
IRIDIS

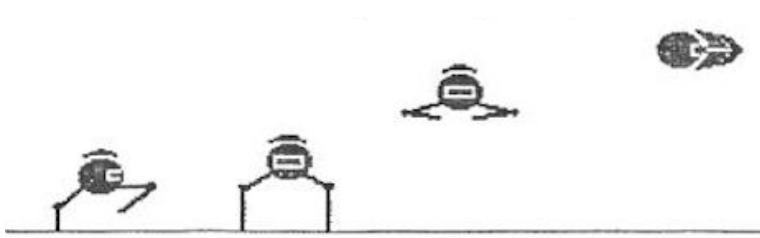
ALPHA

THEORY

SCRAPBOOK EDITION

ROB HOGAN

IRIDIS ALPHA THEORY



Concept sketch for Iridis Alpha by Jeff Minter.

© Rob Hogan 2022-2023, All Rights Reserved.
'Iridis Alpha' © Jeff Minter 1986.

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



Contents

1 We Need to Talk About Binary	13
2 A Little Archaeology	21
2.1 The Madness Begins	23
2.2 After Our First Real Byte	26
2.3 A Loader for your Loader	30
2.4 Putting an End to the Madness	39
3 Some Disassembly Required	43
3.0.1 Important Concept Number One: High Bytes and Low Bytes . . .	47
3.0.2 Important Concept Number Two: Interrupts	48
4 The First 16 Milliseconds	51
4.1 Sprites	52
4.2 Waiting for the Beam	53
4.2.1 Drawing the Stripes	57
4.2.2 Drawing the Text	60
4.3 Racing the Beam	63

CONTENTS

4.3.1	A Complication	67
4.3.2	Back to the Beam	70
4.4	Enter The Gilbies	73
4.5	Title Text	77
5	Making Planets for Nigel	81
5.1	Step One: Creating the Sea	84
5.2	Step Two: Creating the Land	87
5.3	Step Three: Structures Structures Structures	91
5.4	Step Four: Add the warp gate	96
5.5	Inactive Lower Planet	97
5.6	Drawing the Lower Planet	98
5.6.1	Flipping the Byte	103
5.6.2	Flipping the Byte's Position	106
6	Blasting, Fast and Slow	111
6.1	Updating Enemy Sprite Positions	114
6.2	Scrolling the Planets	115
6.2.1	Pixel Movement	116
6.2.2	Character Movement	118
6.3	Jumping Up and Down	123
6.4	Sound Effects	128
6.4.1	The Sound Effect Data Structure	128
6.4.2	Multiplexing	133
6.4.3	Unused Sound Effects	138

7 Enemies and their Discontents	141
7.1 Something Simple - Byte 0: The Enemy's Color	143
7.2 Bytes 1-4: Sprite Animation	144
7.3 Bytes 18-21: Enemy Movement	148
7.3.1 What is going on with Byte 6?	150
7.4 Bytes 6-8: Alternate Enemy Waves	152
7.5 Bytes 22-23: The Bytes of Death	156
7.6 Byte 35: Energy Multiplier	158
7.7 Byte 34: Score for Hitting the Enemy	160
8 Congoatulations Hotshot	161
8.1 Entry Sequence	161
8.2 Generating Maps	165
8.2.1 Choosing a Map	174
8.3 Some Very Ugly Sprites	177
8.4 IBalls	177
9 A Hundred Thousand Billion Theme Tunes	179
9.1 Some Basics	182
9.1.1 Structure	184
9.1.2 Phrasing	188
9.1.3 Seeding the Random	193
10 Another 16^4 Tunes	199
10.1 Taurus:Torus	201

CONTENTS

10.2 Taurus/Torus Two	204
11 Made in France	209
12 A Pause Mode for your Pause Mode	211
13 An Oscillator in 4 Parts	221
14 Iridis Oops!	231
14.1 The Byte that Broke	231
14.2 Reappearing Enemies	233
14.3 A Sort of Cheat	234
A Sprite Atlas	235
B Enemy Data	243
C Planet Data	343
D 18/100,000,000,000,000 Theme Tunes	355
E Bonus Phase - Map Segments	377
F Bonus Phase - Map Rows	383
G Bonus Phase - Tilesheet	387
H Bumph	389
Notes & References	395

About This Book

This book describes the inner workings of a relatively obscure video game created by an eccentric Englishman by the name of Jeff Minter in 1986 for the Commodore 64.

If you are curious about old computers such as the Commodore 64 or the detailed mechanics of making a glorified digital breadboard produce something on a screen that flashes, bleeps, and fascinates then this book is hopefully for you. Iridis Alpha was not an enormous success on its release but it is widely regarded as one of the great achievements on the Commodore 64 hardware. It embodies an 8-bit arcade aesthetic that was unique to its time, it was slightly mad in its concept and execution, and its emphasis on speed and unforgiving gameplay influenced generations of game developers in the years that followed.

Since the source code of Iridis Alpha is no longer available, I have had to [unpack and reinterpret](#) it from the game binary Jeff Minter [released into the public domain in 2019](#). I describe this reverse-engineering process in the opening chapters because I think it is an interesting exercise in and of itself to go from a binary blob to a set of fully commented source code that provides insight to the inner workings of the game. [A Little Archaeology](#) describes how we extract the game binary from the cassette tape it was originally distributed. [Some Disassembly Required](#) shows you how to go from a very long list of bytes to a full source code listing. Hopefully you are here because you enjoy this kind of gory detail too.

If you are just interested in learning about the mechanics of the game itself you can flick straight to [The First 16 Milliseconds](#), which begins by covering the loading the title screen and all that goes on in there. [Making Planets for Nigel](#), [Blasting Fast and Slow](#), and [Enemies and Their Discontents](#) cover the main gameplay and dive into how the game levels are created and how the speed of the game is achieved. These chapters go deep into the game's reassembled code and look closely at how Jeff Minter designed its core engine.

I dedicate a full chapter [A Hundred Thousand Billion Theme Tunes](#) to the ingenious

procedural programming behind Iridis Alpha's theme tunes and trace its inspiration back to an article 'Musical Fractals' in a 1986 issue of Byte magazine. [Another 16⁴ Tunes](#) looks at the early experimentations with this fractal music in the demo 'Torus' Minter released while developing the game. [An Oscillator in 4 Parts](#) delves into the animation experiment in 'Torus', something that was used in the bonus screen in Iridis Alpha itself.

Iridis Alpha is full of little additional extras. [Congoatulations Hotshot!](#) unpacks the vertical scrolling mini-game Minter inserted as a bonus sequence. Even if the game itself is a little half-baked this bonus sequence is full of intrepid effects and some clever techniques for generating the game maps. [Made in France](#) and [A Pause Mode for Your Pause Mode](#) unpick the pause mode games bundled with the main game.

As a final coda, [Iridis Oops!](#) takes a look at some of the bugs that slipped through to the final release.

If you are reading the PDF version of this book the [Appendices](#) contain a data dump of sprite sheets, character sets, maps and tables that should provide hours of rewarding bedtime reading, or more likely, deep undisturbed sleep.

Note on the Text

The version you are reading is little more than a scrapbook still many 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, Dublin 2022-2023
[@mwenge](#)

IRIDIS ALPHA THEORY

We Need to Talk About Binary

Here we are, ground zero in every book about assembly programming. Where the reader finds themselves in an awkward sit-down discussion about the B word. It's a necessary evil of course. Nothing will ever make sense without a rudimentary but slowly growing grasp of what bytes are and why they keep being referred to in a confusing mixture of letters and numbers such as \$D012 and \$F4.

So what am I supposed to do with you here? Succeed where every other author has usually failed? I assume that's the expectation. OK here goes.

You already know, OK maybe you already know, that computers, even those in the 1980s, understand everything in terms of 1s and 0s. All it knows is a world in which something is on or off, high or low, present or absent. The reason for this is simply because yes/no and on/off are so fundamental to the reality of very small things. Once you break anything down far enough you are always left with a simple question of there being somewhere or nothing. The idea of a 1 or a 0 as the simplest possible building blocks for every single thing is not so outrageous. You put enough 1s and 0s together in a row you can build patterns that repeat and create larger patterns and suddenly you find yourself with a string of 1s and 0s that means something.

The idea of on/off is so simple even a computer can understand it. If you feed it a signal all it has to do is determine whether it is one or the other and store the result. Then read the next signal. And before long the computer has a sequence of 1s and 0s it might be able to do something with.

So when you are programming a computer or in our case trying to understand how it was programmed to give us Iridis Alpha you have to be able to visualize how the computer has stored all the ones and zeros it has been given and how it shuffles them around. An unruly mound of 1s and 0s is not much use when operating on them intensively. There has to be some benefit to splitting up the blob in some way that allows

some order to be imposed. Some way of segmenting a string of 1s and 0s that is both useful to us as the programmers and an efficient way of directing the computer to make effective use of the amorphous blob fed into it.

Trial and error has eventually arrived at an optimal arrangement, one which you have definitely heard of: this is the byte. The idea of the byte is simply to divide and conquer. We take any string of 1s and 0s and split them up into segments of eight. We now consider ourselves to have a string of bytes and we call each of the individual 1s and 0s bits.

Any scheme that reduces the number of items we have to deal with is a boon, but is there any rhyme or reason to choosing 8 as our magic number instead of say 7, or 12? Believe it or not, the number 8 was chosen almost solely after much experimentation with others because it proved the easiest and most convenient for people to deal with when understanding how they would make computers work.

The reason a byte of 8 bits is so convenient to deal with is because of the number of different values it can represent and the compact notation this particular number allows. Having a convenient notation system that allows humans to compose larger values from smaller values is surprisingly important when it comes down to it. It is humans doing the important part of the computing: making the big decisions, figuring out where things go and how they should work. The computer is just a glorified bit shuffler for which everything is on or off and there is no bigger picture. Humans need to be able to at least intuit some of this shuffling with a mental model of what happens when one set of ones and zeros is clashed with another. Using the magic number of 8 as the denominator for batches of bits enables this simple mental model.

The first great thing about 8 is that it can be divided into two. That is, into two segments of 4 bits. The great thing about a sequence of four 1s and 0s is that it has 16 possible permutations. That's to say it can be ordered in sixteen unique ways. Yet another way, is to say that a sequence of 4 bits can store up to sixteen different values.

Value	Bits	Value	Bits
0	0000	8	1000
1	0001	9	1001
2	0010	10	1010
3	0011	11	1011
4	0100	12	1100
5	0101	13	1101
6	0110	14	1110
7	0111	15	1111

The 16 different permutations of four 0s and 1s.

You have to agree this is marvellous stuff. But what is so convenient about a number between 0 and 15? If you're using decimal notation it's not convenient at all, in some cases you end up having to write two whole characters rather than one. We could avoid that if we invented a system that allowed a single character for each of the 16 values.

For example 0-9 for 0-9, and A-F for 10-15. Seems a bit clunky. But it works.

Value	Bits	Value	Bits
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

The 16 different permutations of four 0s and 1s but with letters too.

On the other hand, congratulations, we've just invented hexadecimal.

The genius of this system is that it allows us to write every possible sequence of 8 bits with just two characters.

Value	Bits	Value	Bits
00	00000000	80	10000000
01	00000001	81	10000001
02	00000010	82	10000010
.	.	.	.
7D	01111101	FD	11111101
7E	01111110	FE	11111110
7F	01111111	FF	11111111

The 256 different permutations of eight 0s and 1s.

But that is not its only benefit. When we look at any specific number using this notation we can almost immediately tell how the bits in it are set, without needing to memorize all 256 possible permutations.

This is possible because we can start by looking at either character in the two-

CHAPTER 1. WE NEED TO TALK ABOUT BINARY

character notation and quickly divine the 1s and 0s that make it up. We can do this because each character represents a value between 0 and 15 that made is up of some combination of 8, 4, 2, and 1 - each representing a bit from left to right.

8	4	2	1	Decimal	Hex
0	0	0	0	0	0
0	0	0	1	1	1
0	0	1	0	2	2
.	.	.	.		
1	1	0	1	13	E
0	1	1	0	14	D
1	1	1	1	15	F

The 16 different permutations of four 0s and 1s but with letters too.

"All" we have to do when confronted with a single hexadecimal digit like 9 is figure out that its made up of 8 and 1 in order to know that the bits are set as follows:

8	4	2	1	Decimal	Hex
1	0	0	1	9	9

Likewise with the slightly more inscrutable D. Since F is 15, D must be 13. And 13 is made up of 8, 4, and 1:

8	4	2	1	Decimal	Hex
1	1	0	1	13	D

The ability to do this is especially important when doing the low level programming that is typical of assembly. We'll see plenty of examples in future chapters but here is a taste of the kind of operation we ask a computer to perform by expressing our intentions using hexadecimal notation for binary.

Imagine we have a value from some random source stored in a byte. Imagine we happen to be in need of a random number, for example to select a random planet to use the next time we run attract mode in Iridis Alpha.

This random number we have stored in a byte sounds like it should be useful to us, but

since its random it can be any value between 0 and 255 and since we only have four planets to choose from what we need is a number between 0 and 3.

A simple binary operation provides a way of solving this problem and our hexadecimal notation gives us a straightforward way of expressing it.

First the binary operation itself. Let's say our random value is a string of bits like 01011010 which is 5A in hexadecimal.

8	4	2	1	8	4	2	1	Decimal	Hex
0	1	0	1	1	0	1	0	90	5A

Since we only want a value between 0 and 3 we can extract out just the last two bits to give us that number. In other words, if we set all the other bits to zero but that last two we would have a number between 0 and 3:

8	4	2	1	8	4	2	1	Decimal	Hex	
0	0	0	0	0	0	0	1	0	2	02

We can ask the computer to do this us by asking it to compare 01011010 with 00000011 and give use as a result all the bits where both are 1.

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
\$5A	0	1	0	1	1	0	1	0
\$03	0	0	0	0	0	0	1	1
Result: \$02	0	0	0	0	0	0	1	0

AND'ing \$5A and \$03 gives \$01 (1). For AND to give a 1 both bits must 1 or both must be 0.

Notice that in this new bit string we've set the bits that we're interested in to 1. This is called an AND operation. We have used 00000011 to mask out the bits we're not interested in and hence turn a random number between 0 and 255 into a random number between 0 and 3.

Our choice of 8 as a magic number becomes slightly less magical when we realize that we never had all possible numbers to choose from in the first place. Sure we could have chosen 7 or 9 or 13 but how practical would this really be when the actual value assigned to any individual bit is always going to be a power of 2. Even if we could

never precisely intuit the reasons as to why, a magic number that is also the power of 2, e.g. 8,16,32,64, is always going to make things easier over the long haul even if we can't articulate them.

So now some jargon we might have heard in the past is starting to make sense. The Commodore 64 was an '8-bit computer'. What this means is that its basic unit of currency is the 8-bit byte we've invented here. Everytime it performs an operation it does so using a single byte as its fundamental building block. This is true whether we talking about the AND operation or storing a value for use later on. When we give the C64 a value it's always a single byte. The subsequent era of 16-bit computing (e.g. the Amiga) took this a step further by introducing a two-byte block as its basic unit of currency. 32-bit and 64-bit computing are the lingua franc of modern processors - no matter how small the value we're using it will be managed as 4-byte or 8-byte value respectively by 32-bit and 64-bit computer architectures.

But even the C64 had use for values larger than 255. In fact it could handle staggering numbers as large as 65,535 using a little sleight of hand to put two bytes side by side and treat them as a single number.

This was as far as its ken could reach however. And since it could look no further than 65,535 that imposed an upper bound on the number of values it could reach for at any given time. This finite horizon is known as its address space. When we want to store a value in the C64's memory we only have 65,536 slots available to specify or select from. The address we choose must be between \$0000 (0) and \$FFFF (65,535).

43	36	34	00	40	..	41	50	45	DE	A2
\$0000	\$0001	\$0002	\$0003	\$0004	..	\$FFFC	\$FFFD	\$FFFE	\$FFFF	

Figure 1.1: The C64's memory visualized as a long strip of 65,536 slots, each holding a byte represented in hexadecimal.

This upper limit is what gives the Commodore 64 its name. It has 64 kilobytes of memory.

Hex	Decimal	Meaning	Hex	Decimal	Meaning
\$0400	1024	1 Kilobyte	\$0400	1024	1 Kilobyte
\$FFFF	65,535	64 Kilobytes	\$0400	1024	1 Kilobyte

Some magic numbers when dealing with bytes.

This is all we need to get started. The rest we can pick up along the way. So let's start.

A Little Archaeology

Iridis Alpha was distributed on cassette tape by the publisher Hewson Consultants. Normally used to play audio, cassette tapes were the cheap and ubiquitous medium du jour of the 1980s and a natural choice for the nascent 8-bit game industry to distribute its wares.

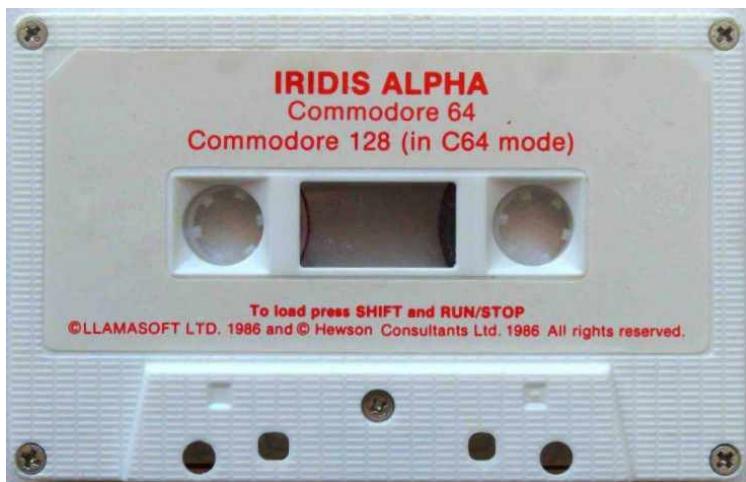


Figure 2.1: It should be simple getting bytes off this right?

Playing a cassette tape for a C64 game such as Iridis Alpha on a normal cassette tape player would be an ear-splitting mistake. Instead of music you would be subjected to a cacophony of mechanical chirruping. This is the tape attempting to convey to you its long stream of binary data in the only language available to it: lots of sound waves of varying length.

Without knowing how it's actually done, it's tempting to imagine a variety of possible schemes that might have been used. For example, one sound wave denoting a '0' and another one denoting a '1'. The actual method used isn't very far away from such a thing but there is plenty of intricacy layered on top, particularly in a bid to spend as little time as possible loading data from the tape.

In order to get something to work with our first step is to somehow convert the contents of the cassette tape into a binary file so that we can emulate the steps the C64's tape player performed to read the sounds from the tape and load them into memory as something that could be run as a computer program.

The earliest and most durable attempt at standardizing this was from Per Hakan Sundell in 1997. He invented what is now known as the 'tap' format for representing the beeps and bleeps encoded on the tape as a file of bits and bytes. The idea is that each byte in the file represents the length of a pulse. It is the length of these pulses that will ultimately tell us whether we should interpret a value of 1 or 0. When we get eight 1s and 0s we have a byte. We get enough bytes, we have a program we can run!

Someone, somewhere has kindly decoded the contents of the Iridis Alpha cassette tape distribution to a 'tap' file for us. So we have something to dig into. This is going to be a slightly bonkers journey into the bowels of decoding a 54kb game file from over 500kb of raw data. Every time you think you are nearly done there will be yet another convolution to wrap your head around. But at the end of it we will finally have our binary game file and will be ready to figure out how to decipher it into something approximating the original assembly language.



Figure 2.2: Simple, right? Here each short-pulse sound is represented by a light grey pixel, each medium-duration pulse by a blue pixel, and each long pulse by a pink pixel. Roughly speaking, gray is a shorthand for 0 bits and dark pixels for 1 bits.

2.1 The Madness Begins

This is what the start of our `iridis-alpha.tap` file looks like:

CHAPTER 2. A LITTLE ARCHAEOLOGY

43	36	34	2D	54	41	50	45	2D	52	41	57	00	00	00	00
5A	0A	08	00	00	5D	32	2F	30	2F	2F	30	30	31	30	31
30	31	31	2F	31	31	30	31	30	30	31	30	30	31	30	30
31	31	30	31	31	30	31	30	30	31	30	31	31	30	30	30
31	31	31	30	30	31	30	31	31	31	30	30	30	31	31	30
30	30	31	30	31	30	30	30	31	30	31	31	30	30	30	31
31	31	30	30	31	30	32	31	30	31	30	30	32	31	30	30

Figure 2.3: The leading material read by the Megasave loader in the run up to retrieving game data. The Pilot Bytes are in red, the 'Sync Train' in blue, and the Data Header fields from the grey cell onwards.

Field Description	Field Value	Note
TAP Format Header Description	43 36 34 2D 54 41 50 45 2D 52 41 57	'C64-TAPE-Raw' in ASCII
Version	00	Version Number 0
Reserved	00 00 00	Used by format versions > 0
File Size	5A 0A 08	\$080A05, in decimal: 526,853 bytes long.
Start of Data	00 00 5D 32 2F 30 2F 2F 30 30 31 30 31 30 31 31 2F 31 31 30 31 30 30 31 30 30 31 30 31 31 30 31 31 30 31 30 30 31 30 31 30 30 30 31 31 31 30 30 31 30 31 31 30 30 30 31 31 30 30 30 31 30 31 30 30 31 30 31 31 30 30 30 31 31 31 30 30 31 30 32 31 30 31 30 30 32 31 30 30	Bytes representing the individual pulses/sounds on the tape.

Figure 2.4: The meaning of the first batch of data we've read in.

After the header information described above, each byte in the tap file represents the pulse length or duration of a single sound emitted by the tape. A pair of sounds taken together represent a single bit, i.e. a 0 or a 1. A medium length sound followed by a short one represents a 1, a short one followed by a medium one represents a 0.

The following table shows us whether we should consider a byte on the tape to represent short, medium, or long duration sound:

Sound Length	Lower Range	Upper Range
Short	\$24	\$36
Medium	\$37	\$49
Long	\$4A	\$64

Figure 2.5: Values for short, medium, and long pulses. For example, ny byte value on the tap file between \$24 and \$36 would be considered a 'Short' pulse.

Remarkably the first 27,000 or so pulses on the Iridis Alpha tape are nothing but short sounds (values between \$2F and \$31) so cannot be interpreted as anything. It's not until the 27,157th byte in the tape that we start to encounter real data:

```
00006a10: 3031 3031 3156 4144 3130 4231 4243 3130 01011VAD10B1BC0
00006a20: 4231 4131 4244 312f 4057 4032 4231 4231 B1A1BD1//@2B1B1
00006a30: 4244 312f 4231 4231 4243 3142 3055 4144 BD1/B1B1BC1B0UAD
00006a40: 3142 3143 3130 4231 4231 4232 4244 3142 1B1C10B1B1B2BD1B
00006a50: 3055 4131 4243 3142 3130 4231 4231 4231 0UA1BC1B10B1B1B1
00006a60: 4244 3130 4056 4143 3230 4143 3130 4231 BD10@VAC20AC10B1
00006a70: 4231 4232 4244 312f 4056 4232 4231 4243 B1B2BD1//@B2B1BC
00006a80: 3230 4231 4231 4231 4243 3142 2f54 4245 20B1B1B1BC1B/TBE
00006a90: 3141 3130 4231 4231 4230 4230 4243 3030 1A10B1B1B0B0BC00
```

Listing 2.1: Data finally gets started at 56 41 in the first line above.

The first meaningful twenty bytes therefore are:

```
56 41 44 31 30 42 31 42 43 31 30 42 31 41 31 42 44 31 2F 40
```

You get a sense of how wasteful, or ahem redundant, this encoding scheme is when you learn that these twenty pulses are required to give us a single byte. The table below shows how we interpret them to construct a series of 1s and 0s.

First Byte	Second Byte	First Byte Pulse Length	Second Byte Pulse Length	Meaning
\$56	\$41	Long	Medium	Start of Byte Indicator
\$44	\$31	Medium	Short	\$01
\$30	\$42	Short	Medium	\$00
\$31	\$42	Short	Medium	\$00
\$43	\$31	Medium	Short	\$01
\$30	\$42	Short	Medium	\$00
\$31	\$41	Short	Medium	\$00
\$31	\$42	Short	Medium	\$00
\$44	\$31	Medium	Short	\$01
\$2F	\$40	Short	Medium	Parity Bit of \$00

Figure 2.6: Interpretation of the first 20 meaningful bytes creating a byte. The parity bit at the end is a \$00 if there are an odd number of 1s and \$01 if there are an even number of 1s. 10010001 has an odd number of 1s.

We can visualize the twenty bytes as a square sound wave. When reading the tape the C64 would interpret these sound pulses as long, short, or medium to construct a meaning for the entire sequence.

The First 'Real' Byte of Data on the Tape as a Square Sound Wave.

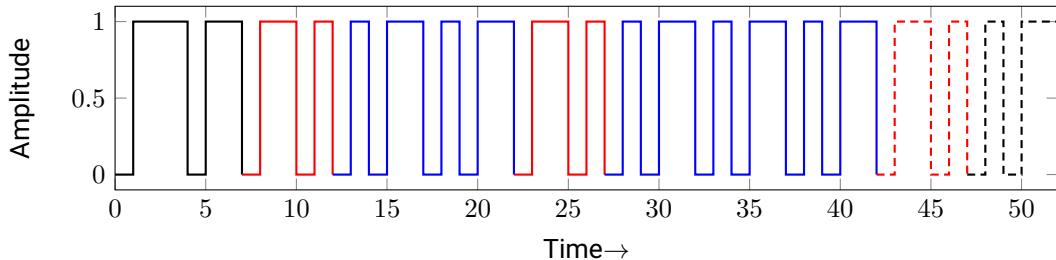


Figure 2.7: Medium-Short pairs in red represent a '1' bit, Short-Medium pairs in blue represent a '0' bit. So this gives us '10010001'. The black wave form at the beginning is the 'Start of Byte' indicator and the one at the end is a parity bit.

Once we've extracted our result of 10010001 from these twenty bytes we now must reverse it. We must do this because the bits are encoded on the tape with the 'most significant bit' first and we are used to reading binary with the 'least significant bit first'. In hexadecimal the reversed bit-string of 10001001 is \$89.

We have our first byte! It's 89!

2.2 After Our First Real Byte

With this precious commodity in hand we now continue reading off bytes in the same manner from the tape. Eventually we encounter a signal that tells us we've reached the end of the data block, a 'Long-Short' sequence. When this happens we find we've read 202 bytes in total:

CHAPTER 2. A LITTLE ARCHAEOLOGY

89	88	87	86	85	84	83	82	81	03	A7	02	04	03	49	52
49	44	49	53	00	00	00	00	00	00	00	00	00	00	78	A9
6E	8D	06	DD	A2	01	20	D4	02	26	F7	A5	F7	C9	63	D0
F5	A0	64	20	E7	03	C9	63	F0	F9	C4	F7	D0	E8	20	E7
03	C8	D0	F6	C9	00	F0	D6	20	E7	03	99	2B	00	99	F9
00	C8	C0	0A	D0	F2	A0	00	84	90	84	02	20	E7	03	91
F9	45	02	85	02	E6	F9	D0	02	E6	FA	A5	F9	C5	2D	A5
FA	E5	2E	90	E7	20	E7	03	C8	84	C0	58	18	A9	00	8D
A0	02	20	93	FC	20	53	E4	A5	F7	45	02	05	90	F0	03
4C	E2	FC	A5	31	F0	03	4C	B9	02	A5	32	F0	03	6C	2F
00	20	33	A5	A2	03	86	C6	BD	F3	02	9D	76	02	CA	D0
F7	4C	E9	02	A9	07	85	F8	20	D4	02	26	F7	EE	20	D0
C6	F8	10	F4	A5	F7	60	00	00	E4						

Figure 2.8: The data we've read in so far. The unshaded section is machine code.

Fortunately this data has a meaning. It contains the first part of a machine code program the C64 can execute:

Figure 2.9: The meaning of the data we've read in so far.

This small program, once we have loaded the rest of it, is where the fun starts. But before we do that we have lots more busywork to do. How about reading in all of the

CHAPTER 2. A LITTLE ARCHAEOLOGY

above data again from the tape? Yup, that's correct. As we keep reading the tape we will find that it contains all of the above data a second time, with the slight difference that the 'Data Block Header' will be 09 08 07 06 05 04 03 02 01 instead of 89 88 87 86 85 84 83 82 81.

We have to read another 23,000 or so more pulses before we get to something new that we're interested in, the second and final part of the program that we can execute.

When it arrives it looks like this:

89	88	87	86	85	84	83	82	81	A9	80	05	91	4C	EF	F6
A9	A7	78	8D	28	03	A9	02	8D	29	03	58	A0	00	84	C6
84	C0	84	02	AD	11	D0	29	EF	8D	11	D0	CA	D0	FD	88
D0	FA	78	4C	51	03	AD	0D	DC	29	10	F0	F9	AD	0D	DD
8E	07	DD	4A	4A	A9	19	8D	0F	DD	60	20	8E	A6	A9	00
A8	91	7A	4C	74	A4	52	D5	0D	00	00	00	00	00	00	00
00	00	8B	E3	AE	02	53									

Figure 2.10: The data we've read in so far. The unshaded section is machine code.

Field Description	Field Value	Note
Countdown	89 88 87 86 85 84 83 82 81	Data Block Header
	A9 80 05 91 4C EF F6 A9 A7 78 8D 28 03 A9 02 8D 29 03 58 A0 00 84 C6 84 C0 84 02 AD 11 D0 29 EF 8D 11 D0 CA D0 FD 88 D0 FA 78	
Machine Code	4C 51 03 AD 0D DC 29 10 F0 F9 AD 0D DD 8E 07 DD 4A 4A A9 19 8D 0F DD 60 20 8E A6 A9 00 A8 91 7A 4C 74 A4 52 D5 0D 00 00 00 00 00 00 00 00 00 8B E3 AE 02	This is the rest of machine code of the program to execute.
Checksum	53	How is this calculated?

Figure 2.11: The meaning of the second batch of data we've read in.

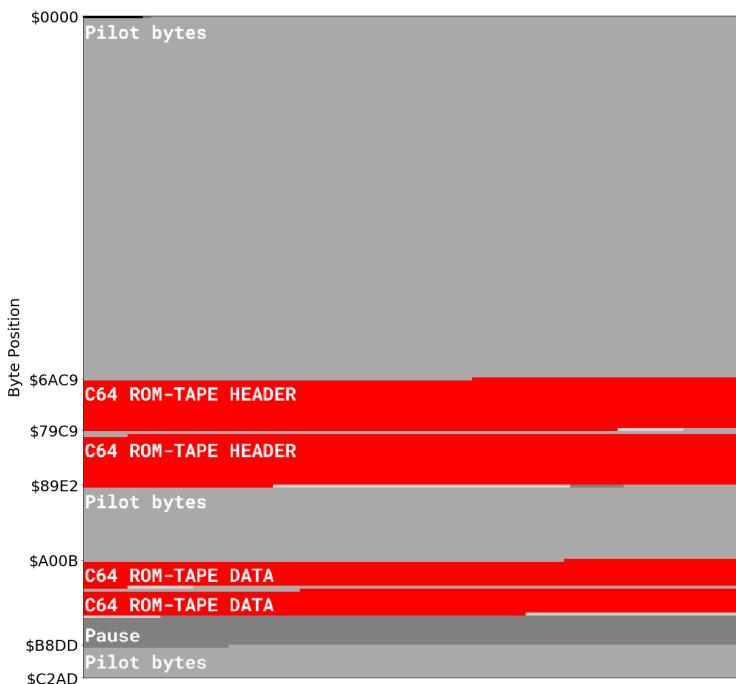


Figure 2.12: The bytes we've read so far from the tape.

Would you be surprised to learn that we have to read in this payload a second time from the tape before we're done? Let's just assume that you're not and let's move swiftly on to wondering what we're supposed to do with 100 or so bytes of data we've finally managed to read after listening to some 50,000 chirrups and clicks from cassette tape.

The answer is simple. We load the data we've received into RAM and execute it. We load the second chunk of data we received at address \$02A7 (this was given in the 'Load Address' field) and the first chunk of data we received directly after it.

What is this program? It seems a bit short to be Iridis Alpha right? What it is is a whole new program for reading the rest of the data from the tape. That's right: we've read all this data from the tape to get a program for reading data from the tape. This type of program is called a 'loader', or perhaps in an effort to justify its existence, a 'turbo loader'.

2.3 A Loader for your Loader

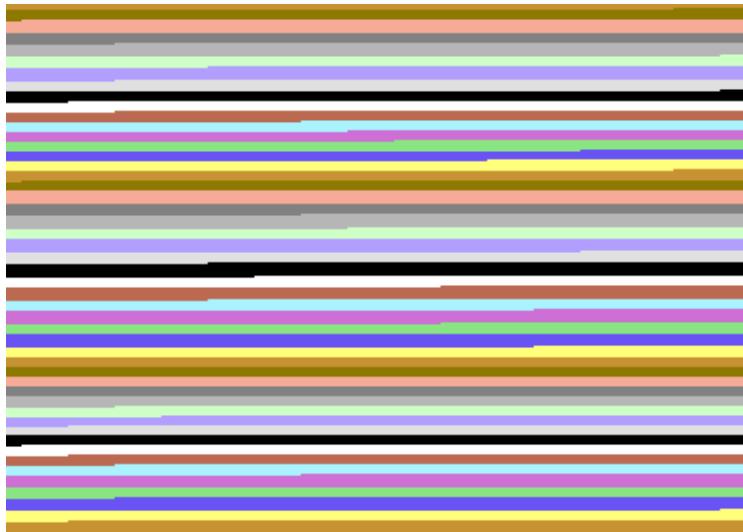


Figure 2.13: MegaSave loader pictured.

The idea is that this little program will do a better job of reading data from the tape and more quickly than the C64 can manage by itself. There is a whole menagerie of these programs that proliferated in the 1980s with exotic names such as Jetload, Easy-tape, Audiogenic and so on. Luigi di Fraia maintains a utility called tapclean that does a great job of identifying and emulating these loaders and thanks to him we have a disassembled version of the loader we've just found on the Iridis Alpha tape. It has the quintessentially 1980s moniker 'MegaSave' and as you can see in the listing reproduced below is relatively compact.

```
; This disassembly has been adapted from
; https://www.luigidifraia.com/doku/doku.php?id=commodore:tapes:loaders:mega-save
;
; From: Cauldron
; Note: Assemble with 64tass
;
; ****
; * CBM Header *
; ****
*$033C
;
; Cassette I/O Buffer - Header
T033C .byte $03,$A7,$02,$04,$03,$49,$52,$49,$44,$49,$53,$00,$00,$00,$00,$00
T034C .byte $00,$00,$00,$00
;
AlignAndSynchronizeLoop
    SEI
    LDA #$6E      ; Set the read bit timer/counter threshold to 0x0107
    STA $DD06
```

```

LDX #$01

; Byte-align with the incoming stream by shifting bits in until the first pilot byte is read

_align JSR rd_bit ; Read a bit

ROL $F7 ; Shift each of them into the byte receive register, MSbF
LDA $F7
CMP #$63 ; Until the first occurrence of 0x63 (pilot byte) is received
BNE _align

LDY #$64 ; Pre-load the first expected sync byte in Y

; Read the whole pilot sequence

_pilot JSR rd_byte ; Keep reading bytes until the pilot train is over
CMP #$63
BEQ _pilot

; Check that the sync sequence is as expected (Note: $F7 = byte read by means of a call to rd_byte)

_sync CPY $F7 ; Is the currently expected sync byte following?
BNE _align ; Start over if not

JSR rd_byte ; Read a byte

INY ; Bump the expected sync byte value
BNE _sync ; Read the whole sync sequence (0x64-0xff inclusive)

CMP #$00 ; In absence of issues A is set to 0x01 here
BEQ AlignAndSynchronizeLoop

; Read and store file header

_header JSR rd_byte ; Read 10 header bytes
STA $002B,Y ; Overwrite BASIC program pointers
STA $00F9,Y ; Also store them in RAM where they will be changed
INY
CPY #$0A
BNE _header

LDY #$00
STY $00 ; Zero the status flags (not set by this code)
STY $02 ; Zero the checkbyte register

; Read and store data from tape into RAM

_data JSR rd_byte ; Read a byte
STA ($F9),Y ; Store in RAM

EOR $02 ; Update the checkbyte value
STA $02

INC $F9 ; Bump the destination pointer
BNE _check_complete
INC $FA

._check_complete
LDA $F9 ; And check if the file is complete
CMP $2D ; by comparing the destination pointer
LDA $FA ; to its one-past-end value
SBC $2E
BCC _data

JSR rd_byte ; Read a byte

INY
STY $C0 ; Control motor via software

CLI
CLC

LDA #$00 ; Make sure the call to $FC93 does not restore the standard IRQ
STA $02A0
JSR $FC93 ; Disable interrupts, un-blank the screen, and stop cassette motor

JSR $E453 ; Copy BASIC vectors to RAM

LDA $F7 ; Compare saved and computed checkbytes
EOR $02
ORA $90 ; And check that the status flags do not indicate an error
BEQ *+5

JMP $FCE2 ; Soft-reset if any of the above checks fails

```

CHAPTER 2. A LITTLE ARCHAEOLOGY

```
; Code execution after a file is completely loaded in

LDA $31      ; Check flag #1
BEQ *+5      ; If not set move on
JMP J02B9    ; Otherwise re-execute the loader

LDA $32      ; Check flag #2
BEQ *+5      ; If not set move on in order to issue a BASIC RUN command
JMP ($002F)  ; Otherwise execute a custom routine pointed by the vector $2f/$30

JSR $A533    ; Relink lines of tokenized program text

LDX #$03     ; Set the number of characters in keyboard buffer to 3
STX $C6

LDA T02F4-1,X ; And copy R, <shift> + U, <return> into the buffer
STA $0276,X
DEX
BNE *-7

JMP J02E9

; -----
; Read byte: read 8 bits from tape, grouping them MSbF
; Returns: the read byte in A

rd.byte LDA #07    ; Prepare for 8 bits
STA $F8    ; Using $F8 as a counter

B03EB JSR rd_bit ; Read a bit
ROL $F7    ; Shift each of them into the byte receive register, MSbF
INC $D020
DEC $F8    ; And loop until 8 bits are received
BPL B03EB
LDA $F7    ; Return read byte in A
RTS

.error * > $03FC, "The CBM Header code is too long to fit in the tape buffer!"

; -----
.align $03FC, $00 ; Padding

; *****
; * CBM Data *
; *****

*=$02A7

NMIH LDA #$80
ORA $91
JMP $F6EF

J02AE LDA #<NMIH
SEI
STA $0328 ; Disable <Run Stop> + <Restore>
LDA #>NMIH
STA $0329

J02B9 CLI
LDY #$00
STY $C6 ; No char in keyboard buffer
STY $C0 ; Enable tape motor
STY $02 ; Zero the checkbyte register (also done later on)
LDA $D011 ; Blank the screen
AND #$EF
STA $D011

DEX      ; Wait a bit
BNE *-1
DEY
BNE *-4
```

```

SEI
JMP AlignAndSynchronizeLoop ; Execute the main loader code
; -----
; Read bit: loops until a falling edge is received on the read line and uses
;           the read bit timer/counter to discriminate the current bit value
;           ; Returns: the read bit in the Carry flag
rd_bit LDA $DC0D ; Loop until a falling edge is detected
        AND #$10 ; on the read line of the tape port
        BEQ rd_bit

        LDA $DD0D
        STX $DD07
        LSR
        LSR ; Move read bit into the Carry flag

        LDA #$19 ; Restart the bit read threshold timer/counter
        STA $DD0F

        RTS
; -----
J02E9 JSR $A68E ; Reset pointer to current text character to the beginning of program text
        LDA #$00
        TAY
        STA ($7A),Y
        JMP $A474 ; Print READY
; -----
; Characters to be injected in the keyboard buffer, if required
T02F4 .byte $52,$D5,$0D ; R, <shift> + U, <return>
        .cerror * > $0300, "The CBM Data code is too long to fit in front of the vector table!"
; -----
; Overwrite BASIC vectors in RAM
        .align $0300, $00 ; Padding
T0300 .word $E38B ; Leave IERROR unchanged
        .word J02AE ; Autostart the turbo loader, once loaded, by overwriting IMAIN

```

Listing 2.2: The data we have just loaded translated back into assembly language. This is the MegaSave loader.

What does MegaSave do that makes it so much faster than the default C64 tape loader? The simple answer is that it cuts corners and strips away all of the cautious redundancy we observed when loading the loader itself. Instead of reading 20 bytes (or pulses) from the tape in order to construct a single byte it just needs 8. It does what we naively thought at the beginning might be the way to read data from the tape: a long pulse is a 0, a short pulse is a 1, you read 8 of them you have the 8 bits for your byte.

This simplicity is risky. With no repetition of data blocks, no parity check on each individual byte, and no delimiters between bytes, the loader is vulnerable to corruption of the tape itself or any hardware flakiness in the cassette reader. The fact that it generally works is simply due to a lack of conservatism paying off in practice. In addition, the MegaSave loader isn't totally bereft of precautions. There is a slightly elaborate dance it goes through to gain some assurance that the tape medium is going to yield

a reliable string of bytes.

The first thing it does is look for a sentinel value of \$63.

The First Pilot Byte of '63' as read by the loader from the tape.

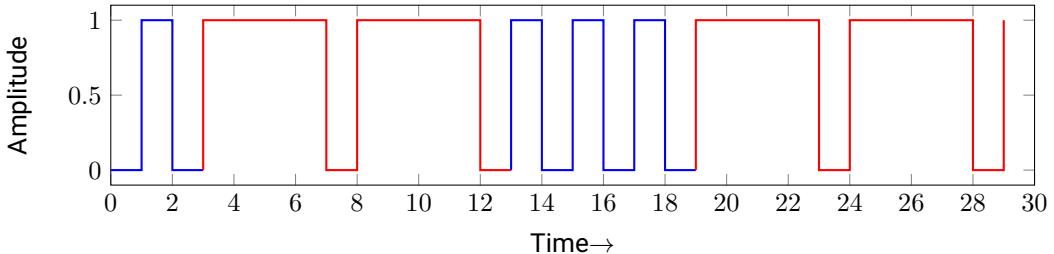


Figure 2.14: Long square waves in red represent pulses giving a '1' bit, Short square waves in blue represent a '0' bit. So this gives us '01100011' i.e. \$63. Unlike the default tape loader, MegaSave expects the 'most significant bit first' - which is the natural way of representing bits on paper so we don't need to reverse the bits to 'make sense' of them.

There is an inordinately long string of these, followed by an ascending sequence of byte values from \$63 to \$FF. This sequence has the catchy name of a 'Sync Train':

63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63
63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63
63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63
63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63
63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63
63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63
63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63
63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63
63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63
63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63
63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63
63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63
63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63
6F	70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	7F
80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F	90
91	92	93	94	95	96	97	98	99	9A	9B	9C	9D	9E	9F	A0	A1
A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC	AD	AE	AF	B0	B1	B2
B3	B4	B5	B6	B7	B8	B9	BA	BB	BC	BD	BE	BF	C0	C1	C2	C3
C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF	D0	D1	D2	D3	D4
D5	D6	D7	D8	D9	DA	DB	DC	DD	DE	DF	E0	E1	E2	E3	E4	E5
E6	E7	E8	E9	EA	EB	EC	ED	EE	EF	F0	F1	F2	F3	F4	F5	F6
F7	F8	F9	FA	FB	FC	FD	FE	FF	01	00	08	FF	BF	00	00	01
02	CA	00														

Figure 2.15: The leading material read by the Megasave loader in the run up to retrieving game data. The Pilot Bytes are in red, the ‘Sync Train’ in blue, and the Data Header fields from the grey cell onwards.

The bytes after the blue cells above are our first bit of raw meat in a while. Here’s what they mean:

CHAPTER 2. A LITTLE ARCHAEOLOGY

Field Description	Field Value	Note
Header Sentinel	01	Indicates the start of the header, expected to be non zero.
Load Address	00 08	Address to load to: \$0800
End Address	FF BF	End Address to load to: \$BFFF
Execution Address	00 00	Filename: "IRIDIS"
Next Action Indicator	01	Resume loading data when done or execute the loaded data.
Execution Type	02	How to execute the code.

Figure 2.16: The meaning of the Data Header values read in by MegaSave.

What the loader can garner from this is that the data that follows should be read in and stored at \$0800 and that rather than execute it straight away it should then resume loading more data from the tape.

The entire game is stored across four separate chunks on the tape. Once it has loaded this first one, the loader reads in the next three chunks.

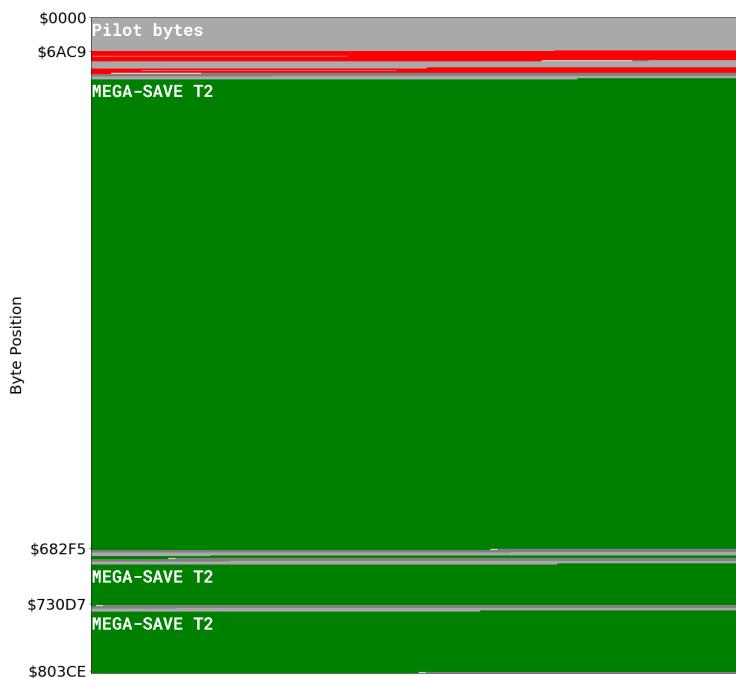


Figure 2.17: All the data that has been read from the tape. The four chunks of game data are in green the third is only a sliver. The relative sizes of the red data (the MegaSave loader which is only actually 200 or so bytes long) and the green data (representing over 50,000 bytes of game data) illustrates how efficient the MegaSave loader's storage is by comparison with the default.



Figure 2.18: Our image of the spool from the start of this chapter but this time with each section colored in as described in the previous image.

When it has completed it has loaded the following to RAM:

Start Address	End Address	Note
0800	BFFE	.
BF00	BFFF	.
C000	CFE	.
E000	F7FF	.

Figure 2.19: The four chunks of game data.



Figure 2.20: Where the different parts of the game end up in memory.

