintPixel
    LDX initialPixelYPosition
    LDY initialPixelXPosition
    LDA screenLinesLoPtrArray ,X
    STA colorRAMLoPtr
    LDA screenLinesHiPtrArray ,X
    CLC
    ADC #OFFSET_TO_COLOR_RAM
    STA colorRAMHiPtr
    LDA (colorRAMLoPtr),Y
    AND #$0F
    CMP currentValue
    BEQ ActuallyPaintPixel

    TAX
    LDA colorComparisonArray ,X
    CMP lastColorValue
    BEQ ActuallyPaintPixel
    BPL ActuallyPaintPixel
    RTS

ActuallyPaintPixel
    LDA currentColor
    STA (colorRAMLoPtr),Y
    RTS

colorComparisonArray
    .BYTE ORANGE,ORANGE,WHITE,ORANGE,BLUE,PURPLE,YELLOW,CYAN
    .BYTE RED,ORANGE,ORANGE,ORANGE,ORANGE,ORANGE,GREEN,ORANGE
    .BYTE ORANGE
```

Listing 11.5: All the pattern data structures in Psychedelia organized into a set of arrays.

**Lines 504-528.** PaintPixel: This is much less cryptic than previous versions of this routine. The first part simply figures out if the current location chosen for drawing is the same as the current background color (`currentColorValue`) - if so, we paint it.

Otherwise, we check the current color at the chosen location against the `colorComparisonArray` using the color value itself as an index. If it's greater or equal we paint our new chose color there.



**colourspace  
complex**



# developing a bit of a complex

## Jeffrey Says



This was the next major evolution of the lightsynth. The Atari machine had a much more extensive palette than the Commodore machines, plus you could make all kinds of interesting screen modes using the Display List Interrupt system, so it seemed like a natural machine to develop a next-generation lightsynth on. I still used the same basic algorithm as *Psychedelia*, but introduced a lot more colour variation, and colour cycling. There was also a provision for designing static graphics and logos on the screen and having the generated patterns flow over or under them, and many more symmetry modes afforded by the flexible hardware. Screens could have a cylindrical "curved" look since you could vary the vertical pixel size on a scanline by scanline basis, thanks to the DLIs.

Just a few months before the mini-version of *Psychedelia* included in *Batalyx* appeared, Minter released his next major iteration in his life-long sequence of light synthesizers. This was *Colourspace* and it was a major improvement on its predecessor. The core code of the game remained the same (the algorithms we have covered in detail in *Psychedelia* remain unchanged) but the hardware of the Atari 8 bit computers created opportunities for much more colour and complexity.

For this reason alone it is worth us devoting the final parts of this book to the effects of this second generation in the series. There will be much less code for us to cover because so much of it was simply dropped in from *Psychedelia*. The important, and interesting, differences lie mainly in the work required to take advantage of the Atari 400/800's graphical capabilities. These we will cover in our next chapter '[painting pixels is so complicated](#)'. There we'll take a look at some of the things Minter had to learn about the Atari and adapt to the purpose of making a new generation of *Psychedelia* and this will give us a clear sense of the scale of the undertaking in such a short time.

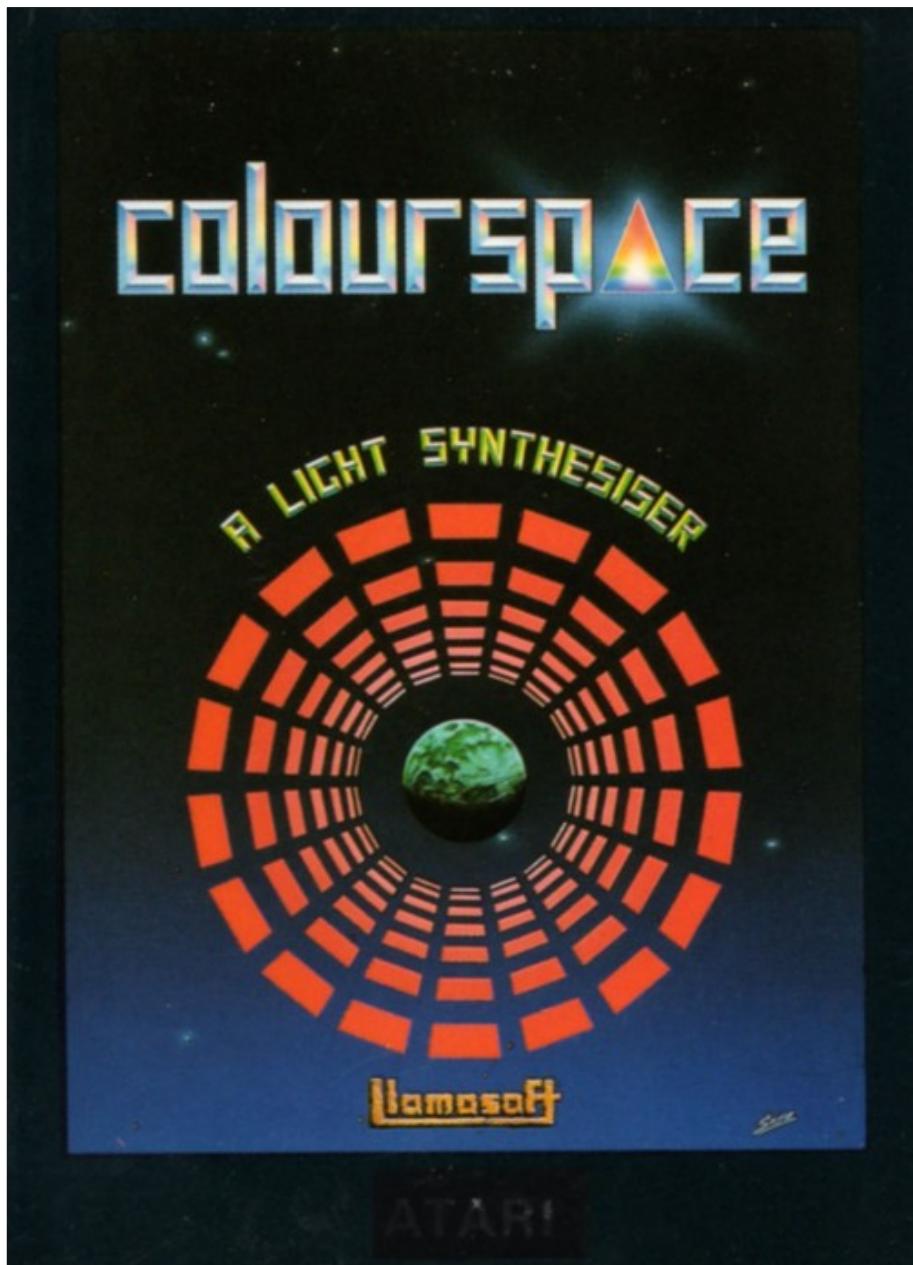


Figure 12.1: Cover art by Steinar Lund for the commercial edition of Colourspace

After that we are free to appreciate the fruit of this Atari port without detaining ourselves too long over the code required to achieve it since we have already seen it in action in our previous chapters on *Psychedelia*. For this reason our final chapters provide a restful close to this book in which we can enjoy the colours and configurations of Colourspace and I can get away with as little writing as possible. '[all the little light-forms](#)' sets out the 28 new patterns available in *Colourspace*, while '[plentiful presets](#)' shows off all 78 presets. Finally, in '[colourflow](#)' we get to see the full range of colour schemes the game employs.



# painting pixels is so complicated

**Jeffrey Says**



By using the unique Atari screen hardware and colour palette, the effect of the program is much improved. The difference between Psychedelia and Colourspace is as pronounced as the difference between a Mini and a Ferrari. Using the Atari you can get curved screens, hardware reflections, interlace effects, stroboscopics, dynamic colourflows, and variable resolution screens.

Up to now we have only had to deal with the relative simplicity of painting permitted by the Commodore 64. The screen was represented in RAM by two arrays of 1000 bytes. The first, starting at \$4000, contained a reference to an 8x8 pixel character to paint at each position in the 40x25 screen. The second array, starting at \$D800, specified the color of the character at each position. Since *Psychedelia* was primarily interested in painting colors, this second array was the focus of most of its operations.

```
$D800 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
$D828 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
$D850 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
$D878 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
$D8A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
$D8C8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
$D8F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 02 00 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
$D918 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
$D940 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 05 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
$D968 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
$D990 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 02 00 07 07 07 03 05 02 00 00 00 00 00 00 00 00 00  
$D9B8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 02 05 03 07 07 07 03 05 02 00 00 00 00 00 00 00 00 00  
$D9E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03 00 00 07 07 00 00 00 00 00 00 00 00 00 00 00 00 00  
$DA08 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
$DA30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
$DA58 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
$DA80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
$DAA8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
$DADO 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
$DAF8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
$DB20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
$DB48 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
$DB70 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
$DB98 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 13.1: Color RAM on the C64. 40 blocks wide by 25 blocks high.



the Atari 800 has a whopping 256. This means that although we can only display 8 colours at a time we get to choose from a much larger palette and have the opportunity to choose tasteful combinations for our displays, for varying definitions of tasteful!

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17		
18	19	1A	1B	1C	1D	1E	1F	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F		
38	39	3A	3B	3C	3D	3E	3F	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F		
58	59	5A	5B	5C	5D	5E	5F	60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F		
78	79	7A	7B	7C	7D	7E	7F	80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F		
98	99	9A	9B	9C	9D	9E	9F	A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC	AD	AE	AF		
B8	B9	BA	BB	BC	BD	BE	BF	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF		
D8	D9	DA	DB	DC	DD	DE	DF	E0	E1	E2	E3	E4	E5	E6	E7	E8	E9	EA	EB	EC	ED	EE	EF		
F8	F9	FA	FB	FC	FD	FE	FF											F0	F1	F2	F3	F4	F5	F6	F7

Atari 800: A broad spectrum of tasteful colors to choose from.

00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

Commodore 64: Not many colours and none of them particularly tasteful.

The initial palette defined for Colourspace is given by `colorPallete`:

```
colorPallete    .BYTE $00,$18,$38,$58,$78,$98,$B8,$D8
```

These translate as follows, the same palette we see in use in our figure on the page opposite.

00 18 38 58 78 98 B8 D8

The initial palette defined for Colourspace.

Colourspace allows us to cycle this palette using a 'simultaneous adder' by pressing the 'H' key. This confusing term 'simultaneous adder' means we can add a fixed value to each element in the array that will increment all of the values at once. This has the effect of shifting the palette without changing any of the component colours.

The code that does this is as follows. We can see `simlAdder`, which has a default value of \$10(i.e. 16), gets added to each element in the array:

```
UpdateColorValues
    LDY #$07
    INX
UpdateColorValuesLoop
    LDA colorPallete,X
    CLC
    ADC simlAdder
    STA colorPallete,X
    INX
```

painting pixels is so complicated

---

```
DEY  
BNE UpdateColorValuesLoop
```

The result, as we can see, is to shift each value in `colorPalette` by \$10:

00	18	38	58	78	98	B8	D8	10	28	48	68	88	A8	C8	E8
20	38	58	78	98	B8	D8	F8	30	48	68	88	A8	C8	E8	08
40	58	78	98	B8	D8	F8	18	50	68	88	A8	C8	E8	08	28
60	78	98	B8	D8	F8	18	38	70	88	A8	C8	E8	08	28	48
80	98	B8	D8	F8	18	38	58	90	A8	C8	E8	08	28	48	68
A0	B8	D8	F8	18	38	58	78	B0	C8	E8	08	28	48	68	88
C0	D8	F8	18	38	58	78	98	D0	E8	08	28	48	68	88	A8
E0	F8	18	38	58	78	98	B8	F0	08	28	48	68	88	A8	C8

Cycling through the palettes using an adder with the default value of \$10(16).

We get a better result if we change the *simultaneous adder* to a different value (which we can do by pressing the J key). This is what cycling the palette using an adder of \$0F instead of \$10 looks like:

00	18	38	58	78	98	B8	D8	0F	27	47	67	87	A7	C7	E7
1E	36	56	76	96	B6	D6	F6	2D	45	65	85	A5	C5	E5	05
3C	54	74	94	B4	D4	F4	14	4B	63	83	A3	C3	E3	03	23
5A	72	92	B2	D2	F2	12	32	69	81	A1	C1	E1	01	21	41
78	90	B0	D0	F0	10	30	50	87	9F	BF	DF	FF	1F	3F	5F
96	AE	CE	EE	0E	2E	4E	6E	A5	BD	DD	FD	1D	3D	5D	7D
B4	CC	EC	OC	2C	4C	6C	8C	C3	DB	FB	1B	3B	5B	7B	9B
D2	EA	OA	2A	4A	6A	8A	AA	E1	F9	19	39	59	79	99	B9
F0	08	28	48	68	88	A8	C8	FF	17	37	57	77	97	B7	D7

Cycling through the palettes using an adder with the default value of \$0F(15).

Unlike when we use \$10, the palette will never cycle around so using this uneven adder we will always get a new, if not always aesthetically pleasing, result.

Now that we have a grasp of how our colors are populated, by using a value between 0 and 7 as an index into an 8 byte array containing our colour palette, we can start to take a look at how the screen itself is populated with these colours. A good example that will help us unpick the machinery involved is the banner screen. The data used to populate the banner screen consists of 3 arrays and stored in `foregroundPixelData`. The end result, when the banner is displayed on screen, looks as follows:

```
$7000: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$7028: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$7050: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$7078: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$70A0: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$70C8: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$70F0: 11100022200300004440500056666000777700111000000000000000000000000000000000000000000000000000000000000000000000000  
$7118: 000002000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$7140: 000000200000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$7168: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$7190: 000000200000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$71B8: 000000200000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$71E0: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$7208: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$7230: 23456712345671234567123456712345671234567123456712345671234567123456712345671234567123456712345671234567123  
$7258: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$7280: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$72A8: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$72D0: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$72F8: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$7320: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$7348: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$7370: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$7398: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$73C0: 011010000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$73E8: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$7410: 011011000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$7438: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$7460: 765432000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$7488: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$74B0: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$74D8: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$7500: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$7528: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$7550: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
$7578: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

The 35 lines of screen RAM. Each line is 40 bytes wide. Each half of each byte references a separate color.  
The RAM starts at \$7000 in memory.

Our starting point is to wonder how this is achieved. The answer lies in the use to which we put each of 3 arrays. Each array is 1000 bytes long. The first contains the X co-ordinate of the pixel we want to paint:

```
$00 , $00 , $00 , $00 , $00 , $01 , $02 , $03  
$01 , $02 , $03 , $04 , $04 , $06 , $06 , $06  
$06 , $06 , $07 , $08 , $09 , $07 , $08 , $09
```

The second contains the Y co-ordinate:

```
$07 , $08 , $09 , $0A , $0B , $0C , $0C , $0C  
$06 , $06 , $06 , $07 , $0B , $07 , $08 , $09  
$0A , $0B , $0C , $0C , $0C , $06 , $06 , $06
```

And the third contains the index into our colour palette:

```
$01 , $01 , $01 , $01 , $01 , $01 , $01 , $01  
$01 , $01 , $01 , $01 , $01 , $02 , $02 , $02  
$02 , $02 , $02 , $02 , $02 , $02 , $02 , $02
```

On the following two pages we give a full account of each array and illustrate the progressive rendering of the banner as each line in the arrays is consumed.



	C	O	C	O'
COL	COLP	COLO'	COLOL	COLOU'
COLOUR	COLOUR'	COLOURS	COLOURS	COLOURSI
COLOURSP	COLOURSP^	COLOURSP^	COLOURSP^	COLOURSP^
COLOURSP▲	COLOURSP▲	COLOURSP▲	COLOURSP▲C	COLOURSP▲CL
COLOURSPACE:	COLOURSPACE: A	COLOURSPACE: A L	COLOURSPACE: A LI	COLOURSPACE: A LIC'
COLOURSPACE: A LICP	COLOURSPACE: A LIGHT	COLOURSPACE: A LIGHT S	COLOURSPACE: A LIGHT SY	COLOURSPACE: A LIGHT SWT
COLOURSPACE: A LIGHT SWT	COLOURSPACE: A LIGHT SYNT	COLOURSPACE: A LIGHT SYNTH	COLOURSPACE: A LIGHT SYNTHE	COLOURSPACE: A LIGHT SYNTHES
COLOURSPACE: A LIGHT SYNTHES	COLOURSPACE: A LIGHT SYNTHESI	COLOURSPACE: A LIGHT SYNTHESISE	COLOURSPACE: A LIGHT SYNTHESISE	COLOURSPACE: A LIGHT SYNTHESISER b'
COLOURSPACE: A LIGHT SYNTHESISER bq .	COLOURSPACE: A LIGHT SYNTHESISER bq .	COLOURSPACE: A LIGHT SYNTHESISER bq J.	COLOURSPACE: A LIGHT SYNTHESISER bq Je	COLOURSPACE: A LIGHT SYNTHESISER bq Je'
COLOURSPACE: A LIGHT SYNTHESISER bq Je[	COLOURSPACE: A LIGHT SYNTHESISER bq Je[r	COLOURSPACE: A LIGHT SYNTHESISER bq Je]]	COLOURSPACE: A LIGHT SYNTHESISER bq Jeff	COLOURSPACE: A LIGHT SYNTHESISER bq Jeff ]
COLOURSPACE: A LIGHT SYNTHESISER bq Jeff M	COLOURSPACE: A LIGHT SYNTHESISER bq Jeff M-	COLOURSPACE: A LIGHT SYNTHESISER bq Jeff Mu	COLOURSPACE: A LIGHT SYNTHESISER bq Jeff Min	COLOURSPACE: A LIGHT SYNTHESISER bq Jeff Minl
COLOURSPACE: A LIGHT SYNTHESISER bq Jeff Mint	COLOURSPACE: A LIGHT SYNTHESISER bq Jeff Mint-	COLOURSPACE: A LIGHT SYNTHESISER bq Jeff Minter	COLOURSPACE: A LIGHT SYNTHESISER bq Jeff Minter	COLOURSPACE: A LIGHT SYNTHESISER bq Jeff Minter
COLOURSPACE: A LIGHT SYNTHESISER bq Jeff Minter	COLOURSPACE: A LIGHT SYNTHESISER bq Jeff Minter	COLOURSPACE: A LIGHT SYNTHESISER bq Jeff Minter	COLOURSPACE: A LIGHT SYNTHESISER bq Jeff Minter	COLOURSPACE: A LIGHT SYNTHESISER bq Jeff Minter
<b>COLOURSPACE: A LIGHT SYNTHESISER bq Jeff Minter</b>				

Each screenshot represents the state of the screen after a line of each of the arrays on the facing page has been processed. The first array provides the X position of the pixel, the second the Y position, and the last array the color to paint.

In order to convert the three arrays of X positions, Y positions and colour values into dots on the screen runs a little piece of code as follows on each trio of values from the arrays as follows:

```
LDA (foregroundPixelsLoPtr),Y
..
PaintTheForegroundPixels
; Get the X position for this pixel.
CLC
ADC drawForegroundAtXPos
STA previousPixelXPosition

; Get the Y position for this pixel.
LDA foregroundPixelsHiPtr
PHA
CLC
ADC #$04
STA foregroundPixelsHiPtr
LDA (foregroundPixelsLoPtr),Y
CLC
ADC drawForegroundAtYPos
STA previousPixelYPosition

; Get the color value for this pixel.
LDA foregroundPixelsHiPtr
CLC
ADC #$04
STA foregroundPixelsHiPtr
LDA (foregroundPixelsLoPtr),Y
ORA #$40
STA currentPaintState

; Now paint the pixel.
JSR PaintPixelForCurrentSymmetry
```

This plucks the X, Y, and colour values from each array and writes them to the corresponding position in our 40 by 35 table of values starting at position \$7000 in RAM. This is the table we visualized three pages previously.

However, we are not done yet. The RAM at \$7000 is just a holding pen - it does not represent in full what will get displayed on the screen. This is because, unlike the Commodore 64, the Atari 800 has a much more complicated system than mere RAM up its sleeve. This system offers enormous flexibility, which we will see Colourspace take full advantage of, but it comes at the price of providing the programmer, and you the lay reader, something of a headache.

The core of this apparently perverse system is something called a 'Display List'. This is a *list* of commands that tell the Atari hardware what to *display* on screen. The commands are written in their own inscrutable little language of hexadecimal values. In

order to fill a screen with pixels you must add commands to the 'Display List' that tells the Atari graphics processor what to put where, and where to get it from. A typical entry that we might add to the Display List might look like the following:

```
$4F $F070
```

This is mysterious-looking to say the least. However there is a potential clue in there. Doesn't \$F070 remind you of something, especially when you recall us mentioning that two-byte addresses often have their order reversed so that the CPU can process the least significant byte first? If we reverse \$F070 we get \$70F0, which I hope you remember is very close to the address in RAM that we stored our pixel data, \$7000. (It was only a few paragraphs ago after all.)

So the entry in the display list is doing something with one of the rows of our pixel data, this one:

```
$70F0: 01110002220030000044400500050666600077770011110000020000000033300444400000000000
```

If you refer to the full figure given a few pages previously you'll see this is the first row of the 'Colourspace' banner. So whatever we are doing here we are probably doing it with the whole of this row of pixel data. We have no way of independently divining the meaning of the byte \$4F that precedes \$F070. So we have to look it up, possibly resorting to photostats of an out-of-print manual from the early 1980s. There we find that \$4F means 'Read 40 bytes from the following address'. So from that we can now deduce that the full meaning of our *Display List* command is: 'Read 40 bytes from the address \$70F0 and write their colour values as a row of pixels to the screen'.

Now that we know what a single *Display List* command can look like we have to figure out how they might be used in aggregate. The straightforward answer is that we issue the commands in the order that we expect them to be executed, from the top of the screen down to the very bottom. On the following two pages we'll take a look at the *Display List* used to render our title banner in its plainest possible mode, with it filling the screen more or less as we would expect. Since the screen is 280 pixels high and each command writes a single line of pixels we must issue at least 280 commands in total to cover the screen. You'll even notice we have a shorthand command available to us for writing blank lines which we use at the very start: \$70, which writes 8 completely blank lines.

## Display List: Variable Resolution Mode

## Display List Entries: 1-61

Display List Entries: 62-122

### Display List Entries: 123-182



Figure 13.2: 'Variable Resolution' Display Mode

Notice that on the page opposite we are writing each line from our pixel data a total of five times. This repetition means that each 'line' is 5 pixels high, which is just enough to fill the screen more or less in its entirety. This number is configurable: we can use the Z key and SHIFT + Z to increase or decrease the number of pixel lines we write for each row, making the banner smaller or taller as required. With a 'Variable Resolution' of 4 for example, we will only write each line 4 times instead of 5. The current value is stored in `numberOfLinesToDraw` and referenced when we're generating our Display List commands for each row in the table. It will keep writing out the same command until `numberOfLinesToDraw` (copied to X) has reached zero:

```

WriteDisplayListLoop
    LDX  numberOfLinesToDraw      ; Load number of lines to X.
_Loop   LDA  #$4F              ; Our $4F command.
        JSR  WriteValueToDisplayList
        LDA  foregroundPixelsLoPtr
        JSR  WriteValueToDisplayList
        LDA  foregroundPixelsHiPtr
        JSR  WriteValueToDisplayList
        DEX      ; Decrement X.
        BNE  _Loop   ; Keep looping until we've drawn all lines.

```

This is pretty straightforward, but since we can control the number of pixel lines we display for each row in our data, other effects are possible. It's safe to say Jeff Minter has a pretty good go at exploring this potential to its fullest. So let's take a look at what he gets up to.





Figure 13.3: 'Hi-Resolution + Hard Reflect' Display Mode

For this mode the trick is to simply write one pixel line for each row in our RAM. Note that in the following loop we only write the address of the row once, rather than use an inner loop to write it multiple times:

```

UseHiResHardReflectMode
    LDX #\$59
_Loop   JSR WriteValuesFromMemoryToDisplayList
        LDA foregroundPixelsLoPtr
        CLC
        ADC #NUM_COLS_40
        STA foregroundPixelsLoPtr
        LDA foregroundPixelsHiPtr
        ADC #$00
        STA foregroundPixelsHiPtr
        DEX
        BNE _Loop

```

This takes care of the banner at the top. For the banner at the bottom we reverse the order in which we write the rows so that it appears reflected. We achieve this by decrementing `foregroundPixelsLoPtr` and `foregroundPixelsHiPtr` back down from their peak of \$7DE8 (highlighted in red in the Display List opposite).

Of course something so simple requires a bit of know-how when it comes to 6502 assembly programming. To subtract \$18 (40 in decimal) from \$7DE8 involves two separate, carefully crafted operations. The first is to actually add a value rather than subtract it from the variable containing the \$E8 part of \$7DE8. By adding \$D8 it will cycle

## painting pixels is so complicated

---

around (like a modulus operation on a byte's maximum value of 255) to \$70. In other words by adding 216 (\$D8) to 232 (\$E8), we end up with 192 (\$70) which is 40 less, our desired result.