2.4 Putting an End to the Madness

With all the data read in you might wonder how the loader knows what to do next (i.e. to run the game) and how it will know where to start running it from. The answer is given in the Header Data for the final chunk of data:

CHAPTER 2. A LITTLE ARCHAEOLOGY

Field Description	Field Value	Note
Header Sentinel	01	Indicates the start of the header, expected to be non zero.
Load Address	00 E0	Address to load to: \$E000
End Address	00 F8	End Address to load to: \$F800
Execution Address	10 08	Address to start execution at: \$0810.
Next Action Indicator	00	00 means start executing code, don't read any more data.
Execution Type	02	How to execute the code: 01 means used the address given in 'Execution Address'.

Figure 2.21: The meaning of the Data Header values in the final chunk of data read in by MegaSave.

So according to the header data, the loader should stop reading data now and start executing what has already been loaded, and it should start doing this at address \$0810.

```
*=$0810
StartExecution
    SEI
    ; Tell the C64 to execute the code at MainControlLoop
    ; the next time an interrupt happens.
    LDA #>MainControlLoop
    STA $0319      ;NMI
    LDA #<MainControlLoop
    STA $0318      ;NMI

    ; Turn off the tape deck.
    LDA #$10
    STA $DD04      ;CIA2: Timer A: Low-Byte
    LDA #$00
    STA $DD05      ;CIA2: Timer A: High-Byte
    LDA #$7F
    STA $DD0D      ;CIA2: CIA Interrupt Control Register
    LDA #$81
    STA $DD0D      ;CIA2: CIA Interrupt Control Register
    LDA #$19
    STA $DD0E      ;CIA2: CIA Control Register A
    CLI
LoopUntilExexcutes
    JMP LoopUntilExexcutes
```

Listing 2.3: The first piece of code that is executed in Iridis Alpha.

This routine does two things: it turns off the tape deck and tells the C64 to execute the code at a different location (`MainControlLoop`) the next time it wakes up and wonders what to do. This will be in a few microseconds time. When it starts executing `MainControlLoop`, Iridis Alpha will get underway.



Figure 2.22: A prg is born.

Some Disassembly Required

We've reached the point where the game has started to execute. We just saw a snippet of code that turned off the tape recorder and prompted the C64 to run a routine called `MainControlLoop`. Though perhaps a little cryptic, and you shouldn't expect to understand it yet, this code is not exactly what the machine 'saw'. Instead it read and executed something far more puzzling looking:

```
78A9 408D 1903 A900 8D18 03A9 108D  
04DD A900 8D05 DDA9 7F8D ODDD A981  
8DOD DDA9 198D 0EDD 584C 3508
```

Listing 3.1: The first piece of machine code that is executed in Iridis Alpha.

That's right it executed a stream of bytes. A stream of bytes commonly referred to as 'machine code'. Each of these bytes is meaningful to the C64 whether individually, or taken in pairs, or groups of three. It can comprehend them as instructions to carry out that will shuffle data around in its memory and ultimately result in a game that can be played.

Before we can dig into the internals of how Iridis Alpha works we have to convert all of the machine code we've loaded into memory in the previous chapter into something we have a chance of reading and understanding. This process is called disassembly and here we're going to explain how it is done and along the way gain a little basic understanding of the human-readable language, called 6502 Assembly Language, that we convert the machine code back into.

The process is called disassembly simply because it is the exact reverse of the process that was originally followed to generate the data on the tape from the assembly language written by Jeff Minter in the first place. Programs that do this are referred to as 'assemblers'. They assemble the instructions written by the programmer into ma-

chine code that the C64 can execute. As self-appointed disassemblers we are going to turn it back into assembly language.

Along the way we will have to invent names for things that are meaningful to us: the names Jeff Minter gave to his functions, routines, and variables are long since lost to us. They were thrown away by the assembler as unnecessary for execution of the machine code. As we proceed, we will see why but it hopefully makes sense to say that the 6502 CPU in the C64 doesn't care what things are called it just cares where in the 65,632 bytes of its RAM things are located. That is to say it only cares about their 'address'.

So let's start by stepping back for a second. Let's put the machine code and the snippet of code we disassembled it back into and see what we can learn:

```
78  
A9 40  
8D 19 03  
A9 00  
8D 18 03  
A9 10  
8D 04 DD  
A9 00  
8D 05 DD  
A9 7F  
8D 0D DD  
A9 81  
8D 0D DD  
A9 19  
8D 0E DD  
58  
4C 35 08
```

Listing 3.2: Machine Code

```
SEI  
LDA #$40  
STA $0319  
LDA #$00  
STA $0318  
LDA #$10  
STA $DD04  
LDA #$00  
STA $DD05  
LDA #$7F  
STA $DD0D  
LDA #$81  
STA $DD0D  
LDA #$19  
STA $DD0E  
CLI  
JMP $0835
```

Listing 3.3: Assembly Language

Straightaway we can infer meanings to some of the bytes:

Byte	Instruction
78	SEI
A9	LDA
8D	STA
58	CLI
4C	JMP

Figure 3.1: Machine code bytes and their assembly language counterparts.

Let's quickly explain what two of these mean as they're extremely common and also fundamental to machine code programming in general.

LDA loads the byte you give it into a small one-byte slot called the 'Accumulator' ('A' for short so LDA is an abbreviation for 'Load to Accumulator'). Think of this slot as a pocket - somewhere for the CPU to store a value for use later on. The technical name for this kind of slot/pocket is a 'register'. The 'Accumulator' is so-called because a lot of the operations performed on bytes stored in this particular pocket have very precise behaviour when addition operations are executed on it. But we never really have to worry about this too often and in the general case it is simply used, as here, as a place to put a value so we can do something else with it.

```
LDA #$40
```

Listing 3.4: Loading the byte \$40 to the Accumulator.

What we do with it for the most part in the code above is go on to store it somewhere else. This is where the STA instruction comes in. This stores the value in the 'Accumulator' pocket at the address in memory that you give to it. As we can see such addresses are not one byte long, but two bytes. Are they ever more than two bytes long? No, and for a very simple reason. The C64 can only understand addresses that are at most two bytes long and this is what ultimately limits it to 64KB of memory. The largest address it can understand is therefore \$FFFF - the largest value that can be expressed by 2 bytes. Which translates to 65,532. 65,532 bytes is 64KB of RAM.

When we look at the disassembly of the STA instructions we see something quite puzzling:

```
8D 19 03
```

```
STA $0319
```

Shouldn't we have expected 8D 19 03 to translate to STA \$1903 rather than STA \$0319? Why are the numbers back to front like that? The reason is due to something you may have heard described with a word before that you've never fully understood and maybe never dared to question. The machine code stores the address 0319 as 1903 because the 6502 CPU in the C64 expects to read addresses with the second half of the number first. When we read numbers we expect it to start with the most significant digits first, i.e. \$0319. But most computer architectures of the 1980s had the opposite expectation - reading the least significant digits first, or the least significant bits first since this is a computer we're talking about, i.e. \$1903. This approach possesses a couple of advantages, the main one being that the computer can start performing an operation on the number (e.g. addition) before it has read all of it. For example, if you're adding 5312 and 2043 you already have enough to get started with once you've read 12 and 43. The carrying etc. can all happen once you've read the rest

of the two numbers. This approach is known as 'little-endian' byte ordering. The opposite, which is closer to what we are familiar with when reading numbers ourselves, is called 'big-endian'. If you had never heard of either of those terms before, then you have now.

The next incremental step in understanding our disassembly is to apply some meaning to these two-byte values that we've decoded. As it happens, all the ones given in this piece of code represent addresses in the C64's memory of \$FFFF bytes, or in the preferred parlance of we humans, 65,532 bytes. Each address here has a particular function so storing a value in it does something. Let's add the meanings, which are still a little cryptic, to our listing:

```
SEI  
LDA #$40  
STA $0319  
LDA #$00  
STA $0318  
LDA #$10  
STA $DD04  
LDA #$00  
STA $DD05  
LDA #$7F  
STA $DD0D  
LDA #$81  
STA $DD0D  
LDA #$19  
STA $DD0E  
CLI  
JMP $0835
```

Listing 3.5:
Assembly

```
SEI  
LDA #$40  
STA $0319 ; Non-Maskable Interrupt  
LDA #$00  
STA $0318 ; Non-Maskable Interrupt  
LDA #$10  
STA $DD04 ; CIA2: Timer A: Low-Byte  
LDA #$00  
STA $DD05 ; CIA2: Timer A: High-Byte  
LDA #$7F  
STA $DD0D ; CIA2: CIA Interrupt Register  
LDA #$81  
STA $DD0D ; CIA2: CIA Interrupt Register  
LDA #$19  
STA $DD0E ; CIA2: CIA Control Register A  
CLI  
JMP $0835 ; Jump to address $0835
```

Listing 3.6: Assembly Language with some comments

As you may remember we said in the previous chapter that this little routine is doing two things: the first is telling the C64 to jump to and execute the routine that starts the actual game the next time it 'wakes up'; the other thing it's doing is turning off the cassette tape reader so that no more data is read from the tape.

The bit that's turning off the tape reader is this:

```
LDA #$10  
STA $DD04 ; CIA2: Timer A: Low-Byte  
LDA #$00  
STA $DD05 ; CIA2: Timer A: High-Byte  
LDA #$7F  
STA $DD0D ; CIA2: CIA Interrupt Control Register  
LDA #$81  
STA $DD0D ; CIA2: CIA Interrupt Control Register
```

```
LDA #$19
STA $DDOE      ;CIA2: CIA Control Register A
```

We are better off treating this series of instructions as a magic formula. The operation of the tape reader is managed by the values stored in a series of bytes between \$DD04 and \$DD0F. The fact that we have to write a variety of values to 5 of them to just stop reading from the tape is just a testament to the power of boring overhead - we'll never interact with the tape reader again so studying the entrails here is not going to benefit us in any way.

The other thing this routine is doing - preparing the C64 to execute the game proper - is more compact and introduces two concepts that are going to be important throughout our tour of the Iridis Alpha code so it is worth spending some time on them here to get them clear.

```
LDA #$40
STA $0319      ;Non-Maskable Interrupt
LDA #$00
STA $0318      ;Non-Maskable Interrupt
```

Listing 3.7: Containing two important concepts.

3.0.1 Important Concept Number One: High Bytes and Low Bytes

On the face of it these four instructions are doing something very simple. They are storing the value \$40 at address \$0319 and the value \$00 at \$0318. What they are actually doing is storing the address \$4000 in a place where the C64's 6502 CPU will be expected to look in a moment's time and take that as a command to start executing whatever is at address \$4000.

43	36	34	00	40	41	50	45
0315	0316	0317	0318	0319	031A	031B	031C

Figure 3.2: The slice of C64 memory at addresses \$0315 and \$031C and the bytes that live there after we've written \$40 to \$0319 and \$00 to \$0318.

The reversed order we spoke about earlier, the 'little-endian' order, in which the 6502 CPU stores and reads pairs of bytes is observed again here. When the CPU reads the contents of \$0318 and \$0319 it interprets them not as \$0040, which is the order in which they appear to us, but as \$4000.

When discussing this storage arrangement we refer to the contents of \$0318 as the 'Low Byte' and the contents of \$0319 as the 'High Byte'. 'High' is just another way of saying 'first', and 'Low' another way of saying 'second'. When talking about values like \$4000 stored in \$0318–\$0319 such as in this case \$40 is the 'High Byte' and \$00 is the 'Low Byte'.

Writing values to a pair of adjacent addresses in memory like this so that they can be subsequently interpreted as yet another address to get something from or do something with is a very common pattern in programming 6502 CPUs such as the C64's and we will encounter it a *lot* in this book.

It's a strange sort of indirection when you first attempt to understand it. Instead of storing actual values at an address we're storing an address in the address. If you are familiar with other programming languages you may already recognize this concept and understand how powerful it can be. If you are not it will probably seem strange and maybe even wasteful. Seeing the many uses it is put to in the Iridis Alpha code may persuade you otherwise, but hopefully when we look at the use it is put to here you may begin to get a flavor of its utility. Let's do that by looking at our second important concept.

3.0.2 Important Concept Number Two: Interrupts

```
LDA #$40
STA $0319      ; Non-Maskable Interrupt
LDA #$00
STA $0318      ; Non-Maskable Interrupt
```

Listing 3.8: It's an interrupt. And it's non-maskable.

We previously waved away what's happening in this code by saying that the address \$4000 will be interpreted as an address to jump to and start executing the next time the 6502 CPU 'wakes up'. That's a lot of hand-waving.

The technical term for this 'waking up' is an 'interrupt'. This waking up happens incredibly frequently, 60 times every second. 60 times a second the C64 will stop what it's doing and execute whatever is given as an address by the bytes at \$0318–\$0319.

The number 60 may ring a bell for you in this context. A common aspiration for graphics-based games, and minimum table stakes today, is that a game runs at 60 'frames per second'. In other words, that at least 60 times a second the display is updated and whatever is on the screen moves a little bit.

While a C64 programmer would never achieve 60 frames per second in practice, 'interrupts' are the C64's mechanism for at least getting some of the way there. They allow

the game developer to at least do something to the display many times per second. Whatever it is, it has to be something short and sweet and at the same time effective enough to actually progress the gameplay. This is why this concept is important to us: most of the important things that happen in Iridis Alpha will be effected during an interrupt. When we look at moving, shooting, and blowing things up, all of them are going to happen in routines that are called multiple times per second by the 6502 CPU executing the code that has been stored at the address \$0318-\$0319 at that point in time.

And this is our first taste of the power of storing an address in an address. Depending on where we are in the game - be it the title screen, the main game, the bonus phase, or in a pause mode sub-game, the code that we want to run during these interrupts will be different.

We can see this in action if we look at what happens when the code at \$4000 is executed. We've given this address \$4000 a 'label' name of `MainControlLoop` in our disassembled code so `MainControlLoop` always refers to whatever lives at \$4000.

```

MainControlLoop
    LDA #$00
    SEI
p4003  LDA #<MainControlLoopInterruptHandler
        STA $318      ;NMI
        LDA #>MainControlLoopInterruptHandler
        STA $319      ;NMI

```

Listing 3.9: The code at \$4000.

As you can see the first thing it does is change the address of the code that should be executed at every 'interrupt'. It's now an address referred to by the label `MainControlLoop-InterruptHandler`. This happens to be address \$6B3E but the beauty of using labels in our code is that we no longer need to worry about the number of the addresses anymore. The label will do the job for us. It does mean we have to know what the syntax `#<MainControlLoopInterruptHandler` means though. What it means is: if `MainControlLoopInterruptHandler` lives at \$6B3E the `#<` decorators refer to the \$3E part of the address, so we're actually saying `LDA #$3E`, i.e. load the value \$3E into the 'Accumulator'. Similarly the syntax `#>MainControlLoopInterruptHandler` refers to the \$6B part of the address.

43	36	34	3E	6B	41	50	45
0315	0316	0317	0318	0319	031A	031B	031C

Figure 3.3: The values in \$0318 and \$0319 after we've updated them in `MainControlLoop`.

CHAPTER 3. SOME DISASSEMBLY REQUIRED

We've almost squeezed as much as we can out of this short and boring snippet of code. There's just one last thing to point out that will be useful to us later. The last instruction in the routine:

```
JMP $0835
```

Listing 3.10: Jump.

This tells the CPU to jump to the address \$0835 and execute whatever is there. What is at \$0835? This:

```
JMP $0835
```

Listing 3.11: Hello again.

That's right - it's jumping back to itself and will execute in an infinite loop repeatedly executing the instruction over and over again. The reason it does this is because it won't be doing it very long. The interrupt will take over and steal execution away so that the C64 can find better things to do that run in circles.

Before we move on let's take one last look at our disassembly accomplishment.

```
StartExecution
    SEI
    ; Tell the C64 to execute the code at MainControlLoop
    ; the next time an interrupt happens.
    LDA #>MainControlLoop
    STA $0319      ;NMI
    LDA #<MainControlLoop
    STA $0318      ;NMI

    ; Turn off the tape deck.
    LDA #$10
    STA $DD04      ;CIA2: Timer A: Low-Byte
    LDA #$00
    STA $DD05      ;CIA2: Timer A: High-Byte
    LDA #$7F
    STA $DD0D      ;CIA2: CIA Interrupt Control Register
    LDA #$81
    STA $DD0D      ;CIA2: CIA Interrupt Control Register
    LDA #$19
    STA $DD0E      ;CIA2: CIA Control Register A
    CLI

LoopUntilExexcutes
    JMP LoopUntilExexcutes
```

Listing 3.12: Fully disassembled

Hopefully with these basics under our belt we can begin to understand how the interesting parts of Iridis Alpha work.

The First 16 Milliseconds

The race is now on to get the title sequence up on the screen. After a little more setup in MainControlLoop we call a routine to set up the main title screen:

```
; Display the title screen. We'll stay in here until the
; player presses fire or we time out and go into attract mode.
JSR EnterMainTitleScreen
```

Listing 4.1: In MainControlLoop

This brings us to a routine called InitializeSpritesAndInterruptsForTitleScreen in which we do some vital setup for the next 16 milliseconds and how we go about getting everything on the screen that we need to:

```
; Set up the our interrupt handler for the title
; screen. This will do all the animation and title
; music work.
LDA #<TitleScreenInterruptHandler
STA $0314      ;IRQ
LDA #>TitleScreenInterruptHandler
STA $0315      ;IRQ

; Acknowledge the interrupt, so the CPU knows that
; we have handled it.
LDA #$01
STA $D019      ;VIC Interrupt Request Register (IRR)
STA $D01A      ;VIC Interrupt Mask Register (IMR)

; Set up the raster interrupt to happen when the
; raster reaches the position we specify in D012.
LDA $D011      ;VIC Control Register 1
AND #$7F
STA $D011      ;VIC Control Register 1
```

```
; Set the position for triggering our interrupt.  
LDA #$10  
STA $D012      ;Raster Position
```

Listing 4.2: In InitializeSpritesAndInterruptsForTitleScreen

You'll notice we've changed our interrupt handler again, this time to a routine called `TitleScreenInterruptHandler`. What we also do is make this interrupt something called a 'Raster Interrupt'. A 'raster' can be thought of as a beam of light that scans across the screen from top to bottom and left to right painting each pixel on the screen one at a time. It travels so quickly down and across the screen painting pixels that it can do so up to 60 times a second. As it makes this journey our 'Raster Interrupt' gives us the opportunity to tell it to stop once it reaches a certain position on the screen and allow us to run some code before it resumes again. We can do this as many times as we want along the journey, but each time we interrupt it we have to be quick. If our code takes too long the display will flicker and the content of the screen become inconsistent.

In this routine we set our first interrupt to line 16 (\$10 on the screen):

```
; Set the position for triggering our interrupt.  
LDA #$10  
STA $D012      ;Raster Position
```

Listing 4.3: In InitializeSpritesAndInterruptsForTitleScreen

This facility is the key that will allow us to do all sorts of magic in the 16 milliseconds it takes to traverse the screen. Every time we get the opportunity to run some code thanks to this interrupt we'll change the location of the screen it should stop at the next time so that we get to stop dozens of times in each single 16 millisecond traversal.

Before we look at how we fit it all in, let's first appreciate just how much we plan to do each time the screen is painted.

4.1 Sprites

The C64 makes 8 sprites available to us. A sprite is a special purpose graphical object that can be up to 24 pixels wide by 20 pixels high. We can place them wherever we want on the screen. They are the core of graphics programming and Iridis Alpha has dozens of them. But if the C64 only has 8 sprites, are we limited to displaying just 8 sprites at once on the screen? The simple answer is that thanks to Raster Interrupts we are not: when we run some code after receiving an interrupt we can place new sprites wherever we like in any position that the raster hasn't reached yet. This means our

only effective limitation is the number of sprites we can place on a single line, which is eight.

If you look carefully at the title screen of Iridis Alpha you'll notice that it is actually split in two. The top half has the title in large letters and the bottom half has a rainbow of jumping gilbies. Each half uses seven sprites to display these assets.

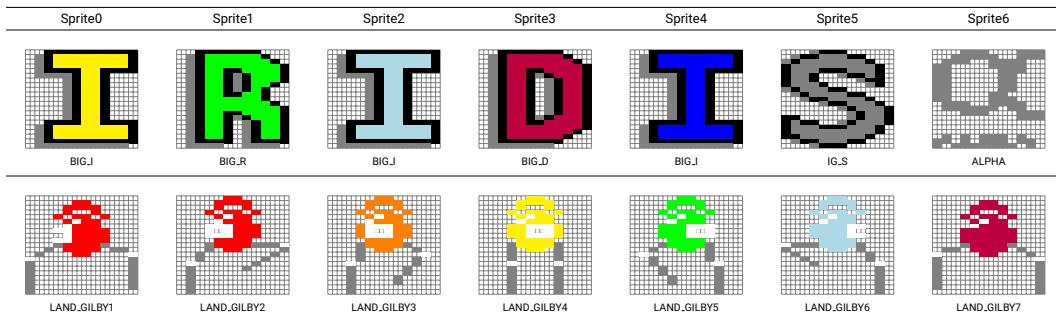


Figure 4.1: The sprites used by the top half of the screen and the bottom half of the screen.

The eighth sprite (Sprite 7) is used on both halves of the screen to display the starfield. This sprite pushes right up against the line limitation. It's painted at intervals throughout the screen but we're careful to avoid it ever being painted twice on the same line. We'll see how this is achieved very soon.

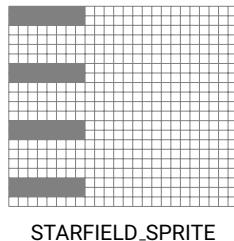


Figure 4.2: The sprite used for painting the starfield. Only a part of the sprite is ever painted!

4.2 Waiting for the Beam

With our 'Raster Interrupt' handler set up as `TitleScreenInterruptHandler` we're ready to react when the raster reaches line 16 on the screen. Since the screen is made

up of 512 lines in total this will be along soon.

Before it comes in we just have time to prepare the relatively light amount of text we want displayed on the screen in memory. We only need to do this once. Throughout the code we refer to this area we write to as SCREEN_RAM. It's an address range between \$0400 and \$07E8 This is a very simple bitmap representation of the entire screen that is 40 characters wide and 25 characters wide, giving a total of 1000 bytes (\$3E8 bytes in hex). If we wanted to think of it as pixels it is 320 pixels wide ($40 * 8$) and 200 pixels high ($25 * 8$). The important thing to remember about this SCREEN_RAM is that it is solely for storing what we call character data. You can think of character data as 'text'. This text gets painted first and then sprites get painted on top of it.

In EnterTitleScreenLoop we call two routines that will prepare the character data for the raster to paint.

The first, DrawStripesBehindTitle writes the rainbow stripes to five lines in the top half of the screen. The second, DrawTitleScreenText writes some text to the bottom half of the screen. Before we look at these in detail we need to understand how this thing SCREEN_RAM works and how we store characters for display in it.

Our starting point for displaying text on screen is to define what our characters look like. We define the appearance of a character using 8 bytes. This is what the definition of the stripe character looks like:

```
characterSetData
.BYTE $FF,$00,$FF,$00,$00,$FF,$00,$FF    ;.BYTE $FF,$00,$FF,$00,$00,$FF,$00,$FF
; CHARACTER $00
; 11111111 *****
; 00000000
; 11111111 *****
; 00000000
; 00000000
; 11111111 *****
; 00000000
; 11111111 *****
```

Listing 4.4: The 'stripe' character.

As you can see each byte translates to a row of 0s and 1s. Each 1 defines a dot and each 0 a blank space. We end up with a character that is 8 pixels wide and 8 pixels high:

1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1

The stripe character

We create this definition for every character we want to display and store it at the address starting at \$2000 in RAM. The order in which we store them determines the reference we use for them later. So for example the stripe character is referred to as \$00, the 'A' character we've defined as \$01 and so on:

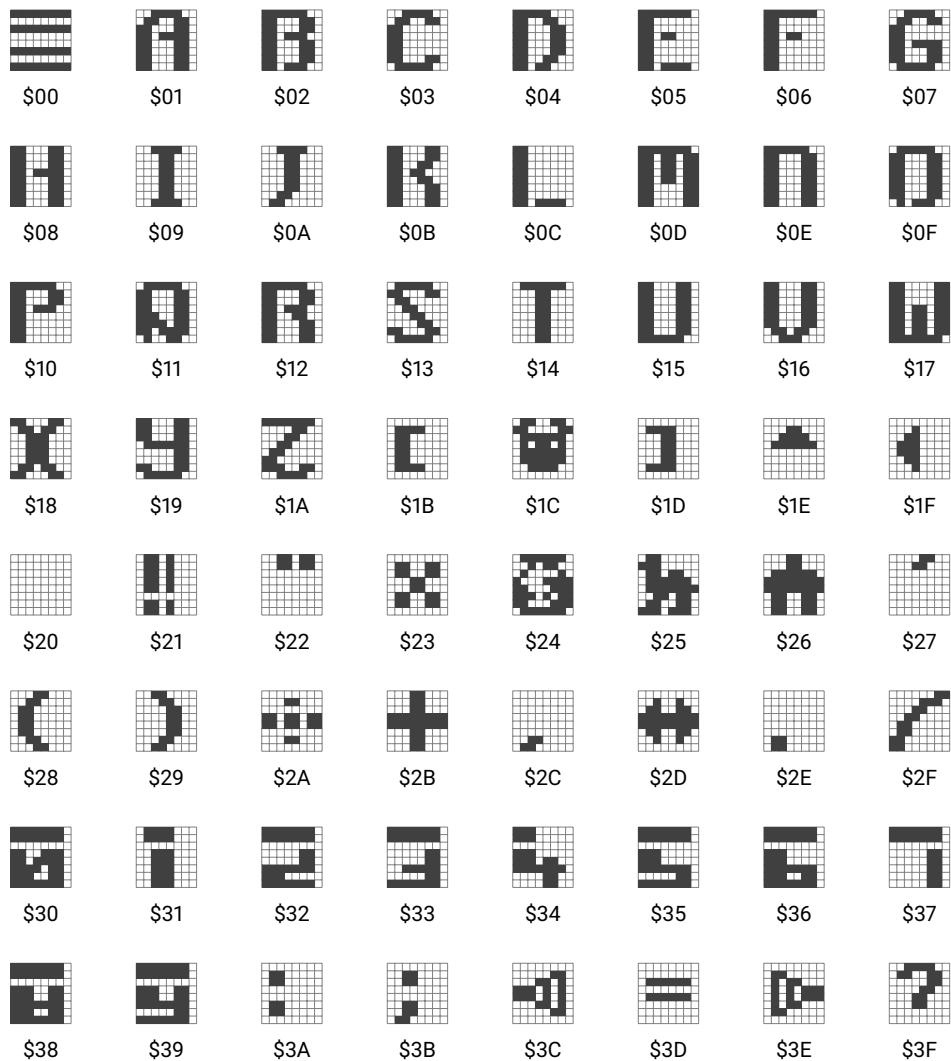


Figure 4.3: Tilesheet: Font Character Set stored at \$2000

With our character set defined we can now write some text to the screen ram. Note that when we write it SCREEN_RAM we're not yet writing it to the actual screen. This is just a place in memory that the raster (our beam of light) will refer to later when it is actually writing dots to the screen. If we write a stripe character to a particulas position in this SCREEN_RAM memory it will know to write it the corresponding position on the screen.

4.2.1 Drawing the Stripes

So let's write some stripes to RAM!

```

DrawStripesBehindTitle
    LDX #$28
    LDA #$00
    STA shouldUpdateTitleScreenColors
DrawStripesLoop
    LDA #RED
    STA COLOR_RAM + LINE2_COL39,X
    LDA #ORANGE
    STA COLOR_RAM + LINE3_COL39,X
    LDA #YELLOW
    STA COLOR_RAM + LINE4_COL39,X
    LDA #GREEN
    STA COLOR_RAM + LINE5_COL39,X
    LDA #LTBLUE
    STA COLOR_RAM + LINE6_COL39,X
    LDA #PURPLE
    STA COLOR_RAM + LINE7_COL39,X
    LDA #BLUE
    STA COLOR_RAM + LINE8_COL39,X
    LDA #$00 ; Stripe character
    STA SCREEN_RAM + LINE2_COL39,X
    STA SCREEN_RAM + LINE3_COL39,X
    STA SCREEN_RAM + LINE4_COL39,X
    STA SCREEN_RAM + LINE5_COL39,X
    STA SCREEN_RAM + LINE6_COL39,X
    STA SCREEN_RAM + LINE7_COL39,X
    STA SCREEN_RAM + LINE8_COL39,X
    DEX
    BNE DrawStripesLoop

```

Listing 4.5: The 'stripe' character.

As you can hopefully see, what we're dealing with here is a loop. We load X with the value \$28 (40 in decimal) and perform everything inside `DrawStripesLoop` until `DEX` has reduced the value of X to zero.

The magic number 40 gives us a clue that what we are doing in each loop is drawing a character in each column of the screen: remember that our screen is 40 columns wide and 25 rows high. The bit actually writing the stripe character to RAM is:

```

LDA #$00 ; Stripe character
STA SCREEN_RAM + LINE2_COL39,X
STA SCREEN_RAM + LINE3_COL39,X
STA SCREEN_RAM + LINE4_COL39,X
STA SCREEN_RAM + LINE5_COL39,X
STA SCREEN_RAM + LINE6_COL39,X
STA SCREEN_RAM + LINE7_COL39,X

```

CHAPTER 4. THE FIRST 16 MILLISECONDS

STA SCREEN_RAM + LINE8_COL39,X

Listing 4.6: In DrawStripesBehindTitle

For the current column, this writes the stripe character (reference by \$00 as we mentioned above) to each of lines 2 to 8. The use of the X in the STA statement is an offset. So where X is 14, for example, it will write to the position referred to by SCREEN_RAM + LINE2_COL39 plus 14.

Figure 4.4: The shaded areas of SCREEN_RAM after they have been written to by DrawStripesBehindTitle.

The other thing we do in `DrawStripesLoop` is set the colors of the stripes. This is achieved using a region of memory similar in concept to `SCREEN_RAM`, that we call `COLOR_RAM`. This lives at \$D800 – \$DBFE. Another region of 1000 bytes, each one controlling the color of the character placed at a position in the $40 * 25$ character rectangle of our screen.

```
LDA #RED
STA COLOR_RAM + LINE2_COL39,X
LDA #ORANGE
STA COLOR_RAM + LINE3_COL39,X
LDA #YELLOW
STA COLOR_RAM + LINE4_COL39,X
LDA #GREEN
```

```
STA COLOR_RAM + LINE5_COL39,X  
LDA #LTBLUE  
STA COLOR_RAM + LINE6_COL39,X  
LDA #PURPLE  
STA COLOR_RAM + LINE7_COL39,X  
LDA #BLUE  
STA COLOR_RAM + LINE8_COL39,X
```

Listing 4.7: In DrawStripesBehindTitle

We've used a meaningful alias for each of the color values that we write, these are defined as:

RED	= \$02
PURPLE	= \$04
GREEN	= \$05
BLUE	= \$06
YELLOW	= \$07
ORANGE	= \$08
BROWN	= \$09
LTBLUE	= \$0E

Listing 4.8: In DrawStripesBehindTitle

So by writing a value to the corresponding place in COLOR_RAM, we're defining the color of the character in that position.

CHAPTER 4. THE FIRST 16 MILLISECONDS

Figure 4.5: The shaded areas of COLOR_RAM after they have been written to by DrawStripesBehindTitle.

4.2.2 Drawing the Text

Next up is to write out the title screen's text to SCREEN_RAM. This we do in `DrawTitleScreenText` using a similar loop to `DrawStripesBehindTitle`.

```
DrawTitleTextLoop
    LDA titleScreenTextLine1 - $01, X
    AND #ASCII_BITMASK
    STA SCREEN_RAM + LINE11_COL39, X
    LDA titleScreenTextLine2 - $01, X
    AND #ASCII_BITMASK
    STA SCREEN_RAM + LINE13_COL39, X
    LDA titleScreenTextLine3 - $01, X
    AND #ASCII_BITMASK
    STA SCREEN_RAM + LINE15_COL39, X
    LDA titleScreenTextLine4 - $01, X
    AND #ASCII_BITMASK
    STA SCREEN_RAM + LINE17_COL39, X
    LDA titleScreenTextLine5 - $01, X
    AND #ASCII_BITMASK
    STA SCREEN_RAM + LINE19_COL39, X

    LDA #GRAY2
```

CHAPTER 4. THE FIRST 16 MILLISECONDS

```
STA COLOR_RAM + LINE11_COL39, X
STA COLOR_RAM + LINE13_COL39, X
STA COLOR_RAM + LINE15_COL39, X
STA COLOR_RAM + LINE17_COL39, X
STA COLOR_RAM + LINE19_COL39, X
DEX
BNE DrawTitleTextLoop
```

Listing 4.9: In DrawTitleScreenText

In this case we're not writing a single character over and over, rather we're writing text we've defined elsewhere in variables `titleScreenTextLine[1-5]`:

```
titleScreenTextLine1    .TEXT "IRIDIS ALPHA..... HARD AND FAST ZAPPING"
titleScreenTextLine2    .TEXT "PRESS FIRE TO BEGIN PLAY.. ONCE STARTED, "
titleScreenTextLine3    .TEXT "F1 FOR PAUSE MODE   Q TO QUIT THE GAME"
titleScreenTextLine4    .TEXT "CREATED BY JEFF MINTER... SPACE EASY/HARD"
titleScreenTextLine5    .TEXT "LAST GILBY HIT 0000000; MODE IS NOW EASY"
```

Listing 4.10: In DrawTitleScreenText

In each iteration of the loop we write a character to all five columns, plucking it from the position in `titleScreenTextLine[1-5]` given by `x`.

Figure 4.6: The shaded areas of SCREEN_RAM after they have been written to by DrawStripesBehindTitle and DrawTitleScreenText.

CHAPTER 4. THE FIRST 16 MILLISECONDS

While writing text for the column we also set the color for each of the text lines to grey:

```
LDA #GRAY2  
STA COLOR_RAM + LINE11_COL39,X  
STA COLOR_RAM + LINE13_COL39,X  
STA COLOR_RAM + LINE15_COL39,X  
STA COLOR_RAM + LINE17_COL39,X  
STA COLOR_RAM + LINE19_COL39,X
```

Listing 4.11: In DrawTitleScreenText

Once it is done the COLOR_RAM has the appropriate lines set to grey:

Figure 4.7: The shaded areas of SCREEN_RAM after they have been written to by DrawStripesBehindTitle and DrawTitleScreenText.

Now that we've looped through all 40 columns we have both SCREEN_RAM and COLOR_RAM fully prepared for painting by the raster. As we watch the screen getting painted in the next section we'll see the following picture we've prepared gradually appear - with the sprites painted on top of course. The magic of adding the sprites to this picture, and animating them while we're at it, is what we will unpick as follow the raster on its journey to the bottom of the screen in the next few milliseconds.



Figure 4.8: The screen as it would appear after `DrawStripesBehindTitle` and `DrawTitleScreenText` have run. The added grid helps compare with our previous figures for `SCREEN_RAM` and `COLOR_RAM`.

4.3 Racing the Beam

Now we're ready to receive our first beam. You may remember we set this to happen when the raster reached line 16:

```
; Set the position for triggering our interrupt.
LDA #$10
STA $D012 ;Raster Position
```

Listing 4.12: In `InitializeSpritesAndInterruptsForTitleScreen`

And that the routine we'll run when that happens is `TitleScreenInterruptHandler` (which itself will pass the work onto `TitleScreenAnimation`):

```
; Set up the our interrupt handler for the title
; screen. This will do all the animation and title
; music work.
LDA #<TitleScreenInterruptHandler
STA $0314 ;IRQ
LDA #>TitleScreenInterruptHandler
```

```
STA $0315 ; IRQ
```

Listing 4.13: In InitializeSpritesAndInterruptsForTitleScreen

The painting of sprites and playing of music as the screen gets painted is all handled by `TitleScreenAnimation`. This routine works by calling one of three different subroutines each time it's called. It picks the one to run depending on some internal state it maintains, all with a view to ensuring that the sprites spelling out the game's title and the sprites depicting the animated gilbies are updated and in place before the raster reaches them.

To ensure it gets called by the interrupt when it's needed it will repeatedly update the line that the next interrupt should happen. We'll trace this as it actually happens, interrupt by interrupt, and sift through what the routine does at each step during the raster's first pass at painting the entire screen.

The first time the raster is called, this is what the screen looks like:



DoStarfieldAnimation (127 Cycles)

Figure 4.9: The state of the screen the first time the raster interrupt is received at line 16.

Of course we never actually see the screen in this state because it only appears for a microsecond or two, much too fast for us to observe. But as you can see the painting has already started. Everything above line 16 indicated in the figure has been painted black, as per our preparation of `SCREEN_RAM` a little earlier.

What our diagram above also tells us is that in this visit from the beam `TitleScreenAnimation` chose to execute the sub-routine `DoStarfieldAnimation` and that it took 127 CPU cy-

cles to complete it. Since it takes the raster 63 cycles to do an entire line this means that by the time we've finished this piece of work, the raster will have moved on to the next line - which is why the diagram shows 17 rather than 16. Every time we get an interrupt, the raster doesn't wait for us. We have to work quickly, especially if we're preparing graphics on lines that its likely to reach soon. This is why each of these subroutines does as little as it can get away with to get the job done.

The three sub-routines the work at each raster interrupt can get divvied out to are `UpdateJumpingGilbyPositionsAndColors`, `DoStarfieldAnimation`, and a cluster of routines starting with `UpdateTitleScreenSpriteColors`. The one that's called the most often is `DoStarfieldAnimation` as it is responsible for sprinkling the screen with animated stars traversing it left to right.

```

; -----
; TitleScreenAnimation
; This handles all the activity in the title screen and is called
; roughly 60 times a second by the Raster Interrupt.
; -----
TitleScreenAnimation
    LDY titleScreenStarFieldAnimationCounter
    CPY #$0C
    BNE MaybeDoStarFieldOrTitleText

    JSR UpdateJumpingGilbyPositionsAndColors
    LDY #$10
    STY titleScreenStarFieldAnimationCounter

MaybeDoStarFieldOrTitleText
    LDA titleScreenStarFieldYPosArray ,Y
    BNE DoStarfieldAnimation

PaintTitleTextSprites
    JSR TitleScreenMutateStarfieldAnimationData

    LDA #$00
    STA titleScreenStarFieldAnimationCounter

    LDA #$10
    STA $D012      ;Raster Position

    ; Acknowledge the interrupt, so the CPU knows that
    ; we have handled it.
    LDA #$01
    STA $D019      ;VIC Interrupt Request Register (IRR)
    STA $D01A      ;VIC Interrupt Mask Register (IMR)

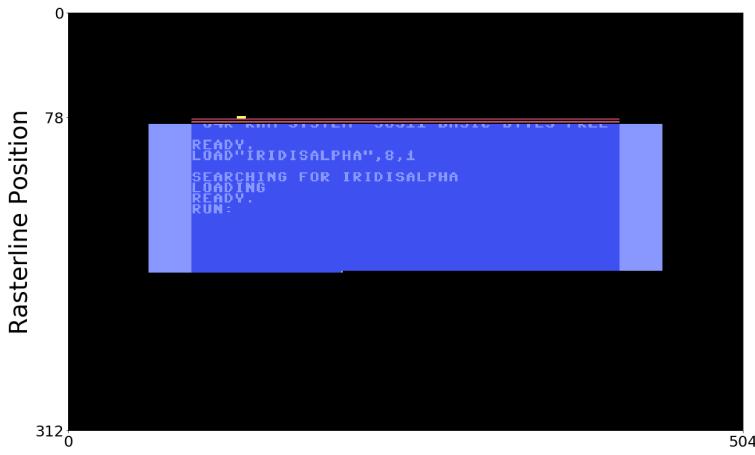
    JSR UpdateTitleTextSprites
    JSR MaybeUpdateSpriteColors
    JSR RecalculateJumpingGilbyPositions
    JSR PlayTitleScreenMusic
    JMP ReEnterInterrupt

```

```
; We're done, returns from function.
```

Listing 4.14: TitleScreenAnimation responsible for choosing what to do at each interrupt.

The internal accounting responsible for choosing the routine to run is tricky to decipher by just looking at the code. So instead let's follow what actually happens in practice. If we roll ahead to the next interrupt we can already see something happening:



DoStarfieldAnimation (99 Cycles)

Figure 4.10: The start of the stripes and a star.

If you look closely, you can see a yellow star painted over the first band of red stripes. This is our first sprite. You may be wondering: what about the title sprites? Shouldn't they be there by now? The answer is no: we will paint them when the raster reaches the end of the screen. When it goes to paint the screen a second time (the second 16 milliseconds) they will be ready for painting. We'll see this in action a little later.

So let's see what DoStarfieldAnimation did to get this sprite ready for painting.

When TitleScreenAnimation chose DoStarfieldAnimation as the sub-routine to run it loaded in a value from titleScreenStarFieldYPosArray to the A register:

```
MaybeDoStarFieldOrTitleText
    LDA  titleScreenStarFieldYPosArray ,Y
    BNE  DoStarfieldAnimation
```

Since Y is zero at this point this means it referenced the first value in the array, which is \$48:

```

titleScreenStarFieldYPosArray .BYTE $48,$4E,$54,$5A,$60,$66,$6C,$72
.BYTE $78,$7E,$84,$8A,$90,$96,$9C,$A2
.BYTE $A8,$AE,$B4,$BA,$C0,$C6,$CC,$D2
.BYTE $D8,$DE,$E4,$EA,$F0,$F6
titleScreenStarFieldXPosArray .BYTE $00,$3A,$1A,$C4,$1B,$94,$7B,$96
.BYTE $5D,$4F,$B5,$18,$C7,$E1,$EB,$4A
.BYTE $8F,$DA,$B3,$6A,$B0,$FC,$68,$04
.BYTE $10,$08,$A7,$B8,$19,$B8

```

So when `DoStarfieldAnimation` is called the first thing it does is set the y position of the star to paint to \$48 (72 in decimal):

```

DoStarfieldAnimation
; A was loaded from titleScreenStarFieldYPosArray
; by the caller.
STA $D00F      ;Sprite 7 Y Pos

; Set the X position of the star.
LDA titleScreenStarFieldXPosArray + $01,Y
STA $D00E      ;Sprite 7 X Pos

```

Listing 4.15: The start of `DoStarfieldAnimation` responsible for painting stars.

You can also see it then sets the X position of the star using values plucked from `titleScreenStarFieldXPosArray`. So we're leaning heavily on these two arrays to decide where to place stars. But so far, so simple. We've placed the star on the screen more or less and when the raster reaches line 72 it will paint it. There's an additional complication to specifying the X coordinate of the star though and we can't really gloss over it here. We'll also encounter this wrinkle elsewhere too so it's worth pausing on for a moment.

The next few lines of the routine do quite a bit of convoluted work to handle something called the `spriteMSBXPosOffset` of the star. This is our complication.

4.3.1 A Complication

```

; Set the rest of the X position of the star
; if it's greater than 255.
LDA titleScreenStarfieldMSBXPosArray + $01,Y
AND #$01
STA spriteMSBXPosOffset

BEQ StarFieldSkipMSB

LDA #$80
STA spriteMSBXPosOffset
StarFieldSkipMSB
LDA $D010      ;Sprites 0-7 MSB of X coordinate
AND #$7F
ORA spriteMSBXPosOffset

```

```
STA $D010      ;Sprites 0-7 MSB of X coordinate
```

Listing 4.16: MSBXPos.. some'it.

If you look at the diagram again you may recall we said the screen we're painting is 504 pixels wide. Fortunately the only part we can paint is the section in the center that is 320 pixels wide and 200 pixels high.

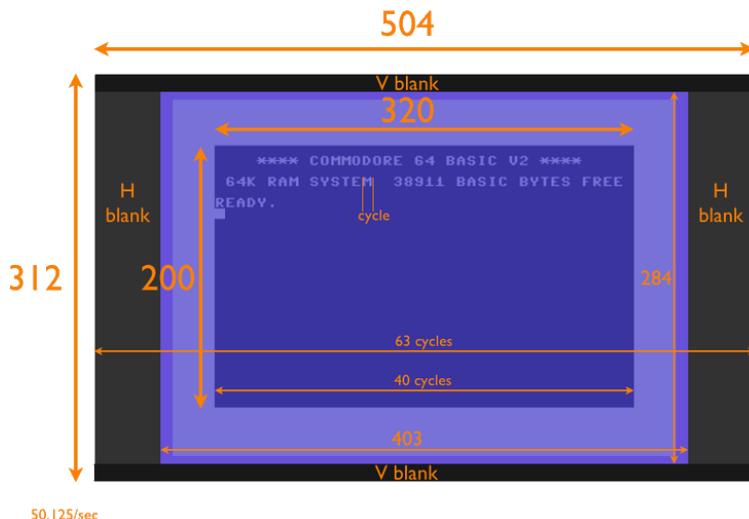


Figure 4.11: The different parts of the screen, we can only paint the bit in the middle. (Source: dustlayer.com)

200 is a value that can be expressed with a single byte. However, 320 is not. A byte can only store a number up to 255 so if we want to specify an X co-ordinate greater than 255 a single byte will not do. The way the C64 works around this is by making a single extra bit for our sprite's X co-ordinate available that brings the available values up from 256 (0 - 255) to 512. Since there are 8 sprites in total we need 8 extra bits to cover this requirement for all of them. For this purpose we use a single byte at address \$D010 that contains the extra bit for all 8 sprites. We refer to this bit as the **MSB** for the X co-ordinate because it is the 'Most Significant Bit', i.e. the left most bit, in the 9-bit number that we store the X co-ordinate in. That's to say our x co-ordinate is given by combining the value between 0 and 255 we store in our 8-bit byte for 'Sprite 7' in \$D00E and the extra bit we store in \$D010.

Sprite 7	Sprite 6	Sprite 5	Sprite 4	Sprite 3	Sprite 2	Sprite 1	Sprite 0
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	0	0	0	0

The most significant bits in \$D010 for each sprite.

Since we are using 'Sprite 7' for painting the starfield the bit we are interested in is bit 7. The way we're going to manage this value for the starfield is by keeping array `titleScreenStarfieldMSBPosXArray` that indicates whether the x co-ordinate for the current index is greater than 255. If the value in there indicates that it is we'll set our bit in \$D010 to 1.

So when we determine from looking in `titleScreenStarfieldMSBPosXArray` that the x co-ordinate of the star is greater than 256..

```
LDA titleScreenStarfieldMSBPosXArray + $01,Y
AND #$01
STA spriteMSBPosXOffset
```

.. we set `spriteMSBPosXOffset` to indicate that that's the case. That's all this step, with the help of the `AND #$01` statement is doing. If 'Bit 1' is set in the value we pluck from `titleScreenStarfieldMSBPosXArray` it is just an indicator that the x co-ordinate for this star is greater than 256. So if we see a value of \$02 in there our operation `AND #$01` will give us a zero result, meaning the intended value of the x co-ordinate is not greater than 256, otherwise it will give us a non-zero result indicating that it is:

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
\$02	0	0	0	0	0	0	1	0
\$01	0	0	0	0	0	0	0	1
Result: \$00	0	0	0	0	0	0	0	0

AND'ing \$02 and \$01 gives \$00 (0). For AND to give a 1 both bits must 1 or both must be 0.