```
LDA foregroundPixelsLoPtr
CLC
ADC #$D8
STA foregroundPixelsLoPtr
```

The virtue of this operation lies in its effect when we come to 'subtract' from `foregroundPixelsHiPtr`. By incrementing using a value of \$D8, overflowing the maximum value of \$FF, and thus cycling around to \$70 (our intended result of 40 less than \$E8) we created a side-effect in the 6502 CPU, specifically we caused it to set something called the `carry bit`. With this set, the next time we add something to the A register (i.e. do an ADC operation), it will add an extra 1 to our result. This means that if we add \$FF (255) to our `foregroundPixelsHiPtr` of \$7D it will initially cycle back around to \$7C but the carry bit will add an extra 1 giving us a final result of \$7D - where we started:

```
LDA foregroundPixelsHiPtr
ADC #$FF
STA foregroundPixelsHiPtr
```

This outcome is correct for our current example, \$7DC0 is 40 less than \$7DE8 so we needed this carry-bit to do its magic in order to keep the right value of \$7D in our `foregroundPixelsHiPtr`.

The usefulness of this technique becomes apparent when we find ourselves in the complementary situation. In other words, when we add \$D8 to `foregroundPixelsLoPtr` and find that it does not need to cycle past the maximum value of \$FF in a byte. When this happens, the carry bit does not get set. This first happens when we have reached a value of \$7D20 for the entry in our display list. Adding \$D8 to \$20 gives us \$F8. The carry bit has not been set, so when we add \$FF to \$7D on this occasion, the CPU does not bump up the result by 1. Instead, our result is \$7C. Again, this is the outcome we wanted since we now have \$7CF8 as our result, which is 40 less than \$7D20. Here is our little loop tying all this together:

```
_Loop JSR WriteValuesFromMemoryToDisplayList
      LDA foregroundPixelsLoPtr
      CLC
      ADC #$D8
      STA foregroundPixelsLoPtr
      LDA foregroundPixelsHiPtr
      ADC #$FF
      STA foregroundPixelsHiPtr
      DEX
      BNE _Loop
```





Figure 13.4: 'Curved Colourspace 1' Display Mode

This mode plays with the same fundamentals as the 'Variable Resolution Mode' we looked at earlier: varying the number of pixel rows we allocate for each row of our pixel data. It consumes the pixel data four rows at a time and writes each row an incrementally increasing number of times. So for example the first four rows of data are written just once each:

Bytes	Description
\$4F \$0070	Read 40 bytes from: \$7000
\$4F \$2870	Read 40 bytes from: \$7028
\$4F \$5070	Read 40 bytes from: \$7050
\$4F \$7870	Read 40 bytes from: \$7078

However the next four rows of data are written twice each:

Bytes	Description
\$4F \$A070	Read 40 bytes from: \$70A0
\$4F \$A070	Read 40 bytes from: \$70A0
\$4F \$C870	Read 40 bytes from: \$70C8
\$4F \$C870	Read 40 bytes from: \$70C8

Bytes	Description
\$4F \$F070	Read 40 bytes from: \$70F0
\$4F \$F070	Read 40 bytes from: \$70F0
\$4F \$1871	Read 40 bytes from: \$7118
\$4F \$1871	Read 40 bytes from: \$7118

Each chunk of 4 rows is incremented in this way, with the subsequent 4 being written three times each, and so on up to a maximum of ten times each. Here is the routine that pulls this together:

```

CurvedColorspaceMode
    LDA #$01
    STA numberOfRowsToWrite
    LDA #$00
    STA bottomMostYPos

CurvedColorSpaceLoop
    LDX #$04

CurvedColorSpaceInnerLoop
    LDA numberOfRowsToWrite
    STA numberOfRowsOfCurvedPixelLines

_Loop    JSR WriteValuesFromMemoryToDisplayList
        DEC numberOfRowsOfCurvedPixelLines
        BNE _Loop

        LDA foregroundPixelsLoPtr
        CLC
        ADC #NUM_COLS_40
        STA foregroundPixelsLoPtr

        LDA foregroundPixelsHiPtr
        ADC #$00
        STA foregroundPixelsHiPtr

        INC bottomMostYPos
        DEX
        BNE CurvedColorSpaceInnerLoop

        INC numberOfRowsToWrite
        LDA numberOfRowsToWrite
        CMP #$0A
        BNE CurvedColorSpaceLoop

```





Figure 13.5: 'Curved Colourspace 2' Display Mode

This display mode is more elaborate than any of our previous. Like 'Curved Colourspace' it varies the height of each pixel row but this time it is creating the effect of a convex curved surface, as though the pixels are being painted on the surface of a cylinder lying on its side. This means generating a gradually increasing number of lines per pixel row, following by a number of lines that gradually decrease. The colour coding in our display list table on the previous page hopefully helps make this clear.

Our routine for drawing the top half is relatively straightforward. We iterate from 1 to 10 and write an increasing number of lines for each RAM address to the display list at each iteration, we are only ever writing lines for 2 addresses (specified by `numberOfPixelRows`):

```

; Number of rows to write in each segment.
LDX #$02
STX numberOfPixelRows

TopHalfOfCurvedColourspace
    ; Start at 1 and increment up to 10.
    LDA #$01
    STA numberOfLinesToDelete

    ; Our loop for drawing the top half.
LoopToDeleteTopHalf
    LDX numberOfPixelRows

    ; Our inner loop for drawing a segment of lines.
DrawSegmentLoop

```

## painting pixels is so complicated

---

```
LDA numberOfLinesToDraw
STA numberOfCurvedPixelLines

; Write the required number of lines to the display list.
_Loop JSR WriteValuesFromMemoryToDisplayList
DEC numberOfCurvedPixelLines
BNE _Loop

; Increment the address we write to the display list
LDA foregroundPixelsLoPtr
CLC
ADC #NUM_COLS_40
STA foregroundPixelsLoPtr

LDA foregroundPixelsHiPtr
ADC #$00
STA foregroundPixelsHiPtr

; Keep looping until we draw the full segment
DEX
BNE DrawSegmentLoop

; Keep looping until we draw all ten segments of the top half
INC numberOfLinesToDraw
LDA numberOfLinesToDraw
CMP #$0A
BNE LoopToDrawTopHalf
```

For the bottom half we reverse the process by starting at 10 as our `numberOfLinesToDraw` and decrementing to 0:

```
DEC numberOfLinesToDraw

; Our loop for drawing the top half.
LoopToDrawBottomHalf
LDX numberOfRowsPixelRows

; Our inner loop for drawing a segment of lines.
DrawSegmentLoop
LDA numberOfLinesToDraw
STA numberOfCurvedPixelLines

; Write the required number of lines to the display list.
_Loop JSR WriteValuesFromMemoryToDisplayList
DEC numberOfCurvedPixelLines
BNE _Loop

; Increment the address we write to the display list
LDA foregroundPixelsLoPtr
CLC
ADC #NUM_COLS_40
STA foregroundPixelsLoPtr
```

```

LDA foregroundPixelsHiPtr
ADC #$00
STA foregroundPixelsHiPtr

DEX
BNE DrawSegmentLoop
DEC numberOfRowsLinesToDraw
BNE LoopToDrawBottomHalf

```

Our next effect can reuse a lot of this code with some small adjustment. Like 'Curved Colourspace 2', 'Curve + Hard Reflect' increments and decrements the number of lines drawn for each row to achieve a curved surface effect, but it has the additional twist of reflecting the top half in the bottom half. So the only thing we need to change are the addresses we reference when writing the display list entries for the bottom half. To do this we reuse the code above but instead decrement the addresses rather than incrementing them using the same trick we covered earlier:

```

DEC numberOfRowsLinesToDraw

; Our loop for drawing the top half.
LoopToDrawBottomHalf
LDX numberOfRowsPixelRows

; Our inner loop for drawing a segment of lines.
DrawSegmentLoop
LDA numberOfRowsLinesToDraw
STA numberOfRowsCurvedPixelLines

; Write the required number of lines to the display list.
_Loop
JSR WriteValuesFromMemoryToDisplayList
DEC numberOfRowsCurvedPixelLines
BNE _Loop

; Decrement, rather than increment, the address
; we write to the display list.
LDA foregroundPixelsLoPtr
CLC
ADC #$D8
STA foregroundPixelsLoPtr

LDA foregroundPixelsHiPtr
ADC #$FF
STA foregroundPixelsHiPtr

DEX
BNE DrawSegmentLoop
DEC numberOfRowsLinesToDraw
BNE LoopToDrawBottomHalf

```





Figure 13.6: 'Curved + Hard Reflect' Display Mode

As we can see above, this has the desired result of a curved surface with a horizontal reflection axis in the centre of the screen.

For our next trick we will find yet another way of reusing these elements of incremental line width and reflection to create a waved surface. To help achieve this we will set `numberOfPixelRows` to 1 instead of 2, so that we increment the number of lines we draw for each individual address rather than for every second one.

Bytes	Description
\$4F \$0070	Read 40 bytes from: \$7000
\$4F \$2870	Read 40 bytes from: \$7028
\$4F \$2870	Read 40 bytes from: \$7028
\$4F \$5070	Read 40 bytes from: \$7050
\$4F \$5070	Read 40 bytes from: \$7050
\$4F \$5070	Read 40 bytes from: \$7050

This will mean that we reach the maximum of 10 lines per row much sooner, so will also decrement back to zero again much sooner as well. When we've finished incrementing and decrementing we will only have painted half the screen.





Figure 13.7: 'Hoopy 4X CurvyReflex' Display Mode

You can see this in the result above - we have performed our loop and only managed to paint a reflected section of the top of quarter of the Colourspace title screen. This means that to complete our effect we must perform the entire operation again in order to fill out the screen completely. This is controlled by a variable called `numberOfLoopsToRun`. By setting this variable to 2 and `numberOfPixelRows` to 1 we can reuse the logic (and even the code) we explained above to achieve this variation on our previous effects.

```
HoopyCurvyMode
    LDA #$01
    STA hardReflectEnabled
    LDA #$01
    STA numberOfRowsPixelRows
    LDA #$02
    STA numberOfLoopsToRun
    JSR CurvedColorspace2Mode
    RTS
```

The routine `CurvedColorspace2Mode` above is the one we have used in the previous two modes, by setting its parameters slightly differently we have been able to reuse it to achieve this distinct 'surface wave' effect.

The final display mode in Colourspace is a variation on the very first one we covered. This again is a hard reflection of the display across the central horizontal axis but with an interlacing of scan lines that has a distinctly 1980s feel.

## Display List: Zarjax Interlace Resolution

Bytes	Description
\$70	Draw 8 Blank Lines
\$70	Draw 8 Blank Lines
\$46 \$C950	Read 40 bytes from: \$50C9
\$90	Draw 2 Blank Lines
\$4F \$0070	Read 40 bytes from: \$7000
\$4F \$E87D	Read 40 bytes from: \$7DE8
\$4F \$2870	Read 40 bytes from: \$7028
\$4F \$007D	Read 40 bytes from: \$7DC0
\$4F \$5070	Read 40 bytes from: \$7050
\$4F \$987D	Read 40 bytes from: \$7D98
\$4F \$7870	Read 40 bytes from: \$7078
\$4F \$707D	Read 40 bytes from: \$7D70
\$4F \$A070	Read 40 bytes from: \$70A0
\$4F \$487D	Read 40 bytes from: \$7D48
\$4F \$C870	Read 40 bytes from: \$70C8
\$4F \$207D	Read 40 bytes from: \$7D20
\$4F \$F070	Read 40 bytes from: \$70F0
\$4F \$F87C	Read 40 bytes from: \$7CF8
\$4F \$1871	Read 40 bytes from: \$7118
\$4F \$D07C	Read 40 bytes from: \$7CD0
\$4F \$A071	Read 40 bytes from: \$7140
\$4F \$A87C	Read 40 bytes from: \$7CA8
\$4F \$6871	Read 40 bytes from: \$7168
\$4F \$807C	Read 40 bytes from: \$7C80
\$4F \$9071	Read 40 bytes from: \$7190
\$4F \$587C	Read 40 bytes from: \$7C58
\$4F \$B871	Read 40 bytes from: \$71B8
\$4F \$307C	Read 40 bytes from: \$7C30
\$4F \$E071	Read 40 bytes from: \$71E0
\$4F \$087C	Read 40 bytes from: \$7C08
\$4F \$0872	Read 40 bytes from: \$7208
\$4F \$E07B	Read 40 bytes from: \$7BE0
\$4F \$3072	Read 40 bytes from: \$7230
\$4F \$B87B	Read 40 bytes from: \$7BB8
\$4F \$5872	Read 40 bytes from: \$7258
\$4F \$907B	Read 40 bytes from: \$7B90
\$4F \$8072	Read 40 bytes from: \$7280
\$4F \$687B	Read 40 bytes from: \$7B68
\$4F \$A872	Read 40 bytes from: \$72A8
\$4F \$407B	Read 40 bytes from: \$7B40
\$4F \$D072	Read 40 bytes from: \$72D0
\$4F \$187B	Read 40 bytes from: \$7B18
\$4F \$F872	Read 40 bytes from: \$72F8
\$4F \$F07A	Read 40 bytes from: \$7AF0
\$4F \$2073	Read 40 bytes from: \$7320
\$4F \$C87A	Read 40 bytes from: \$7AC8
\$4F \$4873	Read 40 bytes from: \$7348
\$4F \$A07A	Read 40 bytes from: \$7AA0
\$4F \$7073	Read 40 bytes from: \$7370
\$4F \$787A	Read 40 bytes from: \$7A78
\$4F \$9873	Read 40 bytes from: \$7398
\$4F \$507A	Read 40 bytes from: \$7A50
\$4F \$C073	Read 40 bytes from: \$73C0
\$4F \$287A	Read 40 bytes from: \$7A28
\$4F \$E873	Read 40 bytes from: \$73E8
\$4F \$007A	Read 40 bytes from: \$7A00
\$4F \$1074	Read 40 bytes from: \$7410
\$4F \$D879	Read 40 bytes from: \$79D8
\$4F \$3874	Read 40 bytes from: \$7438
\$4F \$B079	Read 40 bytes from: \$79B0
\$4F \$D074	Read 40 bytes from: \$7460

Display List Entries: 1-61

Bytes	Description
\$4F \$8879	Read 40 bytes from: \$7988
\$4F \$8874	Read 40 bytes from: \$7488
\$4F \$6079	Read 40 bytes from: \$7960
\$4F \$B074	Read 40 bytes from: \$74B0
\$4F \$3879	Read 40 bytes from: \$7938
\$4F \$D874	Read 40 bytes from: \$74D8
\$4F \$1079	Read 40 bytes from: \$7910
\$4F \$0075	Read 40 bytes from: \$7500
\$4F \$E878	Read 40 bytes from: \$78E8
\$4F \$2875	Read 40 bytes from: \$7528
\$4F \$S078	Read 40 bytes from: \$78C0
\$4F \$5075	Read 40 bytes from: \$7550
\$4F \$9878	Read 40 bytes from: \$7898
\$4F \$7875	Read 40 bytes from: \$7578
\$4F \$7078	Read 40 bytes from: \$7870
\$4F \$A075	Read 40 bytes from: \$75A0
\$4F \$4878	Read 40 bytes from: \$7848
\$4F \$C875	Read 40 bytes from: \$75C8
\$4F \$2078	Read 40 bytes from: \$7820
\$4F \$6876	Read 40 bytes from: \$7668
\$4F \$8077	Read 40 bytes from: \$7780
\$4F \$9076	Read 40 bytes from: \$7690
\$4F \$4076	Read 40 bytes from: \$7640
\$4F \$A877	Read 40 bytes from: \$77A8
\$4F \$6876	Read 40 bytes from: \$7668
\$4F \$8077	Read 40 bytes from: \$7780
\$4F \$9077	Read 40 bytes from: \$7708
\$4F \$E076	Read 40 bytes from: \$76E0
\$4F \$3077	Read 40 bytes from: \$7730
\$4F \$B876	Read 40 bytes from: \$76B8
\$4F \$5877	Read 40 bytes from: \$7758
\$4F \$S077	Read 40 bytes from: \$77D0
\$4F \$8077	Read 40 bytes from: \$7708
\$4F \$9076	Read 40 bytes from: \$7690
\$4F \$6876	Read 40 bytes from: \$7668
\$4F \$A877	Read 40 bytes from: \$77A8
\$4F \$4076	Read 40 bytes from: \$7640
\$4F \$D077	Read 40 bytes from: \$77D0
\$4F \$1876	Read 40 bytes from: \$7618
\$4F \$F877	Read 40 bytes from: \$77F8
\$4F \$F075	Read 40 bytes from: \$75F0
\$4F \$2078	Read 40 bytes from: \$7820
\$4F \$C875	Read 40 bytes from: \$75C8
\$4F \$4878	Read 40 bytes from: \$7848
\$4F \$A075	Read 40 bytes from: \$75A0
\$4F \$7078	Read 40 bytes from: \$7870
\$4F \$7875	Read 40 bytes from: \$7578
\$4F \$9878	Read 40 bytes from: \$7898
\$4F \$5075	Read 40 bytes from: \$7550
\$4F \$C078	Read 40 bytes from: \$78C0
\$4F \$2875	Read 40 bytes from: \$7528
\$4F \$E878	Read 40 bytes from: \$78E8
\$4F \$0075	Read 40 bytes from: \$7500
\$4F \$1079	Read 40 bytes from: \$7910
\$4F \$D874	Read 40 bytes from: \$74D8

Display List Entries: 62-122

Bytes	Description
\$4F \$3879	Read 40 bytes from: \$7938
\$4F \$B074	Read 40 bytes from: \$74B0
\$4F \$6079	Read 40 bytes from: \$7960
\$4F \$8874	Read 40 bytes from: \$7488
\$4F \$8879	Read 40 bytes from: \$7988
\$4F \$6074	Read 40 bytes from: \$7460
\$4F \$B079	Read 40 bytes from: \$79B0
\$4F \$3874	Read 40 bytes from: \$7438
\$4F \$D879	Read 40 bytes from: \$79D8
\$4F \$1074	Read 40 bytes from: \$7410
\$4F \$007A	Read 40 bytes from: \$7A00
\$4F \$E873	Read 40 bytes from: \$73E8
\$4F \$287A	Read 40 bytes from: \$7428
\$4F \$C073	Read 40 bytes from: \$73C0
\$4F \$2073	Read 40 bytes from: \$7230
\$4F \$6873	Read 40 bytes from: \$7668
\$4F \$8072	Read 40 bytes from: \$7280
\$4F \$9072	Read 40 bytes from: \$7200
\$4F \$4072	Read 40 bytes from: \$72D0
\$4F \$A872	Read 40 bytes from: \$72A8
\$4F \$687B	Read 40 bytes from: \$7B68
\$4F \$8072	Read 40 bytes from: \$7280
\$4F \$907B	Read 40 bytes from: \$7B90
\$4F \$6872	Read 40 bytes from: \$7B60
\$4F \$8077	Read 40 bytes from: \$7780
\$4F \$9077	Read 40 bytes from: \$7708
\$4F \$E076	Read 40 bytes from: \$76E0
\$4F \$3077	Read 40 bytes from: \$7730
\$4F \$B876	Read 40 bytes from: \$76B8
\$4F \$5877	Read 40 bytes from: \$7758
\$4F \$S077	Read 40 bytes from: \$77D0
\$4F \$8077	Read 40 bytes from: \$7708
\$4F \$9076	Read 40 bytes from: \$7690
\$4F \$6876	Read 40 bytes from: \$7668
\$4F \$A877	Read 40 bytes from: \$77A8
\$4F \$4076	Read 40 bytes from: \$7640
\$4F \$D077	Read 40 bytes from: \$77D0
\$4F \$1876	Read 40 bytes from: \$7618
\$4F \$F877	Read 40 bytes from: \$77F8
\$4F \$F075	Read 40 bytes from: \$75F0
\$4F \$2078	Read 40 bytes from: \$7820
\$4F \$C875	Read 40 bytes from: \$75C8
\$4F \$4878	Read 40 bytes from: \$7848
\$4F \$A075	Read 40 bytes from: \$75A0
\$4F \$7078	Read 40 bytes from: \$7870
\$4F \$7875	Read 40 bytes from: \$7578
\$4F \$9878	Read 40 bytes from: \$7898
\$4F \$5075	Read 40 bytes from: \$7550
\$4F \$C078	Read 40 bytes from: \$78C0
\$4F \$2875	Read 40 bytes from: \$7528
\$4F \$E878	Read 40 bytes from: \$78E8
\$4F \$0075	Read 40 bytes from: \$7500
\$4F \$1079	Read 40 bytes from: \$7910
\$4F \$D874	Read 40 bytes from: \$74D8

Display List Entries: 123-182

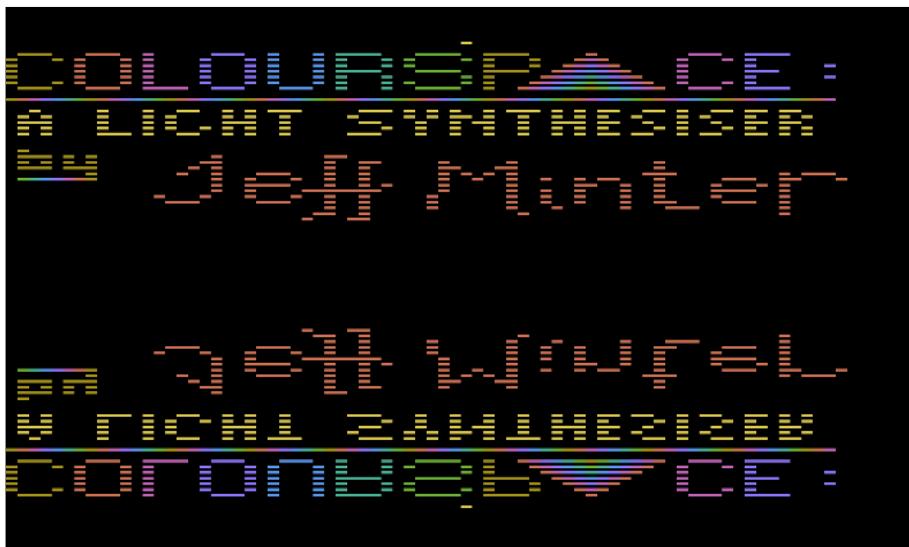


Figure 13.8: 'Zarjax Interlace Res' Display Mode

At first glance we would expect that we must recycle the same techniques we used earlier to achieve the horizontal reflection above - however closer inspection reveals this is not the case. Instead we are achieving the same result by interlacing rows from our RAM table starting at the top with rows from our RAM table starting from the bottom, incrementing the one and decrementing the other. So for example, we populate lines 1 and 3 with \$7000 and \$7028, while populating lines 2 and 4 with \$7DE8 and \$7DC0.

Bytes	Description
\$4F \$0070	Read 40 bytes from: \$7000
\$4F \$E87D	Read 40 bytes from: \$7DE8
\$4F \$2870	Read 40 bytes from: \$7028
\$4F \$C07D	Read 40 bytes from: \$7DC0

By continuing in this vain, incrementing the address we reference in odd-numbered lines and decrementing the address we reference in even-numbered lines we end up painting the same image in both directions. One is the right way up, the other is upside down. By limiting each to 90 lines we fill out the 180 lines of the screen with two interlaced images.

The routine that achieves this relatively compact:

```
ZarjazInterlaceMode
    ; Set up values to decrement from for our
    ; even numbered lines.
    LDA foregroundPixelsLoPtr
    CLC
    ADC #$E8
    STA zarjazLoPtr
    LDA foregroundPixelsHiPtr
    ADC #$0D
    STA zarjazHiPtr

    ; Loop 90 times (giving 180 lines in total).
    LDX #$5A
ZarjazLoop
    ; Write the odd-numbered (right way up) lines.
    JSR WriteValuesFromMemoryToDisplayList

    ; Write the even-numbered (upside down) lines.
    LDA #$4F
    JSR WriteValueToDisplayList
    LDA zarjazLoPtr
    JSR WriteValueToDisplayList
    LDA zarjazHiPtr
    JSR WriteValueToDisplayList

    ; Increment the odd-numbered lines.
    LDA foregroundPixelsLoPtr
    CLC
    ADC #NUM_COLS_40
    STA foregroundPixelsLoPtr
    LDA foregroundPixelsHiPtr
    ADC #$00
    STA foregroundPixelsHiPtr

    ; Decrement the even-numbered lines.
    LDA zarjazLoPtr
    CLC
    ADC #$D8
    STA zarjazLoPtr
    LDA zarjazHiPtr
    ADC #$FF
    STA zarjazHiPtr

    DEX
    BNE ZarjazLoop
```

Now that we've completed our tour of the various display modes used by Colourspace we have broken the back of what makes it so different from Psychedelia. As you have found, there are a number of levels of indirection when it comes to translating a relatively simple image table in RAM into a display list of graphics commands that the Atari 800 then paints to the screen. Our next step on this journey is to take a look at the way patterns are implemented in Colourspace. This will be more familiar territory

to us after covering the methods for generating them in Psychedelia.



# all the little lightforms

## Jeffrey Says



By using the unique Atari screen hardware and colour palette, the effect of the program is much improved. The difference between *Psychedelia* and *Colourspace* is as pronounced as the difference between a Mini and a Ferrari.

Using the Atari you can get curved screens, hardware reflections, interlace effects, stroboscopics, dynamic colourflows, and variable resolution screens.

What once were patterns are now *lightforms*. It won't take you long to realize from glancing through the figures in the following pages that the mechanics underlying the generation of lightforms is fundamentally unchanged from what we covered when picking apart *Psychedelia*. The same data structure is used and it is processed using the same set of routines we covered in '['all the pretty patterns'](#)'. The differences that there are lie solely in some of the implementation details in the routines themselves. We will cover the main ones briefly but there isn't much to detain us without risking repetition.

For that reason, this chapter and the ones that follow, will be not far removed from a glorified appendix. We won't have to strain our concentration too far and instead can just enjoy the improved effects in colour and complexity that *Colourspace* achieved thanks to the superior capability of the Atari 400/800 hardware.

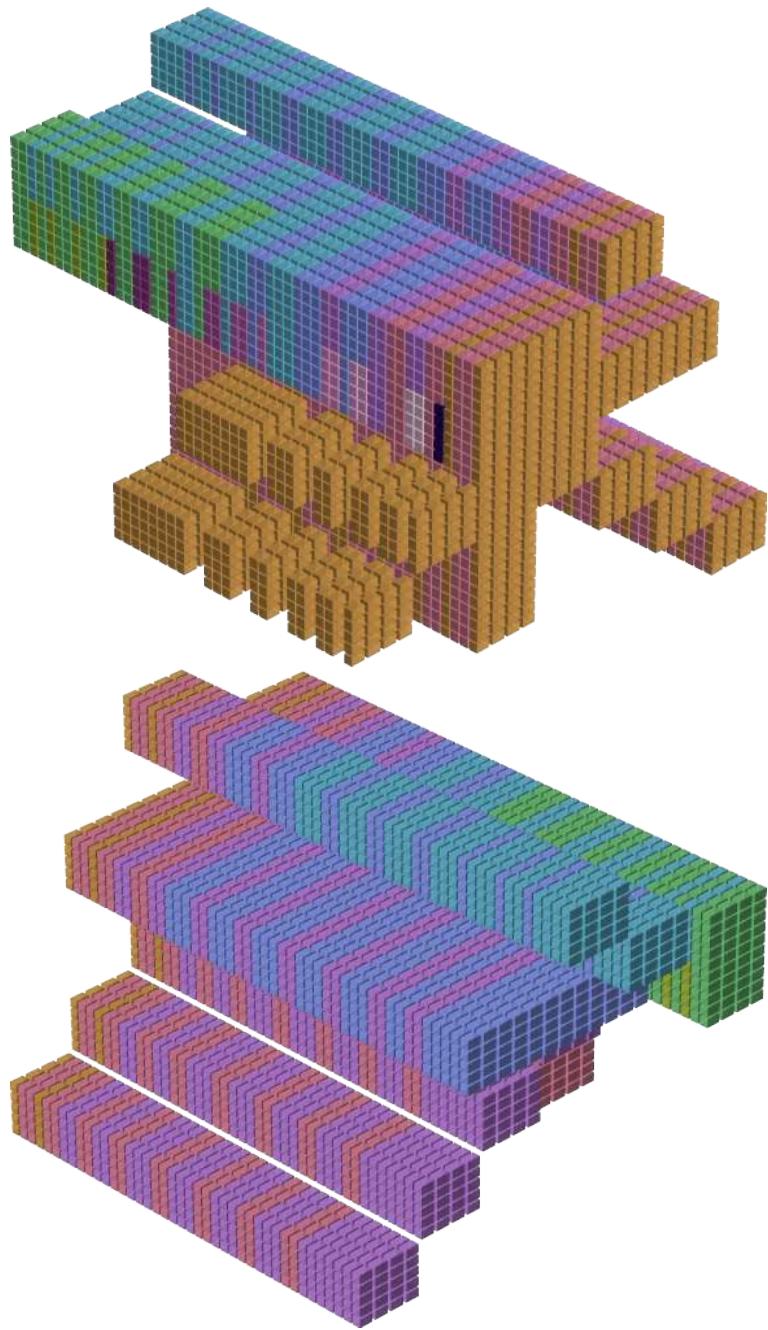


Figure 14.1: Evolution of the 'Twist' pattern.