This zero result allows `BEQ StarFieldSkipMSB` to evaluate as True so we skip ahead to `StarFieldSkipMSB` to set \$D010. It means for us that the value of the x co-ordinate is not going to be greater than 255. If this is not the case, we instead load a value of \$80 to `spriteMSBPosXOffset` to overwrite the 00 there. This will indicate that the value of the x co-ordinate is greater than 255.

```
BEQ StarFieldSkipMSB
```

```
LDA #$80
STA spriteMSBXPoSOffset
StarFieldSkipMSB
LDA $D010 ;Sprites 0-7 MSB of X coordinate
AND #$7F
ORA spriteMSBXPoSOffset
STA $D010 ;Sprites 0-7 MSB of X coordinate
```

The remaining step above is to load the value we've arrived at in `spriteMSBXPoSOffset` to `$D010`. Since we want to do this without affecting any of the other bits in there that have been set for the other sprites we can't just do a `LDA/STA` as that will overwrite what's already there. The combination of the `AND/OR` operations here accomplishes something quite nifty - it allows us to update just the bit (Bit 7) that interests us in `$D010`.

If we suppose the current value in `$D010` is `$F3`, our `AND #$7F` operation clears 'Bit 7' so that it is always set to zero:

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
\$F3	1	1	1	1	0	0	1	1
\$7F	0	1	1	1	1	1	1	1
Result: \$73	0	1	1	1	0	0	1	1

AND'ing `$F3` and `$00` gives `$73`. It clears 'Bit 7' for us of whatever value was there originally.

Now when we perform an 'or' operation on the result with `ORA spriteMSBXPoSOffset` it will have the effect of just setting 'Bit 7' with the value we've stored in `spriteMSBXPoSOffset`. In this case it remains at zero because that's what we have in `spriteMSBXPoSOffset` but if we have `$80` in there it would set it to 1:

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
\$73	0	1	1	1	0	0	1	1
\$00	0	0	0	0	0	0	0	0
Result: \$73	0	1	1	1	0	0	1	1

OR'ing `$73` and `$00` gives `$73`.

4.3.2 Back to the Beam

Now that we've fixed the star's co-ordinates there's just two main things left to do before we're done with handling this raster interrupt. One is to set the color of the star.

We do this using a look-up array where we get the color for the star per our current index and set it:

```
LDA titleScreenStarFieldColorsArrayLookUp ,Y  
TAX  
LDA titleScreenColorsArray - $01,X  
STA $D02E      ;Sprite 7 Color
```

The second, and most important, is that we update the position on the screen that we want the next interrupt to happen. Remember for this visit we were interrupted at line 17. We want to place stars on other lines so we keep a list of the lines we want to write stars on in titleScreenStarFieldYPosArray, i.e. an array that stores the y co-ordinates of our stars. What we do is simply update the Raster Interrupt with the next position from this array so that we get called when the raster reaches it:

```
; Update the raster position for the next interrupt  
; to the current line - 1. This will allow us to  
; draw the sprite multiple times on different lines.  
LDA titleScreenStarFieldYPosArray + $01,Y  
SEC  
SBC #$01  
STA $D012      ;Raster Position
```

In this instance we're setting the value to \$48 (72). So when the next interrupt happens it will reveal the star we just prepared. THis is the one we took a peek at in Figure 1.10.

The next ten interrupts will continue to revisit DoStarfieldAnimation. Let's look at the screen as it unfolds through each of these interrupts:

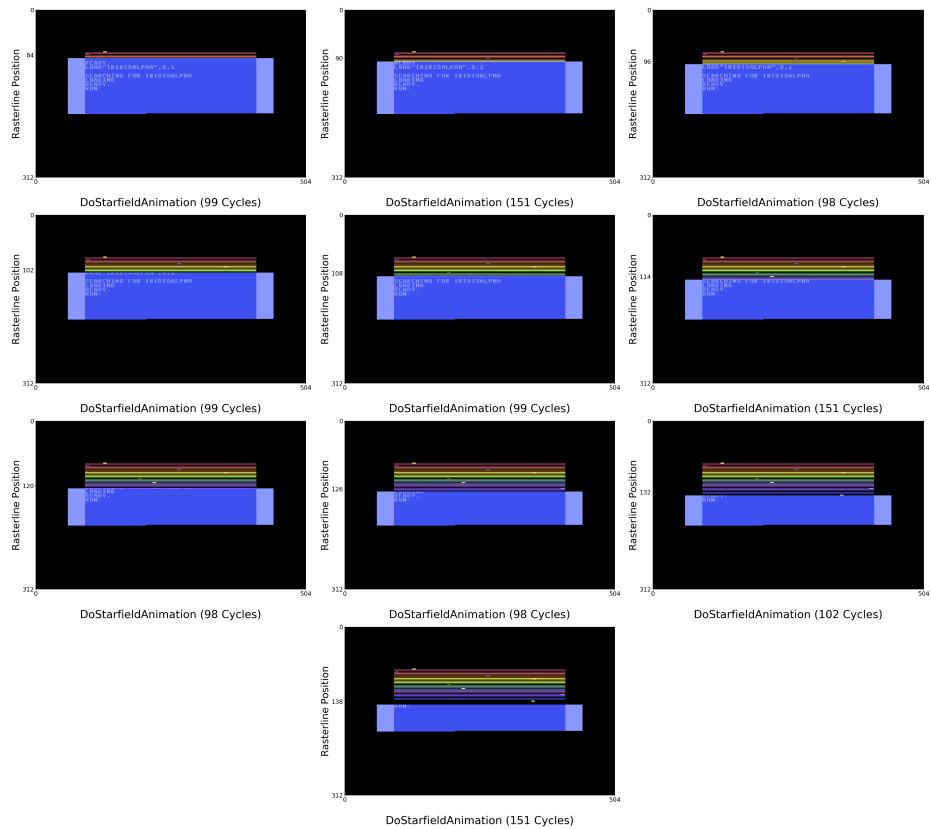
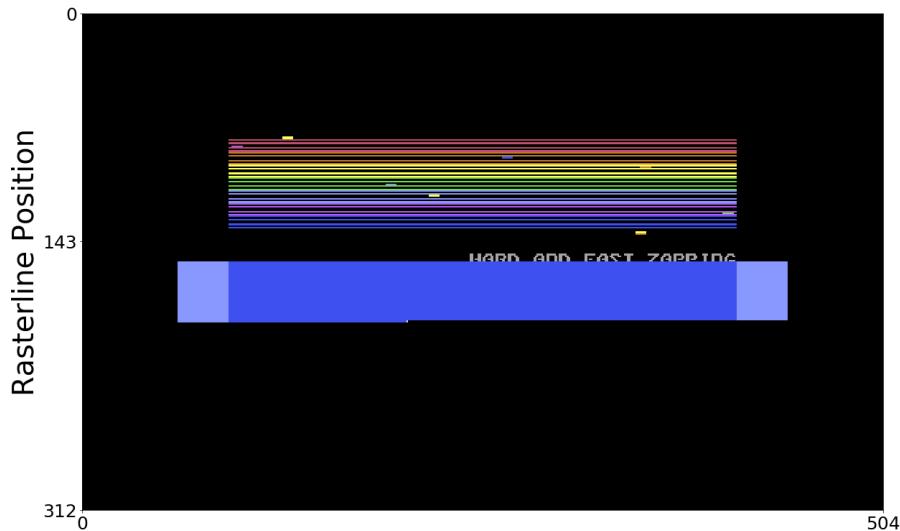


Figure 4.12: The next ten interrupts paint the starfield on the screen until we reach the point at which we want to prepare the gilby sprites.

4.4 Enter The Gilbies



`UpdateJumpingGilbyPositionsAndColors (546 Cycles)`

Figure 4.13: The point we reach in the screen paint when we decide to prepare the gilby sprites.

Finally we've reached a point in the screen where we're not just going to add another star to the background. We've been keeping a count of the number of interrupts we've handled in `titleScreenStarFieldAnimationCounter`. When it reaches \$0C (12) we've handled the raster interrupt twelve times and painted nothing but the text background we prepared earlier and the stars we've added along the way. Now's the time to do something else:

```

TitleScreenAnimation
LDY titleScreenStarFieldAnimationCounter
CPY #$0C
BNE MaybeDoStarFieldOrTitleText

JSR UpdateJumpingGilbyPositionsAndColors
LDY #$10
STY titleScreenStarFieldAnimationCounter

```

The routine we call into here by the name of `UpdateJumpingGilbyPositionsAndColors` will prepare the sequence of jumping rainbow gilbies somewhere in the lower half of the screen before our current raster line. That's why we call it at this point in the

raster's journey - because the raster hasn't reached that position in the screen yet (but will reach it shortly) so now is our opportunity to position the gilbies where we want them.

Since this is an animation sequence we're managing the approach of UpdateJumpingGilbyPositionsAndColors is simply to update their position on the screen. Calculating this new position is something that happens a little later in the routine RecalculateJumpingGilbyPositions when we are nearer the bottom of the screen. This means the position values we're picking up here are the initial ones set in the game's code:

```
titleScreenGilbiesYPosArray      .BYTE $B2,$B6,$BB,$C1,$D0,$C8,$C1  
titleScreenGilbiesXPosArray      .BYTE $54,$58,$5C,$60,$64,$68,$6C
```

The next time around we will pick up the positions as re-calculated by RecalculateJumpingGilbyPositions. So the positioning of the gilbies and the calculation of the updated positions happen separately. The reason for that approach is simple: there isn't enough time right now to do anything but simply update the positions the gilbies are displayed at. Later on, when the raster has passed line 320 we will have a lot more time available to perform complex calculations because we don't need to worry about the raster painting anything on the screen for a while.

Since there are 7 of them, setting the x and y co-ordinates of the seven gilby sprites is handled by a loop:

```
; Loop through the gilby sprites in the title screen and  
; update their position and color  
LDX #$00  
UpdateJumpingGilbiesLoop  
    TXA  
    ASL  
    TAY  
    LDA titleScreenGilbiesXPosArray,X  
    ASL  
    STA $D000,Y ;Sprite 0 X Pos  
    BCC SkipGilbyMSBXPos  
    LDA $D010 ;Sprites 0-7 MSB of X coordinate  
    ORA titleScreenGilbiesMSBXPosArray,X  
    STA $D010 ;Sprites 0-7 MSB of X coordinate  
    JMP UpdateYPosJumpingGilbies  
  
SkipGilbyMSBXPos  
    LDA $D010 ;Sprites 0-7 MSB of X coordinate  
    AND titleScreenGilbiesMSBXPosOffset,X  
    STA $D010 ;Sprites 0-7 MSB of X coordinate  
  
UpdateYPosJumpingGilbies  
    LDA titleScreenGilbiesYPosARray,X
```

```
STA $D001,Y ;Sprite 0 Y Pos
LDA currentTitleScreenGilbySpriteValue
STA SpriteOPtr,X

; Update Gilby color.
LDA titleScreenColorsArray,X
STA $D027,X ;Sprite 0 Color

INX
CPX #$07
BNE UpdateJumpingGilbiesLoop
RTS
```

Listing 4.17: The loop in `UpdateJumpingGilbyPositionsAndColors` updating the x and y position on screen and color of each of the gilby sprites.

We can see in here the verbosity required to handle the most significant bit of the sprite's x co-ordinate. Just as with the starfield we need a separate array (`titleScreenGilbiesMSBXPosArray`) to handle this in addition to arrays to manage the basic x/y positions themselves.

With the gilbies prepared we fall through and update the starfield again. Once that's done we're finished handling the current raster interrupt. This is followed by another dozen or so interrupts where we again just prepare stars for display and as the raster progress our gilbies are revealed.

CHAPTER 4. THE FIRST 16 MILLISECONDS

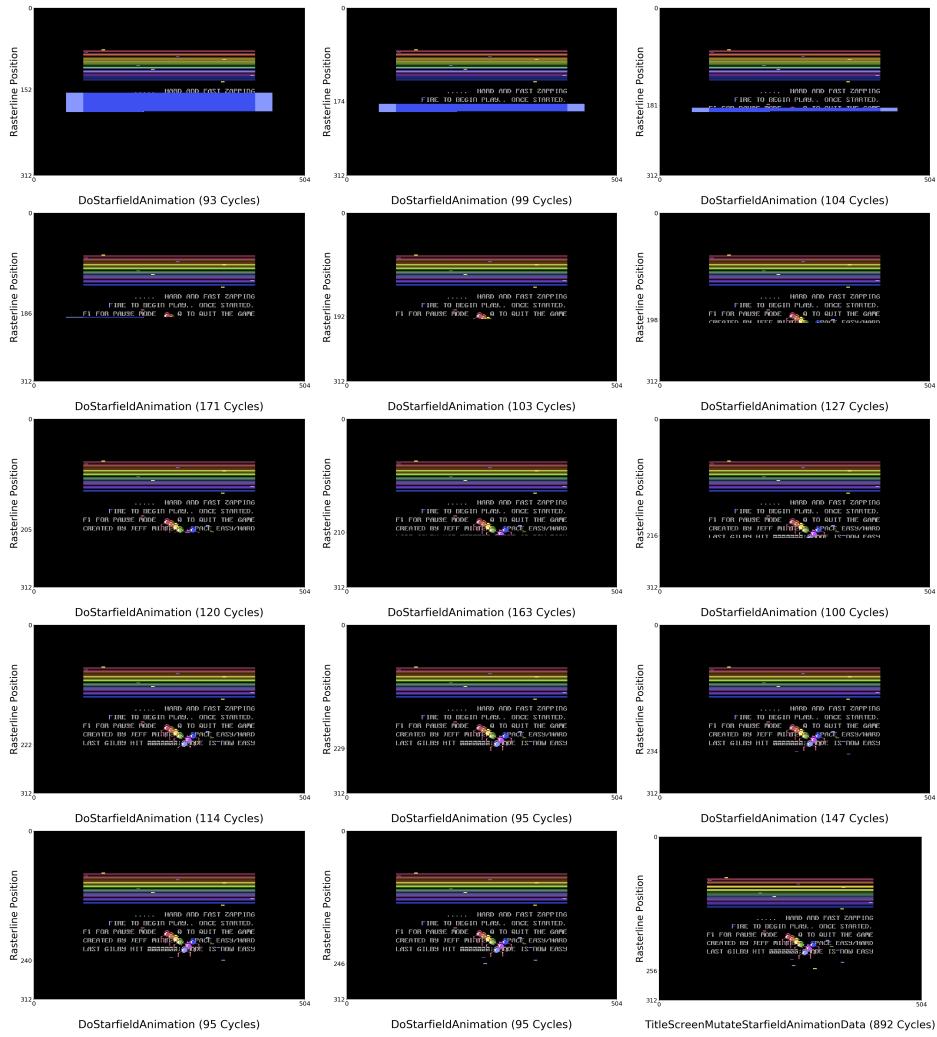


Figure 4.14: Behold the giblies.

4.5 Title Text



UpdateTitleTextSprites and Music (1002 Cycles)

Figure 4.15: We've finally reached the bottom of the screen, with gilbies and stars painted, but still no title.

We've finally reached the bottom of the screen in our first raster paint, at least the bottom of the portion of the screen that we can paint. When the raster hits line 270 we're beyond the point that we can place anything on the screen and into the border area. This gives us time to do some more complicated and time consuming stuff.

There's a relatively full agenda:

```

LDA #$10
STA $D012      ;Raster Position

; Acknowledge the interrupt, so the CPU knows that
; we have handled it.
LDA #$01
STA $D019      ;VIC Interrupt Request Register (IRR)
STA $D01A      ;VIC Interrupt Mask Register (IMR)

; All of this stuff can be done before the raster
; reaches the top of the screen again.
JSR UpdateTitleTextSprites
JSR MaybeUpdateSpriteColors

```

```
JSR RecalculateJumpingGilbyPositions
JSR PlayTitleScreenMusic
JMP ReEnterInterrupt
```

First of all we set the raster interrupt to line 16 at the top of the screen again, then we acknowledge the interrupt. The raster will continue its journey but because we're going to do this while it works its way the next 42 lines at the bottom of the screen we have more time than at any point previously to get things done.

Adding the title sprites is relatively light work. Just as with the gilbies we use a tight loop to paint each of them on the screen. THere's no animation to handle here.

```
titleTextSpriteArray           .BYTE $20,BIG_I,BIG_R,BIG_I,BIG_D,BIG_I,
                                BIG_S
...
PaintSpriteLettersLoop
    ; Assign the sprite.
    LDA titleTextSpriteArray,X
    STA Sprite0Ptr - $01,X

    ; Shift the value in X left 1 bit and assign to Y.
    ; So e.g. 6 becomes 12, 5 becomes 10, 4 becomes 8,
    ; 3 becomes 6 and so on. This allows us to use Y
    ; as an offset to the appropriate item in $D000-
    ; $D012 for updating the sprite's position.
    TXA
    ASL
    TAY

    ; Update the X Position of the sprite
    LDA titleTextXPosArray - $01,X
    STA $D000 - $02,Y

    LDA $D010      ;Sprites 0-7 MSB of X coordinate
    ORA titleTextMSBXPosArray,X
    STA $D010      ;Sprites 0-7 MSB of X coordinate

    ; Update the Y position of the sprite
    LDA #$40
    STA $D000 - $01,Y
    DEX
    BNE PaintSpriteLettersLoop
```

The other complex thing we do is calculate the next step in the jumping gilby animations. RecalculateJumpingGilbyPositions updates the x and y positions of the gilby sprites in titleScreenGilbiesYPosArray and titleScreenGilbiesXPosArray.

Finally we play a single note from the title music, we cover this in detail in a later chapter.

The raster continues on its journey and progresses through its second paint journey of the screen. The title sprites are finally revealed.

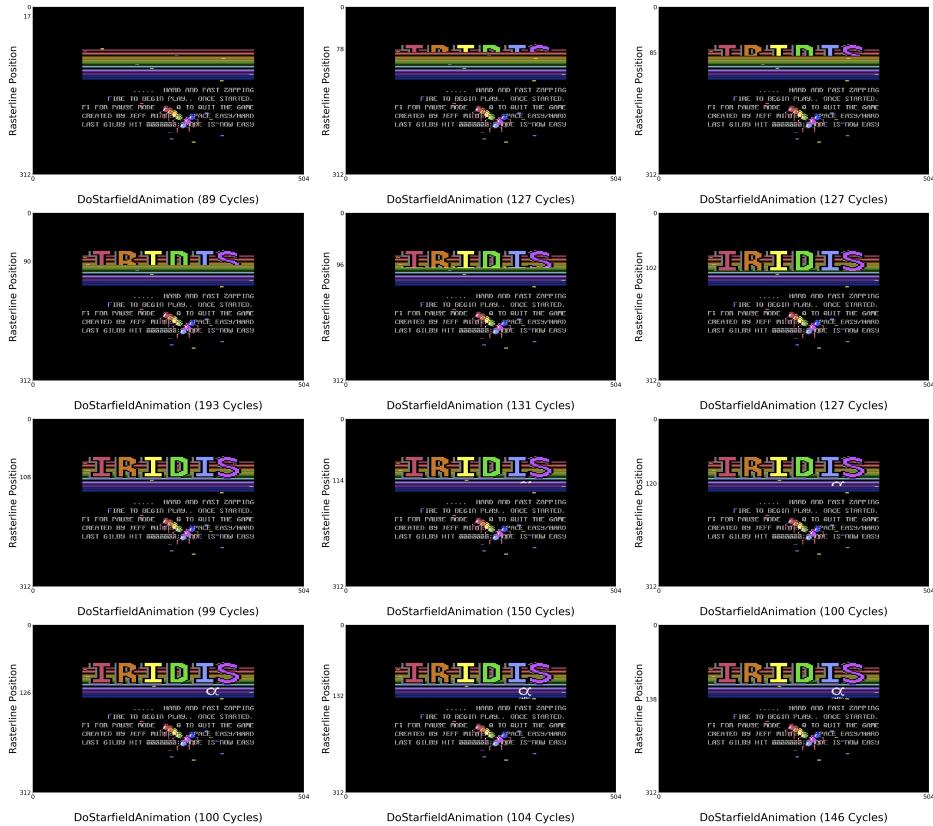
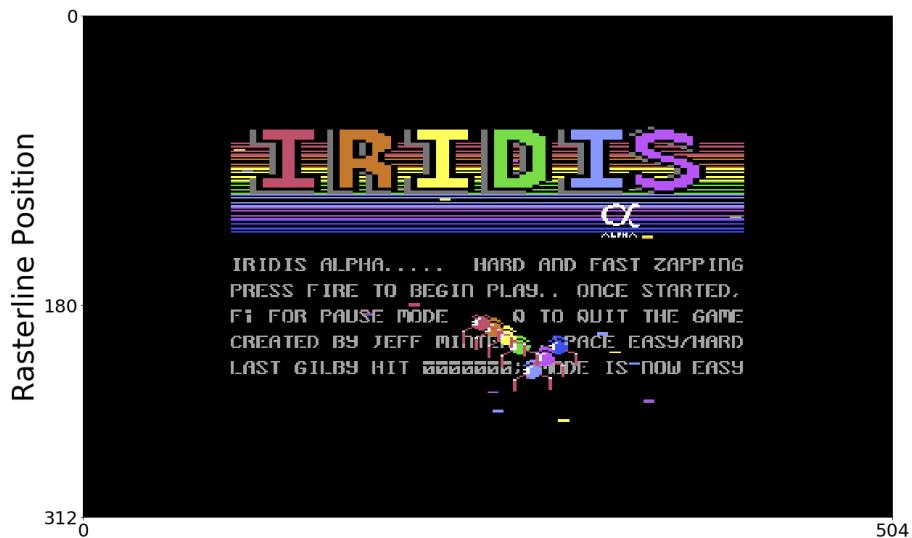


Figure 4.16: The title text is finally revealed

And with that the title sequence is finally up and running after 20 milliseconds or so.



DoStarfieldAnimation (98 Cycles)

Figure 4.17: We're done here.

Making Planets for Nigel

17 February 1986

Redid the graphics completely, came up with some really nice looking metallic planet structures that I'll probably stick with. Started to write the GenPlan routine that'll generate random planets at will. Good to have a C64 that can generate planets in its spare time. Wrote pulsation routines for the colours; looks well good with some of the planet structures. The metallic look seems to be 'in' at the moment so this first planet will go down well. There will be five planet surface types in all, I reckon, probably do one with grass and sea a bit like 'Sheep in Space', cos I did like that one. It'll be nice to have completely different planet surfaces in top and bottom of the screen. The neat thing is that all the surfaces have the same basic structures, all I do is fit different graphics around each one.

— Jeff Minter's Development Diary in Zzap Magazine^[?]

Making planets is easy.

When making a planet, ensure you perform each of the following simple steps in the order given below.



Figure 5.1: Step One: Add the sea across the entire surface of the planet.

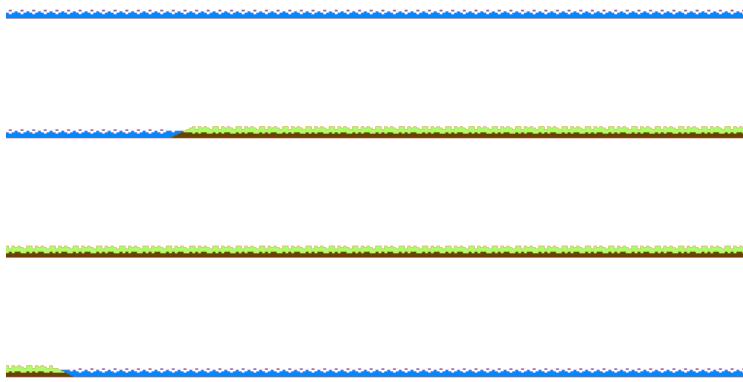


Figure 5.2: Step Two: Insert a land mass at least 32 bytes and at most 128 bytes long.

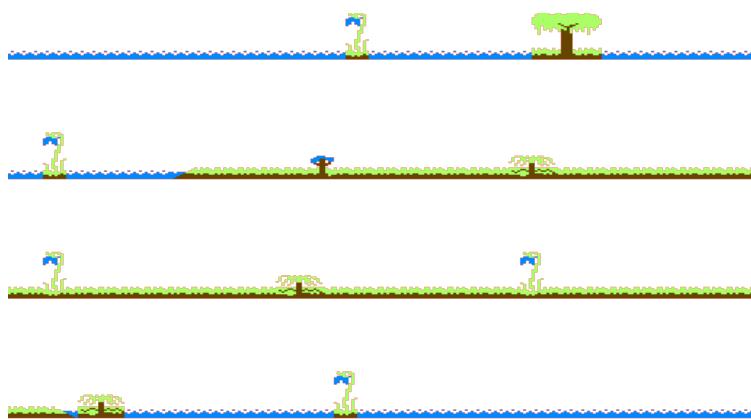


Figure 5.3: Step Three: Add a random structure every 13 to 29 bytes.

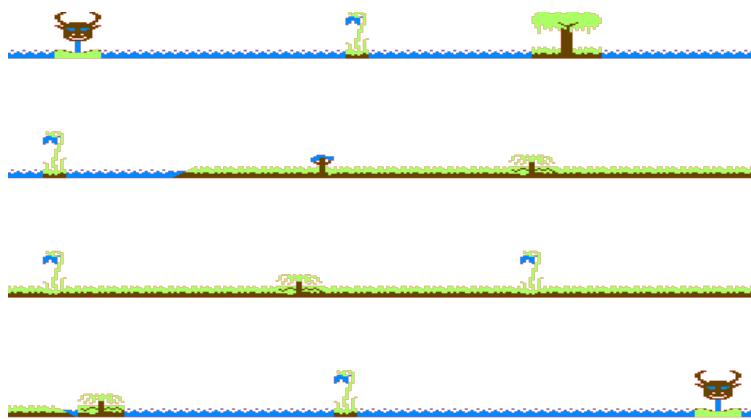


Figure 5.4: Step Four: Add warp gates at the beginning and end of the planet surface.

Now you have not just a layout for one planet, but a layout for all five.

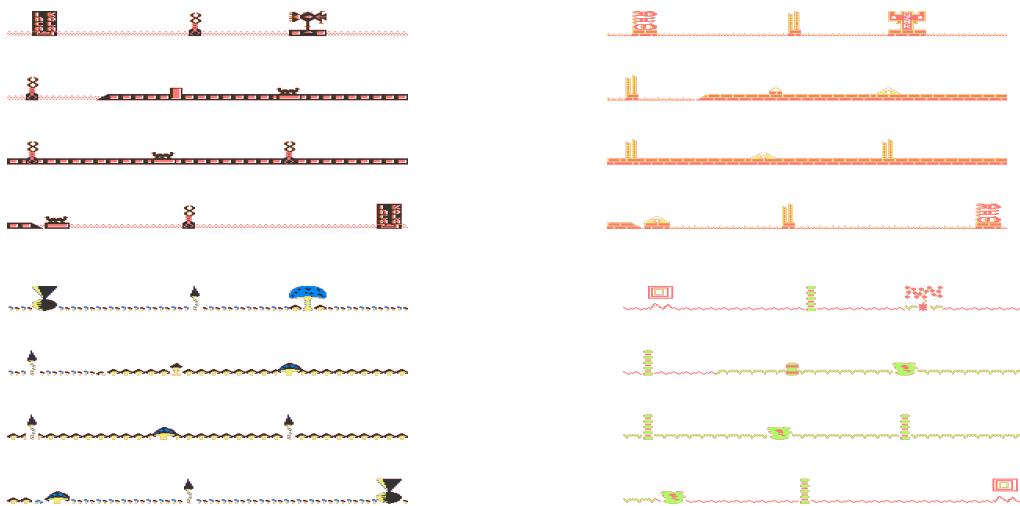
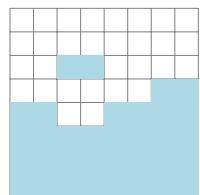


Figure 5.5: A layout that will suit all the planets in your life.

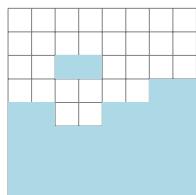
But making planets isn't all simple steps and big picture decisions. There are also trifling details for the little people to wrestle with.

5.1 Step One: Creating the Sea

Making a sea is very easy. You come up with a character than can be repeated 1024 times to fill the surface of the planet.



planet1Charset \$40



planet1Charset \$42

Figure 5.6: There are two characters used for creating the sea and they're both the same! This will make more sense when we look at the land, where they are different.

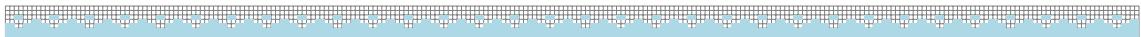


Figure 5.7: planet1Charset Sea

The bit that needs explaining is how you define the character. If it was a simple bitmap then we could imagine the character as 8 rows of 8 bits and where a bit is set to 1 you color that pixel in. That is not the case. You can see how the bits are actually set below:

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	0	0	0	1	0
1	0	0	0	1	0	1	0
1	0	1	0	1	0	1	0
1	0	1	0	1	0	1	0
1	0	1	0	1	0	1	0

Figure 5.8: planet1Charset \$40 representing a tile of sea.

Look closely at the picture above and you should see how it works. What is happening is that we fill two adjacent cells with blue when together they form the value 10. So we create graphic characters not with a simple bit-map but with a map of bit pairs. Each pair of bits is treated as a unit giving us four units on each row. Maybe it's intuitively obvious that 00 means 'blank' or 'background' but I've pointed that out to you now just in case.

```
planet1Charset
.BYTE $00,$00,$20,$02,$8A,$AA,$AA,$AA ; BYTE $00,$00,$20,$02,$8A,$AA,$AA,$AA
; CHARACTER $40
; 00000000
; 00000000
; 00100000      *
; 00000010      *
; 10001010      * * * *
; 10101010      * * * *
; 10101010      * * * *
```

Listing 5.1: Character \$40 representing the sea as it is defined in the source code. A full eight bytes are required to define each character so not cheap.

Is that all there is to it? No. Before we look at how we might color things other than blue, let's look at how we color them with the big blue brush we have so far. The first thing we do is clear down the entire surface of the planet:

```
; Clear down the planet surface data from $8000 to $FFFF.
; There are 4 layers:
```

```
; Top Layer:    $8000 to $83FF - 256 bytes
; Second Layer: $8400 to $87FF - 256 bytes
; Third Layer:  $8800 to $8BFF - 256 bytes
; Bottom Layer: $8C00 to $8FFF - 256 bytes
LDY #$00
ClearPlanetHiPtrs
; $60 is an empty character and gets written to the entire
; range from $8000 to $8FFF.
LDA #$60
ClearPlanetLoPtrs
STA (planetSurfaceDataPtrLo),Y
DEY
BNE ClearPlanetLoPtrs
INC planetSurfaceDataPtrHi
LDA planetSurfaceDataPtrHi
CMP (#>planetSurfaceData) + $10
BNE ClearPlanetHiPtrs
```

Listing 5.2: The surface data is stored from \$8000 to \$8FFF. This code overwrites it all with the value \$60 which is an empty bitmap.

```
.BYTE $00,$00,$00,$00,$00,$00,$00,$00 ;.BYTE $00,$00,$00,$00,$00,$00,$00,$00
; CHARACTER $60
; 00000000
; 00000000
; 00000000
; 00000000
; 00000000
; 00000000
; 00000000
; 00000000
; 00000000
```

Listing 5.3: The empty character bit map (all zeroes) used to overwrite the surface before populating it.

With the planet surface cleared out (overwritten with all \$60s) we can now.. overwrite it all again with sequences of \$40,\$42. No, that's not right. We're only overwriting the bottom layer - the surface layer - this time. This is the layer that contains the land and/or sea and it lives between \$8C00 and \$8FFF which if your hexadecimal arithmetic is better than mine you will realize is 1024 bytes (\$400 in hex).

```
; Fill $8C00 to $8FFF with a $40,$42 pattern. These are the
; character values that represent 'sea' on the planet.
LDA #$8C
STA planetSurfaceDataPtrHi
WriteSeaLoop
LDA #$40
STA (planetSurfaceDataPtrLo),Y
LDA #$42
INY
STA (planetSurfaceDataPtrLo),Y
DEY
; Move the pointers forward by 2 bytes
LDA planetSurfaceDataPtrLo
CLC
ADC #$02
STA planetSurfaceDataPtrLo
```

```

LDA planetSurfaceDataPtrHi
ADC #$00
STA planetSurfaceDataPtrHi
; Loop until $8FFF
CMP #$90
BNE WriteSeaLoop

```

Listing 5.4: Filling the entire bottom surface of the planet with \$40,\$42 which gives us the sea. Our next step is to overwrite some of this with land.



Figure 5.9: That sea again. Our work so far.

5.2 Step Two: Creating the Land

Is that all there is to it? Painting things with blue? No.

There are other possible values aside from 10 and 00 that we could use to paint colors. We could also have 11 and 01. This is useful since we want to color things in with more than one color. We have blue assigned to 10 on Planet 1, while for the land we can use two other colors: 11 which we will assign 'green' and 01 which we will assign 'brown'. We can assign whatever colors we like but we can only choose three, not counting the background. This is the kind of limitation you run into when you only allow two bits for assigning possible colors.

1	1	0	0	1	1	0	0
1	1	0	0	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	0	1	1	0	1	1
0	1	0	1	0	1	0	1
0	1	0	1	0	1	0	1
0	1	0	1	0	1	0	1

planet1Charset \$41

1	1	0	0	0	0	1	1
1	1	1	1	0	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
0	1	1	1	1	1	0	1
0	1	1	0	1	0	1	0
0	1	0	1	0	1	0	1
0	1	0	1	0	1	0	1

planet1Charset \$43

Figure 5.10: Planet 1 Land uses two different characters that alternate to generate the land surface.



Figure 5.11: planet1Charset Land

The location and length of the landmass is randomly generated with a couple of constraints: it must be at least 128 bytes and not more than 256 bytes from the start of surface and it must be at least 32 bytes and not more than 150 bytes long. The result is that the planet surface will be mostly sea since the entire surface is 1024 bytes long.

Picking a random number between 128 and 256 is slightly convoluted in assembly:

```
; Get a random number between 0 and 256 and store
; in A.
JSR PutProceduralByteInAccumulatorRegister
; Ensure the random number is between 128 and 256.
AND #$7F ; e.g. $92 becomes $12.
CLC ; Clear the carry so addition doesn't overflow.
ADC #$7F ; e.g. Adding $7F to $12 gives $91 (145).
; Store the result.
STA charSetDataPtrHi
```

Listing 5.5: Convoluted.

Neat Little Trick?

```
PutProceduralByteInAccumulatorRegister
randomIntToIncrement    =+$01
    LDA randomPlanetData
    INC randomIntToIncrement
    RTS
```

Listing 5.6: Neat.

This little snippet's job is to return a quasi-random byte for use in the planet generation routines. To achieve this, it does something quite fiendish that is more or less unheard of in modern programming: it mutates itself.

When called for the first time it loads a value from the address at `randomPlanetData` to the accumulator. On first run `randomPlanetData` points to the address \$9ABB which contains the value \$42:

```
randomPlanetData
    .BYTE $42,$E4,$3F,$94,$4E,$29,$B0,$59
    .BYTE $2C,$FE,$7F,$B2,$40,$9B,$63,$2B
```

Listing 5.7: Not Quite Random Bytes

Before returning this value as its result it alters itself by changing `randomPlanetData` to reference \$9ABC (INC `randomIntToIncrement`). In other words, it increments the pointer. In the assembly listing we make `randomIntToIncrement` reference the position that holds `randomPlanetData` by positioning it one byte before and adding a 1 to shift its reference beyond the byte holding `LDA` to `randomPlanetData`.

Every time the routine is called it increments the reference again so that the next time it will pick up whatever lies in the bytes beyond 9ABB. The results it returns are never truly random, but random enough to permit the procedural generation of planets that they're used for.

— A

With a random start position selected, a similar convolution is performed to choose the length of the land mass:

```
; Randomly generate the length of the land section, but
; make it at least 32 bytes and not more than 150.
JSR PutProceduralByteInAccumulatorRegister
AND #$7F ; Random number between 0 and 128
```

```
CLC  
ADC #$20 ; Add 32  
STA planetSurfaceDataPtrLo
```

Listing 5.8: A convolution.

Since the random number we get can be anything between \$00 – \$FF (i.e. 0 and 255) and we want a number that's between 0 and 128 we need to do a bitwise AND to mask out Bit 7 which by itself is 128.

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
\$FC	1	1	1	1	1	1	0	0
\$7F	0	1	1	1	1	1	1	1
Result; \$7C	0	1	1	1	1	1	0	0

AND'ing \$FC and \$7F gives \$7C (124).

With the position and length selected we can start laying turf. We don't just plop down our basic land tiles. Posh and proper means giving the shore of the land its own look and feel. This we have in the characters \$5C and \$5E in our character set:



\$5C



\$5E



\$5D



\$5F

Figure 5.12: Character tiles for the left shore (\$5C,\$5E) and the right shore (\$5D,\$5F).

Now we can put the rest of the land down:

```
randomPlanetData  
.BYTE $42,$E4,$3F,$94,$4E,$29,$B0,$59  
.BYTE $2C,$FE,$7F,$B2,$40,$9B,$63,$2B
```

Listing 5.9: Write pairs of \$41,\$43 for the main land mass.

```
; Draw the land from the randomly chosen position for up to  
; 150 bytes, depending on the randomly chosen length of the  
land  
; chosen above and stored in planetSurfaceDataPtrLo.  
DrawLandMassLoop  
INC charSetDataPtrHi  
BNE b7420
```

```

b7420  INC  charSetDataPtrLo
        JSR  StoreRandomPositionInPlanetInPlanetPtr
        LDY  #$00
        LDA  #$41
        STA  (planetPtrLo),Y
        LDA  #$43
        INY
        STA  (planetPtrLo),Y
        DEC  planetSurfaceDataPtrLo
        BNE  DrawLandMassLoop

```

Listing 5.10: Write pairs of \$41,\$43 for the main land mass.

And finally the right shore:

```

; Draw the right short of the land, represented by the chars in
; $5D/$5F.
INY
LDA  #$5D
STA  (planetPtrLo),Y
LDA  #$5F
INY
STA  (planetPtrLo),Y

```

Listing 5.11: Drawing the right hand shore..

5.3 Step Three: Structures Structures Structures

The routines for adding structures to the planet are the opportunity to observe some assembly language cleverness. For each structure we draw we have to decide two things: where to drop it on the surface and what type of structure to draw. Apart from the Warp Gates, there are four structure types available.

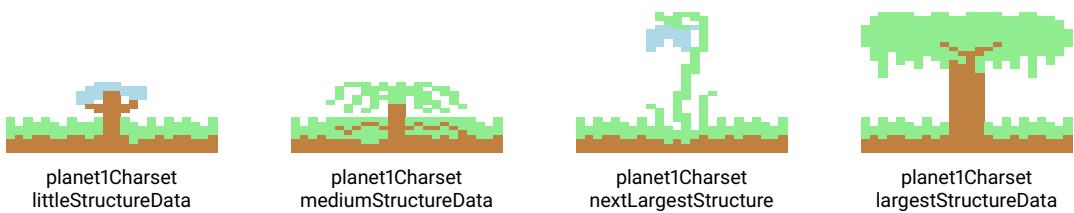


Figure 5.13: The four possible structure types for Planet 1.

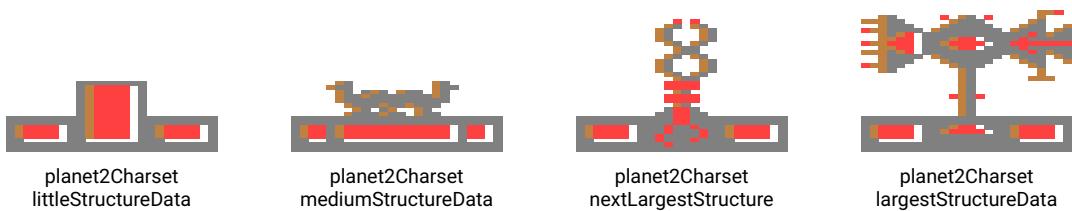


Figure 5.14: The four possible structure types for Planet 2.

You may be getting the sense that there is a sort of economy at work here. The structures are effectively the same for each planet, but with the textures swapped out. Your intuition is correct, the structures are only defined once and the same definition does regardless of which planet we're painting:

```
littleStructureData    .BYTE $45,$47,$FF  
                      .BYTE $44,$46,$FE  
mediumStructureData   .BYTE $65,$67,$69,$6B,$FF  
                      .BYTE $64,$66,$68,$6A,$FE  
largestStructureData  .BYTE $41,$43,$51,$53,$41,$43,$FF  
                      .BYTE $60,$60,$50,$52,$60,$60,$FF  
                      .BYTE $49,$4B,$4D,$4F,$6D,$6F,$FF  
                      .BYTE $48,$4A,$4C,$4E,$6C,$6E,$FE  
nextLargestStructure .BYTE $59,$5B,$FF  
                      .BYTE $58,$5A,$FF  
                      .BYTE $55,$57,$FF  
                      .BYTE $54,$56,$FE
```

Listing 5.12: The definitions of three of the structures above each of which serves all five planets.

The \$FF at the end of each line serves as a sentinel for the drawing routine to know that the subsequent bytes are for the next layer 'up'. The \$FE is a terminator, indicating there is no more data for the structure.

Drawing a structure is relatively straightforward so we'll cover that briefly first. Drawing the littlest structure provides the most compact example of the technique:

```
DrawLittleStructure  
    ; Start iterating at 0.  
    LDX #$00  
DrawLSLoop  
    ; Get the byte in littleStructureData pointed to  
    ; by X.  
    LDA littleStructureData,X  
    ; If we reached the 'end of layer' sentinel, move  
    ; our pointer planetPtrHi to the next layer. The  
    ; BNE 'stays on the same layer' by jumping to  
    ; LS_StayOnSameLayer if the current byte
```

```

; is not $FF.
CMP #$FF
BNE LS_StayonSameLayer
; Switch to the next layer.
JSR SwitchToNextLayerInPlanet
; SwitchToNextLayerInPlanet incremented X for us
; so continue looping.
JMP DrawLSSloop

LS_StayonSameLayer
CMP #$FE
; If we read in an $FE, we're done drawing.
BEQ ReturnFromDrawingStructure
STA (planetPtrLo),Y
; Increment Y to the next position to write to.
INY
; Increment X to get the next byte to read in.
INX
; Continue looping.
JMP DrawLSSloop

```

Listing 5.13: The littlest structure has only two layers.

Given that we're only writing 4 bytes this is a lot of code. As we will see there are separate routines for each of the structures and unfortunately for our search for evidence of coding genius they're all identical. So this is a pretty open-and-shut case of code duplication. It would have been more compact to rationalize them down to a single function and use a pointer to the structure data instead of repeating almost verbatim the same assembly code for each structure.

```

; -----
; DrawMediumStructure ($74B1)
; -----
DrawMediumStructure
    LDX #$00

DrawMSLoop
    LDA mediumStructureData,X
    CMP #$FF
    BNE b74C0
    JSR SwitchToNextLayerInPlanet
    JMP DrawMSLoop

b74C0  CMP #$FE
        BEQ ReturnFromDrawingStructure ; Return
        STA (planetPtrLo),Y
        INY
        INX
        JMP DrawMSLoop

; -----
; DrawLargestStructure ($74CB)

```

```
; -----
; DrawLargestStructure
    LDX #$00

DrawLargeStructureLoop
    LDA largestStructureData,X
    CMP #$FF
    BNE b74DA
    JSR SwitchToNextLayerInPlanet
    JMP DrawLargeStructureLoop

b74DA   CMP #$FE
    BEQ ReturnFromDrawingStructure ; Return
    STA (planetPtrLo),Y
    INY
    INX
    JMP DrawLargeStructureLoop
```

Listing 5.14: DrawMediumStructure and DrawLargestStructure are identical to each other and to DrawLittleStructure and DrawNextLargestStructure.

The cleverness comes a little earlier so let's console ourselves with that. When we've chosen a position to draw our structure we need to pick a type of structure at random. The secret to this is to store the addresses to our regrettably repetitive draw routines in a pair of arrays.

```
;Jump table
structureSubRoutineArrayHiPtr .BYTE >DrawLittleStructure,>DrawMediumStructure
                                .BYTE >DrawLargestStructure,>DrawNextLargestStructure
structureSubRoutineArrayLoPtr .BYTE <DrawLittleStructure,<DrawMediumStructure
                                .BYTE <DrawLargestStructure,<DrawNextLargestStructure
```

Listing 5.15: A 'jump table' containing the addresses to our draw routines. The address for DrawLittleStructure happens to be \$7486 so we store \$74 in the first byte of structureSubRoutineArrayHiPtr and \$86 in the first byte of structureSubRoutineArrayLoPtr.

With this in place our routine consists of getting a random number between 0 and 3, then using that as an index to pick out a value at the same position from structureSubRoutineArrayLoPtr and structureSubRoutineArrayHiPtr. We then store those values in structureRoutineLoPtr and structureRoutineHiPtr respectively. We now have a pointer to one of our draw routines at structureRoutineLoPtr which we can jump to with the simple command: JMP (structureRoutineLoPtr).

```
; -----
; DrawRandomlyChosenStructure
; -----
DrawRandomlyChosenStructure
    ; Pick a random position to draw the structure
    JSR StoreRandomPositionInPlanetInPlanetPtr

    ; Run the randomly chose subroutine, one of:
    ; DrawLittleStructure, DrawMediumStructure,
    ; DrawLargestStructure, DrawNextLargestStructure
```

```

; to draw a structure on the planet surface

; Pick a random number between 0 and 3
JSR PutProceduralByteInAccumulatorRegister
; AND'ing with $03 ensures the number is between
; 0 and 3.
AND #$03
; Move the number to the X register.
TAX
; Use the random number to pick and draw a structure.
LDA structureSubRoutineArrayHiPtr ,X
STA structureRoutineHiPtr
LDA structureSubRoutineArrayLoPtr ,X
STA structureRoutineLoPtr
; With the address of the routine we've chosen copied
; to structureRoutineLoPtr, we jump to that address and
; run the routine.
JMP (structureRoutineLoPtr)
; The routine contains an 'RTS' so does the returning
; for us.

```

Listing 5.16: DrawRandomlyChosenStructure picks a random position and a random draw routine to use at that position.

Rinse and repeat this for the length of the map and we get a surface with sea and land that is dotted with structures of different types.

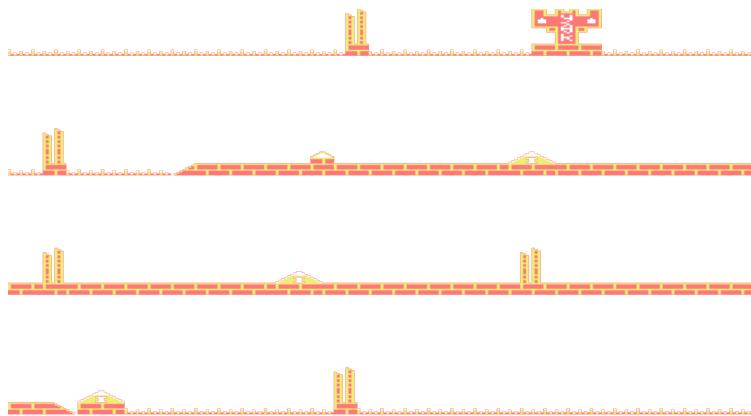


Figure 5.15: Planet 3 once DrawRandomlyChosenStructure has finished its business.

5.4 Step Four: Add the warp gate

Our final step is to add the warp gate.

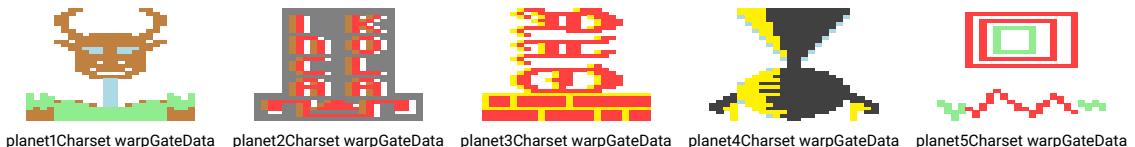


Figure 5.16: The warp gates on each planet.

There's something funny here I haven't figured out yet. The routine for drawing the warp gate draws it twice. Yet each level has only one warp gate. Each one gets an initial position of \$F1 and \$05 respectively. This is used by `StoreRandomPositionInPlanetInPlanetPtr` to point to a position on the surface where the warp gate is drawn.

```
DrawWarpGates
    LDA charSetDataPtrLo
    BEQ GenerateStructuresLoop

    ; Draw a warp gate at the end of the map.
    LDA #$F1
    STA charSetDataPtrHi

    JSR StoreRandomPositionInPlanetInPlanetPtr
    JSR DrawWarpGate
    DEC charSetDataPtrLo

    ; Draw a warp gate at the start of the map.
    LDA #$05
    STA charSetDataPtrHi

    JSR StoreRandomPositionInPlanetInPlanetPtr
    JSR DrawWarpGate
```

Listing 5.17: Why does it draw 2 warp gates when there's only 1? Haven't figured this out yet..

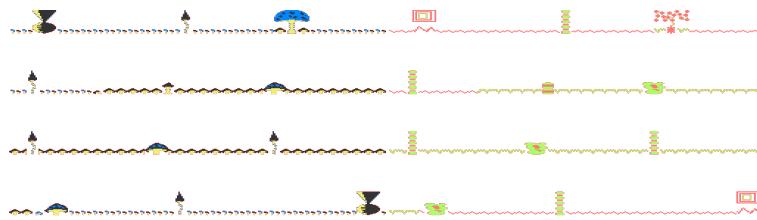


Figure 5.17: The final surfaces for Planets 4 and 5.

5.5 Inactive Lower Planet

When the lower planet is inactive a surface with land, sea, and a warp gate is displayed. This doesn't reuse any of the logic described above. Instead it is generated from some customized data in the routine `DrawLowerPlanetWhileInactive`.

```
; DrawLowerPlanetWhileInactive
; Draws the lower planet for the early levels when it isn't
; active yet.
;
;-----;
; DrawLowerPlanetWhileInactive
    LDA lowerPlanetActivated
    BEQ b6047

    LDX #$28
    DrawLowerTextLoop
        LDA textForInactiveLowerPlanet - $01,X
        AND #$3F
        STA SCREEN.RAM + LINE18.COL39,X
        LDA #WHITE
        STA COLOR.RAM + LINE18.COL39,X
        DEX
        BNE DrawLowerTextLoop

    LDX #$28
    DrawInactiveSurfaceLoop
        LDA surfaceDataInactiveLowerPlanet ,X
        CLC
        ADC #$40
        STA SCREEN.RAM + LINE14.COL39 ,X
        DEX
        BNE DrawInactiveSurfaceLoop

    LDX #$10
    DrawWarpGateInactive
        LDY xPosSecondLevelSurfaceInactivePlanet ,X
        LDA secondLevelSurfaceDataInactivePlanet ,X
        CLC
        ADC #$40
        STA SCREEN.RAM + LINE12.COL4 ,Y
        DEX
        BNE DrawWarpGateInactive
    RTS

; The *-$01 is because the array index starts at 1 rather than 0.
; xPosSecondLevelSurfaceInactivePlanet *=-$01
    BYTE $00,$01,$02,$02,$28,$29,$2A,$2B
    BYTE $50,$51,$52,$53,$57,$79,$7A,$7B
; secondLevelSurfaceDataInactivePlanet *=-$01
    BYTE $30,$32,$38,$3A,$31,$33,$39,$3B
    BYTE $34,$36,$3C,$3E,$35,$37,$3D,$3F
; surfaceDataInactiveLowerPlanet *=-$01
    BYTE $01,$03,$01,$03,$01,$03,$01,$03
```

```
.BYTE $01,$03,$01,$03,$01,$03,$01,$03  
.BYTE $01,$03,$01,$03,$01,$03,$01,$03  
.BYTE $01,$03,$01,$03,$01,$03,$1D,$1F  
.BYTE $00,$02,$00,$02,$00,$02,$00,$02  
  
textForInactiveLowerPlanet  
.TEXT " WARP GATE      GILBY    CORE  NOT-CORE"
```

Listing 5.18: Draw the inactive lower planet.

5.6 Drawing the Lower Planet

Drawing the upper planet is all very well. The data is just there and as we have seen it's just a question of 'writing it to the screen'. For the lower planet, which is just an upside down version of the upper one, we could store the data all over again, but inverted. Or we could try something a little more clever, so let's be clever.

Our clever trick will be to take each character definition for the upper planet and shift its bits around so that we can turn it into an upside down version of the original. This avoids the need to store everything twice. Instead we store it once for the upper planet and just mutate it for the lower planet.

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	0	0	0	1	0
1	0	0	0	1	0	1	0
1	0	1	0	1	0	1	0
1	0	1	0	1	0	1	0
1	0	1	0	0	0	1	0
1	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

planet1Charset \$40

1	0	1	0	1	0	1	0
1	0	1	0	1	0	1	0
1	0	1	0	1	0	1	0
1	0	1	0	0	0	1	0
1	0	1	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Inverted \$40

Figure 5.18: The character set definition for a piece of land and its topsy-turvy counterpart.

Doing this all at once for the full planet would be expensive and there will never be a good time to do it. So what we'll do instead is make a chutney from this pickle and do the heavy lifting while distracting the player's attention with an impressive-looking level-entry sequence.

When the player enters a pair of planets for the first time we'll slowly materialize the planet surfaces while quietly loading the upper planet data and converting it for the lower planet too. If we make this entry sequence long enough we'll have time to populate all 256 bytes of each planet's surface.

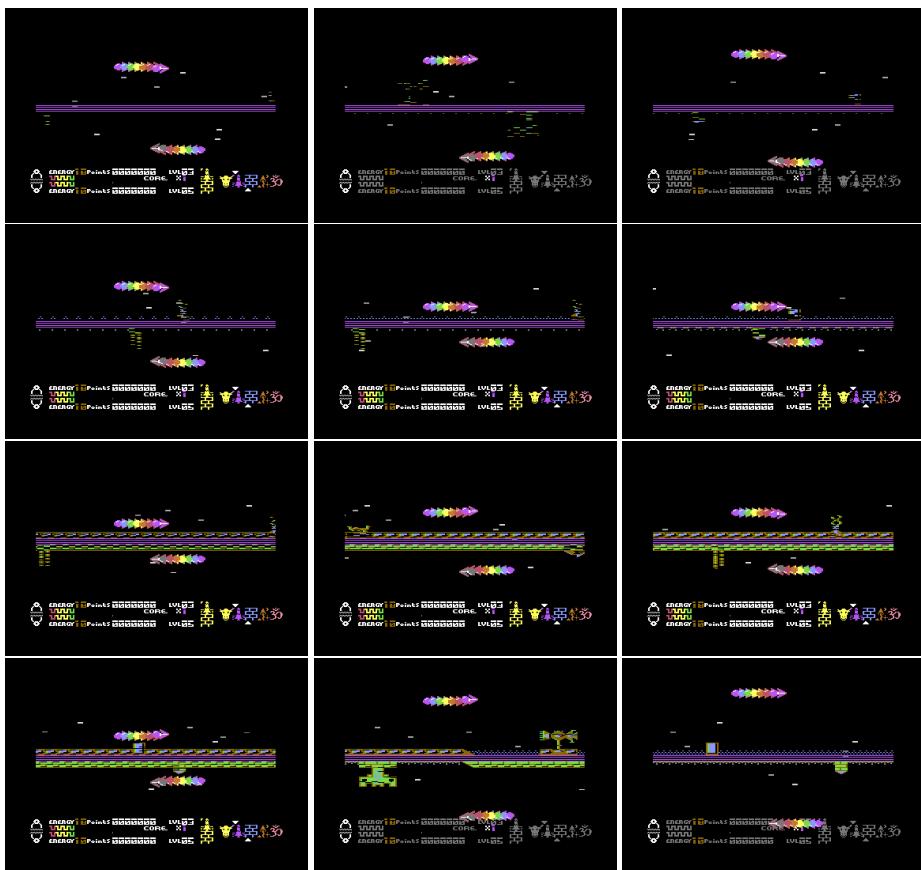


Figure 5.19: The materialization sequence as we enter a new pair of planets.

The trickery takes place in `MaybeDrawLevelEntrySequence`, one of the many routines called in the busiest code junction in the raster interrupt used during main game play. Even though the game isn't properly 'playing' at this point all of the main game code is running during the entry level sequence though nearly all of it except for this routine exits early as there is nothing for most of the routines to do:

```

; -----
; PerformMainGameUpdate
; -----
PerformMainGameUpdate
    LDX currentPlanetBackgroundClr1
    LDA backgroundColorsForPlanets,X
    STA $D022      ;Background Color 1, Multi-Color Register 0
    LDX currentPlanetBackgroundClr2

```

```
LDA backgroundColorsForPlanets ,X
STA $D023      ;Background Color 2, Multi-Color Register 1

LDA $D01F      ;Sprite to Background Collision Detect
STA spriteCollidedWithBackground

JSR CheckKeyboardInGame
JSR ScrollStarfieldAndThenPlanets
JSR AnimateGilbySpriteMovement
JSR PerformMainGameProcessing
JSR CheckForLandscapeCollisionAndWarpThenProcessJoystickInput
JSR PerformGilbyLandingOrJumpingAnimation
JSR AlsoPerformGilbyLandingOrJumpingAnimation
JSR MaybeDrawLevelEntrySequence
JSR PlaySoundEffects
JSR FlashBorderAndBackground
JSR UpdateGilbyPositionAndColor
JSR UpdateAndAnimateAttackShips
JSR UpdateBulletPositions
JSR DrawUpperPlanetAttackShips
JSR UpdateControlPanelColors
; Jump into KERNAL's standard interrupt service routine to
; handle keyboard scan, cursor display etc.
JMP ReEnterInterrupt
;Returns From Interrupt
```

Listing 5.19: PerformMainGameUpdate the spaghetti junction handling nearly everything during main gameplay. We'll see more of this code section later in the book. During the entry level sequence it is MaybeDrawLevelEntrySequence and PlaySoundEffects that do most of the work.

MaybeDrawLevelEntrySequence's principal sleight-of-hand is to make it look like it is slowly filling out the surface of the planets when that is precisely what it is doing. Not just to look fancy, but because inverting the tiles of the lower planet takes a bit of CPU time and would result in the player starting a blank screen for quite a long time if we tried to do it all at once.

The routine maintains a counter from 0 to 255 and every time it is visited it increments this counter and uses it to pick part of a tile in the charset to populate. The thing to remember here is that it is not actually painting the planet itself - rather it is slowly filling out the character set containing the tiles that define the textures on the planet. The game already knows which tiles it wants to put where - it just doesn't yet know what they look like. If we maintained separate copies of the upper planet and lower planet tiles we could just give them to the routine and painting would be easy. But because we don't have a copy of the lower planet tiles we have to generate them on the fly and resort to this bit of entry level sequence trickery.

Since there are 8 bytes in each character set definition, each represeting a line in the tile, each visit to this routine updates just one line in one tile. There are 32 tiles in total so 256 visits to the routine during the entry sequence will be enough to fill out the

tileset completely. This allows us to fill out the tile set in a random-looking way and is responsible for the materialization effect.

The first part of the `MaybeDrawLevelEntrySequence` is the easy part. We just pick the byte we're going to fill at this visit from a jumbled array of every value between 0 and 255 stored in `sourceOfSeedBytes`. This jumbled sequence is the source of the random-looking nature of the materialization.

```
MaybeDrawLevelEntrySequence
    LDA levelEntrySequenceActive
    BNE DrawLevelEntrySequence
ReturnFromEntrySequence
    RTS

DrawLevelEntrySequence
    LDX entryLevelSequenceCounter
    ; 'Y' becomes our 'random' index into the
    ; character set definition picking one of its
    ; 256 bytes to populate a line in one of the tiles.
    LDY sourceOfSeedBytes,X

    ; Pointers to the character set used for the
    ; current planet.
    LDA currentTopPlanetDataLoPtr
    STA planetSurfaceDataPtrLo
    LDA currentTopPlanetDataHiPtr
    STA planetSurfaceDataPtrHi

    ; The easy part, just copy part of the tile
    ; for the upper planet to where it needs to go.
    LDA (planetSurfaceDataPtrLo),Y
    STA upperPlanetsurfaceCharset,Y

    INC planetSurfaceDataPtrHi
    ; Likewise for the character set data used to
    ; define the heads up display.
    LDA (planetSurfaceDataPtrLo),Y
    STA upperPlanethUDCharset,Y

JSR InvertSurfaceDataForLowerPlanet
```

Listing 5.20: The start of `MaybeDrawLevelEntrySequence`. As long as `levelEntrySequenceActive` is non-zero `DrawLevelEntrySequence` will run.

This all just copying and pasting so far. We're slowly populating the tiles for the current upper planet without doing anything computationally intensive. The very last statement is where start work on the lower planet, and that's where the sorcery starts. In `InvertSurfaceDataForLowerPlanet` we repoint ourselves to the data for the lower planet and use a routine called `InvertCharacter` to do the actual turning of the byte upside-down:

```
InvertSurfaceDataForLowerPlanet
    LDA currentBottomPlanetDataLoPtr
    STA planetSurfaceDataPtrLo
    LDA currentBottomPlanetDataHiPtr
    STA planetSurfaceDataPtrHi

    ; This is the routine that does the actual inversion.
    ; Everything below and above is managing the copy/paste
    ; of the inverted character into the appropriate position
    ; in lowerPlanetSurfaceCharset.
    JSR InvertCharacter

    LDA invertedCharToDraw
    ; Note that 'X' was updated by InvertCharacter
    STA lowerPlanetSurfaceCharset,X
    INC planetSurfaceDataPtrHi

    ; Same as above, but for the charset used in the panel.
    JSR InvertCharacter
    LDA invertedCharToDraw
    ; Note that 'X' was updated by InvertCharacter
    STA lowerPlanethUDCharset,X

    RTS
```

Listing 5.21: InvertSurfaceDataForLowerPlanet.

The approach taken by InvertCharacter is two-fold: it has to reverse the chosen byte left to right and it also it has to move the byte to its appropriate position in the 8 byte sequence.

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	0	0	0	1	0
1	0	0	0	1	0	1	0
1	0	1	0	1	0	1	0
1	0	1	0	1	0	1	0
1	0	1	0	1	0	1	0

planet1Charset \$40

1	0	1	0	1	0	1	0
1	0	1	0	1	0	1	0
1	0	1	0	1	0	1	0
1	0	1	0	0	0	1	0
1	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Inverted \$40

Figure 5.20: Inverting 00100000 means transforming it to 00001000 and moving it from position 3 in the 8 byte character set definition to position 5.

5.6.1 Flipping the Byte

The first of these requirements is the more elaborate of the two. We need to look at each of the four bit-pairs in the byte and move it to its corresponding 'inverted' position in the byte. So in a way we treat one half of the byte as a mirror of the other and move the bit pair there as in the following examples.

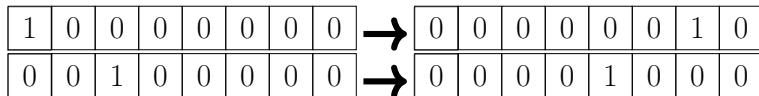


Figure 5.21: Examples of the inversion operation.

The way to do through a combination of bit-shifting and masking. This is what the code responsible looks like. We'll break it down.

```

bitfield1ForInvertingByte .BYTE $00,$40,$80,$C0
bitfield2ForInvertingByte .BYTE $00,$10,$20,$30
bitfield3ForInvertingByte .BYTE $00,$04,$08,$0C

InvertCharacter
    LDA (planetSurfaceDataPtrLo),Y
    PHA
    AND #$03
    TAX

    ; This part of the routine inverts the byte itself.
    LDA bitfield1ForInvertingByte,X
    STA invertedCharToDraw
    PLA
    ROR
    ROR
    PHA
    AND #$03
    TAX

    LDA bitfield2ForInvertingByte,X
    ORA invertedCharToDraw
    STA invertedCharToDraw
    PLA
    ROR
    ROR
    AND #$03
    TAX

    LDA bitfield3ForInvertingByte,X
    ORA invertedCharToDraw
    STA invertedCharToDraw
    LDA (planetSurfaceDataPtrLo),Y

```

```
ROL  
ROL  
ROL  
AND #$03  
ORA invertedCharToDraw  
STA invertedCharToDraw
```

Listing 5.22: InvertSurfaceDataForLowerPlanet.

Our first step is to take the byte we're interested in from the upper planet character set definition. Let's assume it's the one below.



We load this into the accumulator (A) so that we can get work on it. What we're going to do is shift each of the four bit pairs in turn into the rightmost position in the byte and use that value (between 0 and 3) as an index into a trio of helper arrays that give the 'mirrored' bit-pair for that particular position. These helper arrays are `bitfield[1-3]ForInvertingByte`.

We don't have to do any shifting to compare the rightmost bit-pair. We can just AND it with #\$03 and store the result in X:

```
bitfield1ForInvertingByte .BYTE $00,$40,$80,$C0  
InvertCharacter  
    LDA (planetSurfaceDataPtrLo),Y  
    PHA  
    AND #$03  
    TAX
```

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
\$08	0	0	0	0	1	0	0	0
\$03	0	0	0	0	0	0	1	1
Result; \$00	0	0	0	0	0	0	0	0

AND'ing \$08 and \$03 gives \$00.

Since the result is zero, we will then store the value at index zero in `bitfield1ForInvertingByte` as our result in `invertedCharToDraw`:

```
bitfield1ForInvertingByte .BYTE $00,$40,$80,$C0
```

```
LDA bitfield1ForInvertingByte,X  
STA invertedCharToDraw
```

Since our result is zero we've stored zero in `invertedCharToDraw`.

Now we can move on to the second rightmost bitpair. The ones highlighted in blue below:



We pushed our original value onto the stack using PHA. First we retrieve it using PLA and then shift it two bits to the right:

```
PLA  
ROR  
ROR
```

When we've done this our byte looks like this, the bitpair we're interested has moved to the rightmost position:



Now we can AND this byte with \$03 again to ensure we only deal with the last two bits (remember that although all the other bits are zero in the example, they may not be in other cases).

```
AND #$03  
TAX
```

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
\$02	0	0	0	0	0	0	1	0
\$03	0	0	0	0	0	0	1	0
Result; \$02	0	0	0	0	0	0	1	0

AND'ing \$02 and \$03 gives \$02.

This result of \$02 gives us an index into `bitfield2ForInvertingByte` which will give

us the 'mirrored' bit pair:

```
bitfield2ForInvertingByte .BYTE $00,$10,$20,$30
```

```
LDA bitfield2ForInvertingByte,X  
ORA invertedCharToDraw  
STA invertedCharToDraw
```

The byte we get from `bitfield2ForInvertingByte` at index 2 is \$20 (remember that indexes always start counting from 0 rather than 1). When we ORA this with our current value for `invertedCharToDraw` we end up with a value of \$20:

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
\$00	0	0	0	0	0	0	0	0
Result; \$20	0	0	1	0	0	0	0	0

ORA'ing \$20 and \$00 gives \$20.

As we can see, this has achieved an effective mirroring of our original value!



Figure 5.22: The result of our bit-shifting, AND'ing and ORing.

5.6.2 Flipping the Byte's Position

The remaining steps for the other two bit-pairs in the byte are similar. In the case of our example they will have no effect as the result will be always be zero for them. We've effectively mirrored our bitpair already.

The next and last thing we have to do with our mirrored byte is adjust its position in the 8 byte sequence in the character definition. In the example below we are dealing with a byte on the left which occurs as the 5th position in the byte sequence and in order to be inverted needs to be moved to the 4th position.

So whereas we were dealing with a left-right mirroring in the byte itself we are now dealing with an up-down mirroring in the position of the byte in its sequence. The total effect of our inversion operation is to flip the character set from left to right and top to bottom:

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	0	0	1	0	0
1	0	0	0	1	0	1	0
1	0	1	0	1	0	1	0
1	0	1	0	1	0	1	0
1	0	1	0	1	0	1	0

planet1Charset \$40

1	0	1	0	1	0	1	0
1	0	1	0	1	0	1	0
1	0	1	0	1	0	1	0
1	0	1	0	0	0	1	0
1	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Inverted \$40

Figure 5.23

The second half of InvertCharacter that does this looks like:

```

; Now that the byte has been inverted, invert the position in
; the 8 byte charset definition. For example, if the position is
; 0 then the inverted position is 7.

; Mask out everything but the last 3 bits in the current upper
; planet position.
TYA
PHA
AND #$07
TAY
PLA
PHA
AND #$F8
STA charSetDataPtrLo

; Now add the inverted position to get the correct lower planet
; position in the 8-byte charset definition. charSetDataPtrLo is
; just temporary storage here. We store the final value for use
; by the calling routine in 'X' below.
LDA positionInInvertedCharSet,Y
CLC
ADC charSetDataPtrLo

; By storing the updated position in the charset definition to 'X'
;
; here, we're changing the offset used in
lowerPlanetSurfaceCharset
; in InvertSurfaceDataForLowerPlanet.
TAX

PLA
TAY

```

The key to this operation lies in the use of the array positionInInvertedCharSet. If we know that the current byte is number 5 in the 8 byte sequence (i.e. the red row

below):

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	1
1	0	0	0	0	1	0	1
1	0	1	0	1	0	1	0
1	0	1	0	1	0	1	0
1	0	1	0	1	0	1	0

planet1Charset \$40

Figure 5.24

then we can use that as an index into `positionInInvertedCharSet` to retrieve the corresponding position in the inverted character set definition. As we can see value of the the 5th byte in `positionInInvertedCharSet` is \$03:

```
positionInInvertedCharSet      .BYTE $07,$06,$05,$04,$03,$02,$01,$00
```

Which as we can see is the appropriate position for the byte in the inverted character set (the 4th byte or red row starting from the top). (Remember that our index into arrays starts at zero, so an index value of zero will pick the first byte and of three, as in this case, will pick the fourth.)

1	0	1	0	1	0	1	0
1	0	1	0	1	0	1	0
1	0	1	0	1	0	1	0
1	0	1	0	0	0	1	0
1	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Inverted \$40

Figure 5.25

So where do we get the position of the current byte in the uninverted character set? This was passed into `InvertSurfaceDataForLowerPlanet` and `InvertCharacter` in

the Y register. It's a value between 0 and 256 which references the offset of the current byte in the character set data as a whole so all we have to do is clamp it to a value between 0 and 7 to get the value we need within the 8-byte definition for this specific character.

```
; Mask out everything but the last 3 bits in the current upper
; planet position.
TYA
PHA
AND #$07
```

The clamping is achieved by the AND statement. If we imagine that Y has a value of \$D4, we transfer it to the A register (TYA and then do an AND operation with \$07:

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
\$D4	1	0	1	1	0	0	0	0
\$07	0	0	0	1	1	1	1	1
Result: \$04	0	0	0	0	1	0	0	0

AND'ing \$D4 and \$07 gives \$04.

This gives us \$04 as a result, which as we saw returns \$03 when used as an index into `positionInInvertedCharSet`.

With our new position for the byte in the character set definition established we can store it to the X register and use that as the position in `lowerPlanetSurfaceCharset` where we will store the `invertedCharToDraw` we calculated earlier. We do this when we return from the `InvertCharacter` routine and are back in `InvertSurfaceDataForLowerPlanet`:

```
JSR InvertCharacter
LDA invertedCharToDraw
; Note that 'X' was updated by InvertCharacter
STA lowerPlanetSurfaceCharset,X
```

With that done, we've added another brick in the wall. Or more precisely, another byte in the 256 that in total make up the tileset for the upper and lower planets respectively. As the materialization sequence proceeds we slowly fill out the tileset byte-by-byte. After a few seconds we've successfully cheated our way into generating the lower planet's tileset on the fly into `lowerPlanetSurfaceCharset` and the game is ready to play.



Figure 5.26: The planets when we start the materialization sequence, and when we end.

Blasting, Fast and Slow

Every game has a 'main loop'. A tight section of code which is executed multiple times per second and which controls nearly all aspect of the gameplay. In Iridis Alpha this boiler-room is PerformMainGameUpdate:

```
; -----
; PerformMainGameUpdate
; -----
PerformMainGameUpdate
    LDX currentPlanetBackgroundClr1
    LDA backgroundColorsForPlanets,X
    STA $D022      ;Background Color 1, Multi-Color Register 0
    LDX currentPlanetBackgroundClr2
    LDA backgroundColorsForPlanets,X
    STA $D023      ;Background Color 2, Multi-Color Register 1

    LDA $D01F      ;Sprite to Background Collision Detect
    STA spriteCollidedWithBackground

    JSR CheckKeyboardInGame
    JSR ScrollStarfieldAndThenPlanets
    JSR AnimateGilbySpriteMovement
    JSR PerformMainGameProcessing
    JSR CheckForLandscapeCollisionAndWarpThenProcessJoystickInput
    JSR CalculateGilbyVerticalPositionEarthBound
    JSR CalculateGilbyVerticalPositionAirborne
    JSR MaybeDrawLevelEntrySequence
    JSR PlaySoundEffects
    JSR FlashBorderAndBackground
    JSR UpdateGilbyPositionAndColor
    JSR UpdateAndAnimateAttackShips
    JSR UpdateBulletPositions
    JSR DrawUpperPlanetAttackShips
    JSR UpdateControlPanelColors
; Jump into KERNAL's standard interrupt service routine to
```

```
; handle keyboard scan, cursor display etc.
JMP ReEnterInterrupt
;Returns From Interrupt
```

Listing 6.1: PerformMainGameUpdate the spaghetti junction handling nearly everything during main gameplay.

Just like in the title sequence this routine is called dozens of times as the beam of light painting the screen travels from top to bottom up to 25 times per second. PerformMainGameUpdate, along with a routine called AnimateStarFieldAndScrollPlanets whose purpose you can hopefully guess from its name, are the two gears that grind out the gameplay as long as the player is alive and blasting.

For each pass through the screen AnimateStarFieldAndScrollPlanets is executed 8 times while PerformMainGameUpdate is executed once.

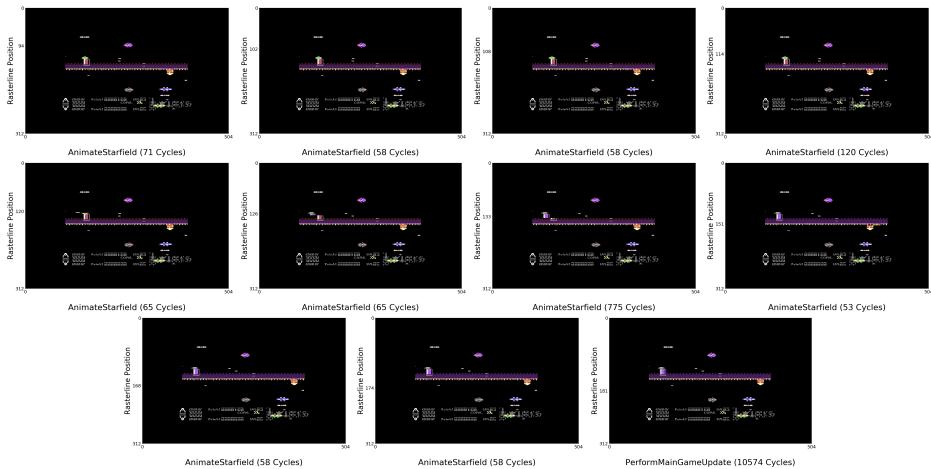


Figure 6.1: The rasterline position when AnimateStarFieldAndScrollPlanets and PerformMainGameUpdate are called.

If you look closely you'll notice that our so-called boiler-room routine is called last, when the raster is nearing the end of the screen. This makes sense as it has the most to do and therefore we need to execute it at a point when most of the screen has been painted and what to paint on the rest of it has already been prepared. PerformMainGameUpdate primarily concerns itself with preparing the screen for the next time it will be painted: updating the position of the enemies on the upper planet, the position of the player's ship, playing sound effects, moving the bullets and so on.

AnimateStarFieldAndScrollPlanets on the other hand only has to worry about painting the parallax starfield in the background, scrolling the planets using the C64's spe-

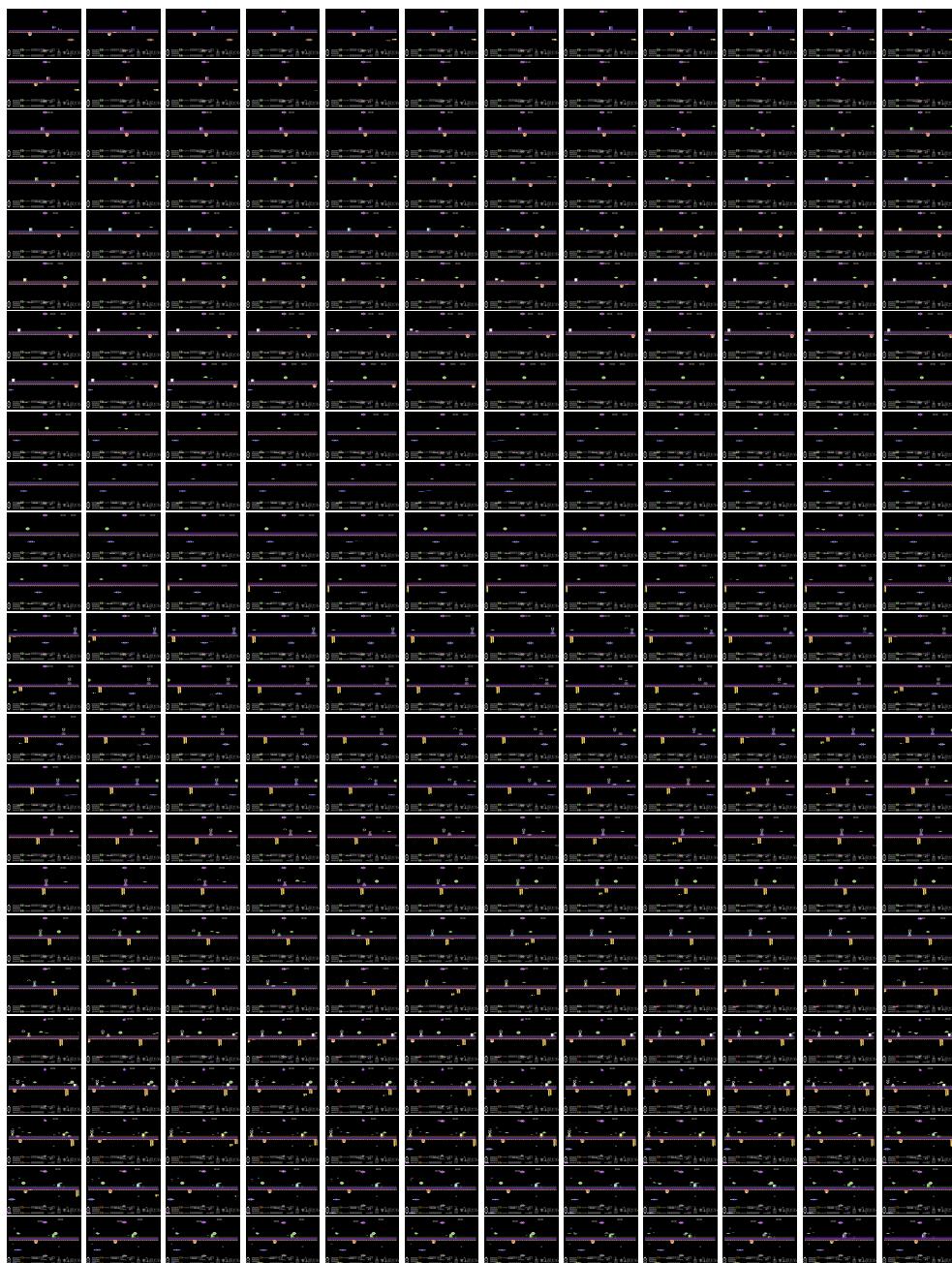


Figure 6.2: The screens displayed in a single second of game time.

cialized hardware, and preparing the position of the enemy sprites on the lower planet. The only reason it is called more than once is because, like in the title screen routine, the starfield is being painted using a single sprite (Sprite 7). Each time it runs it can change the position of this sprite so that it appears at a new position as well as the old one on which the raster has already painted it. We covered the mechanics of this in detail when we dissected the title screen in 'The First 16 Milliseconds'.

6.1 Updating Enemy Sprite Positions

We can get a better sense of how the labour is divided between the two routines if we isolate the position of the raster when the position of the enemy sprites on each planet is updated.

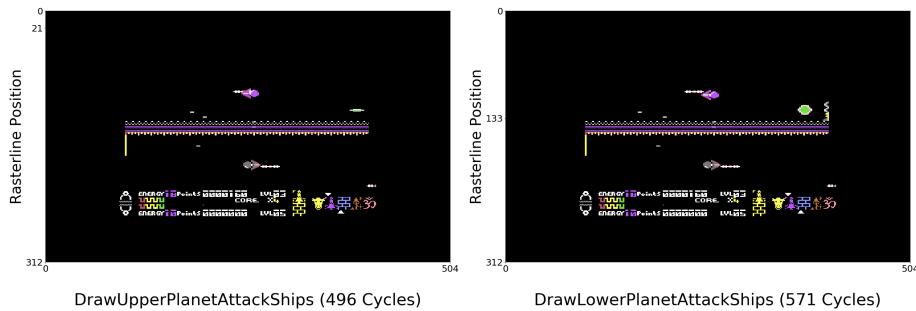


Figure 6.3: The rasterline position when the enemies on the upper and lower planets are updated.

The routines for updating the sprites on the upper and lower planets are identical. If we were writing this game in any other language than assembly we would just have one function and pass the different arrays for each planet in as parameters.

```

DrawUpperPlanetAttackShips
LDX #$0C
LDY #$06
UpperPlanetShipsLoop
LDA upperPlanetAttackShipsXPosArray,Y
STA $D000,X ;Sprite 0 X Pos

LDA attackShipsXPosArray - $01,Y
AND $D010 ;Sprites 0-7 MSB of X coordinate
STA currentMSBXPosOffset

LDA upperPlanetAttackShipsMSBXPosArray,Y
AND attackShipsMSBXPosOffsetArray,Y
ORA currentMSBXPosOffset
STA $D010 ;Sprites 0-7 MSB of X coordinate

; The X-Pos of sprites is fiddly. The MSB manages
; which side of the 512 possible x positions they
; are on.
LDA upperPlanetAttackShipsYPosArray,Y
STA $D001,X ;Sprite 0 Y Pos
STX tempVarStorage

LDX upperPlanetAttackShipsColorArray,Y
LDA colorsForAttackShips,X
STA $D027,Y ;Sprite 0 Color

LDA upperPlanetAttackShipsSpriteValueArray,Y
STA Sprite0Ptr,Y
LDX tempVarStorage

DEX
DEX
DEY
BNE UpperPlanetShipsLoop
RTS

```

```

DrawLowerPlanetAttackShips
LDX #$0C
LDY #$06
LowerPlanetShipsLoop
LDA lowerPlanetAttackShipsXPosArray + $01,Y
STA $D000,X ;Sprite 0 X Pos

LDA attackShipsXPosArray - $01,Y
AND $D010 ;Sprites 0-7 MSB of X coordinate
STA currentMSBXPosOffset

; The X-Pos of sprites is fiddly. The MSB manages
; which side of the 512 possible x positions they
; are on.
LDA lowerPlanetGilbyBulletMSBXPosValue,Y
AND attackShipsMSBXPosOffsetArray,Y
ORA currentMSBXPosOffset
STA $D010 ;Sprites 0-7 MSB of X coordinate

LDA lowerPlanetAttackShipsYPosArray,Y
STA $D001,X ;Sprite 0 Y Pos
STX tempVarStorage

LDX lowerPlanetAttackShipsColorArray,Y
LDA colorsForAttackShips,X
STA $D027,Y ;Sprite 0 Color

LDA lowerPlanetAttackShipsSpriteValueArray,Y
STA Sprite0Ptr,Y
LDX tempVarStorage

DEX
DEX
DEY
BNE LowerPlanetShipsLoop
RTS

```

6.2 Scrolling the Planets

We want scrolling to be smooth and fast. Moving swiftly or slowly across the planet surface depending on how much acceleration we apply is the fundamental dynamic of the game so will be important to get right.

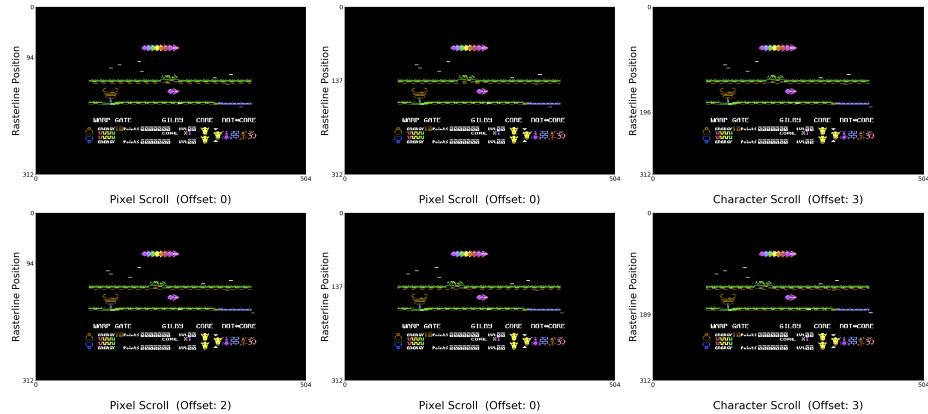
Exerting some fine-grained control on the scrolling requires us to balance two fundamental operations: a pixel-by-pixel scrolling mechanism that preserves the smoothness of movement we want at slower speeds, and a bigger, blunter implement that will accelerate us across larger sections of the planet while preserving an illusion of relative fluidity.

The first is available as a hardware implementation on the C64. The value stored in the last three bits of \$D016 allows us to specify a pixel offset for the planet graphics between 0 and 7, effectively shifting the planet left or right by one or more pixels.

The second is up to us. We need to keep an eye on the speed of our gilby and decide if we should shift the landscape by one or more full characters.

Here are both of these tactics in operation during two paints of the screen while we're warping into the Sheep planet at the start of a new game. Each row represents a single

pass of the raster. As you can see we adjust the pixel position using \$D016 twice on each pass and adjust the character position once.



Since we're moving fairly fast, we're updating the character position by 3 characters on each occasion (notice how much the bush moves). At the same time we're applying a pixel movement to preserve the impression of smoothness.

6.2.1 Pixel Movement

If we look at the code that looks after the pixel-grained movement in `AnimateStarFieldAndScrollPlanets` we can see it is using a variable called `planetScrollSpeed` to control the amount of offset to apply:

```
; Scroll the planet
LDA $D016      ;VIC Control Register 2
AND #$F0
ORA planetScrollSpeed
ORA #$10
STA $D016      ;VIC Control Register 2
```

This parameter is always kept to a value between 0 and 7, for example here in `DrawPlanetScroll` where it gets clamped to the last 3 bits (Bits 0 to 2) by an AND operation:

```
LDA planetScrollSpeed
AND #$07
STA planetScrollSpeed
```

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
\$FC	1	1	1	1	1	1	0	0
\$07	0	0	0	0	0	1	1	1
Result:	\$04	0	0	0	0	1	0	0

AND'ing a notional value of \$FC with \$07 gives \$04.

As you might guess, planetScrollSpeed is controlled by the speed of the gilby itself:

```
LDA planetScrollSpeed
CLC
ADC currentGilbySpeed
STA planetScrollSpeed
```

The more we push on the joystick left or right the greater currentGilbySpeed becomes. The greater currentGilbySpeed becomes, the more we add to planetScrollSpeed. The only thing we have to be careful about when using this simple mechanic is to ensure that when we update \$D016 with planetScrollSpeed we are only updating the lower 3 bits - writing to the rest of them will break things as they are not concerned with scrolling at all. This is why, when we first retrieve \$D016 to the A register we mask out the first four bits:

```
; Scroll the planet
LDA $D016      ;VIC Control Register 2
AND #$F0
```

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
\$DE	1	1	0	1	1	1	1	0
\$F0	1	1	1	1	0	0	0	0
Result:	\$D0	1	1	0	1	0	0	0

AND'ing a notional value of \$DE with \$F0 gives \$D0 preserving the first 4 bits (D).

ORA'ing a notional value for planetScrollSpeed of \$06 with \$D0 gives \$D6, adding our planetScrollSpeed into \$D0 without disturbing what was already there:

```
ORA planetScrollSpeed
```

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
\$D0	1	1	0	1	0	0	0	0
\$06	0	0	0	0	0	1	1	0
Result:	\$D6	1	1	0	1	0	1	1

Finally, ORA #\$10 ensures that the multi-color mode bit in \$D016 is set:

```
ORA #$10
STA $D016 ;VIC Control Register 2
```

So in a nutshell we're not being too fussy about what value we select for the pixel-perfect offset. We just take the overall scroll speed and clamp it to a value between 0 and 7. This is 'good enough' in practice - it ensures we're avoiding the appearance of scrolling purely character-wise by guaranteeing we're nearly always displaying the surface offset by some small number of pixels.

6.2.2 Character Movement

As a quick reminder from our chapter on generating the surfaces of the planets ('Making Planets for Nigel'), the data we came up with for the ever-so-randomly-generated planet was stored between \$8000 and \$8FFF.

```
; Clear down the planet surface data from $8000 to $8FFF.
; There are 4 layers:
; Top Layer:    $8000 to $83FF - 256 bytes
; Second Layer: $8400 to $87FF - 256 bytes
; Third Layer:   $8800 to $8BFF - 256 bytes
; Bottom Layer: $8C00 to $8FFF - 256 bytes
LDY #$00
ClearPlanetHiPtrs
    ; $60 is an empty character and gets written to the entire
    ; range from $8000 to $8FFF.
    LDA #$60
ClearPlanetLoPtrs
    STA (planetSurfaceDataPtrLo),Y
    DEY
    BNE ClearPlanetLoPtrs
    INC planetSurfaceDataPtrHi
    LDA planetSurfaceDataPtrHi
    CMP (#>planetSurfaceData) + $10
    BNE ClearPlanetHiPtrs
```

Listing 6.2: The surface data is stored from \$8000 to \$8FFF. This code overwrites it all with the value \$60 which is an empty bitmap.

As you can see in the code comment above the planet surface itself is 256 bytes long and we store 4 layers of 256 bytes each. The bottom layer is the surface, the other three are used for filling in the structures that dot the planet's surface.

When we initialize the game we use a bunch of pointers to store the position of each of these layers:

```
; The planet data starts at $8000. Each planet
; has 4 lines or layers.
LDA #>planetOneTopLayer
STA planetTextureTopLayerPtrHi
LDA #>planetOneSecondFromTopLayer
STA planetTextureSecondFromTopLayerPtrHi
LDA #>planetOneSecondFromBottomLayer
STA planetTextureSecondFromBottomLayerPtrHi
LDA #>planetOneBottomLayer
STA planetTextureBottomLayerPtrHi
```

Every time we scroll by one ore more characters along the planet surface updating what we see on the screen will just be a simple question of calling a routine called DrawPlanetSurfaces to write wherever in each layer the current pointer is pointing to:

```
DrawPlanetSurfaces
.....
;Draw the upper and lower planets. The lower
; planet is a mirror image of the top.
b71E6 LDX #$27
b71E8 LDA (planetTextureTopLayerPtr),Y
STA SCREEN_RAM + LINE7_COLO,Y
ORA #$CO
STA SCREEN_RAM + LINE15_COLO,X
LDA (planetTextureSecondFromTopLayerPtr),Y
STA SCREEN_RAM + LINE8_COLO,Y
ORA #$CO
STA SCREEN_RAM + LINE14_COLO,X
LDA (planetTextureSecondFromBottomLayerPtr),Y
STA SCREEN_RAM + LINE9_COLO,Y
ORA #$CO
STA SCREEN_RAM + LINE13_COLO,X
LDA (planetTextureBottomLayerPtr),Y
STA SCREEN_RAM + LINE10_COLO,Y
ORA #$CO
STA SCREEN_RAM + LINE12_COLO,X
INY
DEX
CPY #$28
BNE b71E8
RTS
```

So all we have to do as we scroll along the planet is adjust the position in RAM between \$8000-\$83FF that planetTextureTopLayerPtr is pointing to (and the same for the other layers) and we will effect the illusion of movement across the surface of the planet.

This means that our job is a simple one: how many characters should we move along the planet?

This decision happens in the `ScrollPlanets` routine, inside the `PerformMainGameUpdate` loop. As we have seen earlier, this routine is called around the time the raster reaches just past the halfway point down the screen:



Figure 6.4: The rasterline position at 196 when `ScrollPlanets` is called.

Just like with calculating the more fine-grained pixel offset we used `planetScrollSpeed` to determine how much to move by. But whereas the pixel movement used the lower 3 bits of `planetScrollSpeed` to come up with a value between 0 and 7 to adjust the pixel scroll by, we will here instead use the 3 bits before that.

The reason for doing that is straightforward: if those upper 3 bits are set the number in `planetScrollSpeed` must be fairly large and therefore enough to warrant scrolling an entire character or even more.