```

theTwistXPosArray
    .BYTE $00,$55          ;      2
    .BYTE $01,$02,$55       ;     12
    .BYTE $01,$02,$03,$55   ;     333
    .BYTE $01,$02,$03,$04,$55 ;   654
    .BYTE $00,$00,$00,$55    ; 6 5 4
    .BYTE $FF,$FE,$55        ; 5   4
    .BYTE $55                  ;      4
theTwistYPosArray
    .BYTE $FF,$55
    .BYTE $FF,$FE,$55
    .BYTE $00,$00,$00,$55
    .BYTE $01,$02,$03,$04,$55
    .BYTE $01,$02,$03,$55
    .BYTE $01,$02,$55
    .BYTE $55

```

Listing 14.1: Source code for the Twist



Figure 14.2: Pattern Progression for 'The Twist'

## all the little lightforms

---

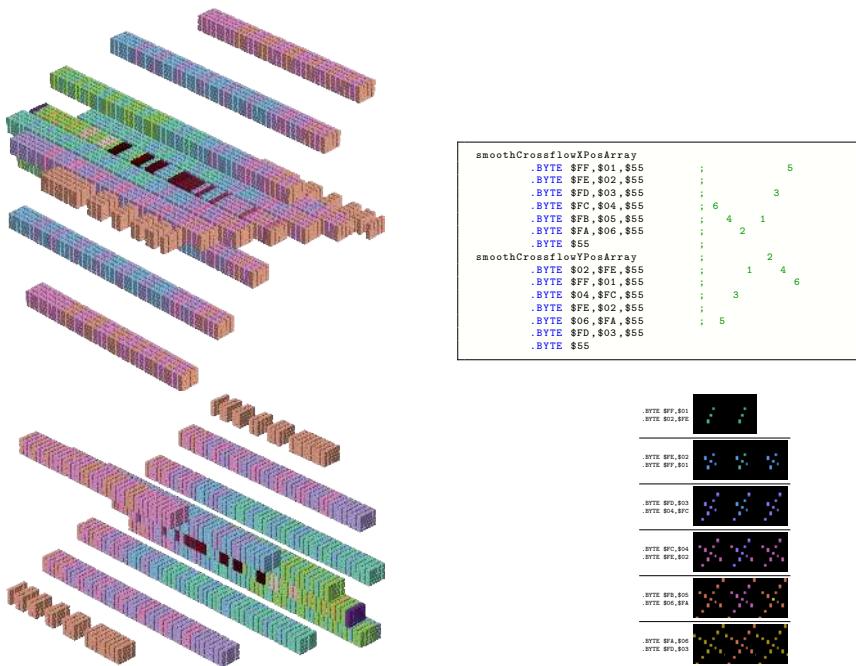


Figure 14.3: The 'Smooth Crossflow'.

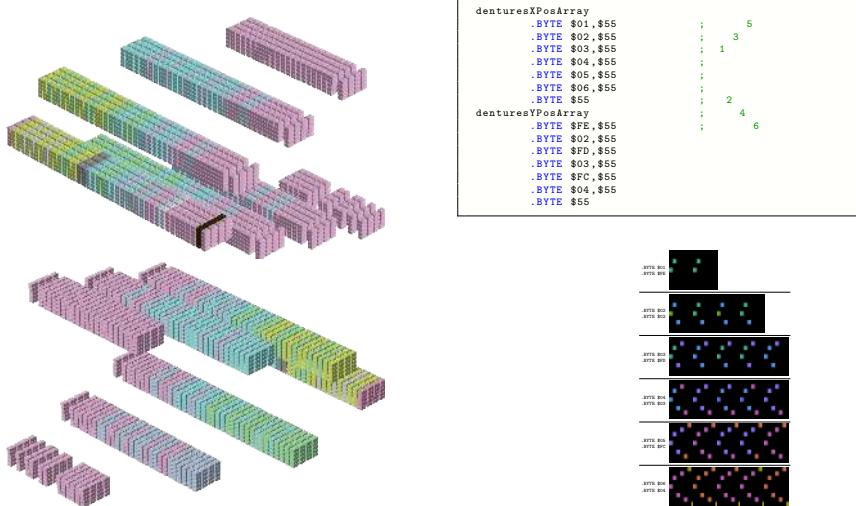
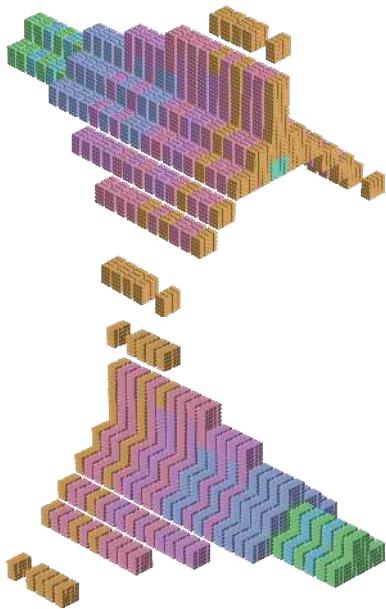


Figure 14.4: 'The Dentures'.



```
deltoidsXPosArray
.BYTE $00,$01,$55 ; 6
.BYTE $55
.BYTE $FF,$00,$01,$02,$55 ; 5
.BYTE $FD,$00,$03,$55 ; 4
.BYTE $FC,$00,$04,$55 ; 3
.BYTE $FA,$00,$06,$55 ; 313
.BYTE $55 ; 31 13
deltoidsYPosArray
.BYTE $00,$FF,$00,$55 ; 4 4
.BYTE $55 ; 5 5
.BYTE $00,$FF,$00,$55 ; 6 6
.BYTE $01,$FD,$01,$55
.BYTE $02,$FC,$02,$55
.BYTE $04,$FA,$04,$55
.BYTE $55
```

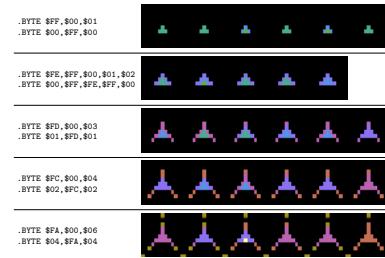
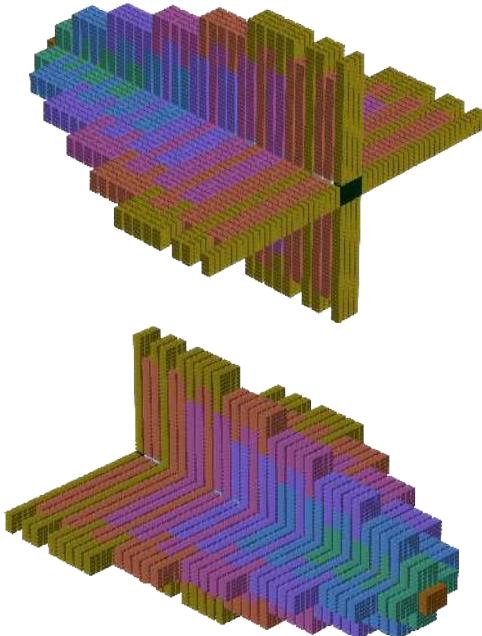


Figure 14.5: 'Deltoids'.



```
pulsarCrossXPosArray
.BYTE $00,$01,$00,$FF,$55 ; 6
.BYTE $00,$02,$00,$FF,$55 ; 5
.BYTE $00,$03,$00,$FD,$55 ; 4
.BYTE $00,$04,$00,$FC,$55 ; 3
.BYTE $00,$05,$00,$FB,$55 ; 2
.BYTE $00,$06,$00,$FA,$55 ; 1
.BYTE $55 ; 654321 123456
pulsarCrossYPosArray
.BYTE $FF,$00,$01,$00,$55 ; 1
.BYTE $FF,$00,$02,$00,$55 ; 2
.BYTE $FF,$00,$03,$00,$55 ; 3
.BYTE $FF,$00,$04,$00,$55 ; 4
.BYTE $FF,$00,$05,$00,$55 ; 5
.BYTE $FF,$00,$06,$00,$55 ; 6
.BYTE $55
```

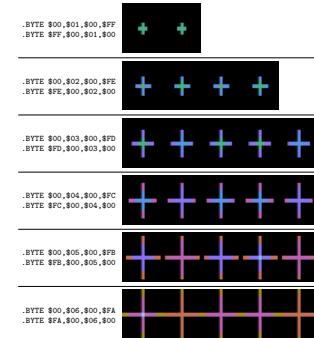


Figure 14.6: 'Pulsar Cross'.

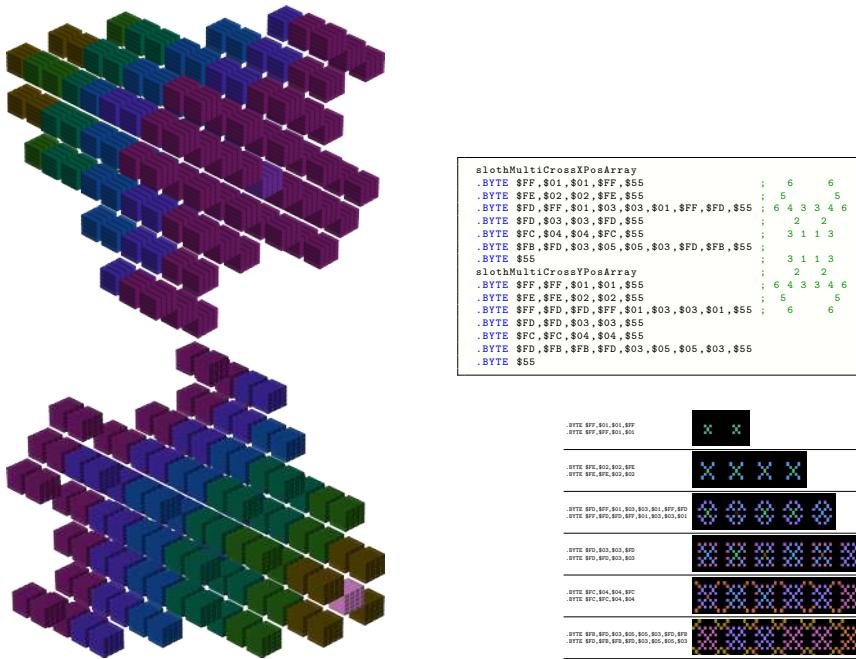


Figure 14.7: 'Sloth MultiCross'.

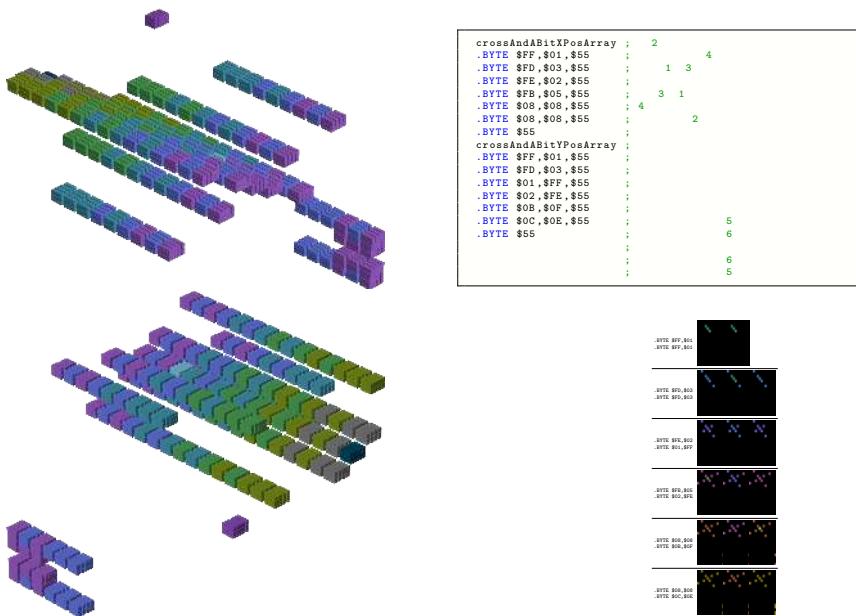
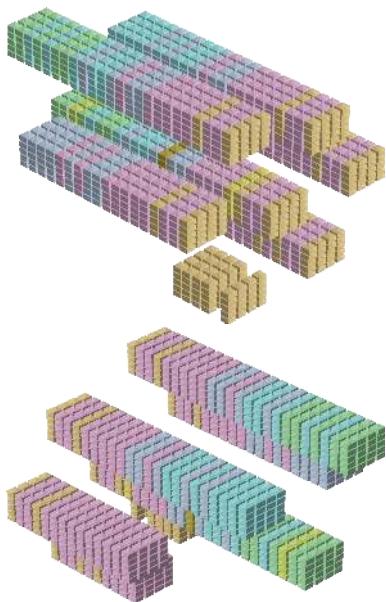


Figure 14.8: 'Cross And a Bit'.



```
star2XPosArray
.BYTE $FF,$55 ; 1
.BYTE $01,$55 ; 2
.BYTE $FE,$55 ; 3
.BYTE $02,$55 ; 4
.BYTE $01,$55 ; 5
.BYTE $FF,$55 ; 6 5
.BYTE $55
start2YPosArray
.BYTE $FD,$55
.BYTE $FF,$55
.BYTE $FF,$55
.BYTE $00,$55
.BYTE $02,$55
.BYTE $02,$55
.BYTE $55
```

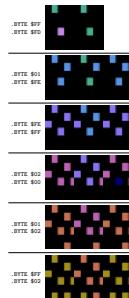
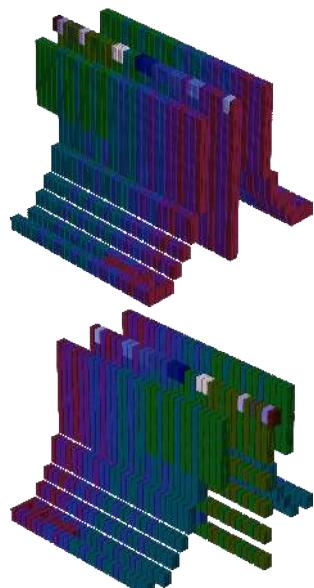


Figure 14.9: 'Star Two'.



```
userLightform0XPosArray
.BYTE $00,$00,$00,$00,$00,$00,$00,$55
; 2 2
.BYTE $04,$04,$04,$04,$04,$FC,$FC,$FC,$FC,$55
; 2 1 2
.BYTE $06,$07,$08,$0A,$55 ; 2 1 2
.BYTE $00,$00,$00,$55 ; 2 1 2
.BYTE $00,$55 ; 2 5 2
.BYTE $F7,$09,$55 ; 3 1 3
.BYTE $55 ; 3 4 3
userLightform0YPosArray
; 3 1 3
.BYTE $01,$03,$02,$05,$07,$09,$0B,$07
; 3 4 3
.BYTE $00,$01,$02,$03,$04,$00,$01,$02,$03,$04,$03
; 3 1 3
.BYTE $05,$06,$07,$08,$09,$0A,$0B,$0B,$05,$06,$07,$08,
$09,$0A,$0B,$0B,$03 ; 3 4 3
.BYTE $0A,$0B,$06,$01 ; 363 1 363
.BYTE $04,$FF
.BYTE $0B,$0B,$07
.BYTE $09,$09,$55
.BYTE $55
```

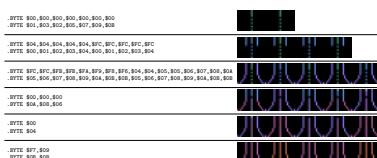


Figure 14.10: 'User LightForm 0'.

## all the little lightforms

---

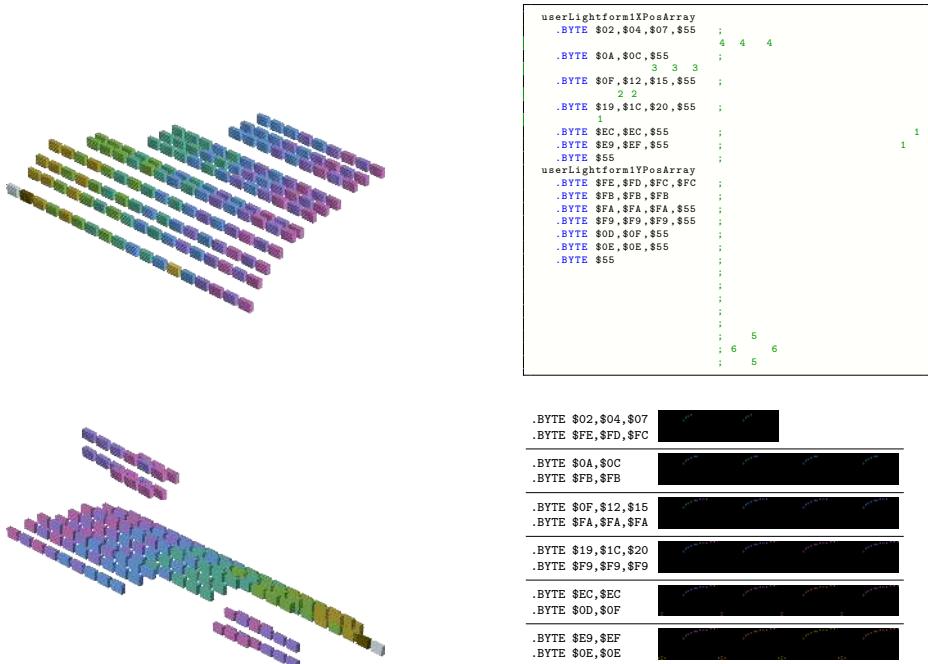


Figure 14.11: 'User LightForm 1'.

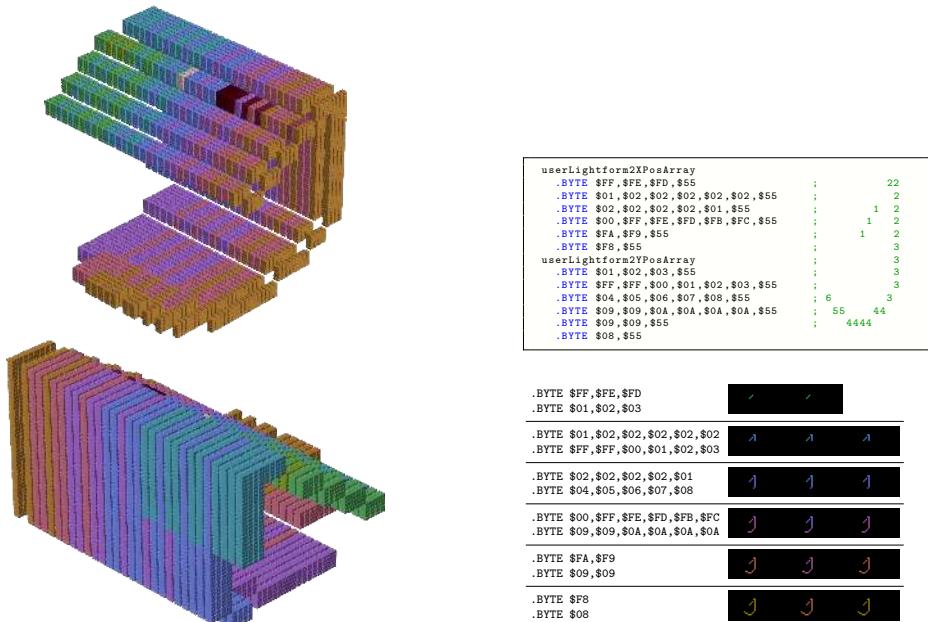
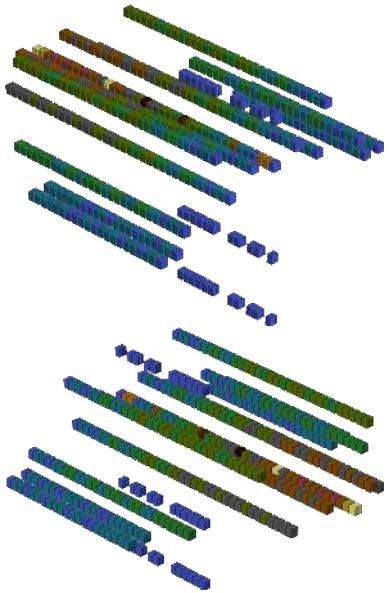


Figure 14.12: 'User LightForm 2'.

## all the little lightforms

---

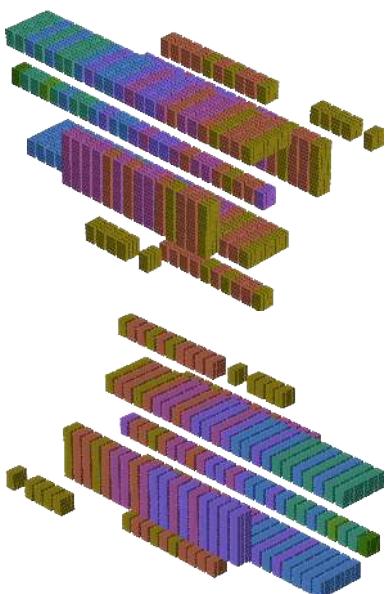


```

userLightform3XPosArray
.BYTE $FD,$03,$55 :      6
.BYTE $FA,$06,$55 :      3   3
.BYTE $FB,$07,$55 :      1   1
.BYTE $FC,$0C,$55 :      2   2
.BYTE $FD,$00,$00,$55 :  4
.userLightform3YPosArray
.BYTE $FF,$FF,$55 :      4
.BYTE $01,$01,$55 :      5   5
.BYTE $55 :               6
.BYTE $FC,$FC,$55 :      6
.BYTE $02,$02,$55 :      6
.BYTE $04,$04,$04,$55 :  6
.BYTE $FB,$0A,$11,$55 :  6
.BYTE $55 :               6
.BYTE $FD,$03,$55 :      6
.BYTE $FA,$06,$55 :      6
.BYTE $FB,$07,$55 :      6
.BYTE $FC,$FC,$55 :      6
.BYTE $FD,$00,$00,$55 :  6
.BYTE $FB,$0A,$11,$55 :  6
.BYTE $55 :               6

```

Figure 14.13: 'User LightForm 3'.



```

userLightform4XPosArray
.BYTE $FF,$00,$01,$55 :      5
.BYTE $FF,$00,$01,$55 :      5
.BYTE $04,$04,$04,$55 :      5
.BYTE $FC,$FC,$FC,$55 :      111
.BYTE $00,$00,$55 :      111
.BYTE $FB,$08,$55 :      4   3
.userLightform4YPosArray
.BYTE $FD,$FD,$FD,$FD,$55 :  6   4   3   6
.BYTE $03,$03,$03,$55 :      4   4
.BYTE $FF,$00,$01,$55 :      222
.BYTE $FF,$00,$01,$55 :      222
.BYTE $FA,$06,$55 :      5
.BYTE $00,$00,$55 :      5

```

Figure 14.14: 'User LightForm 4'.

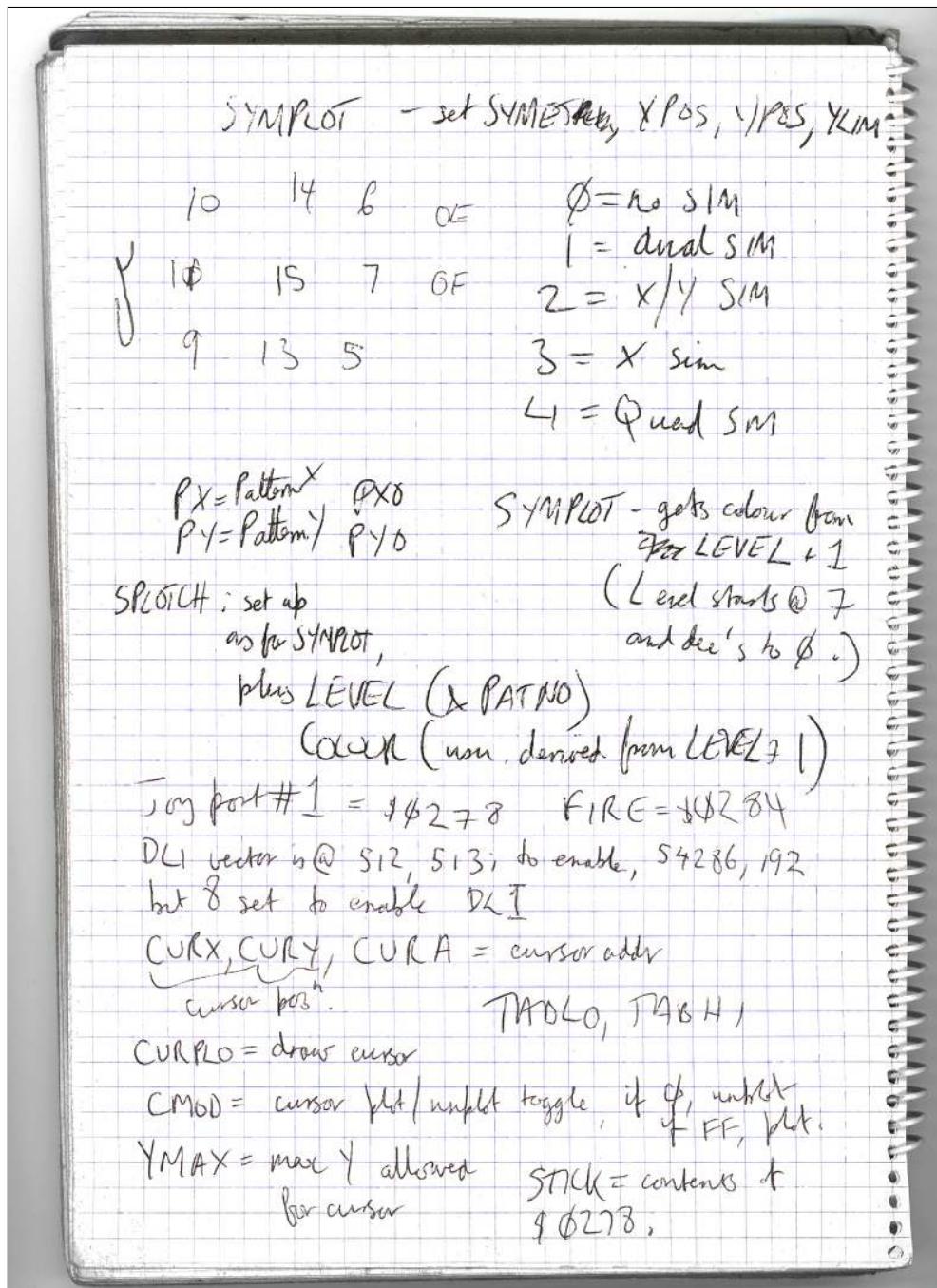


Figure 14.15: From Jeff Minter's development notebook for Colourspace

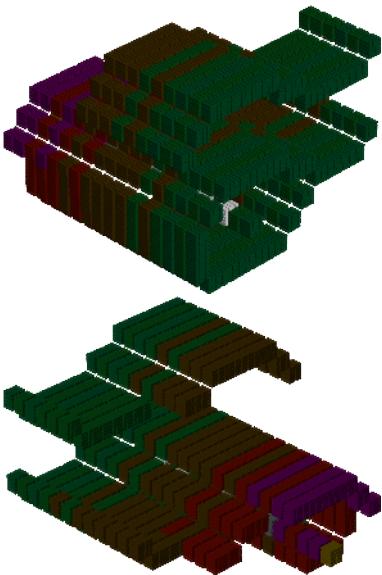
At first glance this notebook could be for Psychedelia as much as for Colourspace. However there is a tell-tale reference to 'dlists', i.e. 'display lists', which are a feature of the Atari 800 architecture rather than the Commodore series of computers.

**First Half.** SYMPLOT: The first half of this excerpt from from Jeff Minter's development notes sketch out some initial details on the routine we call `PaintPixelForCurrentSymmetry`, and which he calls SYMPLOT (i.e. 'symmetry plot'). As he says, the purpose of the routine is to 'set symm[etry] using the `XPOS` (`pixelXPosition`) and `YPOS` (`pixelYPosition`).

The final routine makes no use of a `YLIM` variable, instead it looks at `#NUM_ROWS` as a constant value to determine the bottom of the screen. The extract also notes that SYMPLOT gets its 'colour from LEVEL + 1 (Level starts @ 7 and dec's to 0)'. This is the mechanic we described in detail in '[soul of a light machine](#)'.

## all the little lightforms

---