In a situation where we're moving to the right, this is the logic that figures out how many characters to move and updates `planetTextureTopLayerPtr` with the updated position:

```

ScrollPlanetRight
    LDA planetScrollSpeed
    EOR #$FF
    CLC
    AND #$F8
    ROR
    ROR
    ROR
    STA tempHiPtr1
    INC tempHiPtr1
    LDA planetTextureTopLayerPtr
    CLC
    ADC tempHiPtr1
    STA planetTextureTopLayerPtr
  
```

This is more complicated than we actually had reason to expect. Surely if the value in `planetScrollSpeed` is greater than 7 we could just shift the bits over there and use that instead? For example with a notional value of \$1F in `planetScrollSpeed` if we just did the following:

```
ScrollPlanetRight
    LDA planetScrollSpeed
    AND #$F8
    ROR
    ROR
    ROR
```

We would get a value of \$03 for our number of characters to move by. THis is because AND'ing \$1F and \$F8 gives us \$18:

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
\$1F	0	0	0	1	1	1	1	1
\$F8	1	1	1	1	1	0	0	0
Result: \$18	0	0	0	1	1	0	0	0

And then using ROR to shift the bits to the right three times results in \$03:

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
\$18	0	0	0	1	1	0	0	0
Result: \$03	0	0	0	0	0	0	1	1

Figure 6.5: ROR performed three times shifts everything to the right by three bits.

But instead of this we're doing an exclusive-or with the value in `planetScrollSpeed` first:

```
ScrollPlanetRight
    LDA planetScrollSpeed
    EOR #$FF
```

The reason we have to do this is because the value in `planetScrollSpeed` isn't what we might have assumed: a linear value between 0 and 40 for example that goes up and down a sliding scale depending on how fast we're going. Instead, it's something slightly different. It's fed by the value in `currentGilbySpeed` that reflects the current velocity of the gilby and this always starts out at a value of \$EA:

```
setUpGilbySprite
    LDA #GILBY_AIRBORNE_RIGHT
    STA currentGilbySprite
```

```
...
LDA #$EA
STA currentGilbySpeed
...
RTS
```

There's a trick at work here. The gilby can move left or right and `currentGilbySpeed` is being used to store both the speed **and** direction of the gilby. When the value is between 00 and 80 the gilby is moving to the left and the value reflects its relative velocity. When the value is between 80 and FF the gilby is moving to the right and the value reflects its relative velocity.

When Moving Left	Stationary	When Moving Right
\$04 03 02 01 00		FF FE FD FC

Figure 6.6: Value of `currentGilbySpeed` when moving left and right.

Sure enough, when we look at the calculation using for the character scroll offset when moving left, it's what we originally envisaged:

```
ScrollPlanetLeft
    LDA planetScrollSpeed
    CLC
    ADC currentGilbySpeed
    STA planetScrollSpeed
    ...
b72CF   CLC
        ROR
        ROR
        ROR
    STA tempHiPtr1

    LDA planetTextureTopLayerPtr
    SEC
    SBC tempHiPtr1
```

So the extra steps we see in `ScrollPlanetRight` are to handle the fact that the speed is going to be some value between \$80 and \$FF. The exclusive-or (`EOR`) has the effect of reversing the value in `planetScrollSpeed` and transforming it into a number between 0 and 16 that we can then clamp to a number between 0 and 7.

```
ScrollPlanetRight
    LDA planetScrollSpeed
    EOR #$FF
    CLC
    AND #$F8
    ROR
    ROR
```

```
ROR
STA tempHiPtr1
INC tempHiPtr1
LDA planetTextureTopLayerPtr
CLC
ADC tempHiPtr1
STA planetTextureTopLayerPtr
```

6.3 Jumping Up and Down

```
; -----
; PerformMainGameUpdate
; -----
PerformMainGameUpdate
...
JSR CalculateGilbyVerticalPositionEarthBound
JSR CalculateGilbyVerticalPositionAirborne
...
```

Listing 6.3: The routines responsible for updating the Gilby's vertical position.

When the gilby jumps on the surface of the planet it exhibits a smooth and pleasing acceleration in ascent and descent that eloquently suggests the gravity of the planet. This is achieved with a remarkably simple mechanism. Rather than define distinct behaviours for the journey upwards and the journey back, we use a single continuous movement based on incrementing the gilby's vertical position with an offset that 'cycles around' and transitions naturally from ascent to descent.

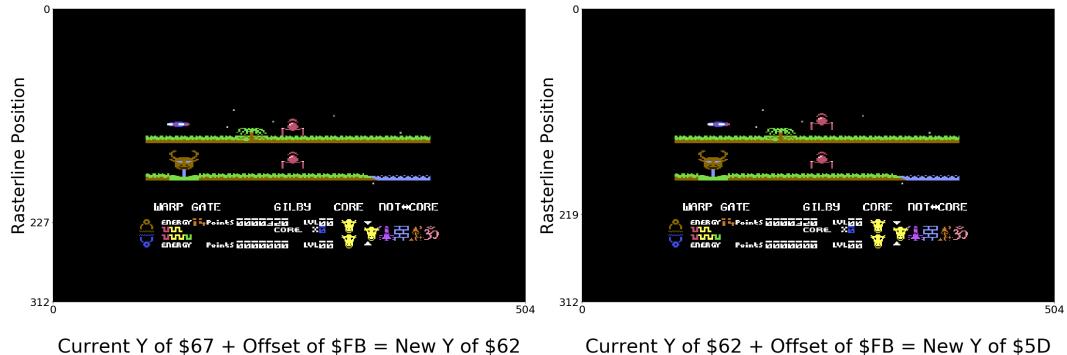
Achieving the complementary movements of ascent and descent with the same operation depends on a simple property of byte values when we increment them. If we keep adding to a value and it eventually reaches the maximum value of 255 (i.e. \$FF) that can be stored in a single byte, the next time we add 1 to it it will cycle around to \$00. So if we add \$FB, for example, to \$09 we get \$05.

So let's take our starting vertical position (our Y co-ordinate) on the planet to be \$6D. This is the Y-coordinate on the screen that coincides with the surface of the planet. If we're planning to move upwards we might think that the natural number to subtract from this position to move us up the screen is something like 3 or 4. And then when we're moving down the screen later on we would increment this position value by 3 or 4, or some smaller number depending on how quickly we want to move. This intuition is correct, up to a point, but it would mean keeping track of which direction we're going and complicate our code.

Given the circular property of byte arithmetic we described above we can increment or

decrement our position by simply adding a carefully chosen value. If we add a number that will force the result to cycle around and come out less than the current Y value then we've successfully subtracted from Y while adding to it! This is the essence of our trick.

If we choose our offset value (`gilbyLandingJumpingAnimationYPosOffset`) as \$FB and add it to our current Y co-ordinate of \$67 we get a result of \$62: we move the gilby up the screen by five pixels:



This is what the code achieving this effect looks like:

```

UpdateGilbyPosition
    CLC
    ADC gilbyLandingJumpingAnimationYPosOffset
    STA gilbyVerticalPositionUpperPlanet

    ; Check if we've hit the surface.
    AND #$F0
    CMP #YPOS_PLANET_SURFACE + $03
    BNE StorePositionAndReturn

    LDA #YPOS_PLANET_SURFACE
    STA gilbyVerticalPositionUpperPlanet

StorePositionAndReturn
    ; Update the position on screen.
    LDA gilbyVerticalPositionUpperPlanet
    STA $D001      ;Sprite 0 Y Pos
    RTS

```

Our choice of \$FB as an initial value for the offset is very deliberate. As we increment this offset it will eventually reach \$FF. Each time we do so the resulting degree in movement it creates gets smaller. So the gilby will appear to move quickly at first but slow down the further it gets from the planet:

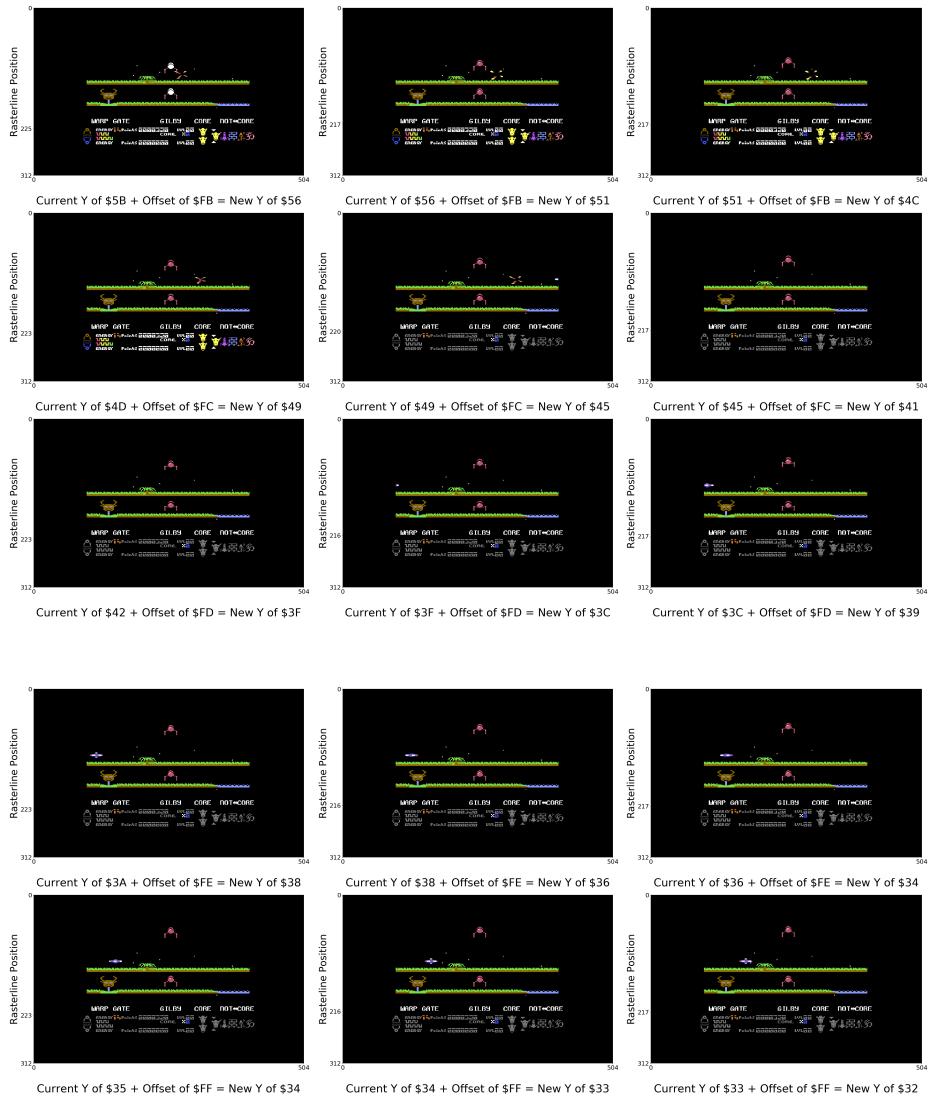


Figure 6.7: Each increment in the offset, performed every three movements, results in a deceleration effect.

When our offset value `gilbyLandingJumpingAnimationYPosOffset` reaches `$FF` it is time for modular arithmetic to kick in the next time we increment it: the movement-increment it gives us is `$00` and for the next three 'movements' the gilby appears to hang in the sky at Y co-ordinate `$33`:



Now that `gilbyLandingJumpingAnimationYPosOffset` is \$00 adding it to our current Y position will no longer have the effect of cycling around and arriving a smaller value than the current Y position. Instead it will increment the Y position and move the gilby down the screen again. Just as when we were moving upwards the resulting movement increases in degree as we get closer to the land, creating an acceleration effect due to gravity.

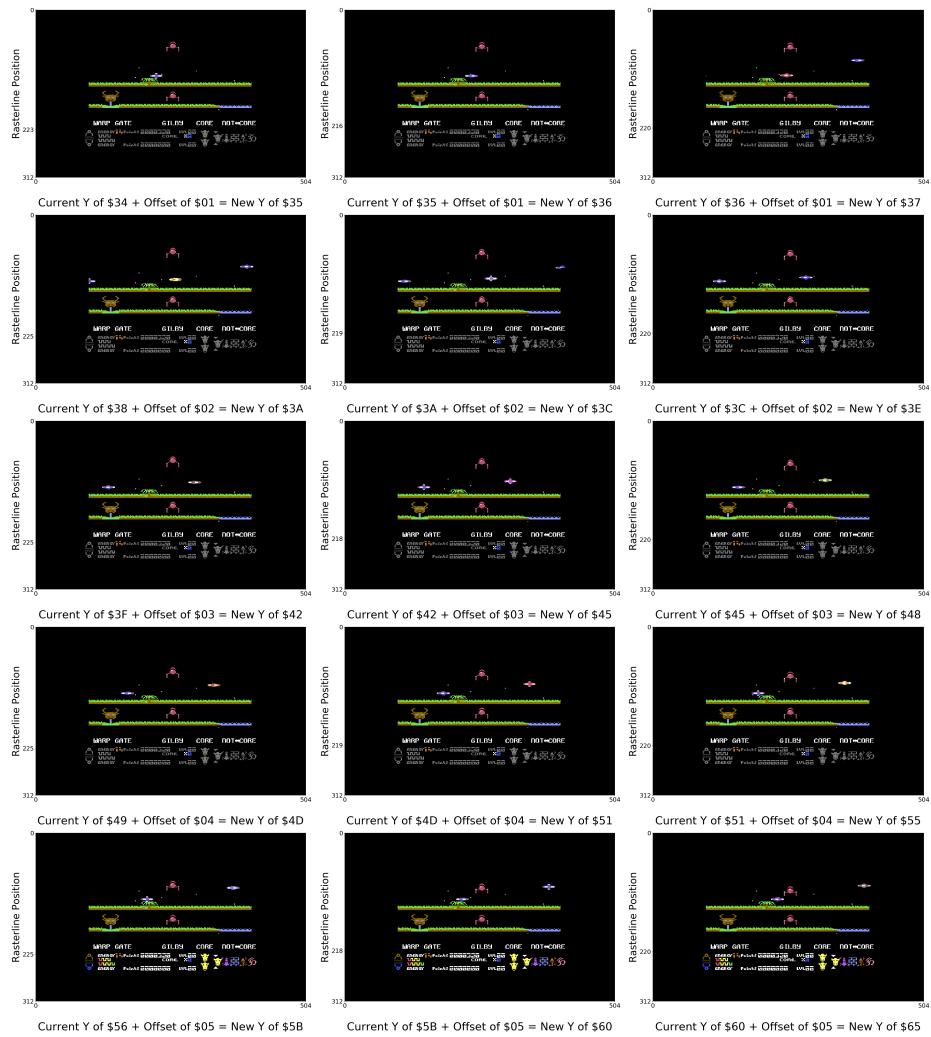


Figure 6.8: Each increment in the offset, performed every three movements, results in an acceleration effect.

With a simple, consistent addition operation each time it comes to position the gilby we've managed to achieve a jumping and landing effect with a neat gravity effect built in!

6.4 Sound Effects

```
; -----
; PerformMainGameUpdate
; -----
PerformMainGameUpdate
    ...
    JSR PlaySoundEffects
    ...
```

Iridis Alpha has a rich weave of sound effects during play. The game is as much an assault on the ears as it is on the eyes. In order to create the impression that there are multiple sounds going on at once so it is necessary to... have multiple sounds going at once. The fancy word for this is 'multiplexing'. If we want the player to experience the world of Iridis Alpha as one in which things are not happening sequentially but simultaneously we will need the sounds they experience to interleave with each other, rather than just playing whatever the 'current' sound is then waiting for it to finish before we play the next. Playing two sounds at once is more than enough multiplexing to be going on with, so that is what we do.

The way to achieve this is to come up with a data structure for sound effects that segments the full sound effect into multiple frames and to process a few frames of each data structure every time `PlaySoundEffects` is called by the raster interrupt. We also need `PlaySoundEffects` to behave like a 'state machine'. This means that when it is called a little later it can pick up from where it left off on each data structure, recognize when it has finished playing the frames for that interrupt and also when it has finished playing the full effect. This will allow us to manage two different sound effects concurrently. Before we look at how this multiplexing is achieved lets look at how a single sound effect is managed in general.

6.4.1 The Sound Effect Data Structure

As ever, the key to managing complexity isn't the cleverness of our code but the simplicity of the data structure we choose. Iridis Alpha solves this with a remarkably compact solution where each frame in the sound effect is represented by a meagre 5 bytes and the whole sound effect is encoded by a sequence of these 5-byte records. In order to unpack what this looks like in practice, let's look at the iconic effect played when we enter a new planet. The data structure here is just one part of the level entry sound effect, the duller part. It plays a dull bass note that fades away. (We'll examine the more interesting leg of the overall sound effect, the one that it's multiplexed with, right after this. But this is a relatively nice and simple one for us to get an idea of how the data structure works.)

```

PLAY_SOUND = $00
PLAY_LOOP = $05
LINK = $80
VOICE2_HI = $08
VOICE2_CTRL = $0B
VOICE3_HI = $0F
VOICE2_ATK_DEC = $0C
VOICE2_SUS_REL = $0D
VOICE3_CTRL = $12
VOICE3_ATK_DEC = $13
VOICE3_SUS_REL = $14
planetWarpSoundEffect .BYTE $00,PLAY_SOUND,$0F,VOICE2_ATK_DEC,$00
                     .BYTE $00,PLAY_SOUND,$0F,VOICE3_ATK_DEC,$00
                     .BYTE $00,PLAY_SOUND,$0F,VOLUME,$00
                     .BYTE $00,PLAY_SOUND,$00,VOICE2_SUS_REL,$00
                     .BYTE $00,PLAY_SOUND,$00,VOICE3_SUS_REL,$00
                     .BYTE $00,PLAY_SOUND,$03,VOICE2_HI,$00
                     .BYTE $00,PLAY_SOUND,$03,VOICE3_HI,$00
                     .BYTE $00,PLAY_SOUND,$21,VOICE2_CTRL,$00
                     .BYTE $00,PLAY_SOUND,$08,VOICE3_LO,$00
                     .BYTE $00,PLAY_SOUND,$00,VOICE2_LO,$00
                     .BYTE $00,PLAY_SOUND,$21,VOICE3_CTRL,$01
                     .BYTE $18,PLAY_LOOP,$00,<pwLoop,>pwLoop
                     .BYTE $00,PLAY_SOUND,$20,VOICE2_CTRL,$00
                     .BYTE $00,PLAY_SOUND,$20,VOICE3_CTRL,$00
                     .BYTE $00,LINK,<setVolToMax,>setVolToMax,$00
pwLoop

```

This data is like a piano roll fed into PlaySoundEffects. In simplified terms each line is a 'frame' containing a single note for it to play. For the lines with PLAY_SOUND it really is that simple. Byte 3 in each of those lines is the 'note' to play, and Byte 4 (e.g. VOICE2_ATK_DEC) is the 'key on the piano' to play it on. PlaySoundEffects will keep playing each line in the roll until it hits one with a value in Byte 5 that is not \$00. In this case that happens when it hits the line that starts with pwLoop, notice the \$01 at the very end.

Let's be more concrete about what's happening when each PLAY_SOUND record is processed as it's quite simple. The value in Byte 3 is written to the position in the SID register given by Byte 4. The SID register is an array of bytes in the C64's ROM that controls the production of sound and they live between addresses \$D400 and \$D418. So in the first record in planetWarpSoundEffect we are going to write \$0F to the address \$D40C in the SID register.

What does that do you might ask? Does it play a note? Well, no, as it happens this address in the SID register is responsible for controlling how much a sound rises or falls. As you can see in the visualization of the effect below it decays away. This byte we're setting is responsible for setting that on 'Voice 2' in the sound chip.

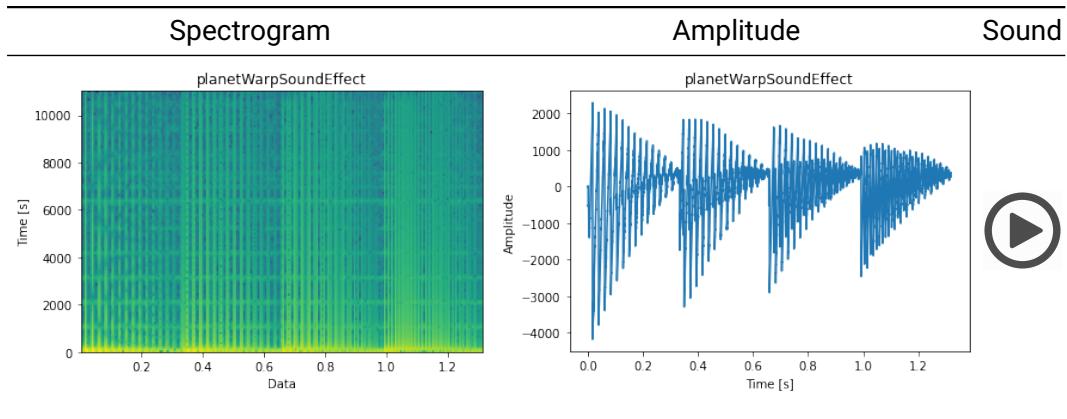


Figure 6.9

The next record does the same for 'Voice 3'. The one after that sets the volume (\$D418) to its maximum value of 15 (\$0F).

```
.BYTE $00, PLAY_SOUND, $0F, VOICE3_ATK_DEC, $00
.BYTE $00, PLAY_SOUND, $0F, VOLUME, $00
```

The next two records ensure Voices 2 and 3 do not sustain their sound by writing \$00 to both. We just want them to die away quickly after all.

```
.BYTE $00, PLAY_SOUND, $00, VOICE2_SUS_REL, $00
.BYTE $00, PLAY_SOUND, $00, VOICE3_SUS_REL, $00
```

Finally we can start playing some notes.

```
.BYTE $00, PLAY_SOUND, $03, VOICE2_HI, $00
.BYTE $00, PLAY_SOUND, $03, VOICE3_HI, $00
```

What we are doing here is writing a frequency value to SID registers for Voice 2 and Voice 3 that will actually make some noise. We then tell the sound device to start playing the note by setting its 'gate' to 1. The 'gate' is the 'least significant bit', i.e. the 1 in the \$21 given in Byte 3:

```
.BYTE $00, PLAY_SOUND, $21, VOICE2_CTRL, $00
```

After adjusting the sound again:

```
.BYTE $00, PLAY_SOUND, $08, VOICE3_LO, $00
```

```
.BYTE $00,PLAY_SOUND,$00,VOICE2_LO,$00
```

We now encounter the use of a new record type called PLAY_LOOP.

```
VOLUME = $18
pwLoop      .BYTE $00,PLAY_SOUND,$21,VOICE3_CTRL,$01
            .BYTE VOLUME,PLAY_LOOP,$01,<pwLoop,>pwLoop
```

What this this particular instance of a PLAY_LOOP record does is gradually lower the volume until it reaches zero.

More generally the way PLAY_LOOP records are processed is as follows. First of all we write to the offset in the SID register given by Byte 1 (in this case the register responsible for volume). The value we write is also derived from Byte 1, but by using it as an index into an array called soundEffectBuffer:

```
soundEffectBuffer .BYTE $00,$94,$00,$00,$11,$0F,$00,$00
                  .BYTE $03,$00,$00,$21,$0F,$00,$08,$03
                  .BYTE $00,$00,$21,$0F,$00,$00,$00,$00
                  .BYTE $02,$00,$00,$00,$00,$00,$00,$00
```

If we look at our record again..

```
.BYTE VOLUME,PLAY_LOOP,$01,<pwLoop,>pwLoop
```

we can see Byte 1 is VOLUME, which is an alias for the value \$18. Using \$18 as an index into soundEffectBuffer retrieves the fourth value on the third line above, \$0F. We then subtract the value in Byte 3 (\$01) and store the result back in the soundEffectBuffer. If it's not zero yet, we then treat the record pointed to by Bytes 4 and 5 as the next one to play. In this case that's given as pwLoop. So in fact we go round in a loop between the two records until \$0F eventually becomes zero. Here's the code that handles this:

```
LDX soundEffectDataStructure_Byte1
LDA soundEffectBuffer,X
SEC
SBC soundEffectDataStructure_Byte3

StorePointersAndReturnIfZero
STA soundEffectBuffer,X
STA $D400,X ;Voice 1: Frequency Control - Low-Byte
BEQ JumpToGetNextRecordInSoundEffect
LDA soundEffectDataStructure_Byte4
LDX indexToPrimaryOrSecondarySoundEffectPtr
STA primarySoundEffectLoPtr,X
LDA soundEffectDataStructure_Byte5
STA primarySoundEffectHiPtr,X
RTS
```

```
JumpToGetNextRecordInSoundEffect
    JMP GetNextRecordInSoundEffect
```

You can see that once we reach zero BEQ JumpToGetNextRecordInSoundEffect becomes true and we end up calling GetNextRecordInSoundEffect which actually lets us move on to the next record in our data structure (the two we've just looped through are given at the start to make it easier to see where picked up from):

```
.BYTE $00,PLAY_SOUND,$20,VOICE2_CTRL,$00
.BYTE $00,PLAY_SOUND,$20,VOICE3_CTRL,$00
```

You can see we write \$20 to VOICE2_CTRL and VOICE3_CTRL. This stops the noise we were making until now. It switches off the sound by setting the 'gate' we set to 1 earlier on back to 0.

The final record we play is a simple 'jump' record. Its processed by simply jumping to the address given in Bytes 4 and 5 and playing whatever is there:

```
.BYTE $00,LINK,<setVolToMax,>setVolToMax,$00
```

In this case it is pointing us to the data structure at setVolToMax. This simply runs in a loop setting the volume back to \$0F, i.e. its maximum value of 15. Notice the way the first record below writes \$0F (Byte 3) to the VOLUME register. The second one, just tells it loop back and do the same thing again. This will keep running until a new sound effect is selected.

```
setVolToMax           .BYTE $00,PLAY_SOUND,$0F,VOLUME,$01
                      .BYTE $00,LINK,<setVolToMax,>setVolToMax,$00
```

We can get a picture of how each record affects the relevant registers in the SID interface by looking at the table below. This is full history of each record that's processed in the planetWarpSoundEffect data structure by PlaySoundEffects in temporal order. Notice the way the column steadily decreases as we go through the PLAY_LOOP loop towards the end.

Data Frame	Voice 2				Voice 3				Volume
	Sound	Gate	Saw	Decay	Sound	Gate	Saw	Decay	
\$00,PLAY_SOUND,\$0F,VOICE3_ATK_DEC,\$00	0	0	0	15	0	0	0	0	15
\$00,PLAY_SOUND,\$0F,VOLUME,\$00	0	0	0	15	0	0	0	0	15
\$00,PLAY_SOUND,\$00,VOICE2_SUS_REL,\$00	0	0	0	15	0	0	0	0	15
\$00,PLAY_SOUND,\$00,VOICE3_SUS_REL,\$00	0	0	0	15	0	0	0	0	15
\$00,PLAY_SOUND,\$03,VOICE2_HI,\$00	768	0	0	15	0	0	0	0	15
\$00,PLAY_SOUND,\$03,VOICE3_HI,\$00	768	0	0	15	768	0	0	0	15
\$00,PLAY_SOUND,\$21,VOICE2_CTRL,\$00	768	1	1	15	768	0	0	0	15
\$00,PLAY_SOUND,\$08,VOICE3_LO,\$00	768	1	1	15	776	0	0	0	15
\$00,PLAY_SOUND,\$00,VOICE2_LO,\$00	768	1	1	15	776	0	0	0	15
\$00,PLAY_SOUND,\$21,VOICE3_CTRL,\$01	768	1	1	15	776	1	1	1	15
\$18,PLAY_LOOP,\$00,<pwLoop,>pwLoop	768	1	1	15	776	1	1	1	15
\$00,PLAY_SOUND,\$21,VOICE3_CTRL,\$01	768	1	1	15	776	1	1	1	15
\$18,PLAY_LOOP,\$00,<pwLoop,>pwLoop	768	1	1	15	776	1	1	1	15
\$00,PLAY_SOUND,\$21,VOICE3_CTRL,\$01	768	1	1	15	776	1	1	1	15
\$18,PLAY_LOOP,\$00,<pwLoop,>pwLoop	768	1	1	15	776	1	1	1	15
\$00,PLAY_SOUND,\$21,VOICE3_CTRL,\$01	768	1	1	15	776	1	1	1	15
\$18,PLAY_LOOP,\$00,<pwLoop,>pwLoop	768	1	1	15	776	1	1	1	15
\$00,PLAY_SOUND,\$21,VOICE3_CTRL,\$01	768	1	1	15	776	1	1	1	15
\$18,PLAY_LOOP,\$00,<pwLoop,>pwLoop	768	1	1	15	776	1	1	1	15
\$00,PLAY_SOUND,\$21,VOICE3_CTRL,\$01	768	1	1	15	776	1	1	1	15
\$18,PLAY_LOOP,\$00,<pwLoop,>pwLoop	768	1	1	15	776	1	1	1	15
\$00,PLAY_SOUND,\$21,VOICE3_CTRL,\$01	768	1	1	15	776	1	1	1	15
\$18,PLAY_LOOP,\$00,<pwLoop,>pwLoop	768	1	1	15	776	1	1	1	15
\$00,PLAY_SOUND,\$21,VOICE3_CTRL,\$01	768	0	1	15	776	1	1	1	15
\$18,PLAY_LOOP,\$00,<pwLoop,>pwLoop	768	0	1	15	776	0	1	1	15

6.4.2 Multiplexing

While playing the fairly dull component we cover above, Iridis Alpha plays a second sequence simultaneously that constitutes the one a player will know and recognize.

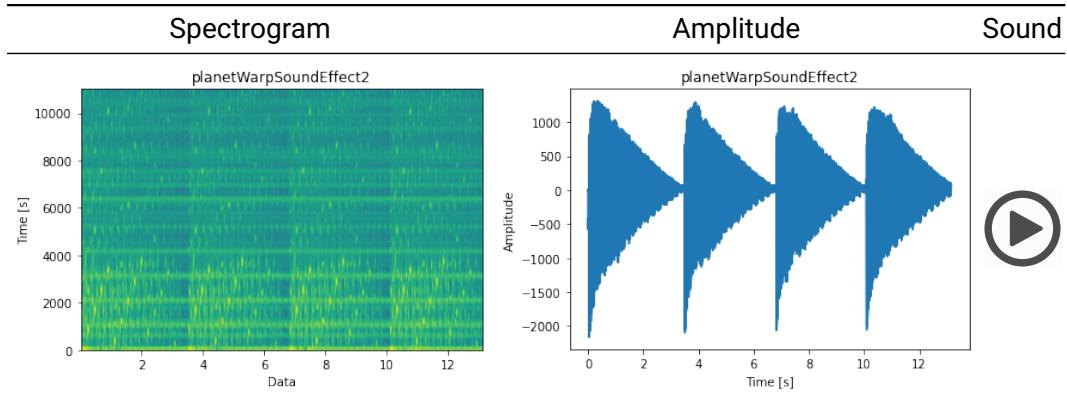


Figure 6.10

The way this concurrency is managed is by storing the addresses for each effect separately. In the case of the sequence where the player is entering a new planet this is done in `PerformPlanetWarp`, the routine responsible for rendering the warp sequence as a whole:

```
PerformPlanetWarp
  ...
  LDA #<planetWarpSoundEffect
  STA secondarySoundEffectLoPtr
  LDA #>planetWarpSoundEffect
  STA secondarySoundEffectHiPtr
  LDA #<planetWarpSoundEffect2
  STA primarySoundEffectLoPtr
  LDA #>planetWarpSoundEffect2
  STA primarySoundEffectHiPtr
```

With the primary and secondary sound effects stored in `primarySoundEffectLo/HIPtr` and `secondarySoundEffectLo/HIPtr` respectively, all the `PlaySoundEffects` routine has to do it ensures that it processes each of the effects every time it is visited by the raster interrupt. Here we see it take in the 'primary' sound effect and process it in `PlayCurrentSoundEffect`:

```
PlaySoundEffects
  LDA #$00
  STA indexToPrimaryOrSecondarySoundEffectPtr
  LDA soundEffectInProgress
  BEQ DontDecrementSoundEffectProgressCounter
  DEC soundEffectInProgress
DontDecrementSoundEffectProgressCounter
  LDA primarySoundEffectLoPtr
```

```

STA currentSoundEffectLoPtr
LDA primarySoundEffectHiPtr
STA currentSoundEffectHiPtr
JSR PlayCurrentSoundEffect

```

And here we see it then move on to processing the secondarySoundEffect. Notice that it falls through to PlayCurrentSoundEffect rather than having to call it directly with JSR:

```

LDA #$02
STA indexToPrimaryOrSecondarySoundEffectPtr
LDA secondarySoundEffectLoPtr
STA currentSoundEffectLoPtr
LDA secondarySoundEffectHiPtr
STA currentSoundEffectHiPtr
;Falls through and plays secondary sound effect.

;-----
; PlayCurrentSoundEffect
;-----
PlayCurrentSoundEffect
LDY #$00
; Read the 5-byte record into working storage.
FillSoundEffectDataStructureLoop
LDA (currentSoundEffectLoPtr),Y
STA soundEffectDataStructure,Y
INY
CPY #$05
BNE FillSoundEffectDataStructureLoop

```

The secondary sound effect for the planet warp sequence, which is the one with all the recognizable noises, looks like this:

```

.pwLoop
    .BYTE $00,PLAY_SOUND,$0F,VOICE1_ATK_DEC,$00
    .BYTE $00,PLAY_SOUND,$00,VOICE1_SUS_REL,$00
    .BYTE $00,PLAY_SOUND,$00,VOICE1_HI,$00
    .BYTE $00,PLAY_SOUND,$11,VOICE1_CTRL,$02
    .BYTE $01,INC_AND_PLAY_FROM_BUFFER,$64,VOICE1_HI,$01
    .BYTE $00,REPEAT_PREVIOUS,$08,$00,$00
    .BYTE $01,INC_AND_PLAY_FROM_BUFFER,$18,VOICE1_HI,$01
    .BYTE VOLUME,PLAY_LOOP,$01,<pwLoop,>pwLoop
    .BYTE $00,PLAY_SOUND,$10,VOICE1_CTRL,$00
    .BYTE $00,LINK,<setVolumeToMax,>setVolumeToMax,$00

```

Just like the primary sound effect we discussed this one contains a loop, achieving the same effect as before. There are a couple of new record types in there though, namely INC_AND_PLAY_FROM_BUFFER and REPEAT_PREVIOUS. The latter is probably self-explanatory, it repeats processing of the previous record for as many times as given by the value in Byte 3 which in this case is \$08 times.

INC_AND_PLAY_FROM_BUFFER does something more interesting, in the first case above it's being used to update the sound we play (i.e. write to register VOICE1_HI) by a regularly incrementing amounts of \$64 given in Byte 3. This value is picked from and updated in position 1 of the soundEffectBuffer, this position being given by Byte 1. The result is the 'iconic' bleeping sound that increases in frequency as we warp into the planet. Here we see the section in PlaySoundEffects that processes records of type INC_AND_PLAY_FROM_BUFFER, and the way it uses each byte in the record to effect the update and the write to the sound register:

```
; Increment the value in the buffer and play it.
LDX soundEffectDataStructure_Byte1
LDA soundEffectBuffer,X
CLC
ADC soundEffectDataStructure_Byte3
LDX soundEffectDataStructure_Byte4
STA soundEffectBuffer,X
STA $D400,X ;Voice 1: Frequency Control - Low-Byte
JMP GetNextRecordAndMaybePlayIt
```

The amount of looping the data structure demands is much greater than the effect we reviewed previously. We can get a sense of this from the truncated trace which shows a total of over 250 writes to the sound register and of which the table below is just a snapshot:

Record Type	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5
Play Sound	Unused	PLAY_SOUND (\$00)	Value to write to offset to \$D400 given by Byte 4.	Offset to \$D400 '01' indicates should play no more records. Anything else indicates the next record to write to.	'00' indicates the next record should be played immediately.
Increment and Play from Buffer	Address of byte to pick from soundEffectBuffer (\$01)	INC_AND_PLAY_FROM_BUFFER	Amount to increment picked byte by.	Offset to \$D400 '01' indicates should play no more records. Anything else indicates the next record to write to.	'00' indicates the next record should be played immediately.
Decrement and Play from Buffer	Address of byte to pick from soundEffectBuffer (\$02)	DEC_AND_PLAY_FROM_BUFFER	Amount to decrement picked byte by.	Offset to \$D400 '01' indicates should play no more records. Anything else indicates the next record to write to.	'00' indicates the next record should be played immediately.
Play Loop	Address of byte to pick from soundEffectBuffer (\$05) and offset to \$D400	PLAY_LOOP	Amount to decrement picked byte by.	Lo Ptr of next record.	Hi Ptr of next record.
Link to Record	Unused.	LINK (\$80)	Lo Ptr of next record.	Hi Ptr of next record.	'00' indicates the next record should be played immediately.
Repeat Previous Record	Unused.	REPEAT_PREVIOUS (\$81)	Number of times to play previous record	Unused.	'01' indicates should play no more records. Anything else indicates the next record should be stored and no more should be played for now.

Figure 6.11: The data structures for sound effects used by Iridis Alpha.

	freq1	gate1	saw1	dec1	sus1	vol
\$00,PLAY_SOUND,\$0F,VOICE1_ATK_DEC,\$00	0	0	0	15	0	15
\$00,PLAY_SOUND,\$00,VOICE1_SUS_REL,\$00	0	0	0	15	0	15
\$00,PLAY_SOUND,\$00,VOICE1_HI,\$00	0	0	0	15	0	15
\$00,PLAY_SOUND,\$11,VOICE1_CTRL,\$02	0	1	0	15	0	15
\$01,INC_AND_PLAY_FROM_BUFFER,\$64,VOICE1_HI,\$01	25600	1	0	15	0	15
\$01,INC_AND_PLAY_FROM_BUFFER,\$64,VOICE1_HI,\$01	51200	1	0	15	0	15
\$01,INC_AND_PLAY_FROM_BUFFER,\$64,VOICE1_HI,\$01	11264	1	0	15	0	15
\$01,INC_AND_PLAY_FROM_BUFFER,\$64,VOICE1_HI,\$01	36864	1	0	15	0	15
\$01,INC_AND_PLAY_FROM_BUFFER,\$64,VOICE1_HI,\$01	62464	1	0	15	0	15
\$01,INC_AND_PLAY_FROM_BUFFER,\$64,VOICE1_HI,\$01	22528	1	0	15	0	15
\$01,INC_AND_PLAY_FROM_BUFFER,\$64,VOICE1_HI,\$01	48128	1	0	15	0	15
\$01,INC_AND_PLAY_FROM_BUFFER,\$64,VOICE1_HI,\$01	8192	1	0	15	0	15
\$01,INC_AND_PLAY_FROM_BUFFER,\$18,VOICE1_HI,\$01	14336	1	0	15	0	15
VOLUME,PLAY_LOOP,\$01,<f5D8D,>f5D8D	14336	1	0	15	0	14
\$00,PLAY_SOUND,\$11,VOICE1_CTRL,\$02	39936	1	0	15	0	14
\$01,INC_AND_PLAY_FROM_BUFFER,\$64,VOICE1_HI,\$01	0	1	0	15	0	14
\$01,INC_AND_PLAY_FROM_BUFFER,\$64,VOICE1_HI,\$01	25600	1	0	15	0	14
\$01,INC_AND_PLAY_FROM_BUFFER,\$64,VOICE1_HI,\$01	51200	1	0	15	0	14
\$01,INC_AND_PLAY_FROM_BUFFER,\$64,VOICE1_HI,\$01	11264	1	0	15	0	14
\$01,INC_AND_PLAY_FROM_BUFFER,\$64,VOICE1_HI,\$01	36864	1	0	15	0	14
\$01,INC_AND_PLAY_FROM_BUFFER,\$64,VOICE1_HI,\$01	62464	1	0	15	0	14
\$01,INC_AND_PLAY_FROM_BUFFER,\$64,VOICE1_HI,\$01	22528	1	0	15	0	14
\$01,INC_AND_PLAY_FROM_BUFFER,\$64,VOICE1_HI,\$01	28672	1	0	15	0	14
\$01,INC_AND_PLAY_FROM_BUFFER,\$18,VOICE1_HI,\$01	28672	1	0	15	0	13
VOLUME,PLAY_LOOP,\$01,<f5D8D,>f5D8D	54272	1	0	15	0	13
Repeats many times!
\$00,PLAY_SOUND,\$10,VOICE1_CTRL,\$00	0	1	0	15	0	14

A sample of the processing of records in the planetWarpSoundEffect2 data structure.

6.4.3 Unused Sound Effects

The PlaySoundEffects code has some leftover routines for record types that appear to have gone unused in the final game. These make slightly more complicated use of the soundEffectBuffer. All they do is mutate the data in the buffer using the 5-byte record. They don't actually play any sounds or make any writes to the SID register.

For example here is the logic applied to records with a 'type byte' in Byte 2 of (\$03):

```
TrySequenceByteValueOf3
    CMP #\$03
    BNE TrySequenceByteValueOf4
    LDX soundEffectDataStructure_Byt1
    LDY soundEffectDataStructure_Byt3
```

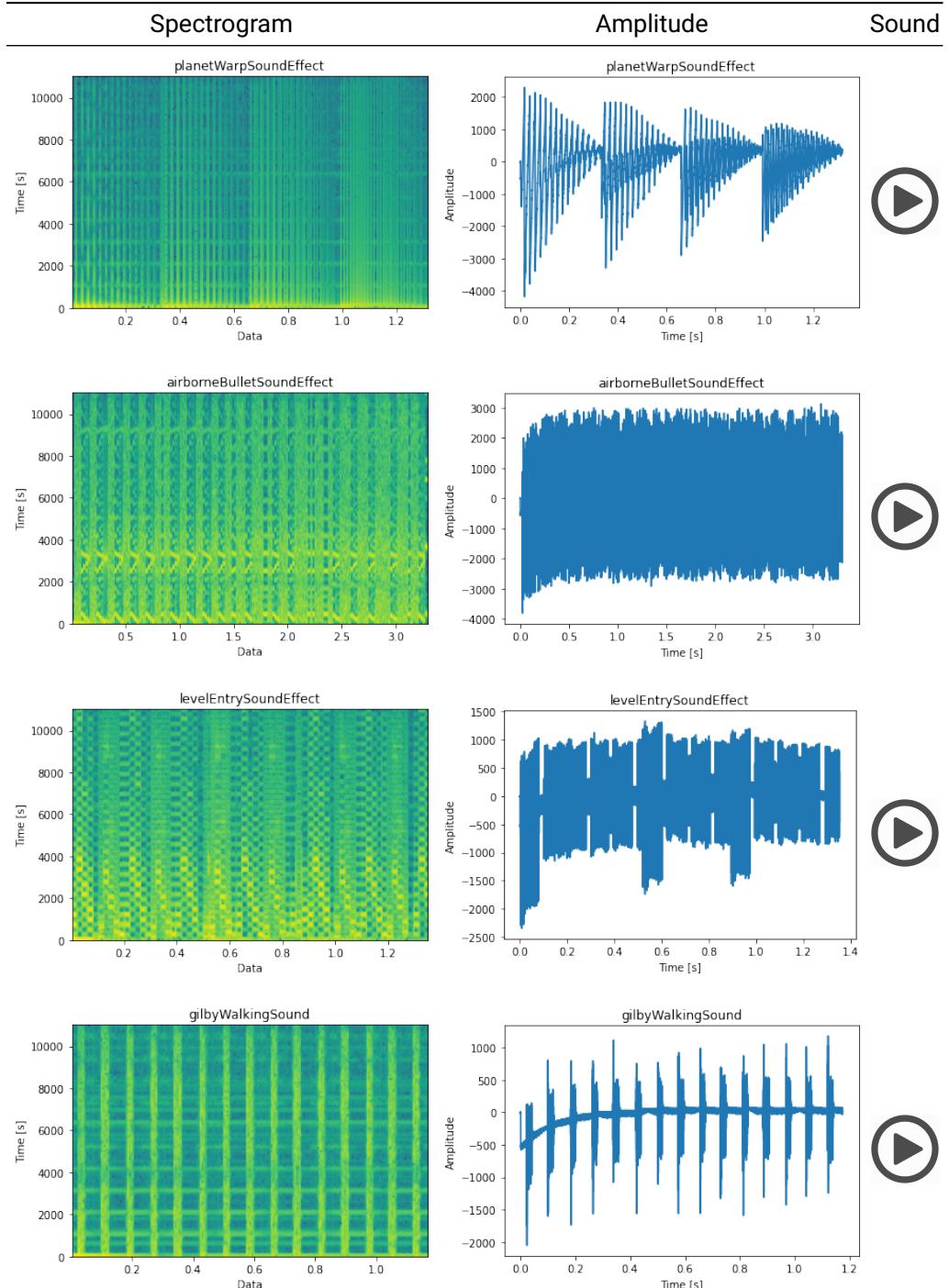
```
LDA soundEffectBuffer,X  
CLC  
ADC soundEffectBuffer,Y  
JMP GetNextRecordInSoundEffectLoop
```

This takes the value in Byte 1 as an offset into `soundEffectBuffer`, adds the value found at the offset in `soundEffectBuffer` given by Byte 3 and stores the result in the A accumulator. It then proceeds directly to reading the next record in the sound effect's data structure.

Logic for records with a type of \$04 does the same thing but subtracts rather than adds:

```
TrySequenceByteValueOf4  
  CMP #$04  
  BNE MaybeIsFadeOutLoop  
  LDX soundEffectDataStructure_Byte1  
  LDY soundEffectDataStructure_BYTE3  
  LDA soundEffectBuffer,X  
  SEC  
  SBC soundEffectBuffer,Y  
  JMP GetNextRecordInSoundEffectLoop
```

So the general idea for these two unused record types seems to have been that the `soundEffectBuffer` could be used to generate a sequence of sounds in some sort of procedural or even fractal manner, similar to the way in which the title music was generated. The experiment obviously didn't work as they ended up on the cutting room floor, with just this leftover code to indicate the attempt.



Enemies and their Discontents

ACONT

This is the bit that I knew would take me ages to write and get glitch free, and the bit that is absolutely necessary to the functioning of the game. The module ACONT is essentially an interpreter for my own 'wave language', allowing me to describe, exactly, an attack wave in about 50 bytes of data. The waves for the first part of IRIDIS are in good rollicking shoot-'em-up style, and there have to be plenty of them. There are five planets and each planet is to have twenty levels associated with it. It's impractical to write separate bits of code for each wave; even with 64K you can run outta memory pretty fast that way, and it's not really necessary coz a lot of stuff would be duplicated. Hence ACONT.

— Jeff Minter's Development Diary in Zzap Magazine^[?]

The bits and bytes that define the behaviour and appearance of wave after wave of Iridis Alpha's enemy formations - twenty across each of the five planets giving one hundred in all - takes up relatively little space given the sheer variety of adversaries the player faces.



Byte	Value	Description
Byte 0	\$06	Index into array for sprite color
Byte 1	FLYING_SAUCER1	First sprite value for the attack ship on the upper planet
Byte 2	FLYING_SAUCER3	Last sprite value for the attack ship on the upper planet
Byte 3	\$03	The animation frame rate for the attack ship.
Byte 4	FLYING_SAUCER1	First sprite value for the attack ship on the lower planet
Byte 5	FLYING_SAUCER3	Last sprite value for the attack ship on the lower planet
Byte 6	\$00	Rate at which to switch to alternate enemy mode.
Byte 7	nullPtr	Lo Ptr for alternate enemy mode
Byte 8	nullPtr	Hi Ptr for alternate enemy mode
Byte 9	nullPtr	Unused Lo Ptr to an arbitrary run of Bytes 18-21.
Byte 10	nullPtr	Unused Hi Ptr to an arbitrary run of Bytes 18-21.
Byte 11	\$00	Unused Rate limite for use of Bytes 9 and 10.
Byte 12	nullPtr	Unused
Byte 13	nullPtr	Unused
Byte 14	\$00	Controls the rate at which new enemies are added.
Byte 15	\$40	Update rate for attack wave
Byte 16	planet1Level1Data2ndStage	Lo Ptr to the wave data we switch to when first hit.
Byte 17	planet1Level1Data2ndStage	Hi Ptr to the wave data we switch to when first hit.
Byte 18	\$06	X Pos movement for attack ship.
Byte 19	\$01	Y Pos movement pattern for attack ship.
Byte 20	\$01	X Pos Frame Rate for Attack ship.
Byte 21	\$01	Y Pos Frame Rate for Attack ship.
Byte 22	\$00	Stickiness factor, does the enemy stick to the player
Byte 23	\$00	Does the enemy gravitate quickly toward the player when its hit?
Byte 24	nullPtr	Lo Ptr for second wave of attack ships.
Byte 25	nullPtr	Hi Ptr for second wave of attack ships.
Byte 26	nullPtr	Lo Ptr for third wave of attack ships.
Byte 27	nullPtr	Hi Ptr for third wave of attack ships.
Byte 28	spinningRings	Lo Ptr for Explosion animation.
Byte 29	spinningRings	Hi Ptr for Explosion animation.
Byte 30	defaultExplosion	Lo Ptr for another set of wave data for this level.
Byte 31	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 32	\$00	Lo Ptr for fourth wave of attack ships.
Byte 33	\$00	Hi Ptr for fourth wave of attack ships.
Byte 34	\$02	Points for hitting the enemy.
Byte 35	\$02	Energy increase multiplier for hitting an enemy.
Byte 36	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 37	\$04	Number of waves in data.
Byte 38	\$18	Number of ships in wave.
Byte 39	\$00	Unused byte.

Sheep Planet - Level 1 - Enemy Data for the First Wave.

7.1 Something Simple - Byte 0: The Enemy's Color

To get an understanding of how the level data is used we can start with the very first byte of the data used for the first wave in the very first level. This is the wave of flying saucers you will already be familiar with if you have played the game (:)):

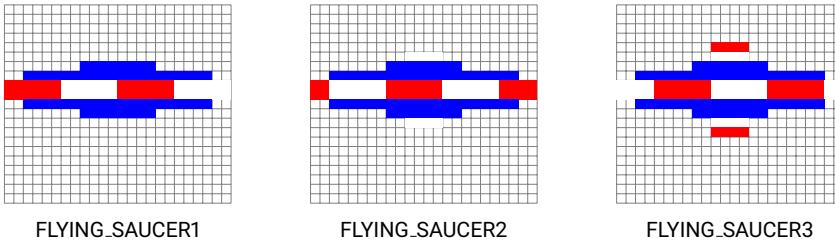


Figure 7.2: The sprites used to animate the 'UFO' in the first level.

The main colour for this sprite is blue. This may not be obvious from looking at the sprites themselves, but the way the sprite color schemes work is that you can select two colors that are available for all sprites and one color that is unique to the sprite itself. In this case, the unique color selected for the flying saucers is determined by Byte 1 in the Level Data. You can see in the table in the page opposite that this is defined with a value of \$06. This is how it looks in the code itself:

```
planet1Level1Data
    ; Byte 0 (Index $00): An index into colorsForAttackShips that
    ; applies a
    ; color value for the ship sprite.
    .BYTE $06
```

This value is an index into the array `colorsForAttackShips`. Starting at zero we count up to 6 and arrive at the 7th item in the list below, giving us the result `BLUE`:

```
colorsForAttackShips
    .BYTE BLACK,WHITE,RED,CYAN,PURPLE,GREEN,BLUE,YELLOW
    .BYTE ORANGE,BROWN,LTRED,GRAY1,GRAY2,LTGREEN,LTBLUE,GRAY3
```

In the routine that draws the attack wave we find the following code segment writing this value to the register `$D027` that determines the color of the sprite:

```
; -----
; DrawUpperPlanetAttackShips Routine
; -----
```

```
DrawUpperPlanetAttackShips
    LDX #$0C
    LDY #$06
UpperPlanetShipsLoop
    ...
    LDX upperPlanetAttackShipsColorArray ,Y
    LDA colorsForAttackShips ,X
    STA $D027,Y ;Sprite Y Color
    ...
    DEX
    DEX
    DEY
    BNE UpperPlanetShipsLoop
RTS
```

Notice that the \$06 was originally stored in a position in the array `upperPlanetAttackShipsColorArray`. This happened in an earlier routine that loads the majority of the data for a level:

```
GetWaveDataForNewShip
; X is the index of the ship in activeShipsWaveDataLoPtrArray
    LDY #$00
    LDA (currentShipWaveDataLoPtr),Y
    STA upperPlanetAttackShipsColorArray + $01,X
```

7.2 Bytes 1-4: Sprite Animation

You pass the interpreter data, that describes exactly stuff like: what each alien looks like, how many frames of animation it uses, speed of that animation, colour, velocities in X- and Y- directions, accelerations in X and Y, whether the alien should 'home in' on a target, and if so, what to home in on; whether an alien is subject to gravity, and if so, how strong is the gravity; what the alien should do if it hits top of screen, the ground, one of your bullets, or you; whether the alien can fire bullets, and if so, how frequently, and what types; how many points you get if you shoot it, and how much damage it does if it hits you; and a whole bunch more stuff like that. As you can imagine it was a fairly heavy routine to write and get debugged, but that's done now; took me about three weeks in all I'd say.

— Jeff Minter's Development Diary in Zzap Magazine^[?]



Figure 7.3: Confirmation that the game developer was a young male. The sprites used to animate attack wave 17 in the Mushroom Planet.

Looking again at the table in the previous page we can see the first 7 bytes are concerned with the appearance and basic behaviour of the enemy. Bytes 1 and 2 define the sprite used for display on the upper planet, Bytes 4 and 5 for the lower planet. The reason there's two in each case is because they are describing the start and end point of the sprite's animation.

We can see this in action in `AnimateAttackShipSprites`. When this routine runs Byte 3 has been loaded to `upperPlanetAttackShipInitialFrameRate` for the upper planet and `lowerPlanetAttackShipInitialFrameRate` for the lower planet. This routine is cycling through the sprites given by Byte 1 as the lower limit and Byte 2 as the upper limit. This is what the animation consists of: an animation effect achieved by changing the sprite from one to another to create a classic animation effect.

```

AnimateAttackShipSprites
    LDA pauseModeSelected
    BEQ AnimateUpperPlanetAttackShips
    RTS

AnimateUpperPlanetAttackShips
    DEC upperPlanetAttackShipAnimationFrameRate - $01,X
    BNE AnimateLowerPlanetAttackShips

    LDA upperPlanetAttackShipInitialFrameRate - $01,X
    STA upperPlanetAttackShipAnimationFrameRate - $01,X
    INC upperPlanetAttackShip2SpriteValue ,X
    LDA upperPlanetAttackShip2SpriteValue ,X

    ; Reached the end of the animation?
    CMP upperPlanetAttackShipSpriteAnimationEnd - $01,X
    BNE AnimateLowerPlanetAttackShips

    ; Reset the animation sprite back to the start.
    LDA upperPlanetAttackShipSpritesLoadedFromBackingData - $01,X
    STA upperPlanetAttackShip2SpriteValue ,X

```

CHAPTER 7. ENEMIES AND THEIR DISCONTENTS

```
AnimateLowerPlanetAttackShips
DEC lowerPlanetAttackShipAnimationFrameRate - $01,X
BNE DontAnimateLowerPlanetAttackShip

LDA lowerPlanetAttackShipInitialFrameRate - $01,X
STA lowerPlanetAttackShipAnimationFrameRate - $01,X
INC lowerPlanetAttackShip2SpriteValue,X
LDA lowerPlanetAttackShip2SpriteValue,X

; Reached the end of the animation?
CMP lowerPlanetAttackShipSpriteAnimationEnd - $01,X
BNE DontAnimateLowerPlanetAttackShip

; Reset the animation sprite back to the start.
LDA lowerPlanetAttackShipSpritesLoadedFromBackingData - $01,X
STA lowerPlanetAttackShip2SpriteValue,X
DontAnimateLowerPlanetAttackShip
RTS
```

Listing 7.1: Routine for Animating Enemy Sprites.

Byte 1 (loaded to `upperPlanetAttackShipAnimationFrameRate` comes into play here. It's decremented and as long as it's not zero yet the animation is skipped, execution jumps forward to `AnimateLowerPlanetAttackShips`:

```
AnimateLowerPlanetAttackShips
DEC lowerPlanetAttackShipAnimationFrameRate - $01,X
BNE DontAnimateLowerPlanetAttackShip

LDA lowerPlanetAttackShipInitialFrameRate - $01,X
STA lowerPlanetAttackShipAnimationFrameRate - $01,X
INC lowerPlanetAttackShip2SpriteValue,X
LDA lowerPlanetAttackShip2SpriteValue,X

; Reached the end of the animation?
CMP lowerPlanetAttackShipSpriteAnimationEnd - $01,X
BNE DontAnimateLowerPlanetAttackShip

; Reset the animation sprite back to the start.
LDA lowerPlanetAttackShipSpritesLoadedFromBackingData - $01,X
STA lowerPlanetAttackShip2SpriteValue,X
```

If it is zero, it instead gets reset to the initial value from Byte 1 (stored in `upperPlanetAttackShipInitialFrameRate`) and the current sprite for the enemy ship is incremented to point to the next 'frame' of the ship's animation:

```
AnimateUpperPlanetAttackShips
DEC upperPlanetAttackShipAnimationFrameRate - $01,X
BNE AnimateLowerPlanetAttackShips

LDA upperPlanetAttackShipInitialFrameRate - $01,X
```

```

STA upperPlanetAttackShipAnimationFrameRate - $01,X
INC upperPlanetAttackShip2SpriteValue ,X
LDA upperPlanetAttackShip2SpriteValue ,X

; Reached the end of the animation?
CMP upperPlanetAttackShipSpriteAnimationEnd - $01,X
BNE AnimateLowerPlanetAttackShips

; Reset the animation sprite back to the start.
LDA upperPlanetAttackShipSpritesLoadedFromBackingData - $01,X
STA upperPlanetAttackShip2SpriteValue ,X

```

Next we check if we've reached the end of the animation by checking the value of Byte 2 (loaded to `upperPlanetAttackShipSpriteAnimationEnd`). If so, we reset `upperPlanetAttackShip2SpriteValue` to the value initially loaded from Byte 1 - and that is what will be used to display the ship the next time we pass through to animate the ship:

```

DrawUpperPlanetAttackShips
    LDX #$0C
    LDY #$06
UpperPlanetShipsLoop
    LDA upperPlanetAttackShipsXPosArray ,Y
    STA $D000,X ;Sprite 0 X Pos

    LDA attackShipsXPosArray - $01,Y
    AND $D010 ;Sprites 0-7 MSB of X coordinate
    STA currentMSBXPosOffset

    LDA upperPlanetAttackShipsMSBXPosArray ,Y
    AND attackShipsMSBXPosOffsetArray ,Y
    ORA currentMSBXPosOffset
    STA $D010 ;Sprites 0-7 MSB of X coordinate

    LDA upperPlanetAttackShipsYPosArray ,Y
    STA $D001,X ;Sprite 0 Y Pos
    STX tempVarStorage

    LDX upperPlanetAttackShipsColorArray ,Y
    LDA colorsForAttackShips ,X
    STA $D027,Y ;Sprite 0 Color

    LDA upperPlanetAttackShipsSpriteValueArray ,Y
    STA Sprite0Ptr ,Y
    LDX tempVarStorage

    DEX
    DEX
    DEY
    BNE UpperPlanetShipsLoop
    RTS

```

7.3 Bytes 18-21: Enemy Movement

Enemy movement is controlled by two parameters in each direction: the number of pixels to move in one go and the number of cycles to wait between each movement. So for movement in the horizontal (or X direction) Byte 18 controls the number of pixels to move at once, while Byte 20 controls the number of cycles to wait between each movement. The same applies to Byte 19 and Byte 21 for the vertical (or Y direction).

If we look at Byte 18 and Byte 20 for Level 1 we can see that the the fast lateral movement of the 'UFO's is implemented by a relatively high value of \$06 for the number of pixels it moves at each step while the interval between steps is relatively low (\$01). Meanwhile the more gradual up and down movement is implemented by a value of \$01 in Byte 19 and Byte 21.

For the second level ('bouncing rings') the horizontal movement is more constrained (Byte 18 is \$00) while the vertical movement is more extreme (Byte 19 is \$24) - achieving the bouncing effect.

Level	Byte 6	Byte 18	Byte 19	Byte 20	Byte 21
1	\$00	\$06	\$01	\$01	\$01
2	\$00	\$00	\$24	\$02	\$01
3	\$00	\$FA	\$01	\$01	\$02

Byte 6 : Whether a specific attack behaviour is used.

Byte 18: X Pos movement for attack ship.

Byte 19: Y Pos movement pattern for attack ship.

Byte 20: X Pos Frame Rate for Attack ship.

Byte 21: Y Pos Frame Rate for Attack ship.

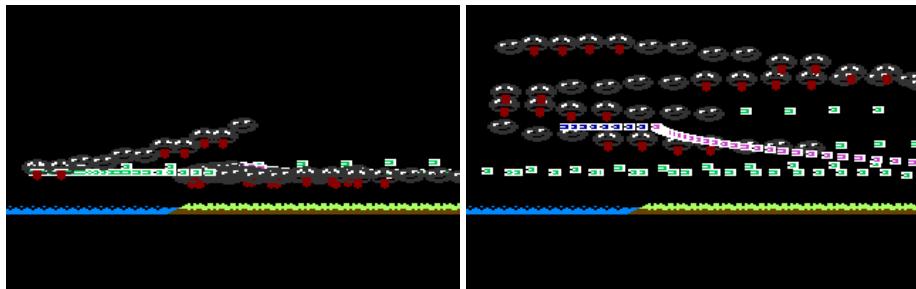
Movement data for the first three levels.

The horizontal movement for Level Three, home to the infamous 'Licker Ships' is \$FA, which would make you think the enemies must be moving horizontally extremely quickly. In fact, when the high bit is set a special behaviour is invoked:

```
LDA xPosMovementForUpperPlanetAttackShip - $01,X  
BMI UpperBitSetOnXPosMovement
```

Listing 7.2: From UpdateAttackShipsXAndYPositions.

This is the special behaviour that makes the Licker Ships on this level such an enormous pain in the arse to play against.



Enemy movement for Sheep planet, level 3 - the infamous licker ships.

When the upper bit is set (e.g. \$FC,\$80) on the value loaded to the accumulator by LDA then BMI will return true and jump to `UpperBitSetOnXPosMovement`.

```
UpperBitSetOnXPosMovement
    ; This creates a decelerating effect on the attack ship's
    ; movement.
    ; Used by the licker ship wave in planet 1 for example.
    EOR #$FF
    STA attackShipOffsetRate
    INC attackShipOffsetRate
    LDA upperPlanetAttackShip2XPos ,X
    SEC
    SBC attackShipOffsetRate
    STA upperPlanetAttackShip2XPos ,X
    BCS DecrementXPosFrameRateLowerPlanet

    LDA upperPlanetAttackShip2MSBXPosValue ,X
    EOR attackShip2MSBXPosOffsetArray ,X
    STA upperPlanetAttackShip2MSBXPosValue ,X
```

This first line `EOR #$FF` performs an exclusive-or between Byte 19 in the Accumulator (\$FA) and the value \$FF. An exclusive-or, remember, is a bit by bit comparison of two bytes which will set a bit in the result if an only if the bit in one of the values is set but the other is not:

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
\$FA	1	1	1	1	1	0	1	0
Result	0	0	0	0	0	1	0	1

X-OR'ing \$FF and \$FA gives \$05.

This result is stored in `attackShipOffsetRate`:

```
UpperBitSetOnXPosMovement
; This creates a decelerating effect on the attack ship's movement.
; Used by the licker ship wave in planet 1 for example.
EOR #$FF
STA attackShipOffsetRate
```

Incremented:

```
INC attackShipOffsetRate
```

And then subtracted from the enemy's X position:

```
SEC
SBC attackShipOffsetRate
STA upperPlanetAttackShip2XPos ,X
```

The net result is a deceleration effect. This is observed in the way the licker ship wave will accelerate out to the center before dialling back again.

7.3.1 What is going on with Byte 6?

Byte 6 comes into play when setting the initial Y position of a new enemy. This initial vertical position is random, but subject to some adjustment:

```
SetInitialRandomPositionUpperPlanet
JSR PutProceduralByteInAccumulatorRegister
AND #$3F
CLC
ADC #$40
STA upperPlanetAttackShipsYPosArray + $01,Y

STY previousAttackShipIndexTmp
; Byte 6 ($06): Usually an update rate for the attack ships.
LDY #$06
LDA (currentShipWaveDataLoPtr),Y
BNE ReturnFromLoadingWaveDataEarly

; Byte 8 ($08): Default initiation Y position for the enemy.
LDY #$08
LDA (currentShipWaveDataLoPtr),Y
BEQ ReturnFromLoadingWaveDataEarly

LDA #$6C
LDY previousAttackShipIndexTmp
STA upperPlanetAttackShipsYPosArray + $01,Y

ReturnFromLoadingWaveDataEarly
```

```
RTS
```

Listing 7.3: The sub-routine SetInitialRandomPositionUpperPlanet in GetWaveDateForNewShip.

The first order of business is to call PutProceduralByteInAccumulatorRegister which gets a random value and stores it in the accumulator.

```
PutProceduralByteInAccumulatorRegister
randomIntToIncrement    =**+$01
    LDA  randomPlanetData
    INC  randomIntToIncrement
    RTS
```

Since A can now contain anything from 0 to 255 (\$00 to \$FF) this needs to be adjusted to a meaningful Y-position value for the upper planet. So if we imagine PutProceduralByteInAccumulatorRegister returned \$85, we now do the following operations on it:

```
AND  #$3F
CLC
ADC  #$40
STA  upperPlanetAttackShipsYPosArray + $01, Y
```

First we do an AND #\$3F with the value of \$85 in A:

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
\$85	1	0	0	0	0	1	0	1
\$3F	0	0	1	1	1	1	1	1
Result	0	0	0	0	0	1	0	1

AND'ing \$3F and \$85 gives \$05.

Our result is \$05. The effect of the AND'ing here is to ensure that the random number we get back is between 0 and 63 rather than 0 and 255. Next we add \$40 (decimal 64) to this result:

```
CLC
ADC  #$40
```

This gives \$45 and this is what we store as the initial Y position for the enemy.

You'll notice that the steps for SetInitialRandomPositionLowerPlanet are identical but with only the constant of the add value of \$98 instead of \$40. This is simply an ad-

ditional offset to ensure that the Y position is lower on the screen for the initial position of the enemy on the lower planet.

We still haven't got into what Byte 7 is doing though. With an initial Y position determined, it looks like the intention was for Byte 6 to specify some adjustment to this value. But this looks like another bit of non-functioning game logic. If Byte 6 contains a value, the function will return early without any further adjustments. If it's zero it will then try Byte 8. If that's zero, it will return early. So the logic needs Byte 6 to be zero and Byte 8 to contain something for anything to happen. That's never the case, so the the adjustment never happens:

```
STY previousAttackShipIndexTmp
; Byte 6 ($06): Usually an update rate for the attack ships.
LDY #$06
LDA (currentShipWaveDataLoPtr),Y
BNE ReturnFromLoadingWaveDataEarly

; Byte 8 ($08): Default initiation Y position for the enemy.
LDY #$08
LDA (currentShipWaveDataLoPtr),Y
BEQ ReturnFromLoadingWaveDataEarly
```

Listing 7.4: An adjustment that never happens. Byte 6 and Byte 8 are never set in this way.

This is definitely some forgotten code. Byte 6 is elsewhere used in combination with Byte 7 and Byte 8 to define an alternate enemy mode for some levels where the ship will supplement any dead ships with alternate enemy types and attack patterns periodically.

7.4 Bytes 6-8: Alternate Enemy Waves

This happens in `MaybeSwitchToAlternateEnemyPattern` in `UpdateAttackShipDataForNewShip`.

```
MaybeSwitchToAlternateEnemyPattern
; Byte 6 ($06): Usually an update rate for the attack ships.
LDY #$06
LDA (currentShipWaveDataLoPtr),Y
BEQ EarlyReturnFromAttackShipBehaviour

DEC rateForSwitchingToAlternateEnemy,X
BNE EarlyReturnFromAttackShipBehaviour

LDA (currentShipWaveDataLoPtr),Y
STA rateForSwitchingToAlternateEnemy,X

; Push the current ship's position data onto the stack.
```

```

TXA
PHA
LDY indexIntoUpperPlanetAttackShipXPosAndYPosArray ,X
LDA upperPlanetAttackShipsXPosArray + $01,Y
PHA
LDA upperPlanetAttackShipsMSBXPosArray + $01,Y
PHA
LDA upperPlanetAttackShipsYPosArray + $01,Y
PHA

; Are we on the top or bottom planet?
TXA
AND #$08
BNE LowerPlanetAttackShipBehaviour

```

Listing 7.5: Byte 6 is used to periodically switch to an enemy mode defined by Bytes 7-8

Byte 6 is used to drive the rate at which this routine switches over to the enemy data-mode defined by Byte 7 and Byte 8.

```

DEC rateForSwitchingToAlternateEnemy ,X
BNE EarlyReturnFromAttackShipBehaviour

LDA (currentShipWaveDataLoPtr),Y
STA rateForSwitchingToAlternateEnemy ,X

```

Listing 7.6: `rateForSwitchingToAlternateEnemy` (Byte 6) is decremented and reloaded each time it reaches zero.

What this routine is going to do is replace the first dead ship it finds in the current wave with the wave data pointed to by Byte 7-8 and create a new enemy with the current ship's position with it.

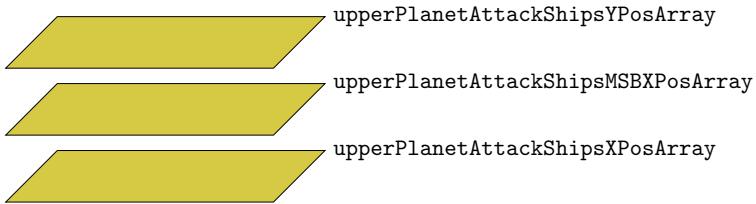
First, we store the current ship's position. The way to do this is get the index (Y) for the current ship X and store each of the X and Y Position information into the accumulator first A and then push it onto the 'stack' (PHA which means 'push A onto the stack').

```

; Push the current ship's position data onto the stack.
TXA
PHA
LDY indexIntoUpperPlanetAttackShipXPosAndYPosArray ,X
LDA upperPlanetAttackShipsXPosArray + $01,Y
PHA
LDA upperPlanetAttackShipsMSBXPosArray + $01,Y
PHA
LDA upperPlanetAttackShipsYPosArray + $01,Y
PHA

```

When this has run the stack of accumulator values now looks like this:



The stack after the code above has run with `upperPlanetAttackShipsXPosArray` at the top.

With our position data safely stashed away on the stack we now decide which planet we're on:

```
; Are we on the top or bottom planet?  
TXA  
AND #$08  
BNE LowerPlanetAttackShipBehaviour
```

If we're on the upper planet we use `SetXToIndexOfShipThatNeedsReplacing` look in the `activeShipsWaveDataHiPtrArray` for any ships that need replacing between positions \$02 and \$06. If we don't find one, we return early:

```
; We're on the upper planet.  
LDX #$02  
ProcessAttackShipBehaviour  
JSR SetXToIndexOfShipThatNeedsReplacing  
BEQ ResetAndReturnFromAttackShipBehaviour
```

If we do find one we can now pull (or 'pop') the positional data we stored away in the stack and assign that to the once-dead ship. First we use the index we retrieved to X to get the ship's index (Y) into the positional arrays:

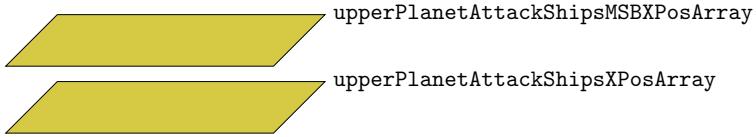
```
LDY indexIntoUpperPlanetAttackShipXPosAndYPosArray ,X
```

Then we pop the first positional item `upperPlanetAttackShipsYPosArray` from the top of the stack and store in the new ship's position in the array:

```
PLA  
STA upperPlanetAttackShipsYPosArray + $01 ,Y
```

Listing 7.7: "PLA remove the top item from the stack and stores it in A

The stack now looks like this, popping from the stack has the effect of removing the first item:



Then we pop the rest of the items one by one and assign them to the new ship. We ignore the sprite's MXB offset if it is zero:

```

PLA
BEQ MSBPosOffsetIzZero

LDA attackShipsMSBPosOffsetArray + $01,X
MSBPosOffsetIzZero
STA upperPlanetAttackShipsMSBPosArray + $01,Y
PLA
STA upperPlanetAttackShipsXPosArray + $01,Y
PLA

; Byte 8 of Wave Data gets loaded now. Bytes 8 and 9
; contain the hi/lo ptrs to the alternate enemy data.
LDY #$07
JMP UpdateWaveDataPointersForCurrentEnemy

```

Listing 7.8: PLA remove the top item from the stack and stores it in A.

Now that we have set up the positional data for the new enemy we load all its other features from the data pointed to by Byte 8-9:

```

; Byte 8 of Wave Data gets loaded now. Bytes 8 and 9
; contain the hi/lo ptrs to the alternate enemy data.
LDY #$07
JMP UpdateWaveDataPointersForCurrentEnemy

```

Let's take a closer look at this routine `UpdateWaveDataPointersForCurrentEnemy`. What it does in this instance is take the address pointed to by Bytes 8 and 9 and load the data there using the routine `GetWaveDataForNewShip`. To be used in this way the values in Bytes 8 and 9 are combined and treated as an address in memory. For example if Byte 8 contains \$70 and Byte 9 contains \$13 they are treated as providing the address \$1370. This is the location where the enemy data for `planet1Level8Data` is kept so that is what is loaded.

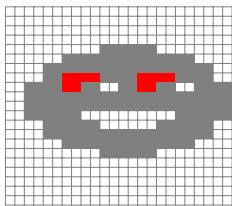
Planet	Level	Byte 6	Bytes 7-8
1	11	\$03	smallDotWaveData
1	14	\$03	planet1Level8Data
2	19	\$0C	landGilbyAsEnemy
3	4	\$04	gilbyLookingLeft
3	6	\$04	planet3Level6Additional
4	19	\$01	planet4Level19Additional
5	3	\$01	planet5Level3Additional
5	5	\$05	planet5Level5Additional
5	14	\$06	llamaWaveData

Byte 6 : Whether a specific attack behaviour is used.

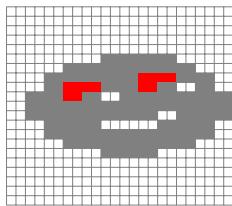
Bytes 7-8 : Lo and Hi Ptr for alternate enemy mode

Table 7.1: Actual use of Bytes 6, 7, and 8. Note that the value in Byte 6 doesn't matter, as long as it's non-zero.

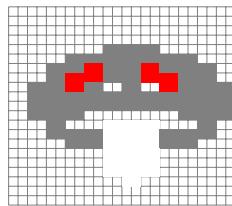
7.5 Bytes 22-23: The Bytes of Death



LICKER_SHIP1



LICKERSHIP2



LICKERSHIP3

Figure 7.4: These arseholes.

An irritating early hurdle in Iridis Alpha's gameplay is the behaviour of the lickerships in the game's third level. When shot the enemies turn into lickerships that seek out the player and then stick to them, sapping the player's energy until they lose a life. This behaviour is defined by Bytes 22 and 23.

```
lickerShipWaveData = $1118
..
; Byte 22 (Index $16): Stickiness factor, does the enemy stick to the
; player
; sapping their energy if they're near them?
.BYTE $01
; Byte 23 (Index $17): Does the enemy gravitate quickly toward the
; player when its
```

```
; been shot? (Typical lickership behaviour)
.BYTE $01
..
```

Implementing each of these is relatively straightforward. Since the gilby is always in the centre of the screen (on the X axis at least), all an enemy has to do is figure out whether the gilby is above or below it and move in that direction. If we leave out the slightly complicated logic to figure out where the gilby is relative to us, we are left with this:

```
MaybeQuicklyGravitatesToGilby
; Byte 23: Does the enemy gravitate quickly towards the gilby when
; it is shot?
LDY #$17
LDA (currentShipWaveDataLoPtr),Y
BEQ MaybeStickyAttackShipBehaviour
...
; Figure out the relative position of the gilby and
; store it in positionRelativeToGilby
...
; Now decide whether to move up or down.
CMP positionRelativeToGilby
BEQ NoVerticalMovementRequired
BMI MoveDownToGilby
MoveUpToGilby
DEC yPosMovementForUpperPlanetAttackShips,X
DEC yPosMovementForUpperPlanetAttackShips,X
MoveDownToGilby
INC yPosMovementForUpperPlanetAttackShips,X
NoVerticalMovementRequired
LDA indexForActiveShipsWaveData,X
TAX
```

Sticking to the gilby involves the same logic but along the horizontal axis. After all if we're sticking to the gilby we want to stay in the centre of the screen.

```
MaybeStickyAttackShipBehaviour
; Byte 22: Does the enemy have the stickiness behaviour?
LDY #$16
LDA (currentShipWaveDataLoPtr),Y
BEQ MaybeSwitchToAlternateEnemyPattern
...
; Figure out the relative position of the gilby and
; store it in positionRelativeToGilby
...
CMP positionRelativeToGilby
BMI MoveRightToGilby
MoveLeftToGilby
DEC xPosMovementForUpperPlanetAttackShip,X
DEC xPosMovementForUpperPlanetAttackShip,X
MoveRightToGilby
```

```
INC xPosMovementForUpperPlanetAttackShip,X  
NoHorizontalMovementRequired  
LDA indexForActiveShipsWaveData,X  
TAX
```

7.6 Byte 35: Energy Multiplier

When an enemy is struck this byte contains the multiplier applied to the player's energy boost:

```
IncreaseEnergyTopOnly  
LDY #$23  
LDA (currentShipWaveDataLoPtr),Y  
BEQ NormalTopEnergyIncrease  
STA energyChangeCounter  
EnergyTopIncreaseLoop  
JSR IncreaseEnergyTop  
DEC energyChangeCounter  
BNE EnergyTopIncreaseLoop  
RTS  
  
NormalTopEnergyIncrease  
JMP IncreaseEnergyTop  
; Returns
```

The above is for the top planet energy counter, the logic for the bottom planet is identical:

```
IncreaseEnergyBottomOnly  
LDY #$23  
LDA (currentShipWaveDataLoPtr),Y  
BEQ NormalBottomEnergyIncrease  
STA energyChangeCounter  
EnergyBottomIncreaseLoop  
JSR IncreaseEnergyBottom  
DEC energyChangeCounter  
BNE EnergyBottomIncreaseLoop  
RTS  
  
NormalBottomEnergyIncrease  
JMP IncreaseEnergyBottom  
; Returns
```

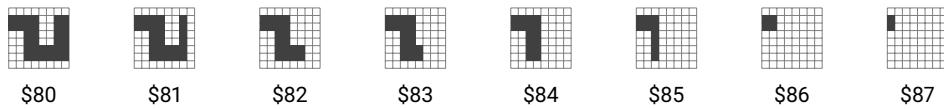


Figure 7.5: Tilesheet: Energy Bar Character Set

Actually writing the updated energy level to the screen uses the tileset above in the routine `IncreaseEnergyTop`:

```
IncreaseEnergyTop
    STX temporaryStorageForXRegister
    LDX currEnergyTop
    DEC SCREEN_RAM + LINE22_COL3,X
    LDA SCREEN_RAM + LINE22_COL3,X
    CMP #$7F
    BNE b547B
    ; Note the reference to the index $80 of the first tile in the set
    ; above.
    LDA #$80
    STA SCREEN_RAM + LINE22_COL3,X
    INX
    STX currEnergyTop
    CPX #$08
    BEQ GilbyDiesFromExcessEnergy
    LDA #$87
    STA SCREEN_RAM + LINE22_COL3,X
    BNE b547B
```

Byte 35 also determines the multiplier applied to the energy sapped from the player in the event of a collision with the enemy:

```
UpdateEnergyLevelsAfterCollision
    ; Check if the enemy saps energy from the gilby?
    LDY #$23
    LDA (currentShipWaveDataLoPtr),Y
    BEQ NoMultiplierAppliedToCollision

    LDA #<shipCollidedWithGilbySound
    STA primarySoundEffectLoPtr
    LDA #>shipCollidedWithGilbySound
    STA primarySoundEffectHiPtr
    JSR ResetRepetitionForPrimarySoundEffect
    LDA #$0E
    STA gilbyExploding
    LDA #$02
    STA starFieldInitialStateArray - $01
    LDA currentGilbySpeed
    EOR #$FF
    CLC
```

```
ADC #$01
STA currentGilbySpeed

LDA setToZeroIfOnUpperPlanet
BEQ EnergyUpdateTopPlanet

LDA extraAmountToDecreaseEnergyByBottomPlanet
BNE NoMultiplierAppliedToCollision
; Y is still $23.
LDA (currentShipWaveDataLoPtr),Y
JSR AugmentAmountToDecreaseEnergyByBountiesEarned
STA extraAmountToDecreaseEnergyByBottomPlanet
BNE NoMultiplierAppliedToCollision

EnergyUpdateTopPlanet
LDA extraAmountToDecreaseEnergyByTopPlanet
BNE NoMultiplierAppliedToCollision
; Y is still $23.
LDA (currentShipWaveDataLoPtr),Y
JSR AugmentAmountToDecreaseEnergyByBountiesEarned
STA extraAmountToDecreaseEnergyByTopPlanet
NoMultiplierAppliedToCollision
LDY #$1E
JMP UpdateWaveDataPointersForCurrentEnemy
; Returns
```

7.7 Byte 34: Score for Hitting the Enemy

This is used to augment the score received for hitting the enemy.

```
; Get the points for hitting enemies in this level
; from the wave data.
AddPointsForHittingEnemy
; Byte 34
LDY #$22
LDA (currentShipWaveDataLoPtr),Y
..
ADC pointsEarnedTopPlanetByte1
STA pointsEarnedTopPlanetByte1
```

Congoatulations Hotshot

This decidedly ropey mini-game has a few interesting features.

8.1 Entry Sequence



Figure 8.1: The entry sequence.

CHAPTER 8. CONGOATULATIONS HOTSHOT

The entry sequence is an animated cascade of colored bars that appears to roll down from the top of the screen. You might assume we achieve this effect by simply drawing a series of colored text-based lines in a tight loop. Not the case:

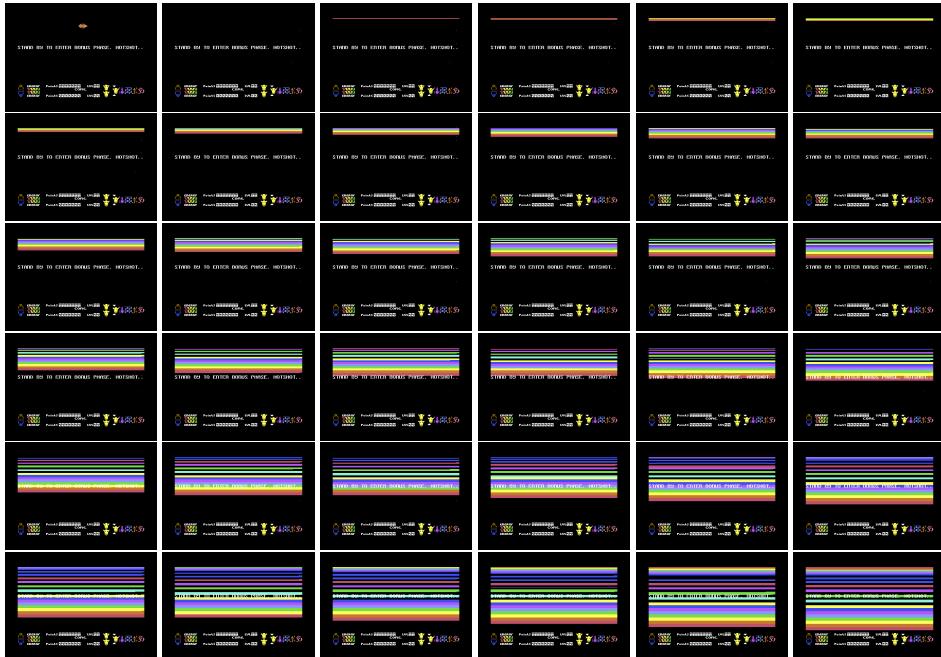


Figure 8.2: The entry effect filling the screen.

```
EnterBonusPhaseInterruptHandler
...
LDY  backgroundColorIndex
LDA enterBPRainbowColors,Y
STA $D021      ;Background Color 0
```

What we actually do is update the screen's background color as the raster travels down the screen. The above three lines do this about 30 times every single paint of the screen, updating the background color that gets painted as we go. The result is that each colored bar reflects the background color of the screen at the time the raster is passing it. The trick is to keep updating the background color at gradually increasing intervals.

After we update the background color we set the raster interrupt to the next position we're interested in:

```

EnterBonusPhaseInterruptHandler
    ...
    LDY  backgroundColorIndex
    LDA  enterBPRainbowColors,Y
    STA  $D021      ;Background Color 0

    ; Check if we've reached the end of the rainbow effect.
    LDA  bpRasterPositionArray,Y
    ...
    ; Update the position of the next interrupt.
    CLC
    ADC  $D012      ;Raster Position
    STA  $D012      ;Raster Position

```

The value we get from `bpRasterPositionArray` reflects our intention of painting increasingly tall bars:

```

bpRasterPositionArray      .BYTE $01,$01,$01,$01,$02,$02,$02,$02
                           .BYTE $03,$03,$03,$03,$04,$04,$04,$04
                           .BYTE $05,$05,$05,$05,$06,$06,$06,$06
                           .BYTE $07,$07,$07,$07,$07,$07,$07,$00

```

Each value in here gets added to the current interrupt position (`ADC $D012`). The further we go down the screen the taller the bars become.

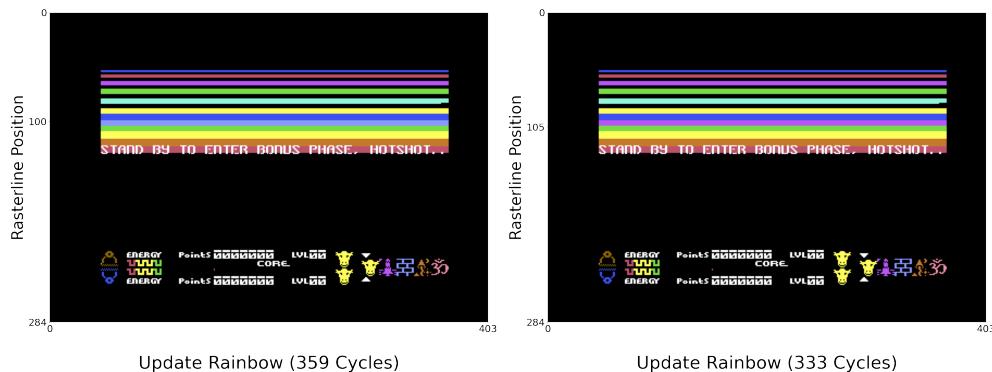


Figure 8.3: Updating the background color at a 5 line interval as given by `bpRasterPositionArray`.

In addition to drawing incrementally larger bars we're also shifting down by one row the color each one is painted at each pass. This is responsible for creating the visual effect of each bar moving independently down the screen.



Figure 8.4: The color sequences shifts down one position after each full screen paint.

This is managed by an array..

```
entryScreenRainbowColors
    .BYTE RED,ORANGE,YELLOW,GREEN,LTBLUE,PURPLE,BLUE,YELLOW
    .BYTE BLACK,CYAN,BLACK,GREEN,BLACK,PURPLE,BLACK,RED
    .BYTE BLACK,BLUE,BLACK,BLUE,BLACK,BLUE,PURPLE,LTBLUE
    .BYTE GREEN,YELLOW,ORANGE,RED
```

.. referenced in `UpdateEntryScreenRainbow`. After every full screen paint we update the an array `previousRoundRainbowColors` to store the position of the previous bar that used it. This allows us to use `previousRoundRainbowColors` as a source for X the next time we finish a screen and shift the colors along by one position:

```
UpdateEntryScreenRainbow
    ...
    LDA entryScreenRainbowColors,X
    STA enterBPRainbowColors,Y
    TYA
    STA previousRoundRainbowColors,X
```

All in all, this is a neat effect and would only have been possible at the speeds achieved by using the raster interrupt in this way. Simply painting the screen with colored text would have been much slower.

8.2 Generating Maps

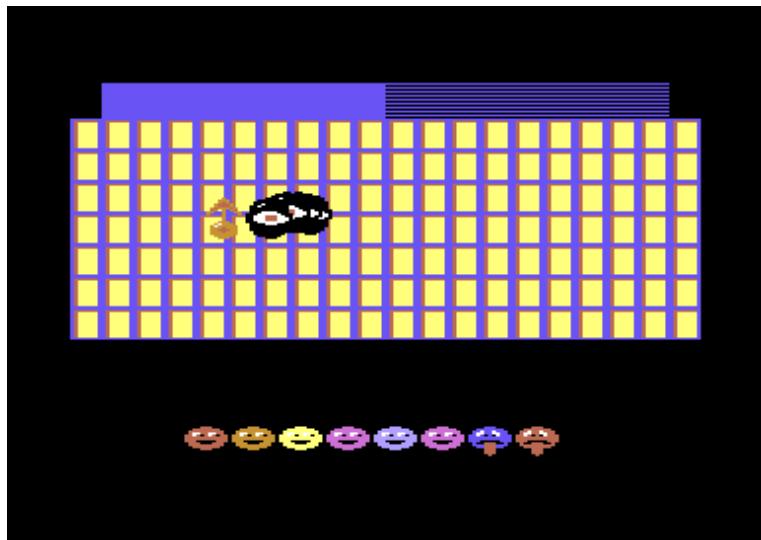


Figure 8.5: The beginning of the first bonus phase.

Every time the player enters the bonus phase we'll procedurally generate a new map. The way we'll manage this is by treating the map a stack of 256 rows and defining our map simply as an array of the rows that make up the map.

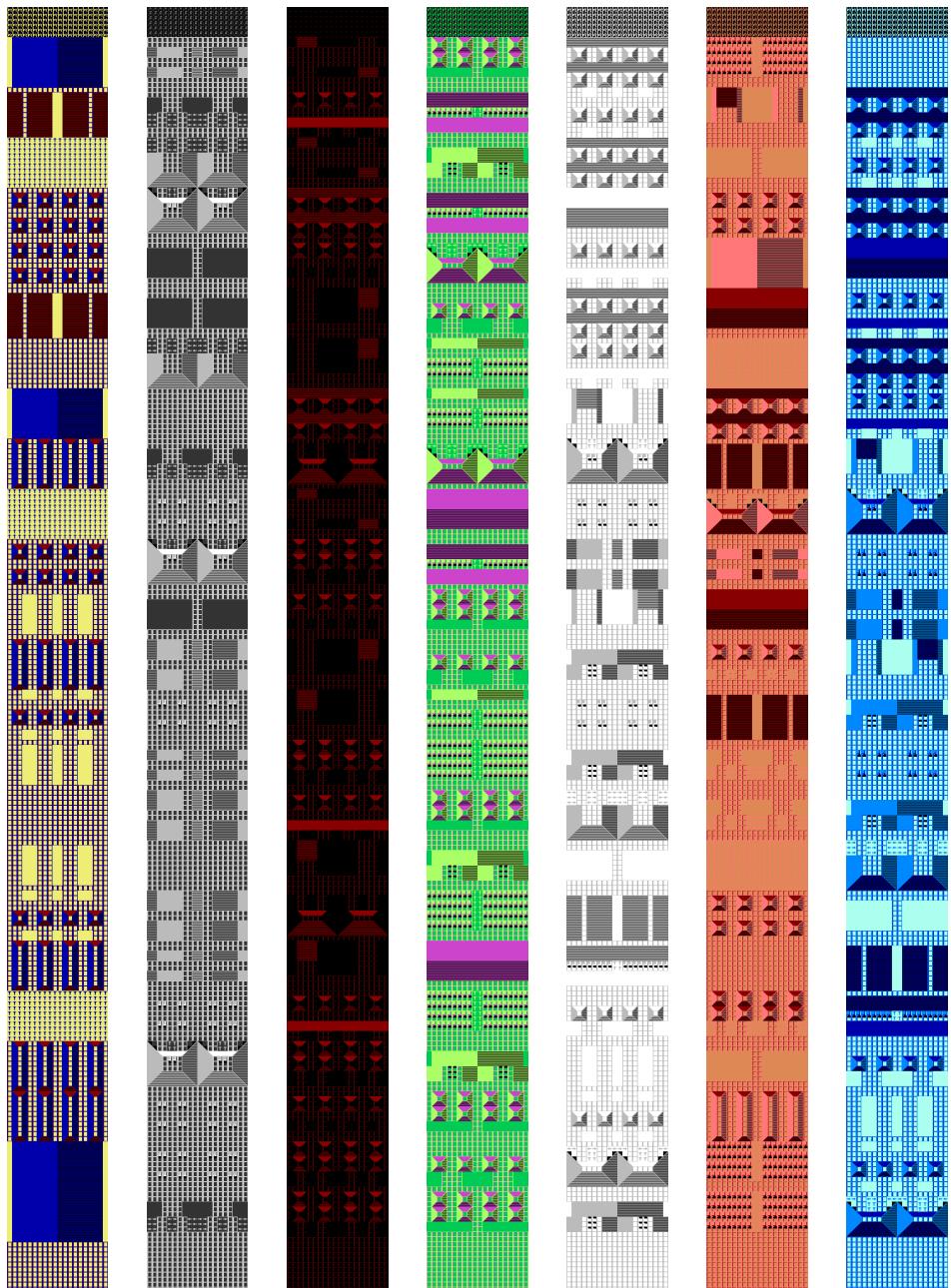


Figure 8.6: The first seven maps used by the bonus phase round.