```

userLightform5XPosArray
.BYTE $FE,$FF,$00,$01,$02,$FD,$FC,$55 : 44444
.BYTE $03,$04,$05,$06,$FC,$FC,$FD,$FF,$FE,$55 : 4 44 5
.BYTE $07,$08,$09,$04,$03,$55 : 4 55555
.BYTE $00,$01,$02,$FB,$FC,$FD,$FE,$FF,$00,$01,$02,$03,$55 :
.BYTE $04,$05,$06,$07,$08,$09,$55 : 11111
.BYTE $00,$55 : 1 22 3
userLightform5YPosArray
.BYTE $FD,$FD,$FD,$FD,$FD,$FD,$FD,$FD,$FD,$FD : 1 2233
.BYTE $FE,$FE,$FE,$FE,$FE,$FE,$FE,$FE,$FE,$FE : 2 6 3
.BYTE $02,$02,$02,$02,$02,$02,$02,$02,$02,$02,$55 : 2 2 3
.BYTE $02,$02,$02,$02,$02,$02,$02,$02,$02,$02,$55 : 222444
.BYTE $FA,$FA,$FA,$FA,$FA,$FA,$FA,$FA,$FA,$FA,$55 : 222444
.BYTE $00,$55

```

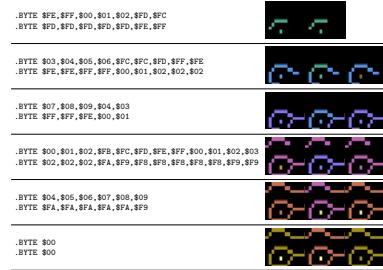
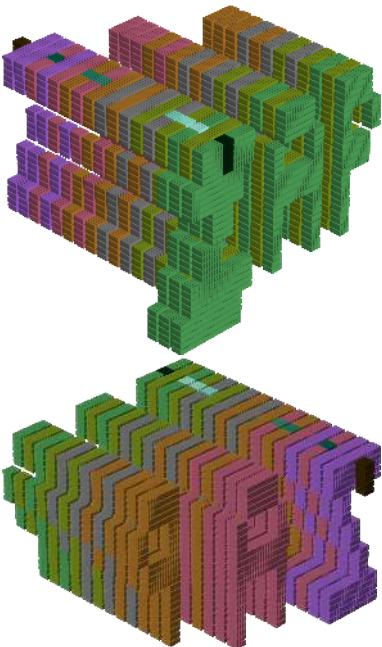


Figure 14.16: 'User LightForm 5'.



```

userLightform6XPosArray
.BYTE $FF,$FE,$01,$02,$01,$00,$FF,$FE : 11 11 222 333
.BYTE $FF,$00,$01,$02,$05 : 66661 2 2 3 3
.BYTE $05,$05,$05,$05,$06,$07,$08,$09 : 66661 2 2 3 3
.BYTE $09,$09,$09,$06,$07,$08,$55 : 61 22222 333
.BYTE $0C,$0C,$0C,$0C,$0C,$0D,$0E,$0F : 61 2 2 3
.BYTE $0D,$0E,$55 : 611111 2 2 3
.BYTE $55 : 666666
.BYTE $55
.BYTE $FD,$FE,$FF,$00,$FF,$FE,$FD,$FC,$55
.BYTE $FD,$FE,$FF,$00,$01,$55
userLightform6YPosArray
.BYTE $01,$02,$03,$04,$01,$02,$03,$04 : 04,04,04,04,04,04,04,04
.BYTE $04,$03,$02,$01,$00,$00,$00,$01 : 02,03,04,02,02,02,02,02
.BYTE $04,$03,$02,$00,$01,$00,$00,$01 : 02,02,02,02,02,02,02,02
.BYTE $05,$05,$05,$05,$06,$07,$08,$09 : 05,05,05,05,05,05,05,05
.BYTE $09,$09,$09,$09,$09,$09,$09,$09 : 05,05,05,05,05,05,05,05
.BYTE $00,$00,$00,$00,$00,$00,$00,$00 : 04,04,04,04,04,04,04,04
.BYTE $04,$03,$02,$01,$00,$00,$00,$01 : 02,02,02,02,02,02,02,02
.BYTE $04,$03,$02,$00,$01,$00,$00,$01 : 02,02,02,02,02,02,02,02
.BYTE $05,$05,$05,$05,$05,$05,$05,$05 : 05,05,05,05,05,05,05,05
.BYTE $01,$01,$01,$01,$01,$02,$03,$04,$05 : 04,04,04,04,04,04,04,04,05

```



Figure 14.17: 'User LightForm 6'.

## all the little lightforms

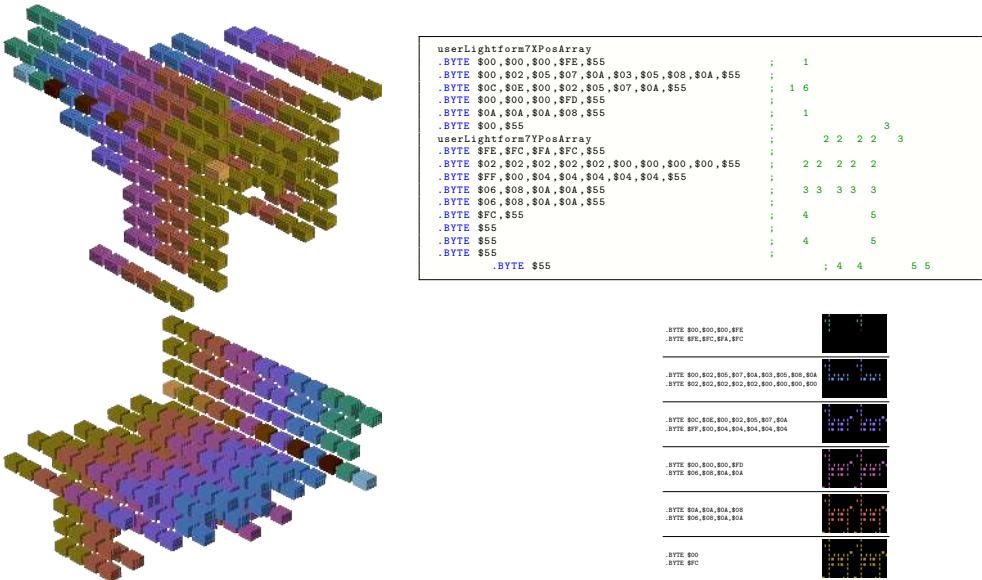


Figure 14.18: 'User LightForm 7'.

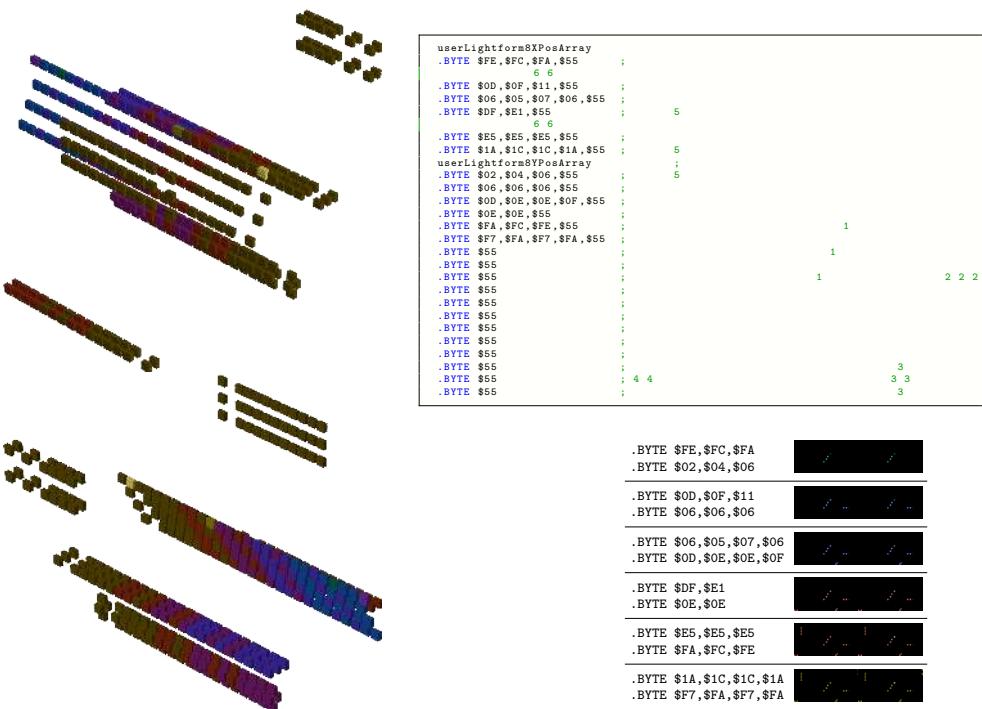


Figure 14.19: 'User LightForm 8'.

## all the little lightforms

---

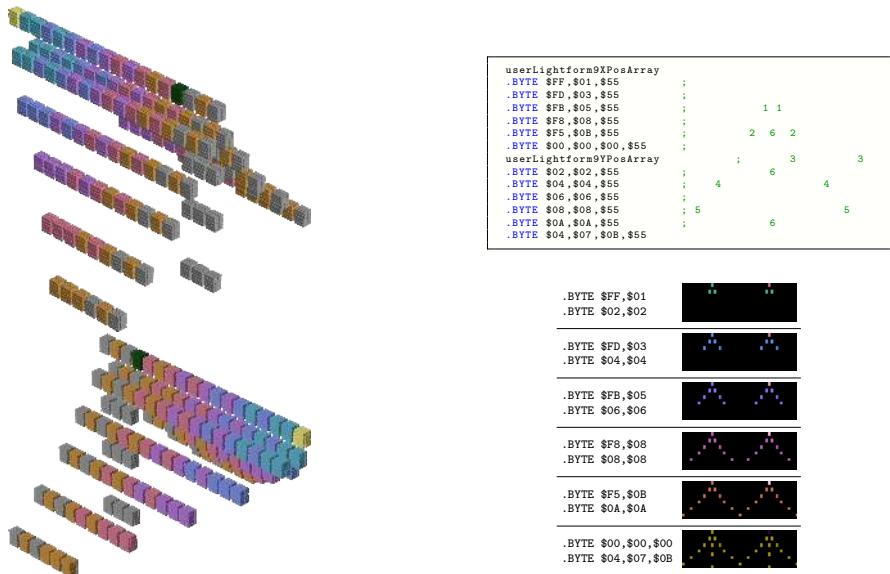


Figure 14.20: 'User LightForm 9'.



Figure 14.21: 'User LightForm 10'.

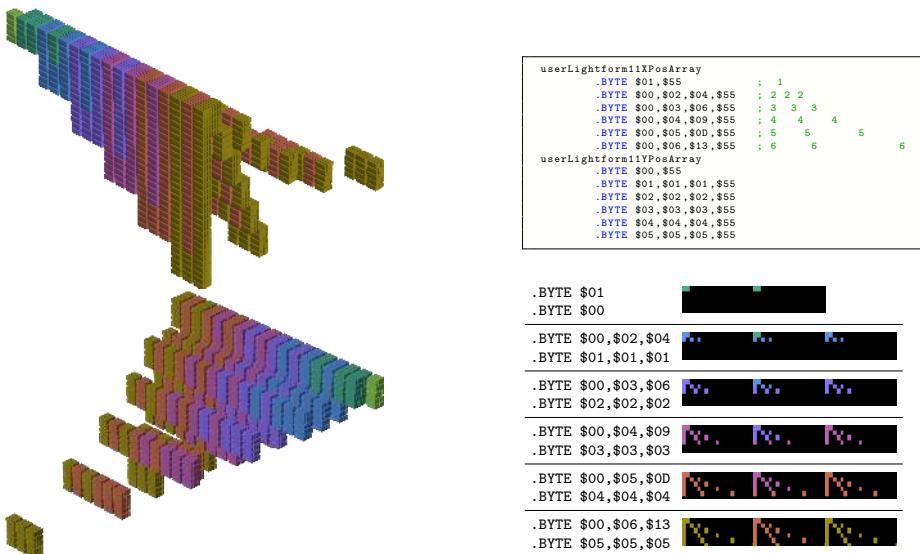


Figure 14.22: 'User LightForm 11'.

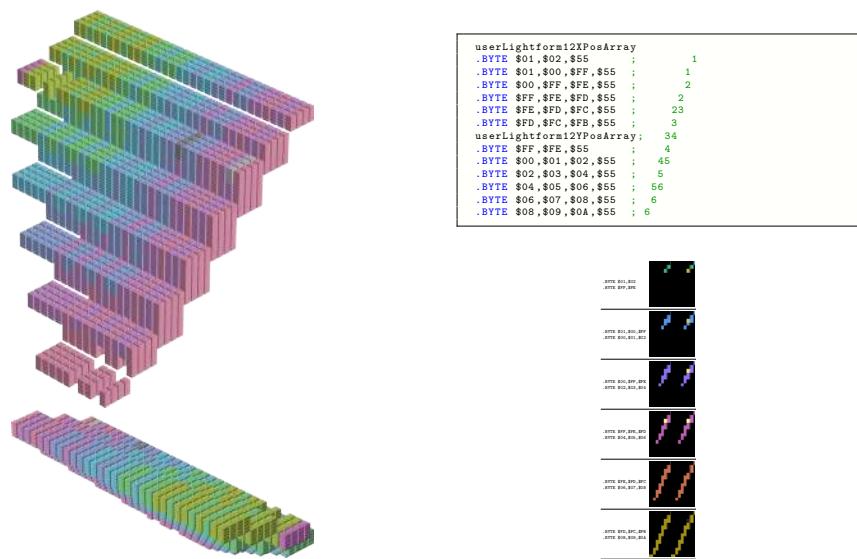
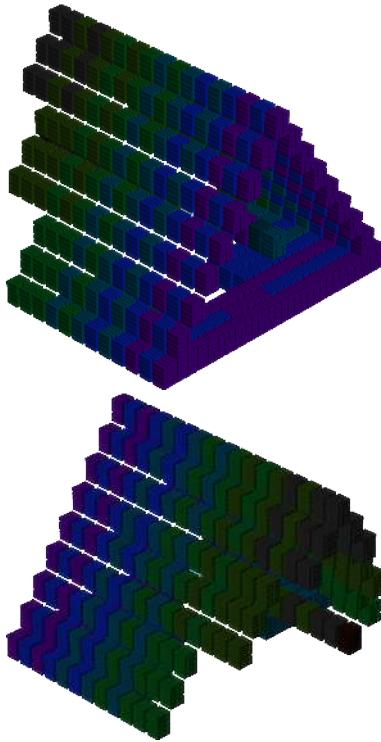


Figure 14.23: 'User LightForm 12'.

## all the little lightforms

---



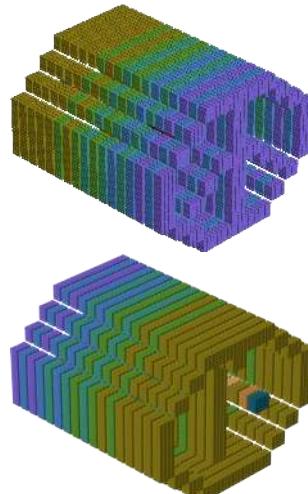
```

userLightform13XPosArray
    .BYTE $00,$FF,$01,$FE,$02,$55      ; 1
    .BYTE $FD,$03,$FC,$04,$FB,$05,$55   ; 1
    .BYTE $FA,$06,$F9,$07,$FB,$08,$55   ; 1
    .BYTE $F9,$FA,$FB,$FC,$FD,$07,$06,$05,$55 ; 2
    .BYTE $04,$03,$02,$01,$55           ; 2
    .BYTE $00,$FF,$FE,$55               ; 2
userLightform13YPosArray      ; 3
    .BYTE $FF,$FC,$FD,$FD,$55          ; 3
    .BYTE $FE,$FF,$FF,$00,$00,$55       ; 3
3444446665555443
    .BYTE $01,$01,$02,$02,$03,$03,$55
    .BYTE $03,$03,$03,$03,$03,$03,$03,$55
    .BYTE $03,$03,$03,$03,$03,$55
    .BYTE $03,$03,$03,$03,$55

```



Figure 14.24: 'User LightForm 13'.



```

userLightform14XPosArray
    .BYTE $FC,$FC,$FC,$FD,$FE,$FF,$00      ; 11111
    .BYTE $01,$03,$03,$02,$04,$05,$06,$06,$06,$05,$04,$01,$55 ; 1 1 1
    .BYTE $01,$01,$01,$01,$01,$01,$01,$01,$55                   ; 1 2 1
    .BYTE $55                                         ; 1 2

    .BYTE $FF,$FE,$02,$03,$04,$55                 ; 1 24
userLightform14YPosArray      ; 1 4 2 4
    1
    .BYTE $00,$01,$02,$03,$04,$05,$05,$05,$05,$FF,$FE,$FD,$FC,$FC ; 1 4 2 4
    .BYTE $FC,$FC,$FC,$FD,$FE,$FF,$00,$01,$02,$03,$04,$05,$05,$05 ; 1 2 1
    .BYTE $55                                         ; 11111
    .BYTE $01,$02,$00,$01,$02,$55

    .BYTE $FC,$FC,$FC,$FD,$FE,$FF,$00
    .BYTE $01,$02,$03,$02,$04,$05,$06,
    .BYTE $06,$06,$06,$05,$04,$01
    .BYTE $00,$01,$02,$03,$04,$05,$05
    .BYTE $05,$05,$FF,$FE,$FD,$FC,$FC
    .BYTE $FC,$FC,$05,$05,$FF,$FE,$FD
    .BYTE $00,$01,$02,$03,$04,$04

```

Figure 14.25: 'User LightForm 14'.

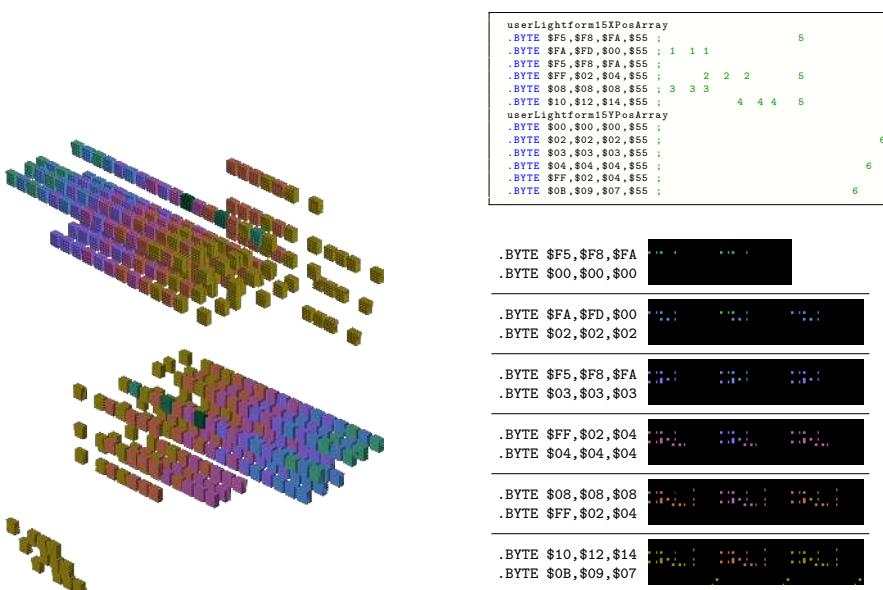


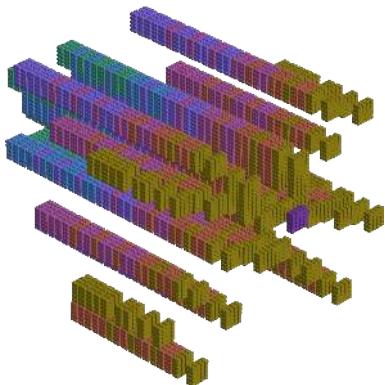
Figure 14.26: 'User LightForm 15'.



# **plentiful presets**

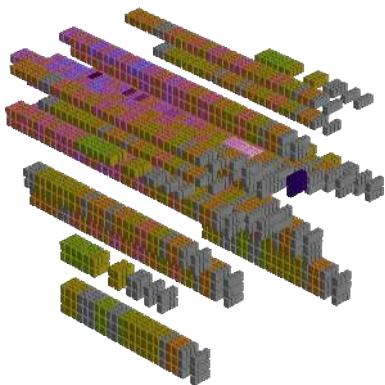
The sheer prettiness of *Colourspace* is most evident in the results you get from its presets. Here we'll take a look at all 78 of them, along with the data that helps generate them.

## plentiful presets



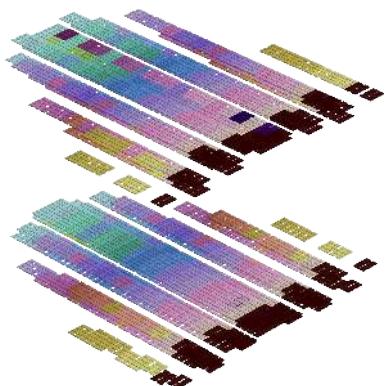
Preset 0.

```
preset0
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $01 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $00 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,RUST8 ; colorPalette
.BYTE DARK_ORANGE8,DARK_LAVENDER8 ; colorPalette
.BYTE ULTRAMARINE_BLUE8,DARK_BLUE8 ; colorPalette
.BYTE OLIVE_GREEN8,DARK_GREEN8 ; colorPalette
.BYTE $00,$00,$00,$00,$00,$00 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode
```



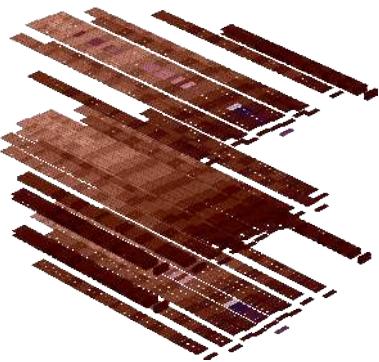
Preset 1.

```
preset1
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $01 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $02 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,RUST8 ; colorPalette
.BYTE DARK_ORANGE8,DARK_LAVENDER8 ; colorPalette
.BYTE ULTRAMARINE_BLUE8,DARK_BLUE8 ; colorPalette
.BYTE OLIVE_GREEN8,DARK_GREEN8 ; colorPalette
.BYTE $00,$03,$03,$03,$03,$03 ; oozeRates
.BYTE $01,$10,$10,$10,$10,$10 ; oozeSteps
.BYTE $0F,$FF,$FF,$FF,$FF,$FF ; oozeCycles
.BYTE $00 ; explosionMode
```



Preset 2.

```
preset2
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $02 ; speedBoostAdjust
.BYTE $01 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $03 ; pulseSpeed
.BYTE $02 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $06 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,RUST8 ; colorPalette
.BYTE DARK_ORANGE8,DARK_LAVENDER8 ; colorPalette
.BYTE ULTRAMARINE_BLUE8,DARK_BLUE8 ; colorPalette
.BYTE OLIVE_GREEN8,DARK_GREEN8 ; colorPalette
.BYTE $00,$01,$01,$01,$01,$01 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode
```

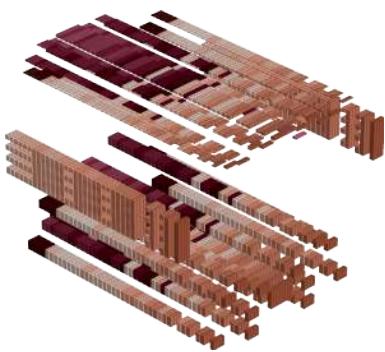


Preset 3.

```

preset3
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $06 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $02 ; currentSymmetry
.BYTE $01 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,DARK_ORANGE1 ; colorPalette
.BYTE DARK_ORANGE2,DARK_ORANGE4 ; colorPalette
.BYTE DARK_ORANGE6,DARK_ORANGE8 ; colorPalette
.BYTE DARK_ORANGE10,DARK_ORANGE12 ; colorPalette
.BYTE $00,$00,$00,$00,$00,$00 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode

```

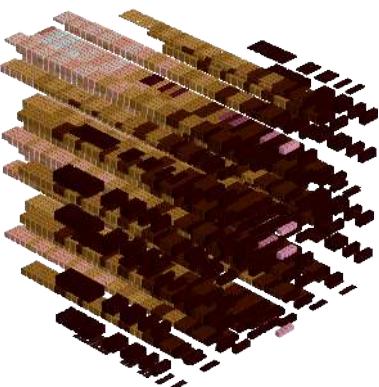


Preset 4.