This means that in order to generate a new map all we have to do is come up with an array of bytes where each byte defines a row in the map. By way of example, this is what the array that defines the map for the first bonus phase round looks like:

```
bonusPhaseMapDefinition
.BYTE $00,$00,$00,$00,$00,$00,$00,$00,$00,$00
.BYTE $11,$11,$11,$11,$11,$11,$11,$11,$11,$11
.BYTE $11,$11,$11,$11,$11,$11,$11,$11,$11,$11
.BYTE $15,$16,$16,$16,$16,$16,$16,$16,$16,$17
.BYTE $15,$16,$16,$16,$16,$16,$16,$16,$16,$17
.BYTE $13,$13,$13,$13,$13,$13,$13,$13,$13,$13
.BYTE $15,$16,$16,$16,$16,$16,$16,$16,$16,$17
.BYTE $14,$14,$00,$15,$16,$17,$00,$00,$14,$14
.BYTE $00,$14,$14,$14,$14,$14,$14,$14,$14,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$14,$14,$14,$14,$14,$14,$14,$14,$00
.BYTE $14,$14,$00,$15,$16,$17,$00,$00,$14,$14
.BYTE $15,$16,$16,$16,$16,$16,$16,$16,$16,$17
.BYTE $00,$14,$14,$14,$14,$14,$14,$14,$14,$00
.BYTE $00,$15,$16,$17,$00,$00,$15,$16,$17,$00
.BYTE $13,$13,$13,$13,$13,$13,$13,$13,$13,$13
.BYTE $15,$16,$16,$16,$16,$16,$16,$16,$16,$17
.BYTE $11,$11,$11,$11,$11,$11,$11,$11,$11,$11
.BYTE $00,$00,$00,$00,$00,$00,$00,$00,$00,$00
.BYTE $12,$12,$12,$12,$12,$12,$12,$12,$12,$00
.BYTE $00,$15,$16,$17,$00,$00,$15,$16,$17,$00
.BYTE $00,$15,$16,$17,$00,$00,$15,$16,$17,$00
.BYTE $13,$13,$13,$13,$13,$13,$13,$13,$13,$13
.BYTE $12,$12,$12,$12,$12,$12,$12,$12,$12,$00
.BYTE $11,$11,$11,$11,$11,$11,$11,$11,$11,$11
.BYTE $10,$10,$10,$10,$10,$10,$10,$10,$10,$10
```

This map definition starts from the bottom up, it does this because we are scrolling upward so it makes sense to begin with what the player will see first. The start of the map is a simple sequence of zeroes:

```
bonusPhaseMapDefinition
.BYTE $00,$00,$00,$00,$00,$00,$00,$00,$00,$00
```

This is the row image each line translates to, a regular square pattern initially followed by rows with some other features.

Index	Image
\$00	
\$00	
\$00	
\$00	
\$00	
\$00	
\$00	
\$00	
\$00	
\$00	
\$00	
\$00	
\$00	
\$00	

Figure 8.7: Snapshot of the map created by the first line definition.

Further along in the definition we reach a segment where some new features are introduced.

```
bonusPhaseMapDefinition
.BYTE $11,$11,$11,$11,$11,$11,$11,$11,$11,$11
.BYTE $15,$16,$16,$16,$16,$16,$16,$16,$16,$17
```

We can see what each of \$11, \$15, \$16 translate to when we map them to their corresponding images:

Index	Image
\$16	
\$16	
\$16	
\$15	
\$11	
\$11	
\$11	

Figure 8.8: Snapshot of the map created by the first line definition.

So how do we get from a value like \$15 to an image that represents a row on the screen?

Index	Image
\$15	

We do this by defining each row with yet another array of bytes. In all we define 32 rows of different types so when we come up with a map for the level we are simply referencing each of these rows by their index in the array. Here is the data structure defining each of the 32 rows:

```

bonusPhaseMapRowDefinitions
.BYTE $00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00 ; 00
.BYTE $0D,$0D,$0E,$0E,$0E,$0E,$00,$00,$0B,$0B,$00,$00,$0D,$0D,$0D,$0D,$0E,$0E
.BYTE $10,$0B,$0B,$0B,$0B,$0B,$0B,$0B,$0B,$11,$10,$0B,$0B,$0B,$0B,$0B,$0B,$0B,$0B,$11
.BYTE $0E,$10,$0B,$0B,$0B,$0B,$0B,$0B,$11,$0D,$0E,$10,$0B,$0B,$0B,$0B,$0B,$0B,$0B,$11,$0D
.BYTE $0E,$0E,$10,$0B,$0B,$0B,$0B,$0B,$11,$0D,$0D,$0E,$0E,$10,$0B,$0B,$0B,$0B,$11,$0D,$0D
.BYTE $0E,$0E,$0E,$0E,$00,$00,$00,$00,$0D,$0D,$0D,$0D,$0E,$0E,$00,$00,$00,$0D,$0D,$0D
.BYTE $0E,$0E,$0E,$00,$0A,$0A,$0A,$0D,$0D,$0D,$0D,$0E,$0E,$00,$0A,$0A,$00,$0D,$0D,$0D
.BYTE $0E,$0E,$0E,$00,$09,$09,$09,$09,$0D,$0D,$0D,$0D,$0E,$0E,$00,$09,$09,$00,$0D,$0D,$0D
.BYTE $0E,$0E,$12,$0C,$0C,$0C,$13,$0D,$0D,$0E,$0E,$12,$0C,$0C,$0C,$13,$0D,$0D
.BYTE $0E,$07,$00,$00,$15,$15,$00,$00,$05,$0D,$0E,$07,$00,$00,$15,$15,$00,$00,$05,$0D
.BYTE $07,$0F,$00,$00,$15,$15,$00,$00,$0F,$05,$07,$0F,$00,$00,$15,$15,$00,$00,$0F,$05
.BYTE $17,$17,$00,$00,$18,$17,$00,$00,$18,$18,$17,$17,$00,$00,$18,$17,$00,$00,$18,$18
.BYTE $0F,$0F,$0F,$00,$00,$00,$00,$0F,$0F,$0F,$0F,$0F,$00,$00,$00,$0F,$0F,$0F,$0F
.BYTE $00,$00,$09,$09,$00,$00,$09,$09,$00,$00,$00,$09,$09,$00,$00,$09,$09,$00,$00
.BYTE $00,$00,$0A,$0A,$0A,$00,$0A,$0A,$00,$00,$00,$00,$0A,$0A,$00,$00,$0A,$0A,$00,$00
.BYTE $0F,$0F,$0F,$0F,$0F,$0F,$0F,$0F,$0F,$0F,$0F,$0F,$0F,$0F,$0F,$0F,$0F,$0F,$0F,$0F ; 0F
.BYTE $14,$14,$14,$14,$14,$14,$14,$14,$14,$14,$14,$14,$14,$14,$14,$14,$14,$14,$14,$14
.BYTE $0F,$0E,$0E,$0E,$0E,$0E,$0E,$0E,$0D,$0D,$0D,$0D,$0D,$0D,$0D,$0D,$0D,$0D,$0D,$0D
.BYTE $0B,$0B,$0B,$00,$0B,$0B,$0B,$0B,$0B,$0B,$0B,$0B,$0B,$0B,$0B,$0B,$0B,$0B,$0B,$0B
.BYTE $15,$15,$15,$15,$15,$15,$15,$15,$15,$15,$15,$15,$15,$15,$15,$15,$15,$15,$15,$15
.BYTE $00,$00,$00,$0F,$0F,$00,$00,$00,$0F,$0F,$00,$00,$00,$0F,$0F,$00,$00,$00,$0F,$0F,$00,$00

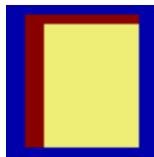
```

```
.BYTE $00,$10,$0B,$11,$00,$00,$10,$0B,$11,$00,$00,$10,$0B,$11,$00 ; 15
.BYTE $00,$0E,$00,$0D,$00,$00,$0E,$00,$0D,$00,$0E,$00,$0D,$00,$00,$00
.BYTE $00,$12,$0C,$13,$00,$00,$12,$0C,$13,$00,$00,$12,$0C,$13,$00
.BYTE $0F,$0E,$00,$0D,$00,$00,$0D,$0F,$0F,$0F,$0F,$0F,$0F,$0F,$0F,$0F
.BYTE $0F,$0E,$00,$0D,$00,$00,$0D,$0F,$0F,$0F,$0F,$0F,$0E,$00,$00,$00
.BYTE $15,$15,$00,$00,$16,$16,$00,$00,$15,$15,$00,$00,$16,$16,$00,$00
.BYTE $00,$00,$16,$16,$00,$00,$15,$15,$00,$00,$16,$16,$00,$00,$15,$15
.BYTE $00,$0B,$0B,$0B,$0B,$0B,$0B,$0B,$0B,$0B,$0B,$0B,$0B,$0B,$0B,$0B
.BYTE $0C,$0C,$0C,$0C,$0C,$0C,$0C,$0C,$0C,$0C,$0C,$0C,$0C,$0C,$0C,$0C
.BYTE $09,$09,$09,$09,$09,$09,$09,$09,$09,$15,$15,$09,$09,$09,$09,$09,$09
.BYTE $0A,$0A,$0A,$0A,$0A,$0A,$0A,$0A,$0A,$15,$0A,$0A,$0A,$0A,$0A,$0A
```

The one at index \$15 (21 in decimal) is this guy:

```
bonusPhaseMapRowDefinitions
.BYTE $00,$10,$0B,$11,$00,$00,$10,$0B,$11,$00,$00,$10,$0B,$11,$00
```

Each byte in this definition represents a square cell of four bytes. So for example, \$00 translates to:



And the first five bytes (which are repeated four times to complete the full row) translate as follows:

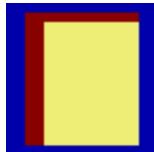


So how do we go from a single byte like \$00 to a four-byte square? Would you be surprised if I told you it involved another array of bytes? In fact it involves two arrays of bytes.

```
cellFirstColumnArray
.BYTE $40,$41,$44,$47,$48,$49,$4F,$4D
.BYTE $50,$51,$54,$56,$5B,$59,$5C,$5D
.BYTE $60,$61,$64,$65,$68,$69,$47,$47
.BYTE $4E,$4E,$57,$57,$5D,$5D,$20,$20
.BYTE $5D,$45,$4B,$47,$4C,$5D,$4E,$52
```

```
.BYTE $7C,$7D,$6C,$6D,$70,$71,$74,$75
.BYTE $78,$79
cellSecondColumnArray
.BYTE $42,$43,$46,$47,$4A,$48,$4E,$4F
.BYTE $51,$53,$56,$57,$5A,$5B,$5E,$5C
.BYTE $61,$63,$66,$67,$6A,$6B,$47,$47
.BYTE $4E,$4E,$57,$57,$5D,$5D,$20,$20
.BYTE $45,$47,$57,$4B,$4E,$4C,$52,$57
.BYTE $7E,$7F,$6E,$6F,$72,$73,$76,$77
.BYTE $7A,$7B
```

So let's look at how we from \$00 to ...



.. using the two arrays above. The value \$00 is treated as an index into each array, so it points us to the first byte in each. However we're not interested in just the first value in each array but the first two. So the bytes that we will use to construct the 4-byte square are:

```
cellFirstColumnArray
.BYTE $40,$41
cellSecondColumnArray
.BYTE $42,$43
```

Each of these bytes is a reference to a byte in the bonus phase character set. When we set out the characters in a table as they are eventually laid out we begin to get a sense of what we must do to turn them into our 4-byte cell:

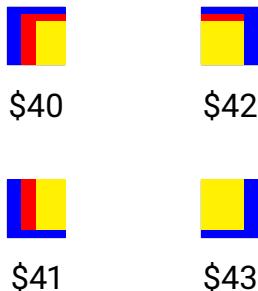


Figure 8.10: Characters making up the four-byte cell referenced by \$00.

What we have done is take the two values at index \$00 in `cellFirstColumnArray` and used each as the first column across two rows. Then we've taken the two values from `cellSecondColumnArray` and used them as the second column across two rows.

Let's repeat this process for the values given by index \$10 highlighted in red.

```
cellFirstColumnArray
    .BYTE $40,$41,$44,$47,$48,$49,$4F,$4D
    .BYTE $50,$51,$54,$56,$5B,$59,$5C,$5D
    .BYTE $60,$61,$64,$65,$68,$69,$47,$47
    ...
cellSecondColumnArray
    .BYTE $42,$43,$46,$47,$4A,$48,$4E,$4F
    .BYTE $51,$53,$56,$57,$5A,$5B,$5E,$5C
    .BYTE $61,$63,$66,$67,$6A,$6B,$47,$47
    ...
```

The values at index \$10 in `cellFirstColumnArray` are \$60,\$61 and in `cellSecondColumnArray` are \$61,\$63. Translated to row and column position this gives:

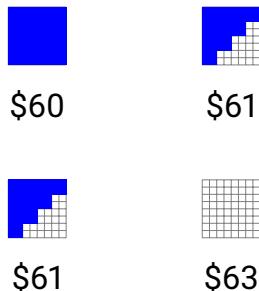
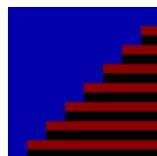


Figure 8.11: Characters making up the four-byte cell referenced by \$10.

Which gives us our second cell..



in the row:



The code that looks after all this is a routine we call `BonusPhaseFillTopLineAfterScrollUp` which is called every time the player scrolls up. There's an equivalent `BonusPhaseFillBottomLineAfterScrollDown` for when the player scrolls down. They're nearly identical.

```
BonusPhaseFillTopLineAfterScrollUp
LDX offsetForScrollUp
LDY bonusPhaseMapDefinition,X
LDA bonusPhaseMapLoPtrArray,Y
STA bonusPhaseMapLoPtr
LDA bonusPhaseMapHiPtrArray,Y
STA bonusPhaseMapHiPtr

LDY #$00
LDX #$00
FillRowLoop
LDA (bonusPhaseMapLoPtr),Y
STY mapOffsetTemp
ASL
CLC
ADC scrollLineOffset
TAY
LDA cellFirstColumnArray,Y
STA SCREEN.RAM,X
LDA cellSecondColumnArray,Y
STA SCREEN.RAM + LINE0.COL0,X
LDY mapOffsetTemp
INX
INX
INY
CPY #$14
BNE FillRowLoop

LDA scrollLineOffset
BNE ReturnEarly

INC offsetForScrollUp
INC offsetForScrollDown
ReturnEarly
RTS
```

```
BonusPhaseFillBottomLineAfterScrollDown
LDX offsetForScrollDown
LDY bonusPhaseMapDefinition,X
LDA bonusPhaseMapLoPtrArray,Y
STA bonusPhaseMapLoPtr
LDA bonusPhaseMapHiPtrArray,Y
STA bonusPhaseMapHiPtr

LDY #$00
LDX #$00
FillRowLoop
LDA (bonusPhaseMapLoPtr),Y
STY mapOffsetTemp
ASL
CLC
ADC scrollLineOffset
TAY
LDA cellFirstColumnArray,Y
STA SCREEN.RAM + LINE18.COL0,X
LDA cellSecondColumnArray,Y
STA SCREEN.RAM + LINE18.COL1,X
LDY mapOffsetTemp
INX
INX
INY
CPY #$14
BNE FillRowLoop

LDA scrollLineOffset
BEQ FillRowLoop

DEC offsetForScrollDown
DEC offsetForScrollUp
LDA offsetForScrollDown
CMP #$FF
BNE ReturnEarly

LDA #$00
STA offsetForScrollDown
LDA #$0A
STA offsetForScrollUp
ReturnEarly
RTS
```

The thing to note about this routine is that it only fills one actual line of characters at a time, whereas as the 'rows' we've defined are two lines deep. It decides which of the two lines its writing by using the `scrollLineOffset` variable to determine which one its writing.

8.2.1 Choosing a Map

So now that we understand how the individual rows of the map are generated, the question arises: how do we procedurally generate entire maps? Do we just pick random rows and join them together? This wouldn't work well, since some rows aren't going to go well together. The solution is to define entire map segments using the building blocks above and let those be the building blocks we use when constructing an entire map.

If we look at our definition for the first bonus phase again we can see it consists of arrays of 10 bytes with each 10-byte array corresponding to a segment in the map and each byte in the array corresponding to a row in the map. So each 10-byte array below gives us a full 20 byte high screen of map data.

```

bonusPhaseMapDefinition
.BYTE $00,$00,$00,$00,$00,$00,$00,$00,$00,$00
.BYTE $11,$11,$11,$11,$11,$11,$11,$11,$11,$11
.BYTE $11,$11,$11,$11,$11,$11,$11,$11,$11,$11
.BYTE $15,$16,$16,$16,$16,$16,$16,$16,$16,$17
.BYTE $15,$16,$16,$16,$16,$16,$16,$16,$16,$17
.BYTE $13,$13,$13,$13,$13,$13,$13,$13,$13,$13
.BYTE $15,$16,$16,$16,$16,$16,$16,$16,$16,$17
.BYTE $14,$14,$00,$15,$16,$17,$00,$00,$14,$14
.BYTE $00,$14,$14,$14,$14,$14,$14,$14,$14,$00
.BYTE $00,$00,$00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$14,$14,$14,$14,$14,$14,$14,$14,$00
.BYTE $14,$14,$00,$15,$16,$17,$00,$00,$14,$14
.BYTE $15,$16,$16,$16,$16,$16,$16,$16,$16,$17
.BYTE $00,$14,$14,$14,$14,$14,$14,$14,$14,$00
.BYTE $00,$15,$16,$17,$00,$00,$15,$16,$17,$00
.BYTE $13,$13,$13,$13,$13,$13,$13,$13,$13,$13
.BYTE $15,$16,$16,$16,$16,$16,$16,$16,$16,$17
.BYTE $11,$11,$11,$11,$11,$11,$11,$11,$11,$11
.BYTE $00,$00,$00,$00,$00,$00,$00,$00,$00,$00
.BYTE $12,$12,$12,$12,$12,$12,$12,$12,$12,$00
.BYTE $00,$15,$16,$17,$00,$00,$15,$16,$17,$00
.BYTE $00,$15,$16,$17,$00,$00,$15,$16,$17,$00
.BYTE $13,$13,$13,$13,$13,$13,$13,$13,$13,$13
.BYTE $12,$12,$12,$12,$12,$12,$12,$12,$12,$00
.BYTE $11,$11,$11,$11,$11,$11,$11,$11,$11,$11
.BYTE $10,$10,$10,$10,$10,$10,$10,$10,$10,$10

```

The trick is that these segments aren't themselves generated procedurally, we defined them in advance. We did this in `bonusMapSegmentArray` given below. We've highlighted the segments used in our map in red:

```

bonusMapSegmentArray
.BYTE $00,$00,$00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$15,$16,$17,$00,$00,$15,$16,$17,$00
.BYTE $00,$14,$14,$14,$14,$14,$14,$14,$14,$00
.BYTE $11,$11,$11,$11,$11,$11,$11,$11,$11,$11
.BYTE $13,$13,$13,$13,$13,$13,$13,$13,$13,$13
.BYTE $12,$12,$12,$12,$12,$12,$12,$12,$12,$00
.BYTE $14,$14,$00,$15,$16,$17,$00,$00,$14,$14
.BYTE $15,$16,$16,$16,$16,$16,$16,$16,$16,$17
.BYTE $00,$00,$00,$00,$00,$00,$00,$00,$00,$00
.BYTE $00,$00,$0F,$0F,$0F,$0F,$0F,$0F,$00,$00
.BYTE $01,$01,$01,$01,$01,$00,$00,$01,$01,$01
.BYTE $00,$00,$0B,$0B,$0B,$0C,$0C,$0C,$00,$00
.BYTE $00,$02,$03,$04,$05,$06,$07,$08,$09,$0A
.BYTE $02,$03,$04,$05,$05,$05,$05,$0B,$0B,$0B
.BYTE $00,$00,$01,$01,$00,$00,$01,$01,$00,$00
.BYTE $00,$00,$0E,$0D,$00,$00,$0E,$0D,$00,$00
.BYTE $00,$02,$03,$04,$05,$08,$09,$0A,$0B,$00
.BYTE $00,$00,$00,$1A,$1A,$1A,$18,$18,$18,$18
.BYTE $00,$00,$00,$1A,$1A,$1A,$19,$19,$19,$19

```

```
.BYTE $00,$00,$18,$18,$00,$00,$00,$19,$19  
.BYTE $00,$00,$1B,$1B,$00,$00,$15,$16,$17,$00  
.BYTE $15,$16,$17,$1D,$1D,$15,$16,$17,$1D,$1D  
.BYTE $14,$14,$1E,$1E,$00,$00,$15,$16,$17,$00  
.BYTE $00,$0B,$0B,$0B,$15,$16,$17,$15,$16,$17  
.BYTE $00,$00,$1D,$1D,$1D,$1E,$1E,$1E,$1E  
.BYTE $00,$00,$20,$1F,$20,$1F,$00,$00,$11,$11  
.BYTE $00,$00,$20,$1F,$20,$1F,$20,$1F,$20,$1F  
.BYTE $00,$1E,$1E,$1E,$20,$1F,$1D,$1D,$1D,$00  
.BYTE $00,$0C,$0C,$0C,$15,$16,$17,$00,$00,$00  
.BYTE $00,$02,$03,$04,$05,$08,$09,$0A,$0B,$00  
.BYTE $00,$00,$06,$06,$06,$11,$11,$11,$00,$00  
.BYTE $00,$00,$0F,$0F,$15,$16,$17,$15,$16,$17
```

By way of example this is what the last segment in the list above looks like when rendered as a section of our map:

Index	Image
\$17	
\$16	
\$15	
\$17	
\$16	
\$15	
\$0F	
\$0F	
\$00	
\$00	

So our procedure for generating a map is simply choosing 30 of these segments in a pseudo-random order and stacking them on top of one another!

8.3 Some Very Ugly Sprites

For some reason the gilby sprite in the bonus phase is made impossibly ugly.

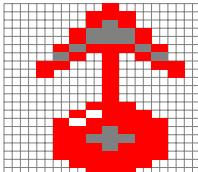
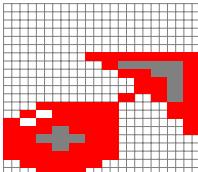
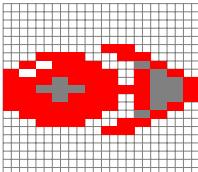
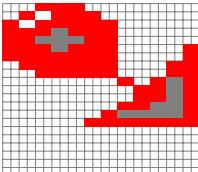
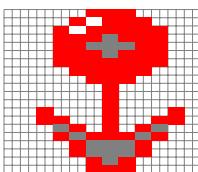
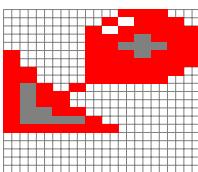
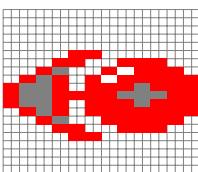
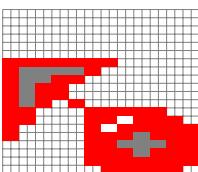
Please	Make	It	Stop
			
UGLY_GILBY1	UGLY_GILBY2	UGLY_GILBY3	UGLY_GILBY4
			
UGLY_GILBY5	UGLY_GILBY6	UGLY_GILBY7	UGLY_GILBY8

Figure 8.12: There is no excuse for this.

8.4 IBalls

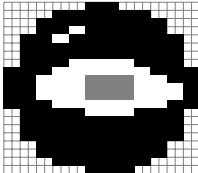
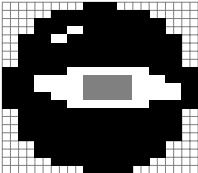
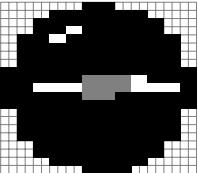
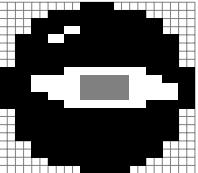
IBall	Another One	A Third	One More
			
BONUS_IBALL1	BONUS_IBALL2	BONUS_IBALL3	BONUS_IBALL4

Figure 8.13: OK this is better.

A Hundred Thousand Billion Theme Tunes

The theme music in Iridis Alpha is procedurally generated. There isn't a chunk of music data that the game plays every time you visit the title screen. Instead a new tune is generated for every visit. There's a distinction to be made here between procedural and random. The music isn't random: the first time you launch Iridis Alpha, and every subsequent time you launch it, you will hear the same piece of music. But as you let the game's attract mode cycle through and return to the title screen you will hear a new, different piece of music. Iridis Alpha has an infinite number of these tunes and it plays them in the same order every time you launch it and as it loops through the title sequence waiting for you to play.

Because the music is generated procedurally, and not randomly, you will hear the same sequence of tunes every time you launch the game so it appears to you as if the music was composed in advance and stored in the game waiting its turn. This is not the case.

Each piece of music is generated dynamically using the same algorithm but because the logic is chaotic enough, the smallest difference in the initial values fed into it will result in a completely different tune being generated.

The routine responsible for creating this music is remarkably short so I've reproduced it here in full before we start to dive in and try to understand what's going on.

```
PlayTitleScreenMusic
    DEC baseNoteDuration
    BEQ MaybeStartNewTune
    RTS

    MaybeStartNewTune
```

```
LDA previousBaseNoteDuration
STA baseNoteDuration

DEC numberOfNotesToPlayInTune
BNE MaybePlayVoice1

; Set up a new tune.
LDA #$C0 ; 193
STA numberOfNotesToPlayInTune

; This is what will eventually time us out of playing
; the title music and enter attract mode.
INC f7PressedOrTimedOutToAttractMode

LDX notesPlayedSinceLastKeyChange
LDA titleMusicNoteArray,X
STA offsetForNextVoice1Note

; We'll only select a new tune when we've reached the
; beginning of a new 16 bar structure.
INX
TXA
AND #$03
STA notesPlayedSinceLastKeyChange
BNE MaybePlayVoice1

JSR SelectNewNotesToPlay

MaybePlayVoice1
DEC voice1NoteDuration
BNE MaybePlayVoice2

LDA #$30
STA voice1NoteDuration

LDX voice1IndexToMusicNoteArray
LDA titleMusicNoteArray,X
CLC
ADC offsetForNextVoice1Note
TAY
STY offsetForNextVoice2Note

JSR PlayNoteVoice1

INX
TXA
AND #$03
STA voice1IndexToMusicNoteArray

MaybePlayVoice2
DEC voice2NoteDuration
BNE MaybePlayVoice3

LDA #$0C
```

```
STA voice2NoteDuration
LDX voice2IndexToMusicNoteArray
LDA titleMusicNoteArray,X
CLC
ADC offsetForNextVoice2Note

; Use this new value to change the key of the next four
; notes played by voice 3.
STA offsetForNextVoice3Note

TAY
JSR PlayNoteVoice2
INX
TXA
AND #$03
STA voice2IndexToMusicNoteArray

MaybePlayVoice3
DEC voice3NoteDuration
BNE ReturnFromTitleScreenMusic

LDA #$03
STA voice3NoteDuration

; Play the note currently pointed to by
; voice3IndexToMusicNoteArray in titleMusicNoteArray.
LDX voice3IndexToMusicNoteArray
LDA titleMusicNoteArray,X
CLC
ADC offsetForNextVoice3Note
TAY
JSR PlayNoteVoice3

; Move voice3IndexToMusicNoteArray to the next
; position in titleMusicNoteArray.
INX
TXA
; Since it's only 4 bytes long ensure we wrap
; back to 0 if it's greater than 3.
AND #$03
STA voice3IndexToMusicNoteArray

ReturnFromTitleScreenMusic
RTS
```

Listing 9.1: Routine responsible for playing the title tune.

9.1 Some Basics

The rudiments of playing music on the Commodore 64 are simple. It has a powerful-for-its-time sound chip that has 3 tracks or 'voices'. You can play any note across 8 octaves on each of these voices together or separately. There are a whole bunch of settings you can apply to each voice to determine the way the note sounds. We'll cover a couple of these settings here but when it comes to playing music these extra settings aren't so important. They're much more useful when generating sound effects.

Playing a note on one of the voices consists of loading a two-byte value into the location (or 'register') associated with that voice. Here's the routine in Iridis used to play a note for the theme tune on Voice '1':

```
PlayNoteVoice1
    LDA #$21
    STA $D404      ;Voice 1: Control Register
    LDA titleMusicLowBytes,Y
    STA $D400      ;Voice 1: Frequency Control - Low-Byte
    LDA titleMusicHiBytes,Y
    STA $D401      ;Voice 1: Frequency Control - High-Byte
    RTS
```

Listing 9.2: Plays a note on Voice 1. The routine is supplied with a value in Y that indexes into two arrays containing the first (Hi) and second (Lo) byte respectively associated with the selected note.

Once the selected bytes have been loaded into \$D400 and \$D401 the new note will start playing. It's as blunt an instrument as that. (Well not quite, we'll cover some other gory details soon).

The full list of available notes is given in the C64 Progammer's Reference Manual. I've adapted and reproduced it below.

CHAPTER 9. A HUNDRED THOUSAND BILLION THEME TUNES

Octave	Note	High Byte	Low Byte	Octave	Note	High Byte	Low Byte	Octave	Note	High Byte	Low Byte
0	C	\$01	\$0C	2	G#	\$06	\$A7	5	E	\$2A	\$3E
0	C#	\$01	\$1C	2	A	\$07	\$0C	5	F	\$2C	\$C1
0	D	\$01	\$2D	2	A#	\$07	\$77	5	F#	\$2F	\$6B
0	D#	\$01	\$3E	2	B	\$07	\$E9	5	G	\$32	\$3C
0	E	\$01	\$51	3	C	\$08	\$61	5	G#	\$35	\$39
0	F	\$01	\$66	3	C#	\$08	\$E1	5	A	\$38	\$63
0	F#	\$01	\$7B	3	D	\$09	\$68	5	A#	\$3B	\$BE
0	G	\$01	\$91	3	D#	\$09	\$F7	5	B	\$3F	\$4B
0	G#	\$01	\$A9	3	E	\$0A	\$8F	6	C	\$43	\$0F
0	A	\$01	\$C3	3	F	\$0B	\$30	6	C#	\$47	\$OC
0	A#	\$01	\$DD	3	F#	\$0B	\$DA	6	D	\$4B	\$45
0	B	\$01	\$FA	3	G	\$0C	\$8F	6	D#	\$4F	\$BF
1	C	\$02	\$18	3	G#	\$0D	\$4E	6	E	\$54	\$7D
1	C#	\$02	\$38	3	A	\$0E	\$18	6	F	\$59	\$83
1	D	\$02	\$5A	3	A#	\$0E	\$EF	6	F#	\$5E	\$D6
1	D#	\$02	\$7D	3	B	\$0F	\$D2	6	G	\$64	\$79
1	E	\$02	\$A3	4	C	\$10	\$C3	6	G#	\$6A	\$73
1	F	\$02	\$CC	4	C#	\$11	\$C3	6	A	\$70	\$C7
1	F#	\$02	\$F6	4	D	\$12	\$D1	6	A#	\$77	\$7C
1	G	\$03	\$23	4	D#	\$13	\$EF	6	B	\$7E	\$97
1	G#	\$03	\$53	4	E	\$15	\$1F	7	C	\$86	\$1E
1	A	\$03	\$86	4	F	\$16	\$60	7	C#	\$8E	\$18
1	A#	\$03	\$BB	4	F#	\$17	\$B5	7	D	\$96	\$8B
1	B	\$03	\$F4	4	G	\$19	\$1E	7	D#	\$9F	\$7E
2	C	\$04	\$30	4	G#	\$1A	\$9C	7	E	\$A8	\$FA
2	C#	\$04	\$70	4	A	\$1C	\$31	7	F	\$B3	\$06
2	D	\$04	\$B4	4	A#	\$1D	\$DF	7	F#	\$BD	\$AC
2	D#	\$04	\$FB	4	B	\$1F	\$A5	7	G	\$C8	\$F3
2	E	\$05	\$47	5	C	\$21	\$87	7	G#	\$D4	\$E6
2	F	\$05	\$98	5	C#	\$23	\$86	7	A	\$E1	\$8F
2	F#	\$05	\$ED	5	D	\$25	\$A2	7	A#	\$EE	\$F8
2	G	\$06	\$47	5	D#	\$27	\$DF	7	B	\$FD	\$2E

Figure 9.1: All available notes on the C64 and their corresponding hi/lo byte values. Note that Iridis Alpha only uses octaves 3 to 7. The available notes in octaves 1 to 2 are never used.

With 96 notes in total available, Iridis only uses 72 of them, omitting the 2 lowest octaves. We can see this when we look at the note table in the game. This pair of arrays are where the title music logic plucks the note to be played once it has dynamically selected one:

```

titleMusicHiBytes : C C# D D# E F F# G G# A A# B
.BYTE $08, $08, $09, $09, $0A, $0B, $0B, $0C, $0D, $0E, $0E, $0F ; 4
.BYTE $10, $11, $12, $13, $15, $16, $17, $19, $1A, $1C, $1D, $1F ; 5
.BYTE $21, $23, $25, $27, $2A, $2C, $2F, $32, $35, $38, $3B, $3F ; 6
.BYTE $43, $47, $4B, $4F, $54, $59, $5E, $64, $6A, $70, $77, $7E ; 7
.BYTE $86, $8E, $96, $9F, $A8, $B3, $BD, $C8, $D4, $E1, $EE, $FD ; 8

titleMusicLowBytes : C C# D D# E F F# G G# A A# B
.BYTE $61, $E1, $68, $F7, $8F, $30, $DA, $8F, $4E, $18, $EF, $D2 ; 4
.BYTE $C3, $C3, $D1, $EF, $1F, $60, $B8, $1E, $9C, $31, $DF, $A5 ; 5
.BYTE $87, $86, $A2, $DF, $3E, $C1, $6B, $3C, $39, $63, $BE, $4B ; 6
.BYTE $0F, $0C, $45, $BF, $7D, $83, $D6, $79, $73, $C7, $97, $97 ; 7
.BYTE $1E, $18, $8B, $7E, $FA, $06, $AC, $F3, $E6, $BF, $F8, $2E ; 8

```

Listing 9.3: The lookup table for all of the notes used in the theme music. The two lowest available octaves are not used by the game. To see this for yourself compare the first entry in titleMusicHiBytes/titleMusicLowBytes (\$08 and \$61 giving \$0861) with the entry highlighted in red in the previous table.

So now that we know where the notes are and how to make them go beep we just have to figure out the order that PlayTitleScreenMusic contrives to play them.

It would certainly help if we could see what the music looks like, so let's do that. Here is the opening title tune as sheet music in Western notation.

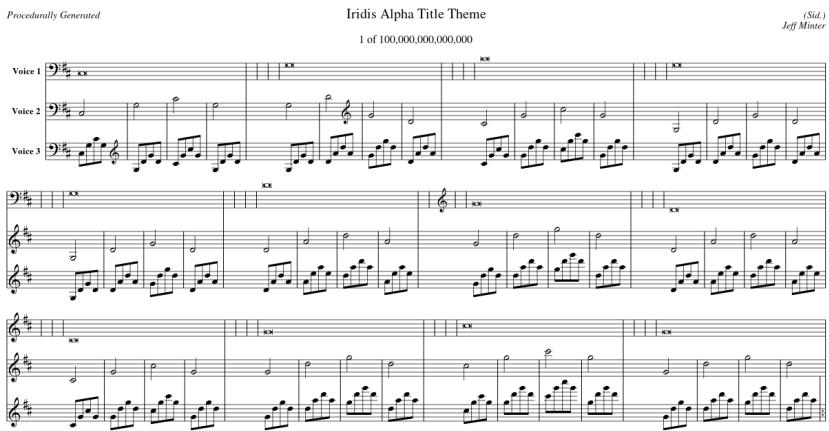


Figure 9.2: The first title tune in Iridis Alpha.

9.1.1 Structure

Even if you can't read sheet music notation some structure should be evident.

Voice 3 carries the main melody.

Iridis Alpha: 1 of 100,000,000,000,000
First 4 Bars of Voice 3

A close-up view of the first four bars of the musical score for Voice 3. The score is in Western notation with a treble clef. The first bar begins with a quarter note followed by a rest. The second bar begins with a half note followed by a quarter note. The third bar begins with a half note followed by a quarter note. The fourth bar begins with a half note followed by a quarter note. The music consists of eighth and sixteenth note patterns.

For every 4 notes Voice 3 plays, Voice 2 chimes in with a new note that it sustains until the next one.

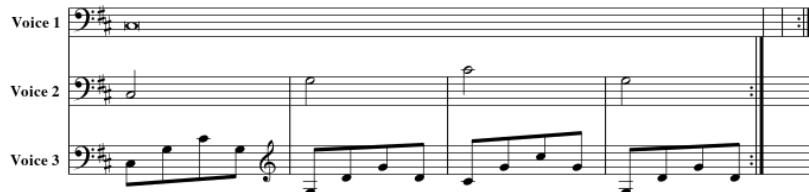
CHAPTER 9. A HUNDRED THOUSAND BILLION THEME TUNES

Iridis Alpha: 1 of 100,000,000,000,000
First 4 Bars of Voice 3 and Voice 2



Voice 1 does the same for every 16 notes that Voice 3 plays and every 4 notes of Voice 2..

Iridis Alpha: 1 of 100,000,000,000,000
Voice 1, 2 and 3.



Armed with this insight we can see it reflected in the logic in `PlayTitleScreenMusic`. This routine is called regularly by a system interrupt, a periodic wake-up call performed by the C64 CPU. So multiple times every second it is run and must figure out what new notes, if any, to play on each of the three voices.

Here it is deciding whether or not to play new note on Voice 1:

```
MaybePlayVoice1
DEC voice1NoteDuration
BNE MaybePlayVoice2

LDA #$30
STA voice1NoteDuration

LDX voice1IndexToMusicNoteArray
LDA titleMusicNoteArray,X
CLC
ADC offsetForNextVoice1Note
TAY
STY offsetForNextVoice2Note

JSR PlayNoteVoice1

INX
TXA
AND #$03
STA voice1IndexToMusicNoteArray
```

Listing 9.4: `MaybePlayVoice1` part of `PlayTitleScreenMusic`.

voice1NoteDuration is used to count the interval between notes on Voice 1. It's decremented on each visit and when it reaches zero it gets reset to 48 (\$30) and a note is played. What's being counted here isn't seconds, it's cycles or 'interrupts'. So this translates to only a few seconds between notes being played.

The same is done for both Voice 2 and Voice 3 but the intervals are shorter: 12 (\$0C) and 3 (\$03). This matches the relationship we see in the sheet music, one note in Voice 1 for every sixteen in Voice 3 ($48/3=16$) and one note in Voice 2 for every four in Voice 3 ($12/3=4$).

```
MaybePlayVoice2
    DEC voice2NoteDuration
    BNE MaybePlayVoice3

    LDA #$0C
    STA voice2NoteDuration
    LDX voice2IndexToMusicNoteArray
    LDA titleMusicNoteArray,X
    CLC
    ADC offsetForNextVoice2Note

    ; Use this new value to change the key of the next four
    ; notes played by voice 3.
    STA offsetForNextVoice3Note

    TAY
    JSR PlayNoteVoice2
    INX
    TXA
    AND #$03
    STA voice2IndexToMusicNoteArray
```

Listing 9.5: MaybePlayVoice2 part of PlayTitleScreenMusic.

```
MaybePlayVoice3
    DEC voice3NoteDuration
    BNE ReturnFromTitleScreenMusic

    LDA #$03
    STA voice3NoteDuration

    ; Play the note currently pointed to by
    ; voice3IndexToMusicNoteArray in titleMusicNoteArray.
    LDX voice3IndexToMusicNoteArray
    LDA titleMusicNoteArray,X
    CLC
    ADC offsetForNextVoice3Note
    TAY
    JSR PlayNoteVoice3

    ; Move voice3IndexToMusicNoteArray to the next
```

```
; position in titleMusicNoteArray.
INX
TXA
; Since it's only 4 bytes long ensure we wrap
; back to 0 if it's greater than 3.
AND #$03
STA voice3IndexToMusicNoteArray
```

Listing 9.6: MaybePlayVoice3 part of PlayTitleScreenMusic.

Extracting the Title Music

Since each tune is dynamically generated there's nowhere for us to pull them from. We could record the tunes as audio files and maybe extract something useful that way. A feature of Vice, the C64 emulator, allows us to do something much simpler. We can log every note that's played to a text file and use that trace to reconstruct the tunes.

We launch Iridis Alpha with x64 using the following command:

```
x64 -moncommands moncommands.txt orig/iridisalpha.prg
```

The `moncommands.txt` file contains a series of debugger directives that tells x64 to log every value stored to the music registers at \$D400–\$D415. This will capture all notes played on all three voices as well as any updates made to the other sound parameters and write them to `IridisAlphaTitleMusicAll.txt`:

```
log on
logname "IridisAlphaTitleMusicAll.txt"
tr store D400 D415
```

We end up with `IridisAlphaTitleMusicAll.txt` full of lines like:

```
TRACE: 1 C:$d400-$d415 (Trace store)
#1 (Trace store d400) 279 052
.C:1598 8D 00 D4 STA $D400 - A:61 X:00 Y:00 SP:e8 ..-..I.. 96135469
#1 (Trace store d401) 279 060
.C:159e 8D 01 D4 STA $D401 - A:08 X:00 Y:00 SP:e8 ..-..I.. 96135477
#1 (Trace store d40b) 280 059
```

This examples gives us the value in A written to each register for Voice 1. For example, \$61 has been written to \$D400 and \$08 has been written to \$D401.

We can now write a short Python notebook that parses this file and for each tune constructs three arrays, each representing a voice, with the sequence of notes played to each. For example, in the extract above we can extract \$0861 as the note 'C' in octave 3 played on Voice 1 (\$D400–\$D401). (Refer to the tables above to see why \$0861 translates to 'C-3'.

With the sequence of notes in three arrays, each representing one of the 3 voices, it is a simple matter to transform this into ABC format, a music notation frequently used for traditional music.

```

%abc-2.2
%pagewidth 35cm
%header "Example page: $P"
%footer "$T"
%gutter .5cm
%barsperstaff 16
%titleformat R-P-Q-T C1 O1, T+T N1
%composerspace 0
X: 2 % start of header
T:Iridis Alpha Title Theme
T:1 of 100,000,000,000,000
C: (Sid.)
O: Jeff Minter
R: Procedurally Generated
L: 1/8
K: D % scale: C major
V:1 name="Voice 1"
C,16 | | | | G,16 | | | | C16 | | | | G,16 | | | |
      | G,16 | | | | D16 | | | | G16 | | | | c16 | | | | | | |
      | C16 | | | | G16 | | | | G16 | | | | G16 | | | |
      | : | | | : | | | | : | | | | : | | | | : | | | |
V:2 name="Voice 2"
C,4 | G,4 | C4 | G,4 | G,4 | D4 | D4 | G4 | D4 | C4 | G4 | c4 | G4 | G,4 | D4
      | G4 | D4 | G,4 | D4 | G4 | D4 | D4 | A4 | d4 | A4 | G4 | G4 | d4 | g4 |
      | d4 | D4 | A4 | d4 | A4 | C4 | G4 | c4 | G4 | G4 | d4 | g4 | d4 |
      | c4 | g4 | c'4 | g4 | G4 | d4 | g4 | d4 | G4 | G4 | d4 | g4 | : |
V:3 name="Voice 3"
C,1G,1C1G,1G,1D1G1D1|C1G1c1G1|G,1D1G1D1|G,1D1G1D1|D1A1d1A1|G1d1g1d1|D1A1d1A1|C1G1c1G1|G1d1g1d1|c1G1c1G1|
      G1d1g1d1|G,1D1G1D1|D1A1d1A1|G1d1g1d1|D1A1d1A1|G,1D1G1D1|D1A1d1A1|G1d1g1d1|D1A1d1A1|D1A1d1A1|A1e1a1e1|
      d1a1d1d1|a1e1a1e1|G1d1g1d1|d1a1d1d1|a1|g1|'g'1'd'|d1a1d1d1|a1|d1a1d1A1|a1e1a1e1|d1a1d1d1|a1e1a1e1|c1G1c1G1
      |G1d1g1d1|c1G1c1G1|G1d1g1d1|G1d1g1d1|d1a1d1d1|a1|g1|'g'1'd'|d1a1d1d1|a1|d1a1d1A1|a1e1a1e1|c1G1c1G1
      1|g1|'g'1'd'|d1a1d1d1|G1d1g1d1|d1a1d1d1|a1|g1|'g'1'd'|d1a1d1d1|a1|d1a1d1A1|a1e1a1e1|c1G1c1G1
      1|g1|'g'1'd'|d1a1d1d1|G1d1g1d1|d1a1d1d1|a1|g1|'g'1'd'|d1a1d1d1|a1|d1a1d1A1|a1e1a1e1|c1G1c1G1

```

Listing 9.7: Title Tune No 1 in ABC format

We can then use the tool ‘`abcm2ps`’ to transform this into an SVG image file giving the music in standard Western notation.

9.1.2 Phrasing

Now that we've identified the underlying 4-bar structure of the arrangement. We can take a closer look at the phrasing of the individual parts. Voice 3 has a simple repetitive structure for each 4-bar phrase:

Iridis Alpha: 1 of 100,000,000,000,000
 Three 4-bar phrases from Voice 3.



Figure 9.3: Bars 2 and 4 are always repeated

Bars 2 and 4 are repeated. Each bar consists of the same tonic formula: three notes rising two notes at a time, falling back on the final note. The difference between bars 1 and 3 is a simple key change.

This 4 note basis is driven by the 4 bytes in `titleMusicNoteArray`. Generating the music for Voice 3 consists of calculating and loading 4 values into this array and using them as an index into `titleMusicLowBytes/titleMusicHiBytes` to play the actual note.

```
; This seeds the title music. Playing around with these first
; four bytes alters the first few seconds of the title music.
; The routine for the title music uses these 4 bytes to determine
; the notes to play.
; This array is periodically replenished from titleMusicSeedArray by
; SelectNewNotesToPlay.
titleMusicNoteArray .BYTE $00,$07,$0C,$07
```

Notice how the values populated in `titleMusicNoteArray` at start-up match the structure of our basic tonic formula, e.g. C3-G3-C4-G3.

<code>titleMusicNoteArray</code>	<code>titleMusicHiBytes</code>	<code>titleMusicLowBytes</code>	Note
	\$00	\$61	C-3
	\$07	\$0C	G-3
	\$0C	\$10	G-3
	\$07	\$0C	G-3

Figure 9.4: The value in `titleMusicNoteArray` is an index into `titleMusicHiBytes/titleMusicLowBytes`.

Playing the 4 note phrase we've stored in this array is done here:

```
; Play the note currently pointed to by
```

```
; voice3IndexToMusicNoteArray in titleMusicNoteArray.  
LDX voice3IndexToMusicNoteArray  
LDA titleMusicNoteArray,X  
CLC  
ADC offsetForNextVoice3Note  
TAY  
JSR PlayNoteVoice3
```

The variable that's doing a bit of extra work here is `offsetForNextVoice3Note`. This is what's shifting the notes for subsequent bars from the base position of C3-G3-C4-G3 to G3-D4-G4-D4. This value has to get updated after every four notes, otherwise we just keep playing the same four notes over and over again.