```

preset4
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $06 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $08 ; smoothingDelay
.BYTE $04 ; currentSymmetry
.BYTE $02 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,DARK_ORANGE1 ; colorPalette
.BYTE DARK_ORANGE2,DARK_ORANGE4 ; colorPalette
.BYTE DARK_ORANGE6,DARK_ORANGE8 ; colorPalette
.BYTE DARK_ORANGE10,DARK_ORANGE12 ; colorPalette
.BYTE $00,$03,$03,$03,$03,$03 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode

```



Preset 5.

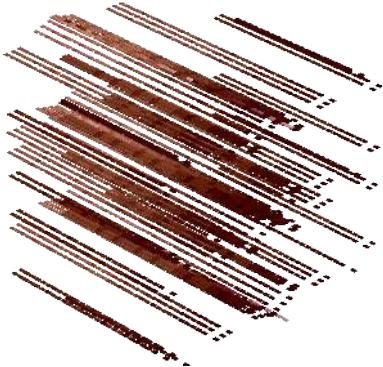
```

preset5
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $01 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $05 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $04 ; stroboscopicsEnabled
.BYTE $04 ; stroboFlashRate
.BYTE BLACK15,RED_ORANGE0 ; colorPalette
.BYTE RED_ORANGE2,RED_ORANGE8 ; colorPalette
.BYTE RED_ORANGE10,DARK_ORANGE12 ; colorPalette
.BYTE DARK_ORANGE14,MEDIUM_BLUE15 ; colorPalette
.BYTE $00,$00,$00,$00,$00,$00 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode

```

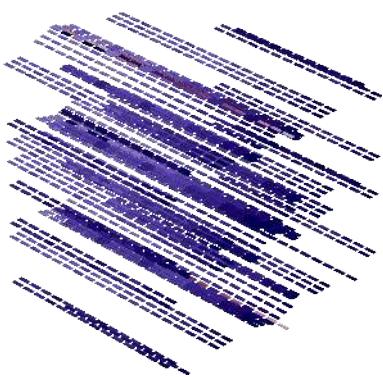
## plentiful presets

---



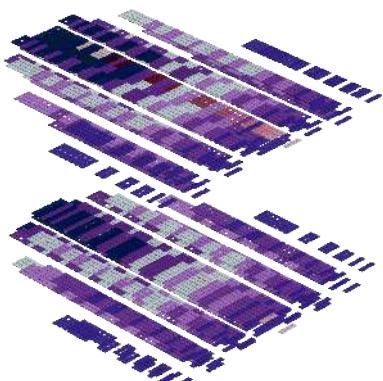
Preset 6.

```
preset6
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $10 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $04 ; currentSymmetry
.BYTE $01 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,DARK_ORANGE1 ; colorPalette
.BYTE DARK_ORANGE2,DARK_ORANGE3 ; colorPalette
.BYTE DARK_ORANGE6,DARK_ORANGE7 ; colorPalette
.BYTE DARK_ORANGE10,DARK_ORANGE12 ; colorPalette
.BYTE $00,$00,$00,$00,$00,$00 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode
```



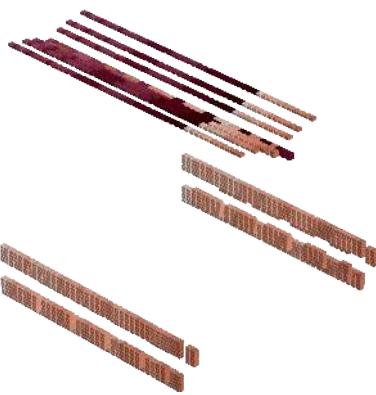
Preset 7.

```
preset7
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $10 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $02 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $04 ; currentSymmetry
.BYTE $01 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,ULTRAMARINE_BLUE0 ; colorPalette
.BYTE ULTRAMARINE_BLUE2,ULTRAMARINE_BLUE4 ;
colorPalette
.BYTE ULTRAMARINE_BLUE6,ULTRAMARINE_BLUES ;
colorPalette
.BYTE ULTRAMARINE_BLUE10,MEDIUM_BLUE0 ;
colorPalette
.BYTE $00,$00,$00,$00,$00,$00 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode
```



Preset 8.

```
preset8
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $01 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $06 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,ULTRAMARINE_BLUE4 ; colorPalette
.BYTE COBALT_BLUE6,COBALT_BLUE10 ; colorPalette
.BYTE MEDIUM_BLUE15,COBALT_BLUE8 ; colorPalette
.BYTE DARK_BLUE1,DARK_BLUE17 ; colorPalette
.BYTE $00,$00,$00,$00,$00,$00 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode
```

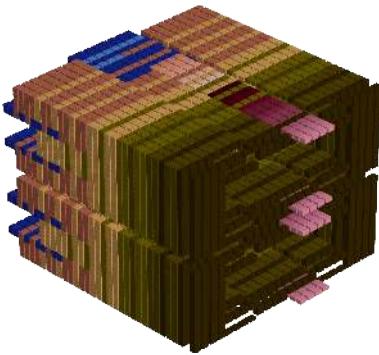


Preset 9.

```

preset9
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $09 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $08 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $02 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,DARK_ORANGE1 ; colorPalette
.BYTE DARK_ORANGE2,DARK_ORANGE1 ; colorPalette
.BYTE DARK_ORANGE6,DARK_ORANGE1 ; colorPalette
.BYTE DARK_ORANGE10,DARK_ORANGE12 ; colorPalette
.BYTE $00,$03,$03,$03,$03,$03,$03 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode

```

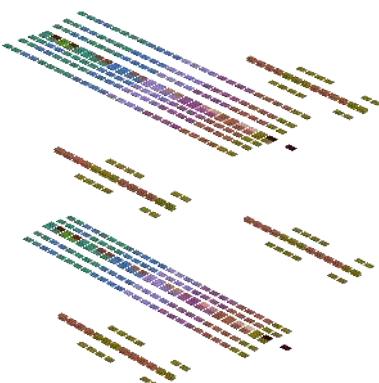


Preset 10.

```

preset10
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $0A ; currentPatternIndex
.BYTE $03 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $05 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,RUST2 ; colorPalette
.BYTE RUST4,RUST6 ; colorPalette
.BYTE RED_ORANGE11,DARK_ORANGE8 ; colorPalette
.BYTE MEDIUM_BLUES3,DARK_BLUE8 ; colorPalette
.BYTE $00,$00,$00,$00,$00,$00,$00 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode

```



Preset 11.

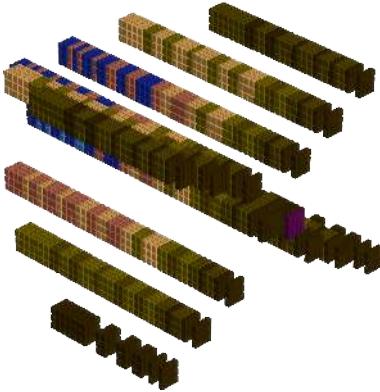
```

preset11
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $02 ; speedBoostAdjust
.BYTE $09 ; currentPatternIndex
.BYTE $02 ; verticalResolutionSPAC
.BYTE $03 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $01 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,RUST8 ; colorPalette
.BYTE DARK_ORANGE8,DARK_LAVENDER8 ; colorPalette
.BYTE ULTRAMARINE_BLUES8,DARK_BLUE8 ; colorPalette
.BYTE OLIVE_GREEN8,DARK_GREEN8 ; colorPalette
.BYTE $00,$00,$00,$01,$00,$00,$00 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode

```

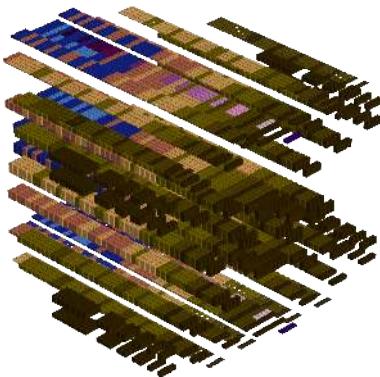
## plentiful presets

---



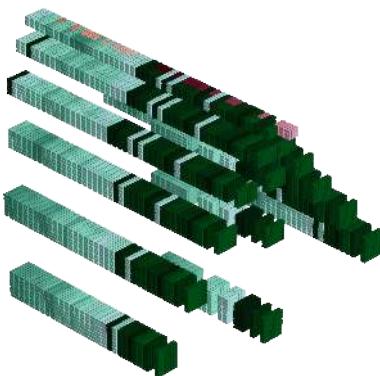
Preset 12.

```
preset12
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $02 ; currentPatternIndex
.BYTE $03 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $00 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,RUST2 ; colorPalette
.BYTE RUST4,RUST6 ; colorPalette
.BYTE RED_ORANGE11,DARK_ORANGES ; colorPalette
.BYTE MEDIUM_BLUES,DARK_BLUE8 ; colorPalette
.BYTE $00,$00,$00,$00,$00,$00,$00 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode
```



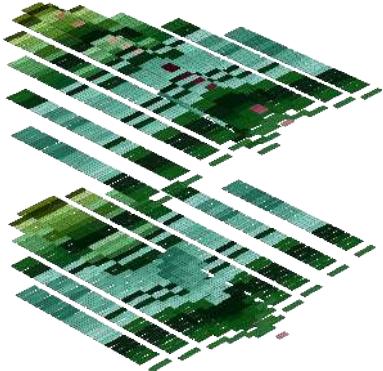
Preset 13.

```
preset13
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $02 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $04 ; currentSymmetry
.BYTE $04 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,RUST2 ; colorPalette
.BYTE RUST4,RUST6 ; colorPalette
.BYTE RED_ORANGE11,DARK_ORANGES ; colorPalette
.BYTE MEDIUM_BLUES,DARK_BLUE8 ; colorPalette
.BYTE $00,$00,$00,$00,$00,$00,$00 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode
```



Preset 14.

```
preset14
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $11 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $03 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,OLIVE_GREEN4 ; colorPalette
.BYTE OLIVE_GREEN6,OLIVE_GREEN8 ; colorPalette
.BYTE OLIVE_GREEN10,OLIVE_GREEN12 ; colorPalette
.BYTE OLIVE_GREEN10,OLIVE_GREEN8 ; colorPalette
.BYTE $00,$01,$01,$01,$01,$01,$01 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode
```

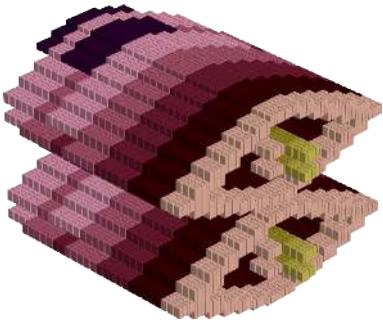


Preset 15.

```

preset15
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $11 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $06 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,OLIVE_GREEN4 ; colorPalette
.BYTE OLIVE_GREEN6,OLIVE_GREEN8 ; colorPalette
.BYTE OLIVE_GREEN10,OLIVE_GREEN12 ; colorPalette
.BYTE OLIVE_GREEN10,OLIVE_GREEN8 ; colorPalette
.BYTE $00,$01,$01,$01,$01,$01,$01 ; oozeRates
.BYTE $01,$02,$02,$03,$03,$05,$05,$06 ; oozeSteps
.BYTE $0F,$20,$20,$20,$20,$20,$20,$20 ; oozeCycles
.BYTE $00 ; explosionMode

```

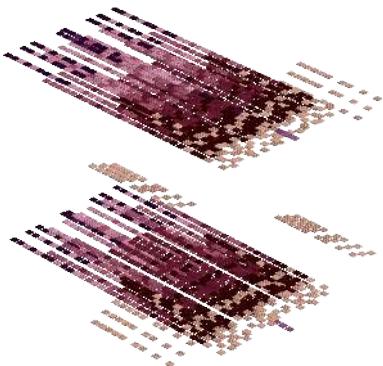


Preset 16.

```

preset16
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $07 ; speedBoostAdjust
.BYTE $15 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $06 ; pulseSpeed
.BYTE $05 ; currentPulseWidth
.BYTE $08 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $05 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,DARK_ORANGE13 ; colorPalette
.BYTE RED0,RED4 ; colorPalette
.BYTE RED8,RED12 ; colorPalette
.BYTE DARK_LAVENDER0,DARK_LAVENDER4 ; colorPalette
.BYTE $03,$00,$00,$00,$00,$00,$00 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $01 ; explosionMode

```



Preset 17.

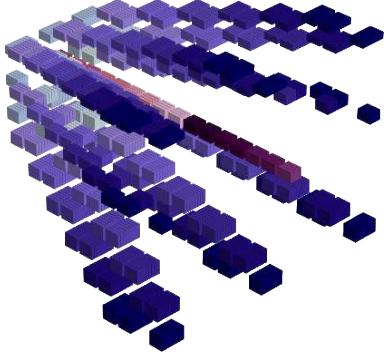
```

preset17
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $17 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $01 ; pulseSpeed
.BYTE $05 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $02 ; currentSymmetry
.BYTE $06 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,DARK_ORANGE13 ; colorPalette
.BYTE RED0,RED4 ; colorPalette
.BYTE RED8,RED12 ; colorPalette
.BYTE DARK_LAVENDER0,DARK_LAVENDER4 ; colorPalette
.BYTE $03,$00,$00,$00,$00,$00,$00 ; oozeRates
.BYTE $10,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $03 ; explosionMode

```

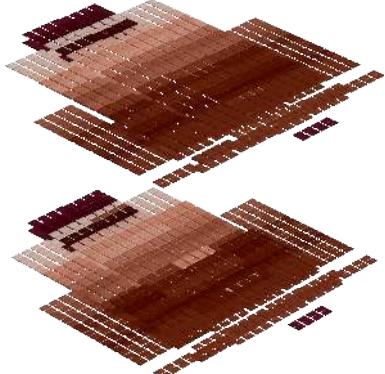
## plentiful presets

---



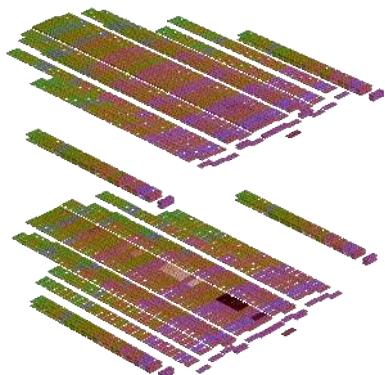
Preset 18.

```
preset18
.BYTE $02 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $0D ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $01 ; pulseSpeed
.BYTE $02 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $02 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,ULTRAMARINE_BLUE0 ; colorPalette
.BYTE ULTRAMARINE_BLUE2,ULTRAMARINE_BLUE4 ;
colorPalette
.BYTE ULTRAMARINE_BLUE6,ULTRAMARINE_BLUE8 ;
colorPalette
.BYTE ULTRAMARINE_BLUE10,MEDIUM_BLUE0 ;
colorPalette
.BYTE $00,$00,$00,$00,$00,$00,$00,$02 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode
```



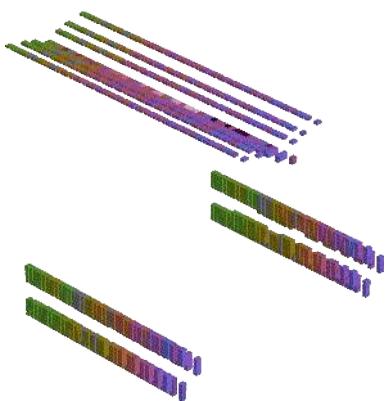
Preset 19.

```
preset19
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $02 ; speedBoostAdjust
.BYTE $07 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $03 ; pulseSpeed
.BYTE $08 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $02 ; currentSymmetry
.BYTE $06 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,DARK_ORANGE1 ; colorPalette
.BYTE DARK_ORANGE2,DARK_ORANGE4 ; colorPalette
.BYTE DARK_ORANGE6,DARK_ORANGE8 ; colorPalette
.BYTE DARK_ORANGE10,DARK_ORANGE12 ; colorPalette
.BYTE $00,$01,$01,$01,$01,$01,$01 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode
```



Preset 20.

```
preset20
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $06 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $01 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,RUST8 ; colorPalette
.BYTE DARK_ORANGE8,DARK_LAVENDER8 ; colorPalette
.BYTE ULTRAMARINE_BLUE8,DARK_BLUE8 ; colorPalette
.BYTE OLIVE_GREEN8,DARK_GREEN8 ; colorPalette
.BYTE $00,$01,$01,$01,$01,$01,$01 ; oozeRates
.BYTE $01,$10,$10,$10,$10,$10,$10 ; oozeSteps
.BYTE $0F,$FF,$FF,$FF,$FF,$FF,$FF ; oozeCycles
.BYTE $00 ; explosionMode
```

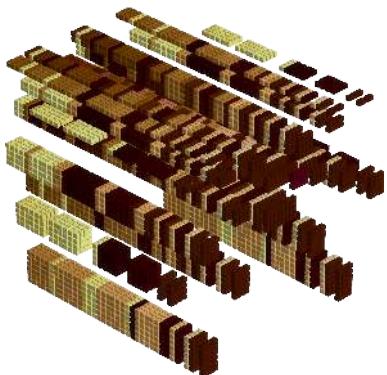


Preset 21.

```

.preset21
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $09 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $02 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,RUST8 ; colorPalette
.BYTE DARK_ORANGE8,DARK_LAVENDER8 ; colorPalette
.BYTE ULTRAMARINE_BLUE8,DARK_BLUE8 ; colorPalette
.BYTE OLIVE_GREEN8,DARK_GREEN8 ; colorPalette
.BYTE $00,$01,$01,$01,$01,$01,$01 ; oozeRates
.BYTE $01,$10,$10,$10,$10,$10,$10 ; oozeSteps
.BYTE $0F,$FF,$FF,$FF,$FF,$FF,$FF ; oozeCycles
.BYTE $00 ; explosionMode

```

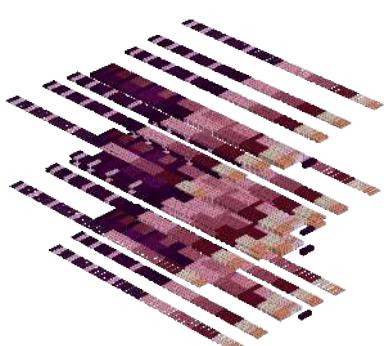


Preset 22.

```

.preset22
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $05 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $08 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $03 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,BLACK2 ; colorPalette
.BYTE BLACK14,BLACK4 ; colorPalette
.BYTE BLACK12,BLACK6 ; colorPalette
.BYTE BLACK10,BLACK8 ; colorPalette
.BYTE $00,$02,$02,$02,$02,$02,$02 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$FF,$FF,$FF,$FF,$FF,$FF ; oozeCycles
.BYTE $00 ; explosionMode

```



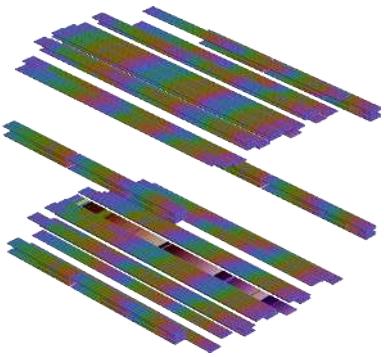
Preset 23.

```

.preset23
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $09 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $08 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $05 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,DARK_ORANGE1 ; colorPalette
.BYTE DARK_ORANGE2,DARK_ORANGE4 ; colorPalette
.BYTE DARK_ORANGE6,DARK_ORANGE8 ; colorPalette
.BYTE DARK_ORANGE10,DARK_ORANGE12 ; colorPalette
.BYTE $00,$03,$03,$03,$03,$03,$03 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$12,$13,$14,$15,$16,$17,$18 ; oozeCycles
.BYTE $00 ; explosionMode

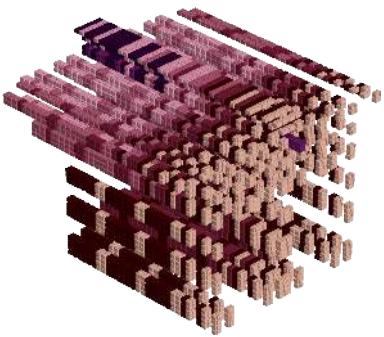
```

## plentiful presets



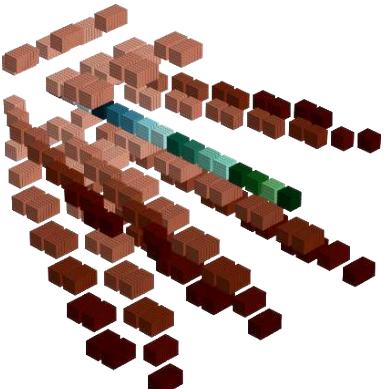
Preset 24.

```
preset24
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $0C ; speedBoostAdjust
.BYTE $06 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $01 ; smoothingDelay
.BYTE $02 ; currentSymmetry
.BYTE $06 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,RUST8 ; colorPalette
.BYTE DARK_ORANGE8,DARK_LAVENDER8 ; colorPalette
.BYTE ULTRAMARINE_BLUE8,DARK_BLUE8 ; colorPalette
.BYTE OLIVE_GREEN8,DARK_GREEN8 ; colorPalette
.BYTE $00,$01,$01,$01,$01,$01,$01 ; oozeRates
.BYTE $01,$10,$10,$10,$10,$10,$10 ; oozeSteps
.BYTE $0F,$FF,$FF,$FF,$FF,$FF ; oozeCycles
.BYTE $00 ; explosionMode
```



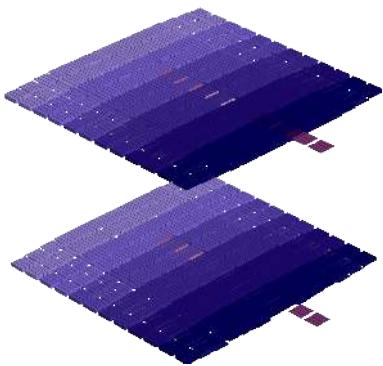
Preset 25.

```
preset25
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $0F ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $01 ; pulseSpeed
.BYTE $05 ; currentPulseWidth
.BYTE $08 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $02 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,DARK_ORANGE13 ; colorPalette
.BYTE RED0,RED4 ; colorPalette
.BYTE RED8,RED14 ; colorPalette
.BYTE DARK_LAVENDER0,DARK_LAVENDER4 ; colorPalette
.BYTE $07,$00,$00,$00,$00,$00,$00 ; oozeRates
.BYTE $10,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $03 ; explosionMode
```



Preset 26.

```
preset26
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $0E ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $06 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $7B ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $00 ; screenMode
.BYTE $24 ; bufferLength
.BYTE $02 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK9,DARK_ORANGE1 ; colorPalette
.BYTE DARK_ORANGE2,DARK_ORANGE4 ; colorPalette
.BYTE DARK_ORANGE6,DARK_ORANGE8 ; colorPalette
.BYTE DARK_ORANGE10,DARK_ORANGE12 ; colorPalette
.BYTE $00,$00,$00,$00,$00,$00,$00 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode
```

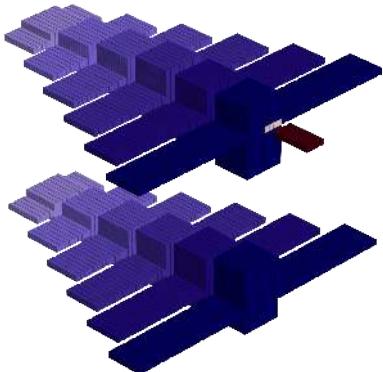


Preset 27.

```

.preset27
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $02 ; speedBoostAdjust
.BYTE $16 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $0B ; pulseSpeed
.BYTE $02 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $01 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopesEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,_ULTRAMARINE_BLUE0 ; colorPalette
.BYTE ULTRAMARINE_BLUE2,_ULTRAMARINE_BLUE4 ;
colorPalette
.BYTE ULTRAMARINE_BLUE6,_ULTRAMARINE_BLUE8 ;
colorPalette
.BYTE ULTRAMARINE_BLUE10,_MEDIUM_BLUE0 ;
colorPalette
.BYTE $00,$00,$00,$00,$00,$00,$02 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode

```

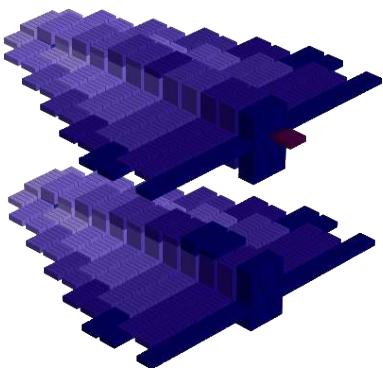


Preset 28.

```

.preset28
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $04 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $05 ; pulseSpeed
.BYTE $01 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $04 ; currentSymmetry
.BYTE $00 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopesEnabled
.BYTE $02 ; stroboFlashRate
.BYTE ULTRAMARINE_BLUE0,_ULTRAMARINE_BLUE0 ;
colorPalette
.BYTE ULTRAMARINE_BLUE2,_ULTRAMARINE_BLUE4 ;
colorPalette
.BYTE ULTRAMARINE_BLUE6,_ULTRAMARINE_BLUE8 ;
colorPalette
.BYTE ULTRAMARINE_BLUE10,_MEDIUM_BLUE0 ;
colorPalette
.BYTE $05,$00,$00,$00,$00,$00,$02 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $03 ; explosionMode

```



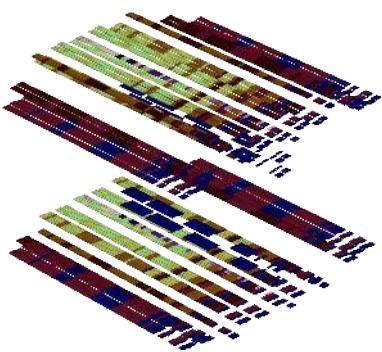
Preset 29.

```

.preset29
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $02 ; speedBoostAdjust
.BYTE $04 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $05 ; pulseSpeed
.BYTE $01 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $04 ; currentSymmetry
.BYTE $00 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $02 ; stroboscopesEnabled
.BYTE ULTRAMARINE_BLUE0,_ULTRAMARINE_BLUE0 ;
colorPalette
.BYTE ULTRAMARINE_BLUE2,_ULTRAMARINE_BLUE4 ;
colorPalette
.BYTE ULTRAMARINE_BLUE6,_ULTRAMARINE_BLUE8 ;
colorPalette
.BYTE ULTRAMARINE_BLUE10,_MEDIUM_BLUE0 ;
colorPalette
.BYTE $05,$00,$00,$00,$00,$00,$02 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $04 ; explosionMode

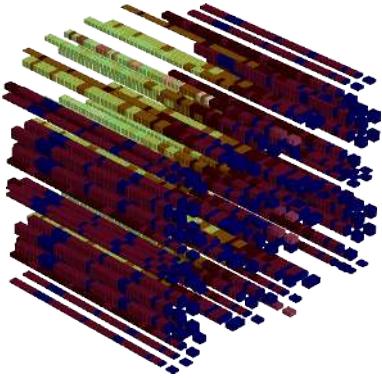
```

## plentiful presets



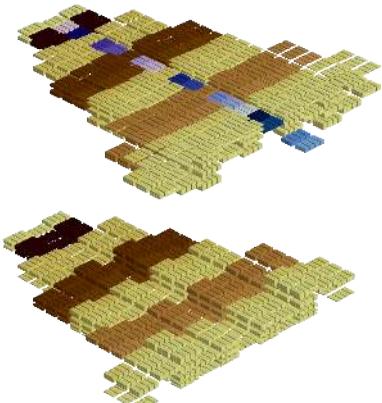
Preset 30.

```
preset30
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $02 ; speedBoostAdjust
.BYTE $0B ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $02 ; pulseSpeed
.BYTE $02 ; currentPulseWidth
.BYTE $09 ; smoothingDelay
.BYTE $02 ; currentSymmetry
.BYTE $06 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,ULTRAMARINE_BLUE0 ; colorPalette
.BYTE RED3,DARK_ORANGE0 ; colorPalette
.BYTE RED_ORANGE6,ORANGE_GREEN12 ; colorPalette
.BYTE DARK_GREEN13,DARK_GREEN15 ; colorPalette
.BYTE $00,$00,$00,$00,$00,$00,$00 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode
```



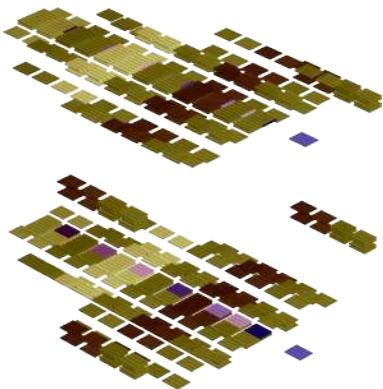
Preset 31.

```
preset31
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $02 ; speedBoostAdjust
.BYTE $0B ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $02 ; pulseSpeed
.BYTE $02 ; currentPulseWidth
.BYTE $09 ; smoothingDelay
.BYTE $02 ; currentSymmetry
.BYTE $05 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,ULTRAMARINE_BLUE0 ; colorPalette
.BYTE RED3,DARK_ORANGE0 ; colorPalette
.BYTE RED_ORANGE6,ORANGE_GREEN12 ; colorPalette
.BYTE DARK_GREEN13,DARK_GREEN15 ; colorPalette
.BYTE $00,$00,$00,$00,$00,$00,$00 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode
```



Preset 32.

```
preset32
.BYTE $02 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $05 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $40 ; smoothingDelay
.BYTE $02 ; currentSymmetry
.BYTE $03 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,BLACK2 ; colorPalette
.BYTE BLACK14,BLACK4 ; colorPalette
.BYTE BLACK12,BLACK6 ; colorPalette
.BYTE BLACK10,BLACK8 ; colorPalette
.BYTE $00,$05,$05,$05,$05,$05,$05 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$FF,$FF,$FF,$FF,$FF,$FF ; oozeCycles
.BYTE $00 ; explosionMode
```

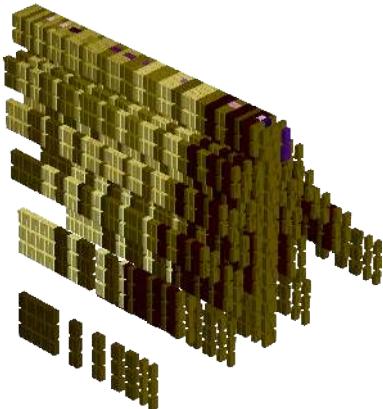


Preset 33.

```

preset33
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $06 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $04 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $40 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $06 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,BLACK2 ; colorPalette
.BYTE BLACK14,BLACK4 ; colorPalette
.BYTE BLACK12,BLACK6 ; colorPalette
.BYTE BLACK10,BLACK8 ; colorPalette
.BYTE $00,$05,$05,$05,$05,$05,$05 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$FF,$FF,$FF,$FF,$FF,$FF ; oozeCycles
.BYTE $00 ; explosionMode

```

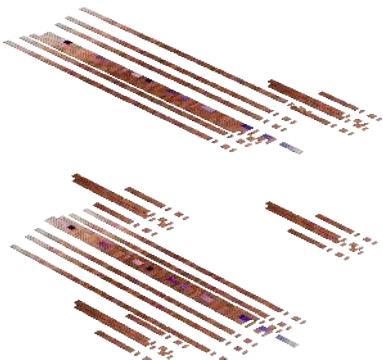


Preset 34.

```

preset34
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $13 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $01 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $0F ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $02 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,BLACK2 ; colorPalette
.BYTE BLACK14,BLACK4 ; colorPalette
.BYTE BLACK12,BLACK6 ; colorPalette
.BYTE BLACK10,BLACK8 ; colorPalette
.BYTE $00,$05,$05,$05,$05,$05,$05 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$FF,$FF,$FF,$FF,$FF,$FF ; oozeCycles
.BYTE $00 ; explosionMode

```



Preset 35.

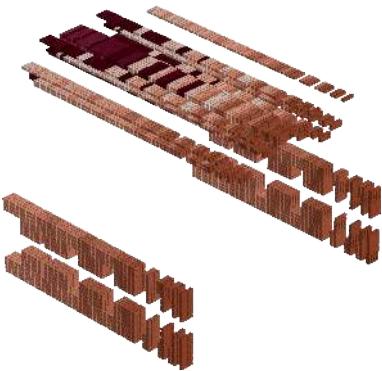
```

preset35
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $06 ; speedBoostAdjust
.BYTE $09 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $04 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $39 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $06 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,DARK_ORANGE1 ; colorPalette
.BYTE DARK_ORANGE2,DARK_ORANGE4 ; colorPalette
.BYTE DARK_ORANGE6,DARK_ORANGE8 ; colorPalette
.BYTE DARK_ORANGE10,DARK_ORANGE12 ; colorPalette
.BYTE $00,$03,$03,$03,$03,$03,$03 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode

```

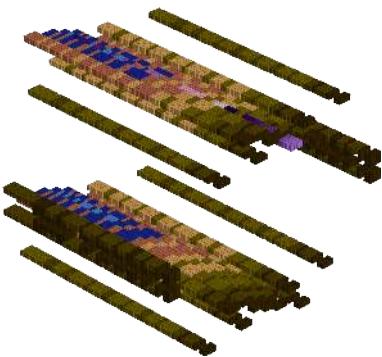
## plentiful presets

---



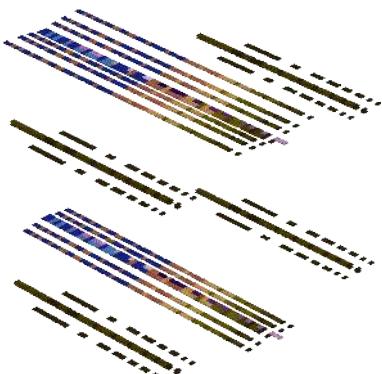
Preset 36.

```
preset36
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $06 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $08 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $02 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,DARK_ORANGE1 ; colorPalette
.BYTE DARK_ORANGE2,DARK_ORANGE4 ; colorPalette
.BYTE DARK_ORANGE6,DARK_ORANGE8 ; colorPalette
.BYTE DARK_ORANGE10,DARK_ORANGE12 ; colorPalette
.BYTE $00,$03,$03,$03,$03,$03 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode
```



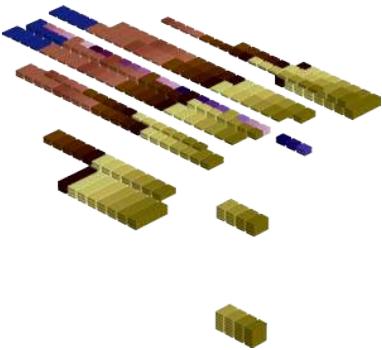
Preset 37.

```
preset37
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $06 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $02 ; currentSymmetry
.BYTE $00 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,RUST2 ; colorPalette
.BYTE RUST4,RUST6 ; colorPalette
.BYTE RED_ORANGE11,DARK_ORANGE8 ; colorPalette
.BYTE MEDIUM_BLUE3,DARK_BLUE8 ; colorPalette
.BYTE $00,$00,$00,$00,$00,$00 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode
```



Preset 38.

```
preset38
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $09 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $01 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,RUST2 ; colorPalette
.BYTE RUST4,RUST6 ; colorPalette
.BYTE RED_ORANGE11,DARK_ORANGE8 ; colorPalette
.BYTE MEDIUM_BLUE3,DARK_BLUE8 ; colorPalette
.BYTE $00,$00,$00,$00,$00,$00 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode
```

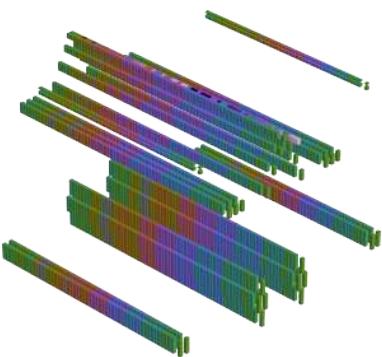


Preset 39.

```

preset39
.BYTE $02 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $0B ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $40 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $02 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,RUST2 ; colorPalette
.BYTE RUST4,RUST6 ; colorPalette
.BYTE RED_ORANGE11,DARK_ORANGE ; colorPalette
.BYTE MEDIUM_BLUE3,DARK_BLUE8 ; colorPalette
.BYTE $00,$02,$02,$02,$00,$00,$00 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode

```

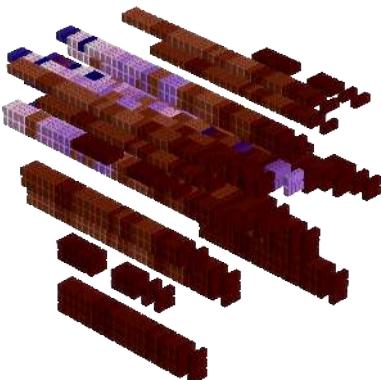


Preset 40.

```

preset40
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $10 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $03 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK4,RUST8 ; colorPalette
.BYTE DARK_ORANGE8,DARK_LAVENDER8 ; colorPalette
.BYTE ULTRAMARINE_BLUE8,DARK_BLUE8 ; colorPalette
.BYTE OLIVE_GREEN8,DARK_GREEN8 ; colorPalette
.BYTE $05,$01,$01,$01,$01,$01,$01 ; oozeRates
.BYTE $10,$10,$10,$10,$10,$10,$10 ; oozeSteps
.BYTE $0F,$FF,$FF,$FF,$FF,$FF,$FF ; oozeCycles
.BYTE $03 ; explosionMode

```



Preset 41.

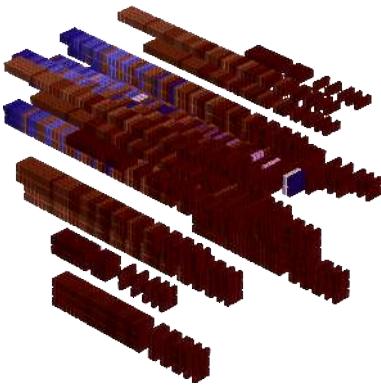
```

preset41
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $02 ; speedBoostAdjust
.BYTE $01 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $03 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $15 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $02 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,DARK_ORANGE1 ; colorPalette
.BYTE DARK_ORANGE2,DARK_ORANGE4 ; colorPalette
.BYTE DARK_ORANGE6,COBALT_BLUE8 ; colorPalette
.BYTE COBALT_BLUE10,COBALT_BLUE12 ; colorPalette
.BYTE $00,$00,$00,$00,$00,$01,$01 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode

```

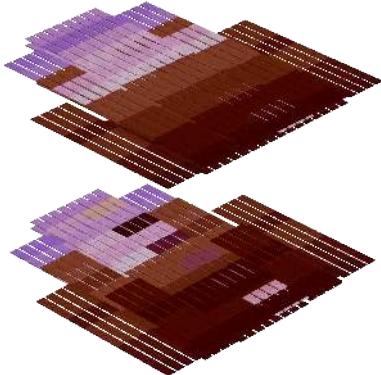
## plentiful presets

---



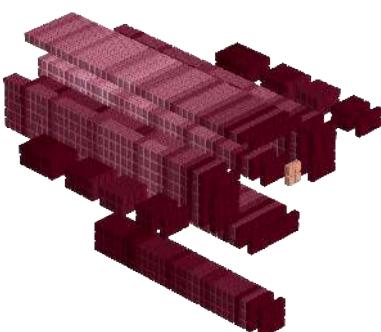
Preset 42.

```
preset42
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $01 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $01 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $15 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $02 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopesEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,DARK_ORANGE1 ; colorPalette
.BYTE DARK_ORANGE2,DARK_ORANGE4 ; colorPalette
.BYTE DARK_ORANGE6,COBALT_BLUE8 ; colorPalette
.BYTE COBALT_BLUE10,COBALT_BLUE12 ; colorPalette
.BYTE $00,$00,$00,$00,$01,$01,$01 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode
```



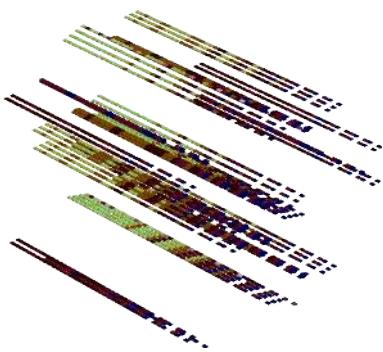
Preset 43.

```
preset43
.BYTE $03 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $08 ; speedBoostAdjust
.BYTE $07 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $01 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $15 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $06 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopesEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,DARK_ORANGE1 ; colorPalette
.BYTE DARK_ORANGE2,DARK_ORANGE4 ; colorPalette
.BYTE DARK_ORANGE6,COBALT_BLUE8 ; colorPalette
.BYTE COBALT_BLUE10,COBALT_BLUE12 ; colorPalette
.BYTE $00,$00,$00,$00,$01,$01,$01 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode
```



Preset 44.

```
preset44
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $0C ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $08 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $02 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopesEnabled
.BYTE $02 ; stroboFlashRate
.BYTE OLIVE_GREEN0,DARK_ORANGE1 ; colorPalette
.BYTE DARK_ORANGE2,DARK_ORANGE4 ; colorPalette
.BYTE DARK_ORANGE6,DARK_ORANGE8 ; colorPalette
.BYTE DARK_ORANGE10,DARK_ORANGE12 ; colorPalette
.BYTE $00,$03,$03,$03,$03,$03,$03 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$12,$13,$14,$15,$16,$17,$18 ; oozeCycles
.BYTE $00 ; explosionMode
```



Preset 45.

```

.preset45
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $10 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $01 ; pulseSpeed
.BYTE $02 ; currentPulseWidth
.BYTE $09 ; smoothingDelay
.BYTE $02 ; currentSymmetry
.BYTE $01 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,ULTRAMARINE_BLUE0 ; colorPalette
.BYTE RED3,DARK_ORANGE0 ; colorPalette
.BYTE RED_ORANGE0,ORANGE_GREEN12 ; colorPalette
.BYTE DARK_GREEN13,DARK_GREEN15 ; colorPalette
.BYTE $00,$00,$00,$00,$00,$00,$00 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode

```

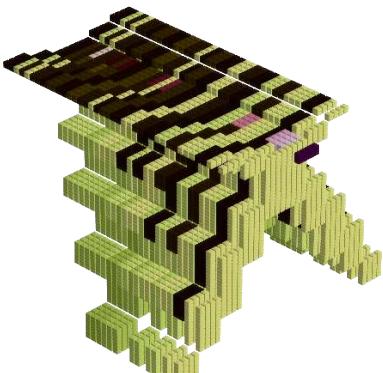


Preset 46.

```

.preset46
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $02 ; speedBoostAdjust
.BYTE $14 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $04 ; pulseSpeed
.BYTE $01 ; currentPulseWidth
.BYTE $04 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $06 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,RUST6 ; colorPalette
.BYTE RUST8,RUST10 ; colorPalette
.BYTE RUST12,RUST14 ; colorPalette
.BYTE RED_ORANGE0,RED_ORANGE2 ; colorPalette
.BYTE $00,$02,$02,$02,$02,$02,$02 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode

```



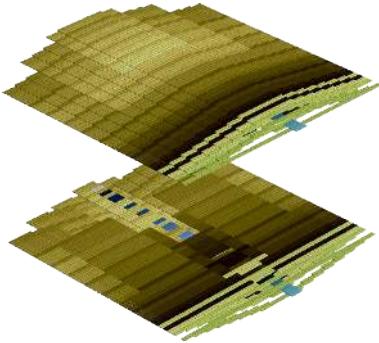
Preset 47.

```

.preset47
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $14 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $01 ; pulseSpeed
.BYTE $10 ; currentPulseWidth
.BYTE $01 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $03 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,ORANGE_GREEN6 ; colorPalette
.BYTE ORANGE_GREEN8,ORANGE_GREEN10 ; colorPalette
.BYTE ORANGE_GREEN12,ORANGE_GREEN14 ; colorPalette
.BYTE ORANGE0,ORANGE2 ; colorPalette
.BYTE $00,$02,$02,$02,$02,$02,$02 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode

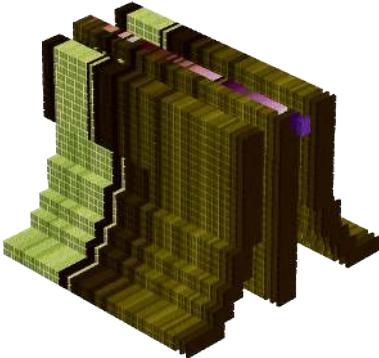
```

## plentiful presets



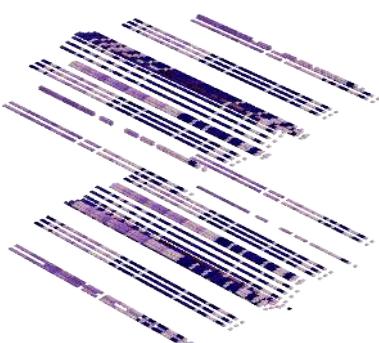
Preset 48.

```
preset48
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $15 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $01 ; pulseSpeed
.BYTE $01 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $06 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,ORANGE_GREEN0 ; colorPalette
.BYTE ORANGE_GREEN8,ORANGE_GREEN10 ; colorPalette
.BYTE ORANGE_GREEN12,ORANGE_GREEN14 ; colorPalette
.BYTE ORANGE0,ORANGE2 ; colorPalette
.BYTE $00,$02,$02,$02,$02,$02,$02 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode
```



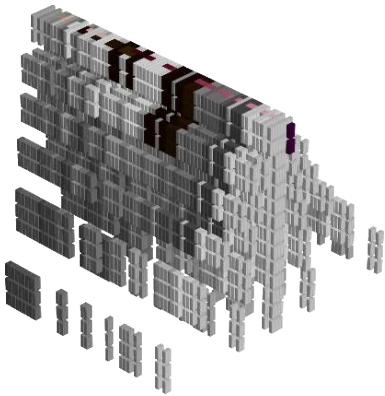
Preset 49.

```
preset49
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $08 ; currentPatternIndex
.BYTE $02 ; verticalResolutionSPAC
.BYTE $01 ; pulseSpeed
.BYTE $01 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $00 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,ORANGE_GREEN0 ; colorPalette
.BYTE ORANGE_GREEN8,ORANGE_GREEN10 ; colorPalette
.BYTE ORANGE_GREEN12,ORANGE_GREEN14 ; colorPalette
.BYTE ORANGE0,ORANGE2 ; colorPalette
.BYTE $00,$02,$02,$02,$02,$02,$02 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode
```



Preset 50.

```
preset50
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $10 ; currentPatternIndex
.BYTE $02 ; verticalResolutionSPAC
.BYTE $01 ; pulseSpeed
.BYTE $01 ; currentPulseWidth
.BYTE $0C ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $01 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,COBALT_BLUE6 ; colorPalette
.BYTE COBALT_BLUE8,COBALT_BLUE10 ; colorPalette
.BYTE COBALT_BLUE12,COBALT_BLUE14 ; colorPalette
.BYTE ULTRAMARINE_BLUE0,ULTRAMARINE_BLUE2 ;
.colorPalette
.BYTE $00,$02,$02,$02,$02,$02,$02 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode
```

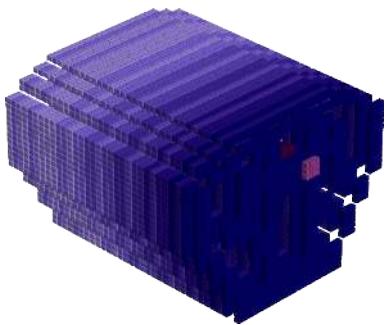


Preset 51.

```

preset51
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $13 ; currentPatternIndex
.BYTE $02 ; verticalResolutionSPAC
.BYTE $01 ; pulseSpeed
.BYTE $01 ; currentPulseWidth
.BYTE $0C ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $02 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,ORANGE_GREEN6 ; colorPalette
.BYTE ORANGE_GREENS,ORANGE_GREEN10 ; colorPalette
.BYTE ORANGE_GREEN12,ORANGE_GREEN14 ; colorPalette
.BYTE ORANGE0,ORANGE2 ; colorPalette
.BYTE $00,$02,$02,$02,$02,$02 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$40,$40,$20,$20,$40 ; oozeCycles
.BYTE $00 ; explosionMode

```

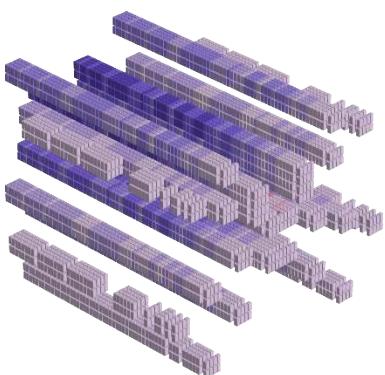


Preset 52.

```

preset52
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $16 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $02 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $03 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboflashRate
.BYTE BLACK0,ULTRAMARINE_BLUE0 ; colorPalette
.BYTE ULTRAMARINE_BLUE2,ULTRAMARINE_BLUE4 ;
colorPalette
.BYTE ULTRAMARINE_BLUE6,ULTRAMARINE_BLUE8 ;
colorPalette
.BYTE ULTRAMARINE_BLUE10,MEDIUM_BLUE0 ;
colorPalette
.BYTE $00,$00,$00,$00,$00,$00 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode

```



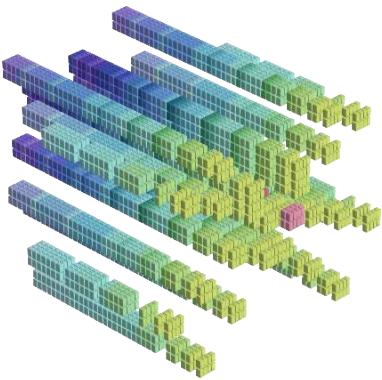
Preset 53.

```

preset53
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $02 ; speedBoostAdjust
.BYTE $01 ; currentPatternIndex
.BYTE $02 ; verticalResolutionSPAC
.BYTE $02 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $0A ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $00 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE DARK_BLUE0,ULTRAMARINE_BLUE14 ; colorPalette
.BYTE ULTRAMARINE_BLUE14,ULTRAMARINE_BLUE12 ;
colorPalette
.BYTE ULTRAMARINE_BLUE10,ULTRAMARINE_BLUE8 ;
colorPalette
.BYTE ULTRAMARINE_BLUE6,ULTRAMARINE_BLUE4 ;
colorPalette
.BYTE $00,$00,$00,$00,$00,$00 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode

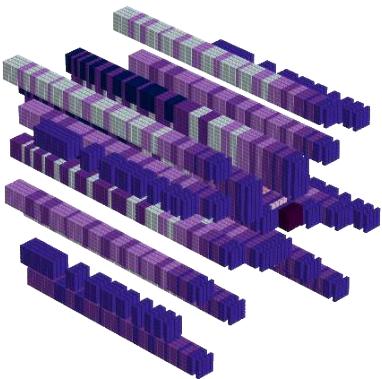
```

## plentiful presets



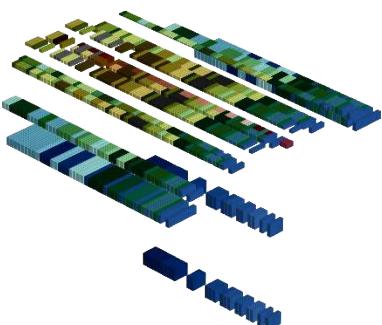
Preset 54.

```
preset54
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $02 ; speedBoostAdjust
.BYTE $01 ; currentPatternIndex
.BYTE $02 ; verticalResolutionSPAC
.BYTE $02 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $0A ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $00 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopesEnabled
.BYTE $02 ; stroboFlashRate
.BYTE ULTRAMARINE_BLUE14,ULTRAMARINE_BLUE14 ; colorPalette
.BYTE ULTRAMARINE_BLUE10,ULTRAMARINE_BLUE8 ; colorPalette
.BYTE ULTRAMARINE_BLUE6,ULTRAMARINE_BLUE4 ; colorPalette
.BYTE $00,$02,$02,$02,$02,$02,$02 ; oozeRates
.BYTE $01,$10,$10,$10,$10,$10,$10 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode
```



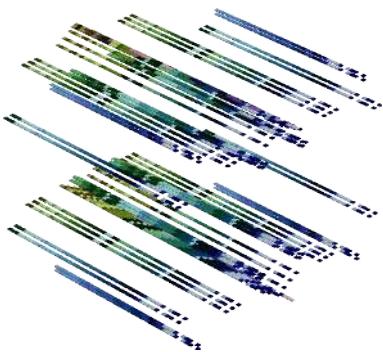
Preset 55.

```
preset55
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $01 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $00 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopesEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,ULTRAMARINE_BLUE4 ; colorPalette
.BYTE COBALT_BLUE6,COBALT_BLUE10 ; colorPalette
.BYTE MEDIUM_BLUE15,COBALT_BLUE5 ; colorPalette
.BYTE DARK_BLUE1,DARK_BLUE17 ; colorPalette
.BYTE $00,$00,$00,$00,$00,$00,$00 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode
```



Preset 56.

```
preset56
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $0B ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $01 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $03 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopesEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK4,ULTRAMARINE_BLUE6 ; colorPalette
.BYTE ULTRAMARINE_BLUE6,ULTRAMARINE_BLUE6 ; colorPalette
.BYTE ULTRAMARINE_BLUE6,ULTRAMARINE_BLUE6 ; colorPalette
.BYTE $20,$02,$02,$02,$02,$02,$02 ; oozeRates
.BYTE $10,$01,$02,$03,$04,$05,$06,$07 ; oozeSteps
.BYTE $0F,$20,$20,$20,$20,$20,$20 ; oozeCycles
.BYTE $07 ; explosionMode
```

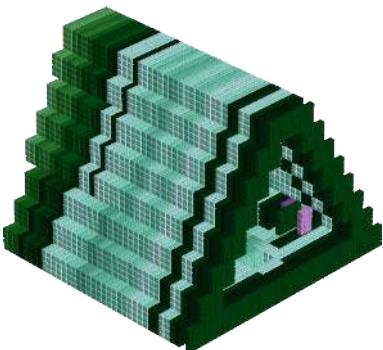


Preset 57.

```

preset57
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $10 ; currentPatternIndex
.BYTE $02 ; verticalResolutionSPAC
.BYTE $01 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $02 ; currentSymmetry
.BYTE $06 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0 ,ULTRAMARINE_BLUE6 ; colorPalette
.BYTE ULTRAMARINE_BLUE6,ULTRAMARINE_BLUE6 ;
colorPalette
.BYTE ULTRAMARINE_BLUE6,ULTRAMARINE_BLUE6 ;
colorPalette
.BYTE $00 ,$02,$02,$02,$02,$02,$02 ; oozeRates
.BYTE $10,$01,$02,$03,$04,$05,$06,$07 ; oozeSteps
.BYTE $0F,$20,$20,$20,$20,$20,$20 ; oozeCycles
.BYTE $00 ; explosionMode

```

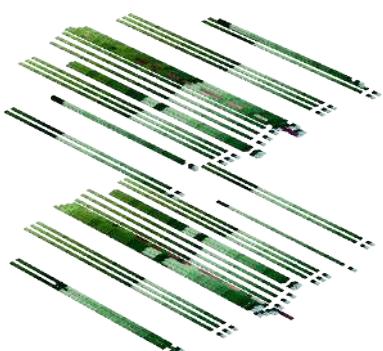


Preset 58.

```

preset58
.BYTE $02 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $15 ; currentPatternIndex
.BYTE $04 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $00 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0 ,OLIVE_GREEN4 ; colorPalette
.BYTE OLIVE_GREEN6,OLIVE_GREEN8 ; colorPalette
.BYTE OLIVE_GREEN10,OLIVE_GREEN12 ; colorPalette
.BYTE OLIVE_GREEN10,OLIVE_GREEN8 ; colorPalette
.BYTE $00,$01,$01,$01,$01,$01,$01 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode

```



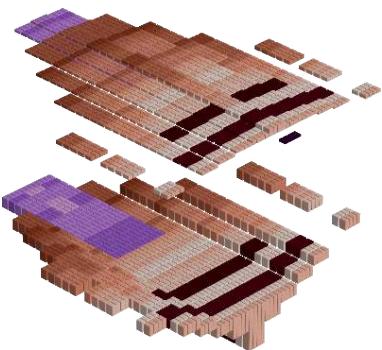
Preset 59.

```

preset59
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $10 ; currentPatternIndex
.BYTE $02 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $01 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0 ,OLIVE_GREEN4 ; colorPalette
.BYTE OLIVE_GREEN6,OLIVE_GREEN8 ; colorPalette
.BYTE OLIVE_GREEN10,OLIVE_GREEN12 ; colorPalette
.BYTE OLIVE_GREEN10,OLIVE_GREEN8 ; colorPalette
.BYTE $00,$01,$01,$01,$01,$01,$01 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$2E,$2E,$2E,$2E,$2E,$2E ; oozeCycles
.BYTE $00 ; explosionMode

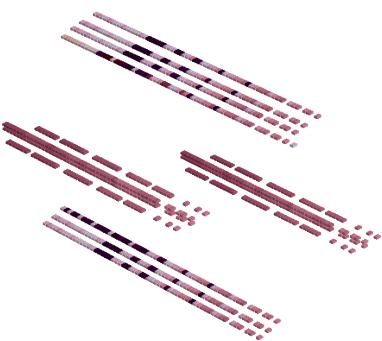
```

## plentiful presets



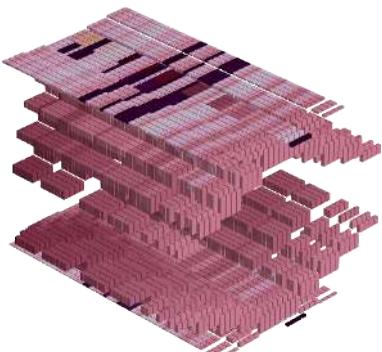
Preset 60.

```
preset60
.BYTE $02 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $03 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $01 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $15 ; smoothingDelay
.BYTE $02 ; currentSymmetry
.BYTE $02 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,DARK_ORANGE1 ; colorPalette
.BYTE DARK_ORANGE2,DARK_ORANGE4 ; colorPalette
.BYTE DARK_ORANGE6,COBALT_BLUE8 ; colorPalette
.BYTE COBALT_BLUE10,COBALT_BLUE12 ; colorPalette
.BYTE $00,$01,$01,$01,$01,$00,$00,$00 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode
```



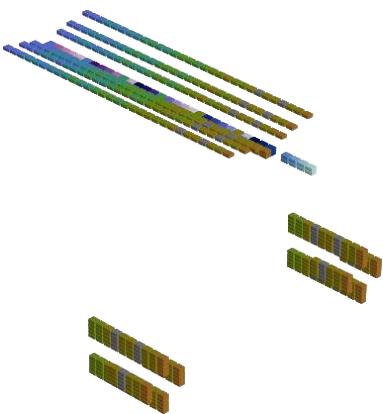
Preset 61.

```
preset61
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $02 ; speedBoostAdjust
.BYTE $09 ; currentPatternIndex
.BYTE $02 ; verticalResolutionSPAC
.BYTE $02 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $08 ; smoothingDelay
.BYTE $02 ; currentSymmetry
.BYTE $00 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,DARK_ORANGE1 ; colorPalette
.BYTE DARK_ORANGE2,DARK_ORANGE4 ; colorPalette
.BYTE DARK_ORANGE6,DARK_ORANGE8 ; colorPalette
.BYTE DARK_ORANGE10,DARK_ORANGE12 ; colorPalette
.BYTE $00,$03,$03,$03,$03,$03,$03,$03 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$40,$40,$40,$40,$40,$40,$40 ; oozeCycles
.BYTE $00 ; explosionMode
```



Preset 62.

```
preset62
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $02 ; speedBoostAdjust
.BYTE $14 ; currentPatternIndex
.BYTE $02 ; verticalResolutionSPAC
.BYTE $02 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $08 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $04 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,DARK_ORANGE1 ; colorPalette
.BYTE DARK_ORANGE2,DARK_ORANGE4 ; colorPalette
.BYTE DARK_ORANGE6,DARK_ORANGE8 ; colorPalette
.BYTE DARK_ORANGE10,DARK_ORANGE12 ; colorPalette
.BYTE $00,$03,$03,$03,$03,$03,$03,$03 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$40,$40,$40,$41,$40,$40,$40 ; oozeCycles
.BYTE $00 ; explosionMode
```

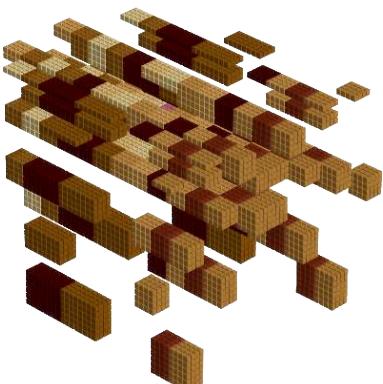


Preset 63.

```

preset63
.BYTE $02 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $09 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $6B ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $02 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,RUST8 ; colorPalette
.BYTE DARK_ORANGE8,DARK_LAVENDER8 ; colorPalette
.BYTE ULTRAMARINE_BLUE8,DARK_BLUE8 ; colorPalette
.BYTE OLIVE_GREEN8,DARK_GREEN8 ; colorPalette
.BYTE $00,$03,$03,$03,$03,$03,$03 ; oozeRates
.BYTE $01,$10,$10,$10,$10,$10,$10 ; oozeSteps
.BYTE $0F,$FF,$FF,$FF,$FF,$FF,$FF ; oozeCycles
.BYTE $00 ; explosionMode

```

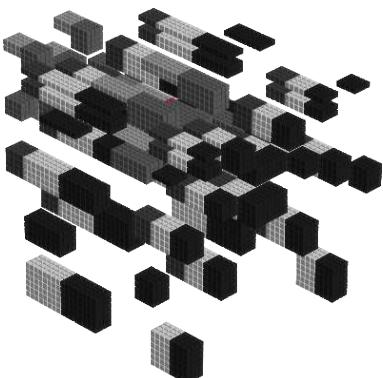


Preset 64.

```

preset64
.BYTE $04 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $04 ; speedBoostAdjust
.BYTE $01 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $08 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $03 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,BLACK2 ; colorPalette
.BYTE BLACK14,BLACK4 ; colorPalette
.BYTE BLACK12,BLACK6 ; colorPalette
.BYTE BLACK10,BLACK8 ; colorPalette
.BYTE $00,$02,$02,$02,$02,$02,$02 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$FF,$FF,$FF,$FF,$FF,$FF ; oozeCycles
.BYTE $00 ; explosionMode

```



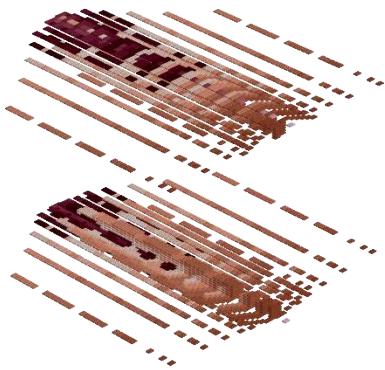
Preset 65.

```

preset65
.BYTE $04 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $04 ; speedBoostAdjust
.BYTE $01 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $08 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $03 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $02 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK7,BLACK2 ; colorPalette
.BYTE BLACK14,BLACK4 ; colorPalette
.BYTE BLACK12,BLACK6 ; colorPalette
.BYTE BLACK10,BLACK8 ; colorPalette
.BYTE $00,$00,$00,$00,$00,$00 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$FF,$FF,$FF,$FF,$FF ; oozeCycles
.BYTE $00 ; explosionMode

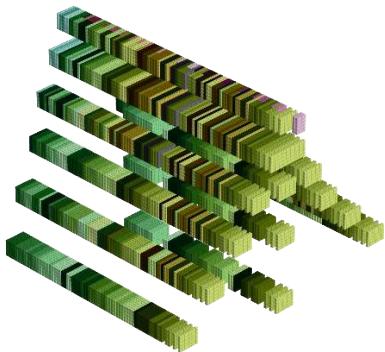
```

## plentiful presets



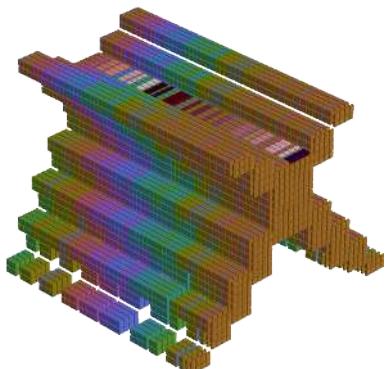
Preset 66.

```
preset66
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $13 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $08 ; smoothingDelay
.BYTE $02 ; currentSymmetry
.BYTE $06 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,DARK_ORANGE1 ; colorPalette
.BYTE DARK_ORANGE2,DARK_ORANGE4 ; colorPalette
.BYTE DARK_ORANGE6,DARK_ORANGE8 ; colorPalette
.BYTE DARK_ORANGE10,DARK_ORANGE12 ; colorPalette
.BYTE $00,$03,$03,$03,$03,$03,$03 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode
```



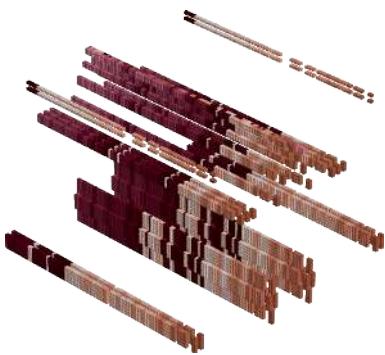
Preset 67.

```
preset67
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $11 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $00 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,OLIVE_GREEN4 ; colorPalette
.BYTE OLIVE_GREEN6,OLIVE_GREEN8 ; colorPalette
.BYTE OLIVE_GREEN10,OLIVE_GREEN12 ; colorPalette
.BYTE OLIVE_GREEN10,OLIVE_GREEN8 ; colorPalette
.BYTE $00,$01,$01,$01,$01,$01,$01 ; oozeRates
.BYTE $01,$02,$02,$03,$03,$05,$06 ; oozeSteps
.BYTE $0F,$20,$20,$20,$20,$20,$20 ; oozeCycles
.BYTE $00 ; explosionMode
```



Preset 68.

```
preset68
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $02 ; speedBoostAdjust
.BYTE $14 ; currentPatternIndex
.BYTE $02 ; verticalResolutionSPAC
.BYTE $02 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $05 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $00 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK4,RUST8 ; colorPalette
.BYTE DARK_ORANGE8,DARK_LAVENDER8 ; colorPalette
.BYTE ULTRAMARINE_BLUE8,DARK_BLUE8 ; colorPalette
.BYTE OLIVE_GREEN8,DARK_GREEN8 ; colorPalette
.BYTE $05,$01,$01,$01,$01,$01,$01 ; oozeRates
.BYTE $10,$10,$10,$10,$10,$10,$10 ; oozeSteps
.BYTE $0F,$FF,$FF,$FF,$FF,$FF,$FF ; oozeCycles
.BYTE $02 ; explosionMode
```

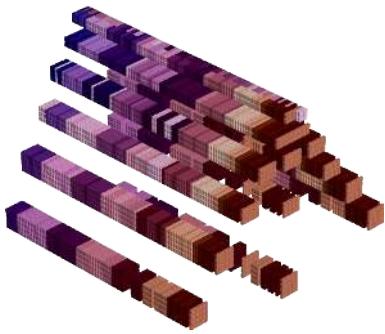


Preset 69.

```

preset69
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $10 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $08 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $02 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,DARK_ORANGE1 ; colorPalette
.BYTE DARK_ORANGE2,DARK_ORANGE4 ; colorPalette
.BYTE DARK_ORANGE6,DARK_ORANGE8 ; colorPalette
.BYTE DARK_ORANGE10,DARK_ORANGE12 ; colorPalette
.BYTE $00,$03,$03,$03,$03,$03,$03 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode

```

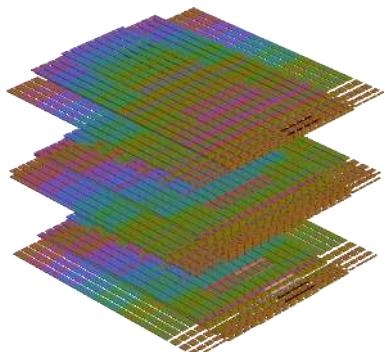


Preset 70.

```

preset70
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $11 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $08 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $02 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,DARK_ORANGE1 ; colorPalette
.BYTE DARK_ORANGE2,DARK_ORANGE4 ; colorPalette
.BYTE DARK_ORANGE6,DARK_ORANGE8 ; colorPalette
.BYTE DARK_ORANGE10,DARK_ORANGE12 ; colorPalette
.BYTE $00,$03,$03,$03,$03,$03,$03 ; oozeRates
.BYTE $01,$08,$08,$08,$08,$08,$08 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode

```



Preset 71.

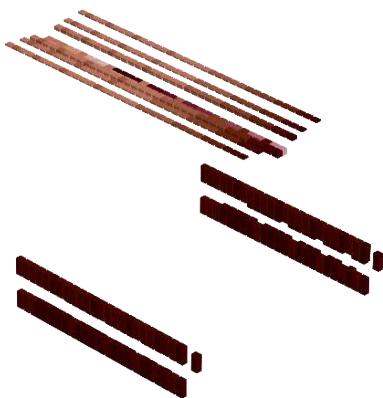
```

preset71
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $02 ; speedBoostAdjust
.BYTE $07 ; currentPatternIndex
.BYTE $06 ; verticalResolutionSPAC
.BYTE $02 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $05 ; smoothingDelay
.BYTE $04 ; currentSymmetry
.BYTE $01 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,RUST8 ; colorPalette
.BYTE DARK_ORANGE8,DARK_LAVENDER8 ; colorPalette
.BYTE ULTRAMARINE_BLUE8,DARK_BLUE8 ; colorPalette
.BYTE OLIVE_GREEN8,DARK_GREEN8 ; colorPalette
.BYTE $00,$01,$01,$01,$01,$01,$01 ; oozeRates
.BYTE $10,$10,$10,$10,$10,$10,$10 ; oozeSteps
.BYTE $0F,$FF,$FF,$FF,$FF,$FF,$FF ; oozeCycles
.BYTE $00 ; explosionMode

```

## plentiful presets

---



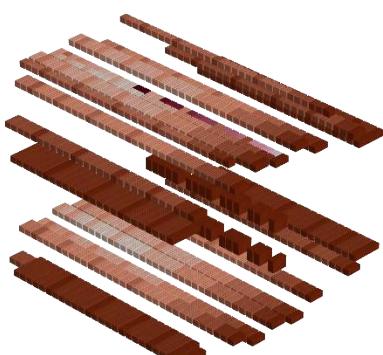
Preset 72.

```
preset72
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $09 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $02 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $03 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,DARK_ORANGE1 ; colorPalette
.BYTE DARK_ORANGE2,DARK_ORANGE4 ; colorPalette
.BYTE DARK_ORANGE6,DARK_ORANGE8 ; colorPalette
.BYTE DARK_ORANGE10,DARK_ORANGE12 ; colorPalette
.BYTE $00,$00,$00,$00,$00,$00 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode
```



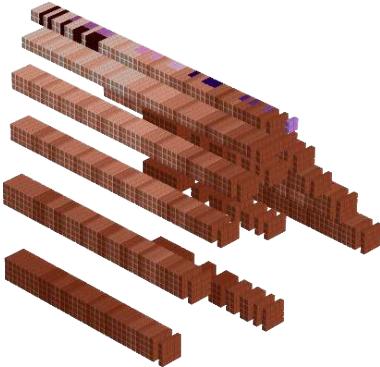
Preset 73.

```
preset73
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $0B ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $01 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $06 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,DARK_ORANGE1 ; colorPalette
.BYTE DARK_ORANGE2,DARK_ORANGE4 ; colorPalette
.BYTE DARK_ORANGE6,DARK_ORANGE8 ; colorPalette
.BYTE DARK_ORANGE10,DARK_ORANGE12 ; colorPalette
.BYTE $00,$14,$14,$14,$14,$14 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$00,$00,$00,$00,$00 ; oozeCycles
.BYTE $00 ; explosionMode
```



Preset 74.

```
preset74
.BYTE $02 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $0B ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $01 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $04 ; currentSymmetry
.BYTE $00 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,DARK_ORANGE1 ; colorPalette
.BYTE DARK_ORANGE2,DARK_ORANGE4 ; colorPalette
.BYTE DARK_ORANGE6,DARK_ORANGE8 ; colorPalette
.BYTE DARK_ORANGE10,DARK_ORANGE12 ; colorPalette
.BYTE $00,$14,$14,$14,$14,$14 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$00,$00,$00,$00,$00 ; oozeCycles
.BYTE $00 ; explosionMode
```

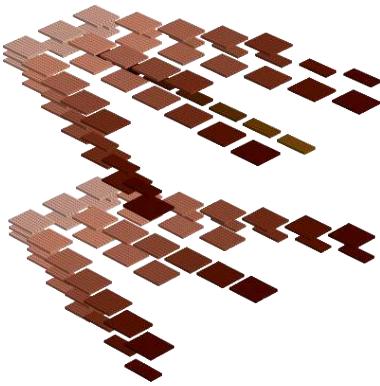


Preset 75.

```

preset75
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $02 ; speedBoostAdjust
.BYTE $11 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $02 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $10 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $02 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,DARK_ORANGE1 ; colorPalette
.BYTE DARK_ORANGE2,DARK_ORANGE4 ; colorPalette
.BYTE DARK_ORANGE6,DARK_ORANGE8 ; colorPalette
.BYTE DARK_ORANGE10,DARK_ORANGE12 ; colorPalette
.BYTE $00,$14,$14,$14,$14,$14,$14 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$00,$00,$00,$00,$00,$00 ; oozeCycles
.BYTE $00 ; explosionMode

```

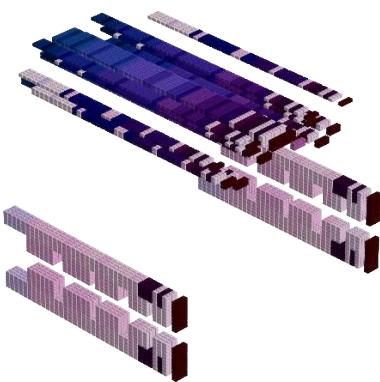


Preset 76.

```

preset76
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $02 ; speedBoostAdjust
.BYTE $01 ; currentPatternIndex
.BYTE $02 ; verticalResolutionSPAC
.BYTE $04 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $06 ; smoothingDelay
.BYTE $04 ; currentSymmetry
.BYTE $00 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $01 ; stroboscopicsEnabled
.BYTE $01 ; stroboFlashRate
.BYTE BLACK5,DARK_ORANGE1 ; colorPalette
.BYTE DARK_ORANGE2,DARK_ORANGE4 ; colorPalette
.BYTE DARK_ORANGE6,DARK_ORANGE8 ; colorPalette
.BYTE DARK_ORANGE10,DARK_ORANGE12 ; colorPalette
.BYTE $00,$14,$14,$14,$14,$14,$14 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$00,$00,$00,$00,$00,$00 ; oozeCycles
.BYTE $00 ; explosionMode

```



Preset 77.

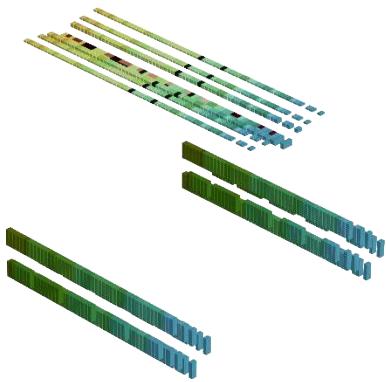
```

preset77
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $05 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $08 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $02 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,DARK_ORANGE1 ; colorPalette
.BYTE DARK_ORANGE2,DARK_ORANGE4 ; colorPalette
.BYTE DARK_ORANGE6,DARK_ORANGE8 ; colorPalette
.BYTE DARK_ORANGE10,DARK_ORANGE12 ; colorPalette
.BYTE $00,$03,$03,$03,$03,$03,$03 ; oozeRates
.BYTE $01,$0F,$0F,$0F,$0F,$0F,$0F ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode

```

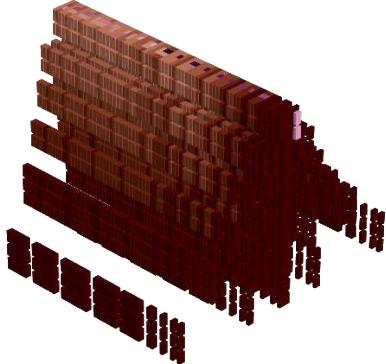
## plentiful presets

---



```
preset78
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $09 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $00 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $08 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $03 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $02 ; stroboFlashRate
.BYTE BLACK0,DARK_ORANGE1 ; colorPalette
.BYTE DARK_ORANGE2,DARK_ORANGE4 ; colorPalette
.BYTE DARK_ORANGE6,DARK_ORANGE8 ; colorPalette
.BYTE DARK_ORANGE10,DARK_ORANGE12 ; colorPalette
.BYTE $01,$03,$03,$03,$03,$03,$03 ; oozeRates
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $01 ; explosionMode
```

Preset 78.

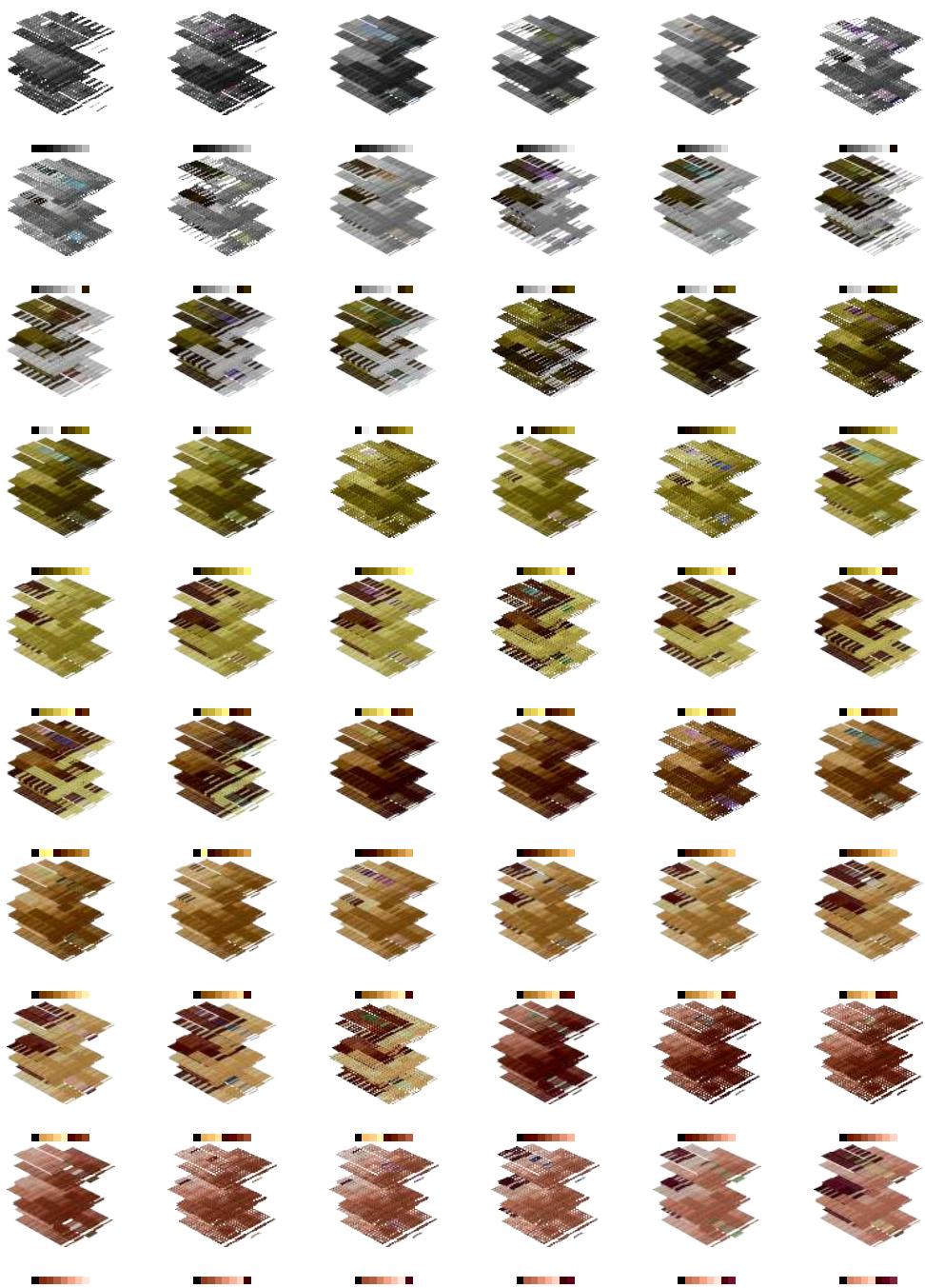


```
preset79
.BYTE $01 ; cursorSpeed
.BYTE $01 ; vectorMode
.BYTE $01 ; speedBoostAdjust
.BYTE $13 ; currentPatternIndex
.BYTE $05 ; verticalResolutionSPAC
.BYTE $01 ; pulseSpeed
.BYTE $00 ; currentPulseWidth
.BYTE $08 ; smoothingDelay
.BYTE $01 ; currentSymmetry
.BYTE $02 ; screenMode
.BYTE $40 ; bufferLength
.BYTE $00 ; stroboscopicsEnabled
.BYTE $05 ; stroboFlashRate
.BYTE DARK_ORANGE13,DARK_ORANGE1 ; colorPalette
.BYTE DARK_ORANGE2,DARK_ORANGE4 ; colorPalette
.BYTE DARK_ORANGE6,DARK_ORANGE8 ; colorPalette
.BYTE DARK_ORANGE10,DARK_ORANGE12 ; colorPalette
.BYTE $00,$00,$00,$00,$00,$00,$00 ; oozeRates
.BYTE $01,$01,$01,$01,$01,$01,$01 ; oozeSteps
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F ; oozeCycles
.BYTE $00 ; explosionMode
```

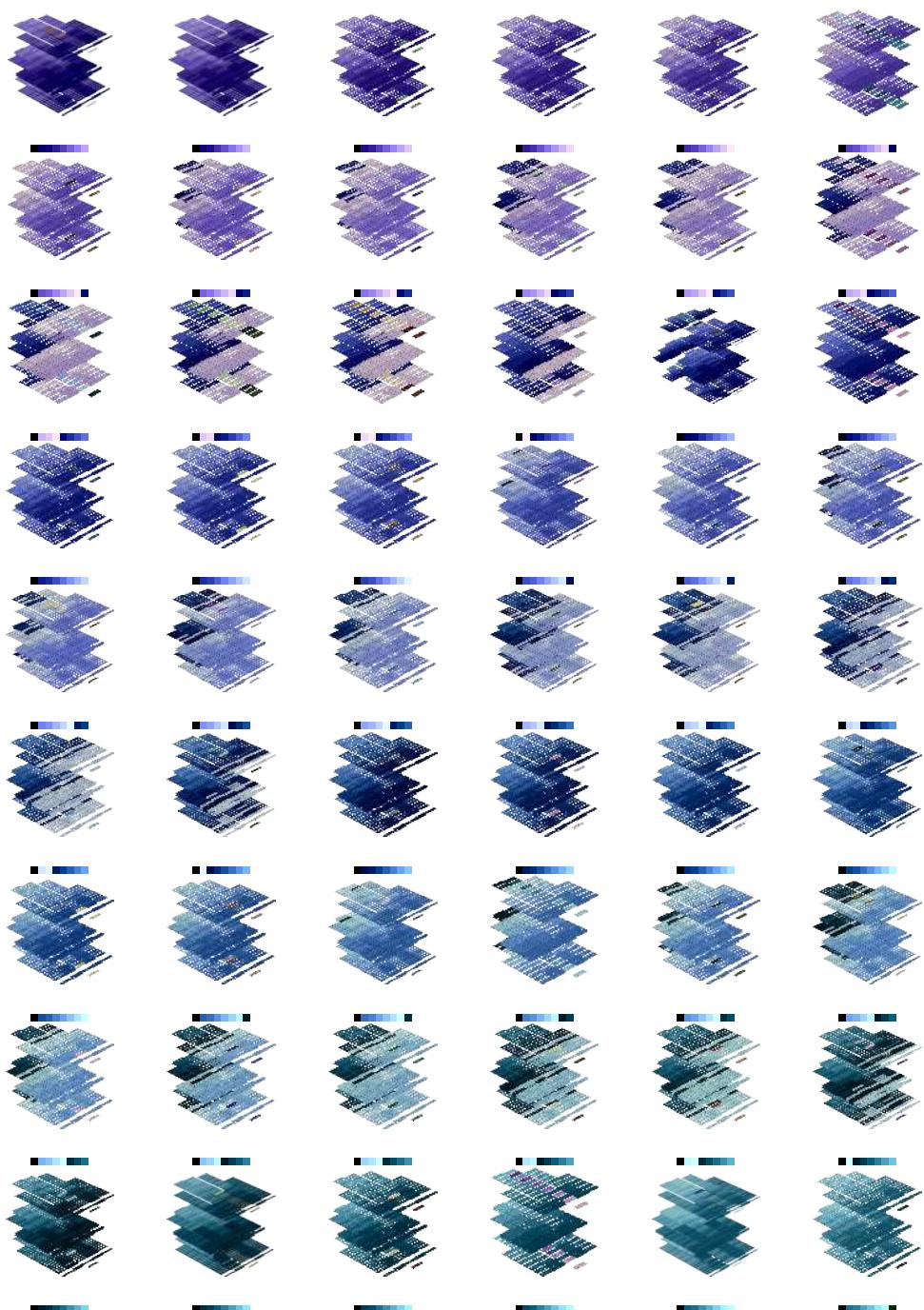
Preset 79.

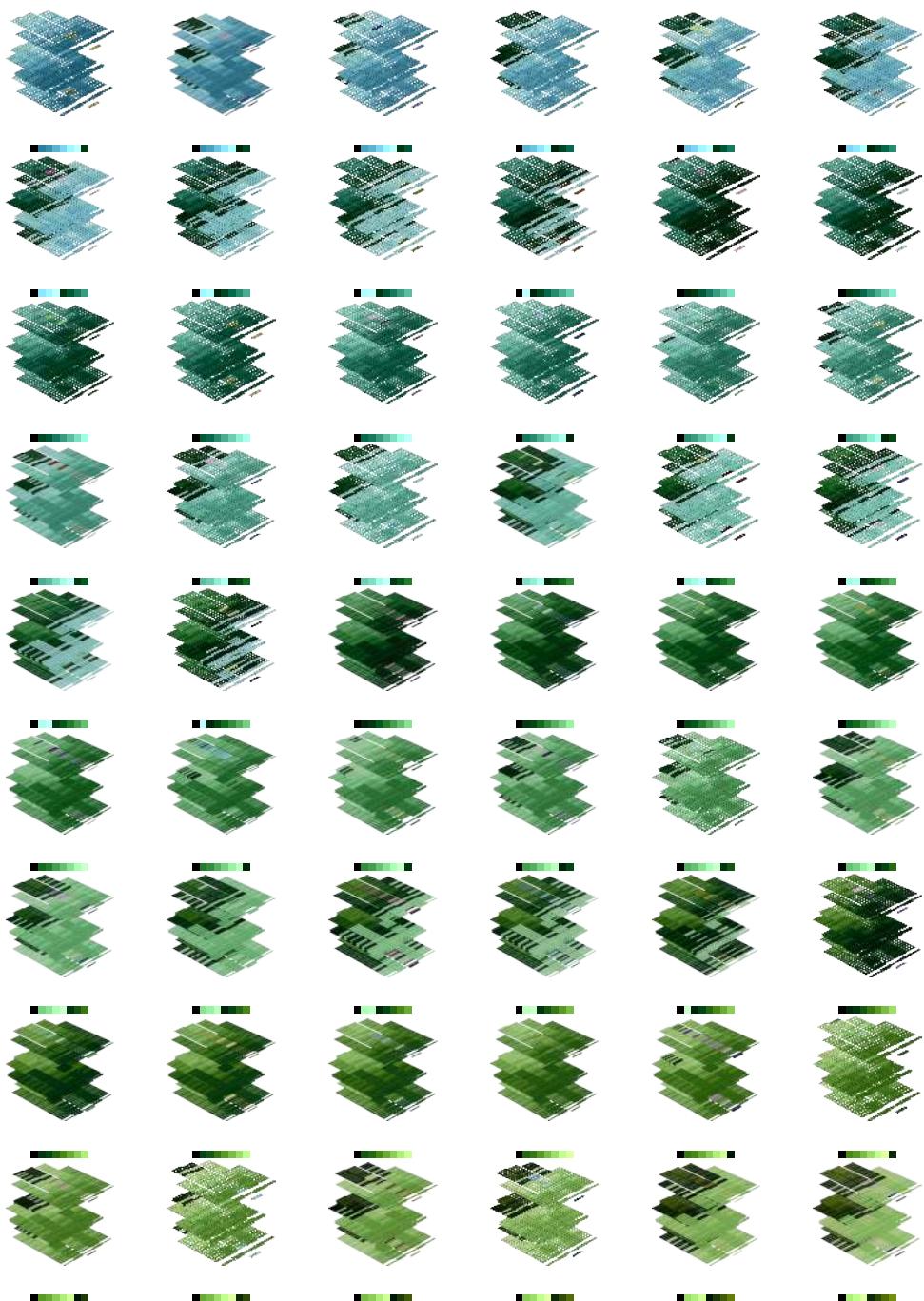
# **colourflow**

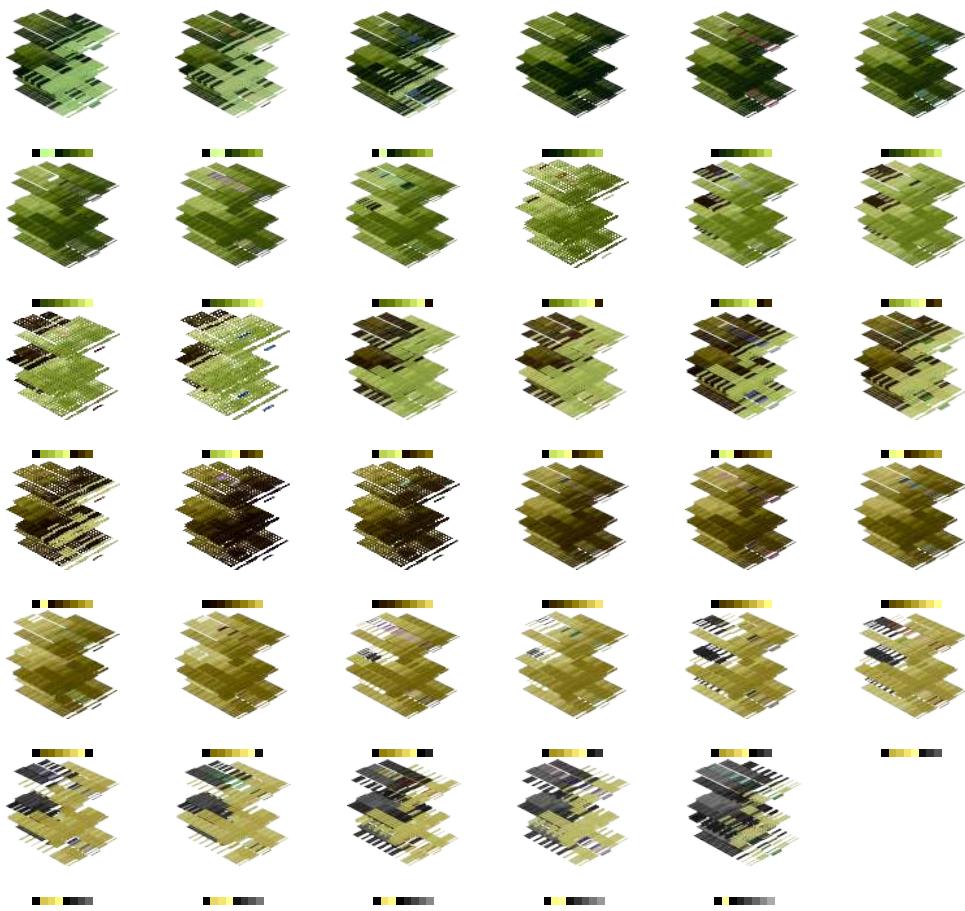
Finally let's take a look at all the potential colour schemes.











# Index

ActivateSequencer, 80, 85, 160, 161  
ActuallyPaintPixel, 18, 33, 34, 38, 62,  
63, 76, 224, 238

baseLevel, 79, 85, 90, 91, 152, 164,  
171–174, 176, 177, 180, 181,  
184, 185

Batalyx, 221, 223, 228, 229, 232, 233

bufferLength, 78, 79, 83–85, 90, 91,  
150, 153, 164, 171–174, 176,  
177, 180, 181, 184–186, 195,  
202, 207, 208

burstGeneratorF1, 84, 92, 141, 146, 147

burstGeneratorF2, 84, 92

burstGeneratorF3, 84, 92, 145–147

burstGeneratorF4, 84, 92

burstSmoothingDelay, 85, 150–152

C64, 16, 34, 39, 41–43, 53, 93, 221, 223,  
225

CheckIfCursorMovedLeftOrRight, 40,  
78

CheckIfEnterPressed, 83

CheckIfPlayerPressedFire, 40, 68, 70,  
71, 78, 79

CheckIfPresetKeysPressed, 174

CheckKeyboardInput, 79–81, 114, 115,  
128, 129, 142, 143, 174, 175,  
194, 195, 200, 206, 207, 212,  
222, 226, 230

CheckKeyboardWhilePromptActive,  
83, 84

ClearLastLineOfScreen, 78, 80, 82–86,  
88, 89, 114, 128, 146, 178, 200,  
212

colorBarScreenRamHiPtr, 82, 83

colorBarScreenRamLoPtr, 82, 83

colorRamHiPtr, 38, 44–47, 76

colorRAMLineTableHiPtrArray, 38, 40,  
44–46, 63, 72, 73, 76, 79

colorRAMLineTableLoPtrArray, 38, 40,  
44, 46, 63, 72, 73, 76, 79

colorRamLoPtr, 38, 44–47, 76

Controls, 171, 199

countStepsBeforeCheckingJoystickInput,  
66

countToMatchCurrentIndex, 29, 30, 38,  
52, 56, 58, 76, 110

currentColorBarOffset, 82, 83

currentColorIndexArray, 25–28, 30, 31,  
40, 50, 52–55, 70, 71, 78, 79,  
82–85, 150, 152, 202, 214,  
215, 218, 226, 230, 231

currentColorSet, 80, 83

currentColorToPaint, 40, 66, 72, 73, 78,  
79, 86, 202

currentDataFree, 84, 85, 146

currentIndexToPixelBuffers, 40, 78, 82,  
84, 136, 218, 230

currentIndexToPresetValue, 86, 88

- currentModeActive, 78, 83, 84, 86, 88, 89, 188  
currentNodeInColorBar, 82, 83  
currentPatternElement, 78–80, 84, 85, 88, 90, 91, 114, 115, 171–173, 176, 177, 180–182, 184, 185, 188, 189  
currentPulseSpeedCounter, 79, 83, 208  
currentPulseWidth, 79, 86, 208, 209  
currentSequencePtrHi, 84, 85, 146, 147, 151–153, 160, 161, 166, 167  
currentSequencePtrLo, 84, 85, 146, 147, 150–153, 160, 161, 164–167  
currentStepCount, 79, 83–85, 150, 153, 164, 202, 208  
currentSymmetrySetting, 78–80, 84, 85, 90–95, 128, 129, 137, 141, 145, 146, 149, 155, 159, 171–173, 176, 177, 180–182, 184, 185, 188, 189, 226, 227  
currentSymmetrySettingForStep, 38, 60, 61, 76, 78, 88, 132–134, 136, 137, 196, 224, 226, 230, 236, 237  
currentValueInColorIndexArray, 29–34, 54, 56–58, 62, 76, 78, 82, 88, 214–219  
currentVariableMode, 80, 82–86, 88, 89, 142, 143, 150, 160, 174, 194, 195, 206, 207  
cursorXPosition, 27, 40, 41, 64, 68–73, 78, 79, 85, 86, 88, 202, 224, 226  
cursorYPosition, 27, 40, 41, 64, 66, 67, 70–73, 78, 79, 85, 86, 88, 202, 214, 215, 224, 226  
customPattern0XPosArray, 79, 96, 106, 120  
customPattern0YPosArray, 79, 96, 106, 120  
customPattern1XPosArray, 79, 96, 106, 121  
customPattern1YPosArray, 79, 96, 106, 121  
customPattern2XPosArray, 79, 96, 106, 121  
customPattern2YPosArray, 79, 96, 106, 121  
customPattern3XPosArray, 79, 96, 106, 122  
customPattern3YPosArray, 79, 96, 106, 122  
customPattern4XPosArray, 79, 96, 106, 122  
customPattern4YPosArray, 79, 96, 106, 122  
customPattern5XPosArray, 79, 97, 106, 123  
customPattern5YPosArray, 79, 97, 106, 123  
customPattern6XPosArray, 79, 97, 106, 107, 123  
customPattern6YPosArray, 79, 97, 106, 107, 123  
customPattern7XPosArray, 79, 97, 106, 107  
customPattern7YPosArray, 79, 97, 106, 107  
customPatternIndex, 86, 88  
customPatternLoPtr, 86, 88  
customPromptsActive, 84, 85, 88  
  
dataFreeDigitOne, 82, 84, 85  
dataFreeDigitThree, 82–85  
dataFreeForSequencer, 84–86  
DecrementDataFreeCounterAndLoop, 84  
DecrementPulseSpeedCounter, 79, 208  
deltoidXPosArray, 79, 80, 106, 107, 116  
deltoidYPosArray, 79, 80, 106, 116  
demoModeActive, 82, 83, 86, 89  
diffusedXPosArray, 79, 80, 106, 107, 117  
diffusedYPosArray, 79, 80, 106, 117

DisplayLoadOrAbort, 80, 88, 89  
DisplayPresetMessage, 80, 83, 174, 175, 178, 179  
displaySavePromptActive, 79, 86, 88  
DisplaySavePromptScreen, 78, 88  
DisplaySequencerState, 82, 85, 160  
DisplayStoppedRecording, 86  
DisplayVariableSelection, 83  
DrawColorValueBar, 82, 83  
DrawCursorAndReturnFromInterrupt, 40, 70–73, 79, 202, 203, 208  
dynamicStorage, 86, 97  
dynamicStorageHiPtr, 86  
DynamicStorageInitLoop, 86  
  
EnterPressed, 88  
enterWasPressed, 83, 84  
  
functionKeyIndex, 80, 84, 142, 143, 146, 147  
  
GetCustomPatternElement, 80, 88, 114  
GetJoystickInput, 78, 86  
GetPresetPointersUsingXRegister, 84, 182, 186, 187  
  
HandleCustomPreset, 78, 88  
  
indexForColorBarDisplay, 79, 83, 90, 91, 171–174, 176, 177, 180, 181, 184, 185  
indexOfCurrentColor, 18, 33, 34, 62, 76  
initialBaseLevelForCustomPresets, 86, 88  
InitializeDynamicStorage, 76, 86  
InitializeProgram, 38, 42–45, 76, 222  
InitializeScreenWithInitCharacter, 38, 76, 84, 88  
initialSmoothingDelayArray, 78, 79, 85, 226, 230  
  
Jeffrey Says, 13, 21, 75, 77, 87, 99, 125, 139, 157, 169, 191, 193, 199, 205, 211, 223  
  
joystickInputDebounce, 89  
joystickInputRandomizer, 89  
JustLPressed, 80, 212, 213  
JustSPressed, 80, 128  
  
laLlamitaXPosArray, 79, 106, 107, 113, 119  
laLlamitaYPosArray, 79, 106, 107, 113, 119  
lastJoystickInput, 78, 79, 86, 89, 224, 226  
lastKeyPressed, 78, 80, 83, 84, 88, 89, 114, 115, 128, 129, 142, 143, 194, 206  
LaunchPsychedelia, 38, 40, 44, 54, 55, 65, 76, 78, 222, 230, 231  
lineModeActivated, 79, 80, 90, 91, 171–174, 176, 177, 180, 181, 184, 185, 212–215  
LoadBurstData, 84, 85, 146, 147, 150–152  
LoadDataForSequencer, 78, 85, 162–166  
LoadOrProgramBurstGenerator, 80, 84, 142, 143, 146, 147  
LoadSelectedPresetSequence, 84, 186–189  
LoadXAndYOfCursorPosition, 40, 41, 72, 73, 79  
LoadXAndYPosition, 17, 31, 38, 62, 63, 76  
  
MainPaintLoop, 25, 27, 40, 41, 53–55, 57, 78, 82, 88, 196, 203, 209, 214, 215, 218  
maxToDrawOnColorBar, 82, 83  
MaybeDisplayDemoModeMessage, 80, 89  
MaybeEditCustomPattern, 83, 86  
MaybeFunctionKeysPressed, 143  
MaybeInColorMode, 83, 194, 206  
MaybeQPressed, 80, 142, 143, 160  
MaybeTPressed, 200  
minIndexToColorValues, 86, 88

Minter, 11, 13, 15, 16, 18, 20, 53, 118, 221  
MovePresetDataIntoPosition, 76, 89  
MoveToNextBurstPosition, 85, 150,  
  152, 153  
MoveToNextPositionInSequencer, 85,  
  164, 166, 167  
multicrossXPosArray, 79, 82, 106, 117  
multicrossYPosArray, 79, 82, 106, 117  
  
NoMoreBytes, 86  
  
offsetForYPos, 82, 216  
offsetToColorBar, 82, 83  
  
PaintCursorAtCursorPosition, 40, 41,  
  66, 72, 73, 78, 79, 86, 202  
PaintLineMode, 78, 82, 214–218  
PaintPixel, 16, 17, 31, 33, 34, 38, 39,  
  60–63, 76, 78, 132–135, 217,  
  224, 227, 235–239  
PaintPixelForCurrentSymmetry, 29, 31,  
  38, 39, 56–61, 76, 77, 82, 110,  
  125, 132–135, 137, 216–219,  
  224, 226, 227, 234, 236, 237  
PaintStructureAtCursorPosition,  
  25–31, 38–40, 54–57, 76–78,  
  88, 110, 111, 215, 226, 227,  
  230, 233–235  
patternIndex, 76, 78, 79, 88, 110, 111,  
  136, 174  
patternIndexArray, 78, 79, 85, 94, 136,  
  141, 145, 149, 151, 152, 155,  
  164, 167, 196  
PixelPaintLoop, 38, 76, 110, 226, 234  
pixelShapeIndex, 82, 88  
pixelXPosition, 17, 25, 29, 31, 38, 40,  
  54, 56–63, 76, 78, 88, 107,  
  110, 111, 132–136  
pixelXPositionArray, 25–28, 38, 40, 50,  
  52–55, 70, 71, 78, 79, 85, 136,  
  151–153, 164, 167, 196, 202,  
  203, 209, 226, 230  
pixelYPosition, 17, 25, 29, 31, 38, 40,  
  54, 56–63, 76, 78, 82, 88, 107,  
  110, 132–136, 216–219  
pixelYPositionArray, 25–28, 38, 40, 50,  
  52–55, 70, 71, 78, 79, 85, 136,  
  151, 152, 164, 167, 196, 202,  
  203, 209, 226, 230  
playbackOrRecordActive, 83, 86, 88, 89  
Popular Computing Weekly, 11, 13, 37  
presetColorValuesArray, 18, 24, 26,  
  30–34, 38, 62, 76, 79, 83, 84,  
  90, 91, 171–174, 176, 177, 180,  
  181, 184–186, 224, 226, 230  
presetHiPtr, 88  
presetLoPtr, 88  
presetSequenceData, 84, 88, 89, 187  
presetSequenceDataHiPtr, 84, 187  
presetSequenceDataLoPtr, 84, 182,  
  186–188  
presetTempHiPtr, 88  
presetTempLoPtr, 88  
presetValueArray, 79, 83, 84, 174, 175,  
  182, 188, 189, 194, 195, 206,  
  207  
previousCursorPosition, 86, 110, 111  
previousCursorPosition, 86  
previousIndexToPixelBuffers, 76, 78,  
  79, 82, 84, 85, 202, 203, 218  
previousPixelXPosition, 76  
previousPixelYPosition, 76, 110  
prevSequencePtrHi, 84–86  
prevSequencePtrLo, 84–86  
prevSymmetrySetting, 85  
PromptToSave, 80, 88, 128, 129  
pubescent, 20  
pulsarXPosArray, 79, 82, 106, 120  
pulsarYPosArray, 79, 82, 106, 120  
PutRandomByteInAccumulator, 38, 50,  
  51, 76, 89  
  
ReachedLastColor, 83  
recordingOffset, 86  
recordingStorageHiPtr, 86  
recordingStorageLoPtr, 86  
RecordJoystickMovements, 86

RefreshPresetData, 84, 182, 183, 186, 187  
ReinitializeScreen, 78, 84, 88  
ReinitializeSequences, 40, 50, 51, 54, 55, 65, 78  
ResetCurrentActiveMode, 84, 186  
ResetSavingStateAndReturn, 88  
ResetSelectedVariableAndReturn, 83  
ResetStateAndClearPrompt, 89  
RestorePositionsAndReturn, 38, 56, 58, 59, 76  
ReturnEarlyFromRoutine, 76  
ReturnFromPromptToSave, 88  
ReturnFromSave, 89  
ReturnPressed, 84, 85  
  
SelectNewPreset, 83, 89  
sequencerActive, 80, 82, 84, 85, 142, 146, 152, 160, 161  
SequencerNotActiveCheckJoystickInput, 162, 163  
sequencerSpeed, 78, 79, 85, 90, 91, 160–162, 171–174, 176, 177, 180, 181, 184, 185  
SetUpInterruptHandlers, 40, 78  
shiftKey, 80, 85, 114, 226  
shiftPressed, 80, 83–86, 89, 114, 128, 146, 147, 160, 178, 182  
skipPixel, 76, 78, 82  
smoothingDelay, 40, 70, 71, 79, 84, 85, 90–95, 141, 145, 146, 149, 155, 159, 171–174, 176, 177, 180, 181, 184, 185, 195–197, 207  
smoothingDelayArray,X, 78, 79, 85, 226, 230  
starOneXPosArray, 23–25, 28, 29, 31, 38, 50, 51, 58, 59, 76, 79, 100–102, 105–107  
starOneYPosArray, 23–25, 28, 29, 31, 38, 58, 76, 79, 100–102, 105–107  
starTwoXPosArray, 79, 106, 107, 116  
starTwoYPosArray, 79, 106, 116  
statusLineBuffer, 80, 82–86, 88, 89, 183  
stepsRemainingInSequencerSequence, 78, 82, 83, 85, 88, 89, 160, 162, 163, 165  
stepsSincePressedFire, 40, 70, 71, 79, 208  
StopOrStartRecording, 82, 86  
symmetrySettingForStepCount, 78, 79, 85, 136, 137, 152, 164, 165, 196  
  
theTwistXPosArray, 79, 106, 107, 109, 111  
theTwistYPosArray, 79, 106, 107, 109, 111  
timerBetweenKeyStrokes, 78, 80, 83, 114  
trackingActivated, 79, 80, 85, 90, 91, 150, 164, 171–174, 176, 177, 180, 181, 184, 185, 200–203, 213  
  
UpdateCurrentActivePreset, 84, 178, 179, 182, 183  
UpdateDataFreeDisplay, 84, 85  
UpdateDataFreeLoop, 84  
  
VIC20, 13  
Virtual Light Machine, 221  
  
WriteLastLineBufferAndReturn, 84, 178, 182, 188  
WriteLastLineBufferToScreen, 80, 82–86, 88, 89, 114, 128, 146, 178, 183, 200, 212