The obvious place to do this is when play a note on Voice 2, which is something we're already doing every 4 notes in Voice 3.

```
MaybePlayVoice2  
    DEC voice2NoteDuration  
    BNE MaybePlayVoice3  
  
    LDA #$0C  
    STA voice2NoteDuration  
    LDX voice2IndexToMusicNoteArray  
    LDA titleMusicNoteArray,X  
    CLC  
    ADC offsetForNextVoice2Note  
  
    ; Use this new value to change the key of the next four  
    ; notes played by voice 3.  
    STA offsetForNextVoice3Note  
  
    TAY  
    JSR PlayNoteVoice2  
    INX  
    TXA  
    AND #$03  
    STA voice2IndexToMusicNoteArray
```

As we can see the mechanics of playing a note for Voice 2 are otherwise the same as Voice 3. We're playing the same phrase encoded in `titleMusicNoteArray` that is played by Voice 3 but just over a longer period of time. And if you look closely again at the first four bars of the first title tune you can see that Voice 2 is in fact playing the exact same 4 notes of the first bar of Voice 3.

Iridis Alpha: 1 of 100,000,000,000,000
Voice 1, 2 and 3.

A musical score for three voices (Voice 1, Voice 2, and Voice 3) over four bars. The score is in common time with a key signature of one sharp. Voice 1 starts with a quarter note followed by a eighth-note pair. Voice 2 starts with a eighth-note pair. Voice 3 starts with a eighth-note pair. In the second bar, Voice 1 has a eighth-note pair, Voice 2 has a quarter note, and Voice 3 has a eighth-note pair. In the third bar, Voice 1 has a eighth-note pair, Voice 2 has a eighth-note pair, and Voice 3 has a eighth-note pair. In the fourth bar, Voice 1 has a eighth-note pair, Voice 2 has a eighth-note pair, and Voice 3 has a eighth-note pair. A blue box highlights the notes of Voice 1 and Voice 2 in the first two bars. A red box highlights the notes of Voice 3 in the first bar.

The same thing happens for Voice 1: it is playing the same notes as the first bar of Voice 3 but over 16 bars (1 every 4 bars).

So ultimately what we have underlying every tune generated by Iridis Alpha is a 16-bar structure where the same 4 notes are played by Voice 3 in its first bar, Voice 2 in its first 4 bars, and Voice 1 over the full 16 bars. This structure recurs every 16 bars, each time using the 4 initial notes from Voice 3.

A musical score for three voices (Voice 1, Voice 2, and Voice 3) over 16 bars. The score is in common time with a key signature of one sharp. The structure is as follows: Bars 1-4: Voice 3 (bottom) plays a eighth-note pair. Voice 2 (middle) plays a eighth-note pair. Voice 1 (top) plays a eighth-note pair. Bars 5-8: Voice 2 (middle) plays a eighth-note pair. Voice 1 (top) plays a eighth-note pair. Bars 9-12: Voice 1 (top) plays a eighth-note pair. Bars 13-16: Voice 1 (top) plays a eighth-note pair. The score shows how the initial 4-bar phrase of Voice 3 is picked up by Voice 2, and the 16th bar phrase is picked up by Voice 1.

Figure 9.5: A full 16 bar passage showing the nested structure of Voices 1 and 2

This is a nested structure with the initial musical phrase that occurs every 4 bars in Voice 3 being picked up by Voice 2 and the one that occurs at every 16th bar being picked up by Voice 1.

The second, finer-grained structure of each tune lies in Voice 3 and consists of selecting a fundamental 4 note pattern (as we discussed above) and applying that same pattern to the key change between each 4 note phrase!

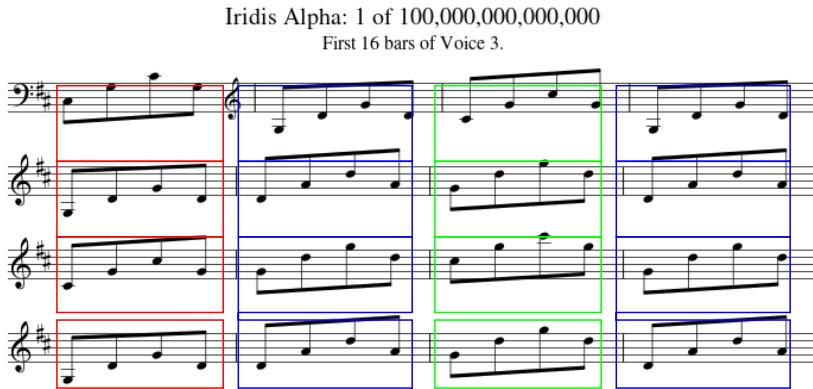


Figure 9.6: The G3-C4-G4-C4 pattern used to construct the 4 note pattern is also used to construct the key changes in each 4-bar sequence (red-blue-green-blue).

This is why we observed the repeating structure of Bars 2 and 4 earlier! It's the same pattern used to construct the 4 note formula.

But how do we choose the key for the start of each 4-bar pattern? By applying the same pattern to the start of each 4-bar section!

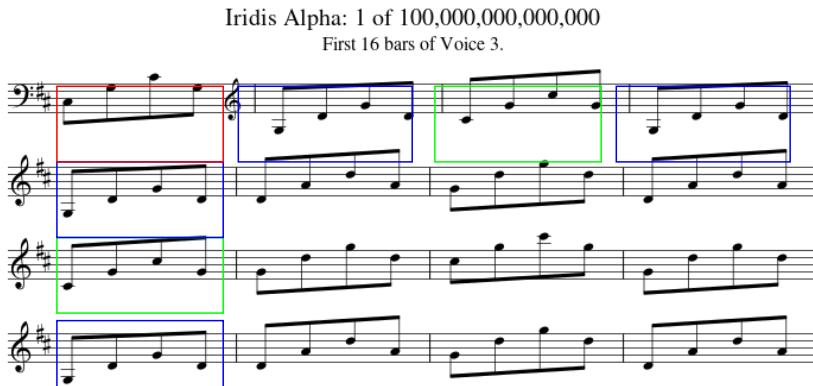


Figure 9.7: The start of each 4 bar pattern in a 16 bar cycle uses each of the 4-note patterns from the first 4 bars.

If we look at two other procedurally generated tunes we can see the same pattern:

Iridis Alpha: 12 of 100,000,000,000,000

Four 4-bar phrases from Voice 3.



Iridis Alpha: 18 of 100,000,000,000,000

Four 4-bar phrases from Voice 3.



Figure 9.8: The same patterns in Tunes 12 and 18.

9.1.3 Seeding the Random

We've established how each tune is built entirely off the same 4-byte sequence, all the way from selecting notes to play to filling out the larger structure of the tune at almost every level. What remains is to see how this 4-byte sequence is selected. We know it's not entirely random since if it was, none of us would ever hear the same tune.

The selection of our 4-byte structure for each tune happens in `SelectNewNotesToPlay`. Once a seed value has been plucked, this is used as an index into `titleMusicSeedArray` and the next four values are populated into our magic 4-byte sequence that determines everything `titleMusicNoteArray`.

```
SelectNewNotesToPlay
    ; Get a random value between 0 and 15.
    JSR PutProceduralByteInAccumulator
    AND #$0F
    ; Jump to InitializeSeedLoop if it's zero.
    BEQ InitializeSeedLoop

    ; Otherwise multiply it by 4. We do this so that
    ; the 4-byte sequence we choose always starts at
    ; a 4-byte offset in titleMusicSeedArray.
    TAX
    LDA #$00
MultiplyRandomNumBy4
    CLC
    ADC #$04
    DEX
    BNE MultiplyRandomNumBy4
```

Listing 9.8: Put a seed byte in the accumulator and multiply this by 4 if it's not zero. This gives us what we need for the next step.

```
InitializeSeedLoop
    ; Put our random number in Y and use it as index into
    ; the seed array.
    TAY
    ; Initialize X to 0, we will use this to iterate up to
    ; 4 bytes for pulling from titleMusicSeedArray.
    LDX #$00

    ; Pick the first 4 bytes in titleMusicSeedArray from our
    ; randomly chosen offset and put them in
    ; titleMusicNoteArray.
MusicSeedArrayLoop
    LDA titleMusicSeedArray,Y
    STA titleMusicNoteArray,X
    INY
    INX
    CPX #$04
    BNE MusicSeedArrayLoop
```

Listing 9.9: Use our seed value to pull 4 bytes from titleMusicSeedArray and store them in titleMusicNoteArray

```
; This is used to replenish titleMusicNoteArray with seed values
; for the procedurally generated title screen music.
titleMusicSeedArray .BYTE $00,$03,$06,$08
                    .BYTE $00,$0C,$04,$08
                    .BYTE $00,$07,$00,$05
                    .BYTE $05,$00,$00,$05
                    .BYTE $00,$06,$09,$05
                    .BYTE $02,$04,$03,$04
                    .BYTE $03,$07,$03,$00
                    .BYTE $04,$08,$0C,$09
```

```
.BYTE $07,$08,$04,$07  
.BYTE $00,$04,$07,$0E  
.BYTE $00,$00,$00,$07  
.BYTE $07,$04,$00,$0C  
.BYTE $04,$07,$00,$0C  
.BYTE $07,$08,$0A,$08  
.BYTE $0C,$00,$0C,$03  
.BYTE $0C,$03,$07,$00
```

Listing 9.10: Our seed bank for 4-byte sequences. It's 64 bytes long giving 16 possible sequences in all.

The real source of variety here is this 'seed value' that we pluck at the very start of the process. This is done by `PutProceduralByteInAccumulator` at the very start of `SelectNewNotesToPlay`.

```
;-----  
; PutProceduralByteInAccumulator  
; This function is self-modifying. Every time it  
; is called it increments the address that  
; sourceOfSeedBytes points to. Since sourceOfSeedBytes  
; initially points to $9A00, it will point to $9A01  
; after the first time it's called, $9A02 after the  
; second time it's called - and so on.  
;  
PutProceduralByteInAccumulator  
srcOfProceduralBytes    =**$01  
    LDA sourceOfSeedBytes  
    INC srcOfProceduralBytes  
    RTS
```

Listing 9.11: Our seed value ultimately comes from `sourceOfSeedBytes`.

And `sourceOfSeedBytes` turns out to be a string of 256 random-looking data at \$9A00:

```
*=$9A00  
sourceOfSeedBytes  
.BYTE $E0,$D3,$33,$1F,$BF,$EC,$EF,$3E  
.BYTE $FA,$70,$DA,$26,$87,$C2,$C9,$9C  
.BYTE $F7,$FB,$C8,$85,$C1,$A9,$64,$AD  
.BYTE $6B,$DE,$8B,$8F,$05,$5E,$54,$51  
.BYTE $78,$0A,$6E,$6F,$FD,$0C,$A5,$32  
.BYTE $F5,$56,$44,$75,$38,$D6,$23,$98  
.BYTE $61,$D5,$49,$C6,$F2,$95,$BA,$08  
.BYTE $C3,$3D,$F4,$F0,$21,$48,$84,$02  
.BYTE $7E,$5B,$68,$55,$04,$92,$AE,$34  
.BYTE $72,$F6,$71,$A1,$39,$4F,$74,$E5  
.BYTE $E8,$31,$9A,$C7,$E3,$86,$6D,$14  
.BYTE $60,$CD,$50,$FF,$82,$52,$66,$9E  
.BYTE $E9,$53,$25,$93,$07,$77,$2E,$D7  
.BYTE $1A,$62,$80,$B7,$0D,$1B,$15,$46  
.BYTE $CE,$AA,$47,$24,$8D,$E1,$18,$67  
.BYTE $6A,$4A,$F1,$B9,$D0,$91,$BC,$EE
```

```
.BYTE $B5,$D1,$7B,$A0,$DB,$36,$45,$E7
.BYTE $11,$22,$81,$FC,$58,$30,$28,$CB
.BYTE $8C,$B1,$0B,$A7,$DC,$B4,$9D,$57
.BYTE $B3,$ED,$3C,$43,$16,$8A,$EA,$D8
.BYTE $0E,$89,$1D,$1E,$DF,$9F,$BD,$BB
.BYTE $F9,$D9,$01,$3B,$7A,$BE,$69,$B8
.BYTE $5A,$A6,$E2,$96,$F8,$AC,$6C,$12
.BYTE $2D,$19,$2A
```

Listing 9.12: Our seed value ultimately comes from `sourceOfSeedBytes`.

As you might have guessed by now, given that there are only 16 possible sequences to choose from the seed bank there must actually be a lot less than a hundred thousand billion possible them tunes. With only 16 sequences there may even be only 16!

Well yes, there are certainly a lot closer to just 16 than a hundred thousand billion. The variety of values we get from `sourceOfSeedBytes` is not really of any account in the number of tunes we can generate. We're just using it to get pseudo-random but relatively predictable values between 0 and 15 and using that to choose one of the 16 4-byte 'tune seeds'.

There's an additional bit of variability that gives us more than just 16 tunes though. This is the value we add to the note's index before we play it:

```
MaybePlayVoice1
    DEC voice1NoteDuration
    BNE MaybePlayVoice2

    LDA #$30
    STA voice1NoteDuration

    LDX voice1IndexToMusicNoteArray
    LDA titleMusicNoteArray,X
    CLC
    ADC offsetForNextVoice1Note
    TAY
    STY offsetForNextVoice2Note
```

Listing 9.13: `offsetForNextVoice1Note` introduces additional tune permutations.

When we select a new tune the value `offsetForNextVoice1Note` may be carrying over a value from the previous tune so rather than be a consistent value every time the 'tune seed' is selected, it will vary in value. The result is that the logic will select a different note-group even though it used the same 4-byte 'tune seed'.

Since the value in `offsetForNextVoice1Note` is ultimately loaded from `titleMusicSeedArray`:

```
LDX notesPlayedSinceLastKeyChange
LDA titleMusicNoteArray,X
```

```
STA offsetForNextVoice1Note
```

Listing 9.14: `offsetForNextVoice1Note` is loaded from `titleMusicNoteArray` and is propagated down to `offsetForNextVoice3Note`.

In practice there are 16 possible voice note sequences and 12 unique possible byte values to load from `titleMusicSeedArray` so there are 192 possible tunes.

[I can only get it to generate 80 so I'm missing something here.]

So the Hundred Thousand Billion is a lie. Iridis will indeed play a hundred thousand billion times if you leave it running long enough but ultimately even when we account for variations in key it can only ever play 192 unique title tunes.

Sorry for getting your hopes up.

Another 16^4 Tunes

It's possible to dig into the making of the title music and how Jeff Minter arrived the music configuration he did thanks to a number of tiny demo programs that survive from the period when he was developing Iridis Alpha.

Jeff distributed these on Compunet in the summer of 1986.

It turns out he was inspired by an article in 'Byte' magazine from June 1986 that described how to make 'Fractal Music'. This article outline a version of the algorithm that Jeff ultimately adopted. The 'self-similarity' we encountered in the way the Iridis Alpha theme tunes are constructed, a four-note structure repeated across different time intervals on each of the three voices, finds its roots in this article.

CHAPTER 10. ANOTHER 16⁴ TUNES

MUSICAL FRACTALS

of the musical structure.

In listing 4, we have limited the number of layers to three, but given the hardware, you could add layers to the musical structure until you achieve a point of aural saturation. You reach saturation when the input exceeds your aural resolution (when an additional layer makes the music move so fast that its tones become blurs) or surpasses the limitations of the synthesis system (when the tones in a new layer are too high for the synthesizer to play).

You can make a musical fractal that sounds less mechanical by incorporating a certain degree of randomness into the previous, non-random example. The idea is to produce random offsets of the original, nonrandom values. The program shown in listing 4 gives you the op-

tion of calling for Brownian random offsets in the selection of pitch intervals on the second and third layers. For the second layer the program will impart a change in the range of 0 to 6 semitones above or below the pitch value specified. The same range and distribution are applied for the third layer to those tones produced for the second layer.

The technique for creating this musical fractal is quite closely related to the technique in computer graphics for making fractal Brownian mountain ranges (see reference 7). This complex computer graphic is made by nesting triangles, where the sides of the triangles are offset by some random amount proportional to the length of that side. In the musical fractals produced by the Brownian variation that is given in listing 4, an analogous

Figure 5a: The relationship in time of four levels of generated notes.

(continued)

A LAYERED STRUCTURE

To understand how the layers are made, you need to understand the concept of *pitch class*. A pitch class is a C or an F#, for example, without regard for its register. In other words, the highest and lowest Cs on the piano keyboard share the same pitch class, C; they differ only in octave.

You don't specify the number of notes in the first layer directly. Instead, you begin by specifying the number of different pitch classes you want to generate in the first layer. You may have noticed that the 1/f algorithm commonly produces repeated notes. Therefore, the contents of the layer are related to pitch-class diversity, not simply to the number of pitches in the layer. The number of pitches in the layer does determine the perceived

Figure 5b: A four-layer musical structure generated using the 1/f-noise algorithm.

192 BYTE • JUNE 1986

Figure 10.1

10.1 Taurus:Torus

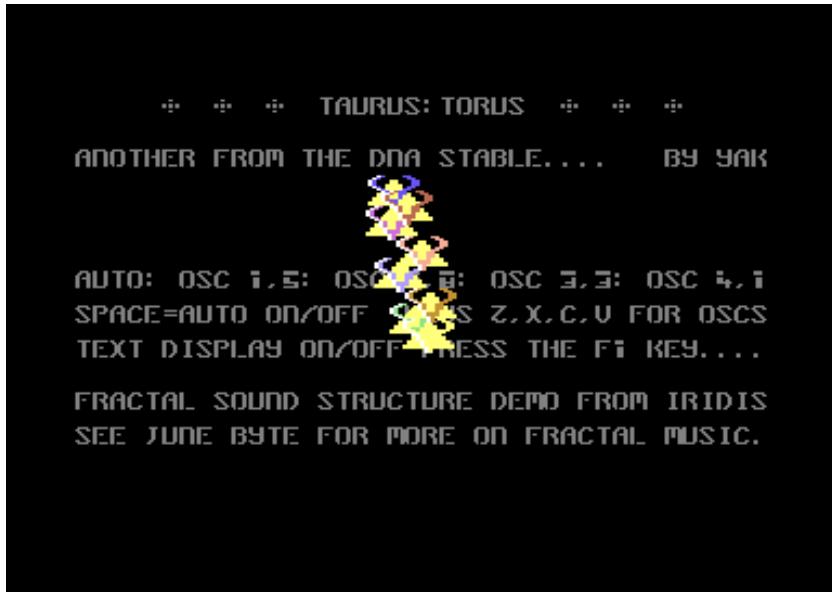


Figure 10.2

This first demo, released in July 1986(?), has a version of Iridis' music-generating algorithm that is nearly fully formed. However, the music it produces is quite different. In fact, it is nearer to a tool for listening to and selecting music than anything else.

The four seed values in `titleMusicNoteArray` that are used to seed all subsequently generated tunes (00 07 0C 07 in Iridis Alpha) can be selected and changed by the user. They're called 'Oscillators' and each can be any value between 0 and 16, i.e. any of 0 1 2 3 4 5 6 7 8 9 A B C D E F.

<pre>----- ; PlayTitleScreenMusic ; (TORUS:TAURUS) ----- PlayTitleScreenMusic</pre>	<pre>----- ; PlayTitleScreenMusic ; (IRIDIS ALPHA) ----- PlayTitleScreenMusic DEC baseNoteDuration BEQ MaybeStartNewTune RTS</pre>
<pre>DEC numberOfNotesToPlayInTune BNE MaybePlayVoice1</pre>	<pre>DEC previousBaseNoteDuration STA baseNoteDuration</pre>
<pre>JSR SelectNewNotesToPlay</pre>	<pre>DEC numberOfNotesToPlayInTune BNE MaybePlayVoice1</pre>

CHAPTER 10. ANOTHER 16⁴ TUNES

```

; Set up a new tune.
LDA #$C0
STA numberOfNotesToPlayInTune

; Set up a new tune.
LDA #$C0
STA numberOfNotesToPlayInTune

; This is what will eventually time us out of
; the title music and enter attract mode.
INC f7PressedOrTimedOutToAttractMode

LDX notesPlayedSinceLastKeyChange
LDA titleMusicNoteArray,X
STA offsetForNextVoice1Note

; We'll only select a new tune when we've reached the
; beginning of a new 16 bar structure.
INX
TXA
AND #$03
STA notesPlayedSinceLastKeyChange

LDX notesPlayedSinceLastKeyChange
LDA titleMusicNoteArray,X
STA offsetForNextVoice1Note

; We'll only select a new tune when we've reached
; the beginning of a new 16 bar structure.
INX
TXA
AND #$03
STA notesPlayedSinceLastKeyChange
BNE MaybePlayVoice1

JSR SelectNewNotesToPlay

MaybePlayVoice1
DEC voice1NoteDuration
BNE MaybePlayVoice2

LDA #$30
STA voice1NoteDuration

LDX voice1IndexToMusicNoteArray
LDA titleMusicNoteArray,X
CLC
ADC offsetForNextVoice1Note
TAY
STY offsetForNextVoice2Note

JSR PlayVoice1

INX
TXA
AND #$03
STA voice1IndexToMusicNoteArray

LDX voice1IndexToMusicNoteArray
LDA titleMusicNoteArray,X
CLC
ADC offsetForNextVoice1Note
TAY
STY offsetForNextVoice2Note

JSR PlayNoteVoice1

INX
TXA
AND #$03
STA voice1IndexToMusicNoteArray

MaybePlayVoice2
DEC voice2NoteDuration
BNE MaybePlayVoice3

LDA #$0C
STA voice2NoteDuration
LDX voice2IndexToMusicNoteArray
LDA titleMusicNoteArray,X
CLC
ADC offsetForNextVoice2Note

; Use this new value to change the key of the next
four
; notes played by voice 3.
STA offsetForNextVoice3Note

TAY
JSR PlayVoice2
INX
TXA
AND #$03
STA voice2IndexToMusicNoteArray

LDX voice2IndexToMusicNoteArray
LDA titleMusicNoteArray,X
CLC
ADC offsetForNextVoice2Note

; Use this new value to change the key of the next
four
; notes played by voice 3.
STA offsetForNextVoice3Note

TAY
JSR PlayNoteVoice2
INX
TXA
AND #$03
STA voice2IndexToMusicNoteArray

MaybePlayVoice3
DEC voice3NoteDuration
BNE ReturnFromTitleScreenMusic

LDA #$03
STA voice3NoteDuration

; Play the note currently pointed to by
; voice3IndexToMusicNoteArray in titleMusicNoteArray.
titleMusicNoteArray
LDX voice3IndexToMusicNoteArray
LDA titleMusicNoteArray,X
CLC
ADC offsetForNextVoice3Note
TAY
JSR PlayVoice3

; Move voice3IndexToMusicNoteArray to the next
; position in titleMusicNoteArray.
INX

; Set up a new tune.
LDA #$C0
STA numberOfNotesToPlayInTune

; This is what will eventually time us out of
; the title music and enter attract mode.
INC f7PressedOrTimedOutToAttractMode

LDX notesPlayedSinceLastKeyChange
LDA titleMusicNoteArray,X
STA offsetForNextVoice1Note

; Set up a new tune.
LDA #$C0
STA numberOfNotesToPlayInTune

; This is what will eventually time us out of
; the title music and enter attract mode.
INC f7PressedOrTimedOutToAttractMode

LDX notesPlayedSinceLastKeyChange
LDA titleMusicNoteArray,X
STA offsetForNextVoice1Note

; We'll only select a new tune when we've reached
; the beginning of a new 16 bar structure.
INX
TXA
AND #$03
STA notesPlayedSinceLastKeyChange
BNE MaybePlayVoice2

JSR PlayNoteVoice1

INX
TXA
AND #$03
STA voice1IndexToMusicNoteArray

MaybePlayVoice2
DEC voice2NoteDuration
BNE MaybePlayVoice3

LDA #$0C
STA voice2NoteDuration
LDX voice2IndexToMusicNoteArray
LDA titleMusicNoteArray,X
CLC
ADC offsetForNextVoice2Note

; Use this new value to change the key of the next
four
; notes played by voice 3.
STA offsetForNextVoice3Note

TAY
JSR PlayNoteVoice2
INX
TXA
AND #$03
STA voice2IndexToMusicNoteArray

LDX voice2IndexToMusicNoteArray
LDA titleMusicNoteArray,X
CLC
ADC offsetForNextVoice2Note

; Use this new value to change the key of the next
four
; notes played by voice 3.
STA offsetForNextVoice3Note

TAY
JSR PlayNoteVoice3
INX

; Set up a new tune.
LDA #$C0
STA numberOfNotesToPlayInTune

; This is what will eventually time us out of
; the title music and enter attract mode.
INC f7PressedOrTimedOutToAttractMode

LDX notesPlayedSinceLastKeyChange
LDA titleMusicNoteArray,X
STA offsetForNextVoice1Note

; Set up a new tune.
LDA #$C0
STA numberOfNotesToPlayInTune

; This is what will eventually time us out of
; the title music and enter attract mode.
INC f7PressedOrTimedOutToAttractMode

LDX notesPlayedSinceLastKeyChange
LDA titleMusicNoteArray,X
STA offsetForNextVoice1Note

; We'll only select a new tune when we've reached
; the beginning of a new 16 bar structure.
INX
TXA
AND #$03
STA notesPlayedSinceLastKeyChange
BNE MaybePlayVoice3

JSR PlayNoteVoice3
INX

```

<pre> TXA ; Since it's only 4 bytes long ensure we wrap ; back to 0 if it's greater than 3. AND #\$03 STA voice3IndexToMusicNoteArray </pre>	<pre> TXA ; Since it's only 4 bytes long ensure we wrap ; back to 0 if it's greater than 3. AND #\$03 STA voice3IndexToMusicNoteArray </pre>
<pre> ReturnFromTitleScreenMusic RTS </pre>	<pre> ReturnFromTitleScreenMusic RTS </pre>

Listing 10.1: The music routine in Torus:Taurus side-by-side with Iridis Alpha.

When it runs the demo cycles through procedural configurations of titleMusicNoteArray of 64 notes each. In other words, exactly the kind of fractal structure we observed in Iridis Alpha proper. The examples below give a flavour of the music it generates:

Procedurally Generated

Torus:Taurus Tune

(Sid.)
Jeff Minter

1 of 100,000,000,000,000

This musical score displays three staves representing different voices. Voice 1 starts with a whole note followed by a rest. Voice 2 has a half note followed by a whole note. Voice 3 has a quarter note followed by a eighth note. The music continues with a series of eighth and sixteenth note patterns. The score is labeled 'Procedurally Generated' at the top left and 'Torus:Taurus Tune' in the center. The copyright information '(Sid.) Jeff Minter' is at the top right. The bar number '1 of 100,000,000,000,000' is centered below the title.

Figure 10.3: Bars 2 and 4 are always repeated

Procedurally Generated

Torus:Taurus Tune

(Sid.)
Jeff Minter

2 of 100,000,000,000,000

This musical score displays three staves representing different voices. Voice 1 has a half note followed by a whole note. Voice 2 has a half note followed by a whole note. Voice 3 has a quarter note followed by a eighth note. The music continues with a series of eighth and sixteenth note patterns. The score is labeled 'Procedurally Generated' at the top left and 'Torus:Taurus Tune' in the center. The copyright information '(Sid.) Jeff Minter' is at the top right. The bar number '2 of 100,000,000,000,000' is centered below the title.

Figure 10.4: Bars 2 and 4 are always repeated

Not all of the tunes are 64-note based. It does generate some that are truncated.

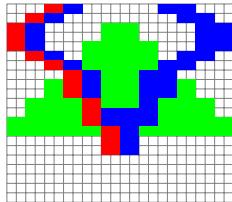


Figure 10.5: BULLHEAD

Figure 10.6: The 'Torus' sprite.

10.2 Taurus/Torus Two



Figure 10.7

```
;-----  
; PlayTitleScreenMusic  
; (TORUS:TAURUS II)  
;  
PlayTitleScreenMusic  
    LDA UnusedValue1  
    STA UnusedValue2  
;  
;  
;-----  
; PlayTitleScreenMusic  
; (IRIDIS ALPHA)  
;  
PlayTitleScreenMusic  
    DEC baseNoteDuration  
    BEQ MaybeStartNewTune
```

CHAPTER 10. ANOTHER 16⁴ TUNES

<pre> MaybeStartNewTune DEC numberOfNotesToPlayInTune BNE MaybePlayVoice1 ; Set up a new tune. LDA #\$C0 STA numberOfNotesToPlayInTune playing LDX notesPlayedSinceLastKeyChange LDA titleMusicNoteArray,X STA offsetForNextVoice2Note ; We'll only select a new tune when we've reached the ; reached the ; beginning of a new 16 bar structure. INX TXA AND #\$03 STA notesPlayedSinceLastKeyChange BNE MaybePlayVoice1 JSR SelectNewNotesToPlay MaybePlayVoice1 DEC voice1NoteDuration BNE MaybePlayVoice2 LDA #\$30 STA voice1NoteDuration LDX voice1IndexToMusicNoteArray LDA titleMusicNoteArray,X CLC ADC offsetForNextVoice2Note TAY STY offsetForNextVoice3Note JSR PlayNoteVoice1 INX TXA AND #\$03 STA voice1IndexToMusicNoteArray </pre>	<pre> RTS MaybeStartNewTune LDA previousBaseNoteDuration STA baseNoteDuration DEC numberOfNotesToPlayInTune BNE MaybePlayVoice1 ; Set up a new tune. LDA #\$C0 ; 193 STA numberOfNotesToPlayInTune ; This is what will eventually time us out of ; the title music and enter attract mode. INC f7PressedOrTimedOutToAttractMode LDX notesPlayedSinceLastKeyChange LDA titleMusicNoteArray,X STA offsetForNextVoice1Note ; We'll only select a new tune when we've ; reached the ; beginning of a new 16 bar structure. INX TXA AND #\$03 STA notesPlayedSinceLastKeyChange BNE MaybePlayVoice1 JSR SelectNewNotesToPlay MaybePlayVoice1 DEC voice1NoteDuration BNE MaybePlayVoice2 LDA #\$30 STA voice1NoteDuration LDX voice1IndexToMusicNoteArray LDA titleMusicNoteArray,X CLC ADC offsetForNextVoice1Note TAY STY offsetForNextVoice2Note JSR PlayNoteVoice1 INX TXA AND #\$03 STA voice1IndexToMusicNoteArray </pre>
<pre> MaybePlayVoice2 DEC voice2NoteDuration BNE MaybePlayVoice3 LDA #\$0C STA voice2NoteDuration LDX voice2IndexToMusicNoteArray LDA titleMusicNoteArray,X CLC ADC offsetForNextVoice3Note ; Use this new value to change the key of the next four ; next four ; notes played by voice 1. STA offsetForNextVoice1Note TAY JSR PlayNoteVoice2 INX TXA AND #\$03 STA voice2IndexToMusicNoteArray TAY JSR PlayNoteVoice2 INX TXA AND #\$03 STA voice2IndexToMusicNoteArray </pre>	<pre> MaybePlayVoice2 DEC voice2NoteDuration BNE MaybePlayVoice3 LDA #\$0C STA voice2NoteDuration LDX voice2IndexToMusicNoteArray LDA titleMusicNoteArray,X CLC ADC offsetForNextVoice2Note ; Use this new value to change the key of the ; notes played by voice 3. STA offsetForNextVoice3Note </pre>
<pre> MaybePlayVoice3 DEC voice3NoteDuration </pre>	<pre> MaybePlayVoice3 DEC voice3NoteDuration </pre>

CHAPTER 10. ANOTHER 16⁴ TUNES

<pre> BNE ReturnFromTitleMusic LDA #\$03 STA voice3NoteDuration ; Play the note currently pointed to by ; voice3IndexToMusicNoteArray in titleMusicNoteArray. titleMusicNoteArray LDX voice3IndexToMusicNoteArray LDA titleMusicNoteArray,X CLC ADC offsetForNextVoice1Note TAY JSR PlayNoteVoice3 ; Move voice3IndexToMusicNoteArray to the next ; position in titleMusicNoteArray. INX TXA ; Since it's only 4 bytes long ensure we wrap ; back to 0 if it's greater than 3. AND #\$03 STA voice3IndexToMusicNoteArray ReturnFromTitleMusic RTS </pre>	<pre> BNE ReturnFromTitleScreenMusic LDA #\$03 STA voice3NoteDuration ; Play the note currently pointed to by ; voice3IndexToMusicNoteArray in ; titleMusicNoteArray. LDX voice3IndexToMusicNoteArray LDA titleMusicNoteArray,X CLC ADC offsetForNextVoice3Note TAY JSR PlayNoteVoice3 ; Move voice3IndexToMusicNoteArray to the next ; position in titleMusicNoteArray. INX TXA ; Since it's only 4 bytes long ensure we wrap ; back to 0 if it's greater than 3. AND #\$03 STA voice3IndexToMusicNoteArray ReturnFromTitleScreenMusic RTS </pre>
---	---

Listing 10.2: The music routine in Taurus:Torus II side-by-side with Iridis Alpha.

Procedurally Generated

Taurus/Torus Two Tune

(Sid.)
Jeff Minter

1 of 100,000,000,000,000

The musical score consists of three staves representing different voices. Voice 1 starts with a rest followed by a series of eighth-note patterns. Voice 2 begins with a quarter note followed by eighth-note patterns. Voice 3 starts with a sixteenth-note pattern followed by eighth-note patterns. The score is set against a background of vertical bar lines and rests. Dynamic markings 'xx' and 'x' are placed above certain measures. The title 'Taurus/Torus Two Tune' is centered at the top, and the credit '(Sid.) Jeff Minter' is to the right. Below the title, it says '1 of 100,000,000,000,000'.

Figure 10.8

CHAPTER 10. ANOTHER 16^4 TUNES

Procedurally Generated

Iridis Alpha Title Theme

(Sld.)
Jeff Minter

1 of 100,000,000,000,000

The musical score for "Iridis Alpha Title Theme" is a procedural generation example. It features three voices (Voice 1, Voice 2, Voice 3) arranged in a 3x4 grid of staves. The time signature is 3/4, and the key signature is G major. The score consists of 12 measures. Voice 1 starts with a whole note rest, followed by eighth-note pairs. Voice 2 starts with a half note, followed by eighth-note pairs. Voice 3 starts with a quarter note, followed by eighth-note pairs. The music is highly repetitive and rhythmic.

Figure 10.9

Made in France



Figure 11.1: Splash screen for the version of 'Made in France' released on CompuNet.

A Pause Mode for your Pause Mode



Figure 12.1: DNA: A pause mode within a pause mode.

```
titleTextLine1    .TEXT "      % % % DNA % % % "
titleTextLine2    .TEXT " CONCEIVED AND EXECUTED BY "
titleTextLine3    .TEXT "Y          Y A K      "
titleTextLine4    .TEXT " SPACE: CANCEL SCREEN TEX"
titleTextLine5    .TEXT "TF5 AND F7 CHANGE COLOURS"
titleTextLine6    .TEXT " LISTEN TO TALKING HEADS."
titleTextLine7    .TEXT ".BE NICE TO HAIRY ANIMALS "
```

Listing 12.1: Defining the text for the DNA screen

CHAPTER 12. A PAUSE MODE FOR YOUR PAUSE MODE

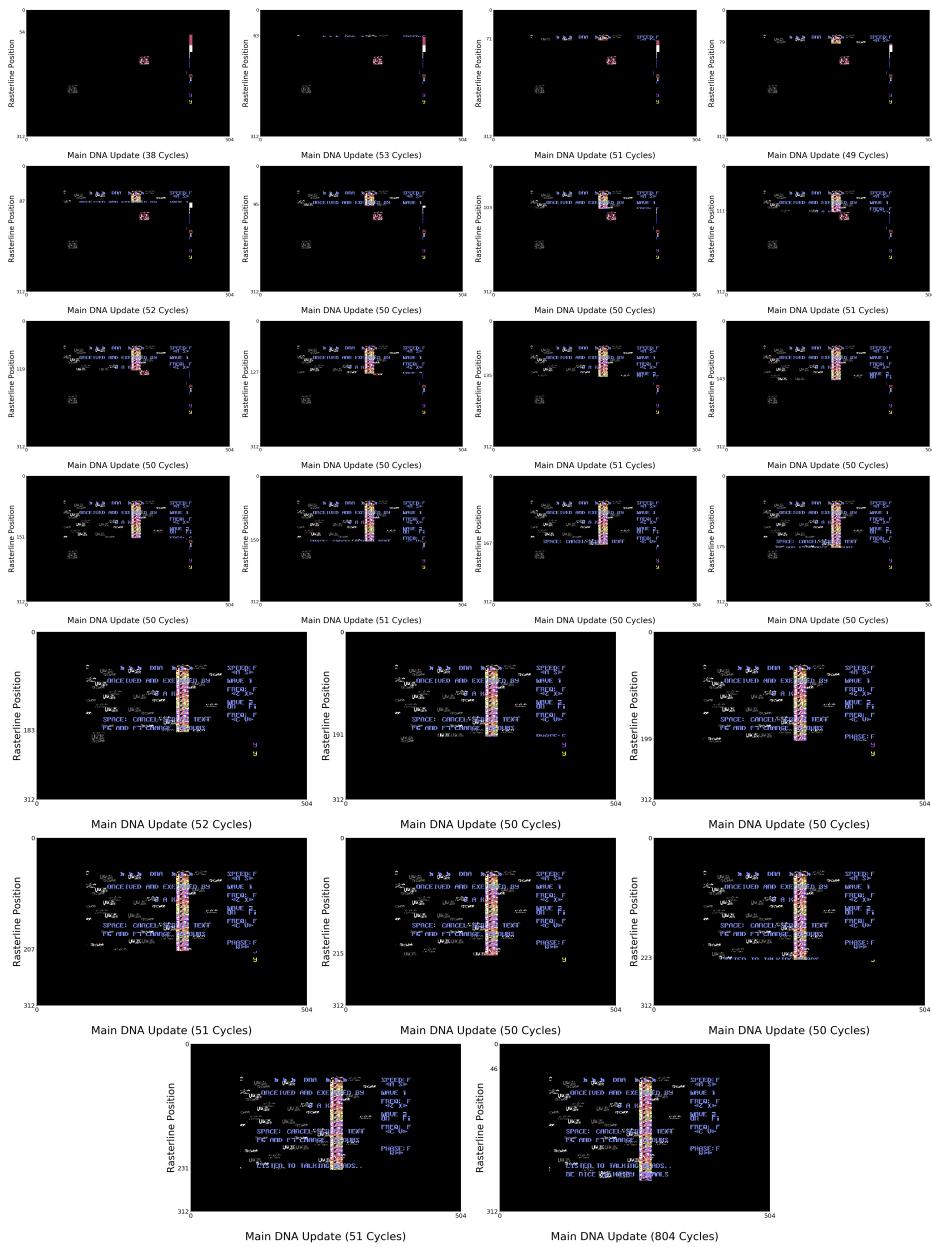


Figure 12.2: The first screen paint in DNA. There are 24 raster interrupts allowing us to paint a long chain of sprites.

Any pause mode must surely be in need of a pause mode. Titled 'DNA' this little entertainment is a cousin of Minter's previous work on Psychedelia for the C64 and Colorspace for the Atari 800 in 1984 and 1985. It isn't accessed directly from the game but instead is invoked by pressing the asterisk key while playing 'Made in France'.

Minter first shared it as a tiny 11K demo in a UK CompuNet forum in the summer of 1986. It followed shortly after 'Torus', an oscillator-based demo, shared at the same time and which we cover elsewhere : both are sprite-based light synthesisers where, like Psychedelia and Colorspace, the player gets to experiment with different configurations that control the behaviour of a frantic assembly of brightly colored, pulsating sprites.

DNA has an unapologetically daft premise: two chains of flashing eyeballs cascade down the screen in an unsettling, blinking helix configuration. Your object as player is to twiddle the available knobs to see if you can get them to do anything interesting while listening to your favorite music.

For its time DNA's most noteworthy feature was the number of sprites being written to the screen. There are 48 eyeballs displayed at any one time, in addition to a pair of parallax starfields drifting past in the background. As you may have gathered by now, the C64 can only support 8 sprites in total so this would have seemed like wizardry to the uninitiated. If you're not dipping into this chapter at random, but have read any of the previous chapters in this book you may already be able to guess the secret to this unsettling feat: raster interrupts.

As elsewhere in Iridis Alpha the trick to filling the screen with sprites is to write a tight piece of code that can run periodically during a single paint of the screen, adding a layer of sprites to each horizontal section. We tell the C64 where we want the raster to interrupt its progress and call our code. This code will then paint as many sprites as possible on the horizontal layer. DNA takes the approach of painting a pair of eyeballs at 8 pixel vertical intervals, so each horizontal layer is 8 pixels tall. These intervals are defined in `dnaSpritesYPositionsArray`:

```
dnaSpritesYPositionsArray      .BYTE $30,$38,$40,$48,$50,$58,$60,$68
                                .BYTE $70,$78,$80,$88,$90,$98,$A0,$A8
                                .BYTE $B0,$B8,$C0,$C8,$D0,$D8,$E0,$E8
                                .BYTE END_SENTINEL
```

While the Y co-ordinates of the sprites are set in stone, their X co-ordinates must be calculated on the fly and indeed the purpose of the twiddling knobs is to control the way these co-ordinates are generated. Initially the X positions are all set to 192 (\$C0):

```
dnaSpritesXPositionsArray    .BYTE $C0,$C0,$C0,$C0,$C0,$C0,$C0,$C0
                                .BYTE $C0,$C0,$C0,$C0,$C0,$C0,$C0,$C0
                                .BYTE $C0,$C0,$C0,$C0,$C0,$C0,$C0,$C0
```

```
.BYTE $00,$00,$00,$00,$00,$00,$00,$00  
.BYTE $00,$00,$00,$00,$00,$00,$00,$00
```

With every full paint of the screen calculates a new X co-ordinate and puts it at the head of the array. Depending on the value given as the 'Speed' setting, it then shifts all the others in the array to the right by one position.

```
DNA_PropagatePreviousXPostToTheRight  
    LDX #$27  
PropagateToRightLoop  
    LDA dnaSpritesXPositionsArray - $01,X  
    STA dnaSpritesXPositionsArray,X  
    DEX  
    BNE PropagateToRightLoop  
    RTS
```

This very quickly fills the array. We can see this in operation from the below snippet in DNA_MainAnimationRoutine. In order to calculate the x position for the right hand chain, it shows the value in dnaCurrentPhase being added to the value dnaCurrentSpritesPosArrayIndex to pick out a value in dnaSpritesXPositionsArray offset by the amount of the 'Phase' setting:

```
LDX dnaCurrentSpritesPosArrayIndex
..
; Add in the phase to our index to the X position
; of the sprite on the right hand chain. If the
; result is greater than the number of values
; in the array ($27) subtract it out again.
; This means the 'Phase' setting acts as an
; offset into the X Position array picking up
; previous values of X Pos from the left hand
; chain.
TXA
..
ADC dnaCurrentPhase
CMP #$27
BMI UpdateXPosWithPhase

; Back out the addition if result greater than $27.
SEC
SBC #$27
UpdateXPosWithPhase
; Finally, use the offset given by the update value
; in 'A', load it to 'X' and use it pick the X
; position from dnaSpritesXPositionsArray.
TAX
LDA dnaSpritesXPositionsArray,X
STA $D004,Y ;Sprite 2 X Pos
```



Listing 12.2: dnaCurrentPhase is set by the 'Q' key.

In the table below we've selected a few frames from the first second or two showing how the array fills up and how values from the array are selected for the X position of the left and right chains. Notice how the 'Phase' setting of 5 effectively means the right-hand chain lags 5 'eyeballs' behind the left chain in terms of the positions.

State	Left Chain	Right Chain
	<pre>dnaSpritesXPositionsArray .BYTE \$58,\$5A,\$60,\$67,\$71,\$7C,\$89,\$96 .BYTE \$A3,\$AE,\$B6,\$BB,\$BC,\$B9,\$B3,\$A9, .BYTE \$9D,\$8F,\$B0,\$C0,\$C0,\$C0,\$C0,\$C0 .BYTE \$C0,\$C0,\$C0,\$C0,\$C0,\$C0,\$C0,\$C0 .BYTE \$C0,\$C0,\$C0,\$C0,\$C0,\$C0,\$C0,\$C0</pre>	<pre>dnaSpritesXPositionsArray .BYTE \$58,\$5A,\$60,\$67,\$71,\$7C,\$89,\$96 .BYTE \$A3,\$AE,\$B6,\$BB,\$BC,\$B9,\$B3,\$A9, .BYTE \$9D,\$8F,\$B0,\$C0,\$C0,\$C0,\$C0,\$C0 .BYTE \$C0,\$C0,\$C0,\$C0,\$C0,\$C0,\$C0,\$C0</pre>
	<pre>dnaSpritesXPositionsArray .BYTE \$6F,\$58,\$61,\$5C,\$59,\$58,\$5A,\$60 .BYTE \$67,\$71,\$7C,\$89,\$96,\$A3,\$AE,\$B6 .BYTE \$B9,\$B3,\$C0,\$B9,\$B3,\$A9,\$9D,\$8F,\$B0 .BYTE \$C0,\$C0,\$C0,\$C0,\$C0,\$C0,\$C0,\$C0 .BYTE \$C0,\$C0,\$C0,\$C0,\$C0,\$C0,\$C0,\$C0</pre>	<pre>dnaSpritesXPositionsArray .BYTE \$6F,\$68,\$61,\$5C,\$59,\$58,\$5A,\$60 .BYTE \$67,\$71,\$7C,\$89,\$96,\$A3,\$AE,\$B6 .BYTE \$B9,\$B3,\$B9,\$B3,\$A9,\$9D,\$8F,\$B0 .BYTE \$C0,\$C0,\$C0,\$C0,\$C0,\$C0,\$C0,\$C0</pre>
	<pre>dnaSpritesXPositionsArray .BYTE \$81,\$7C,\$76,\$6F,\$68,\$61,\$5C,\$59 .BYTE \$58,\$5A,\$60,\$67,\$71,\$7C,\$89,\$96 .BYTE \$A3,\$AE,\$B6,\$BB,\$BC,\$B9,\$B3,\$A9, .BYTE \$9D,\$8F,\$B0,\$C0,\$C0,\$C0,\$C0,\$C0 .BYTE \$C0,\$C0,\$C0,\$C0,\$C0,\$C0,\$C0,\$C0</pre>	<pre>dnaSpritesXPositionsArray .BYTE \$81,\$7C,\$76,\$6F,\$68,\$61,\$5C,\$59 .BYTE \$58,\$5A,\$60,\$67,\$71,\$7C,\$89,\$96 .BYTE \$A3,\$AE,\$B6,\$BB,\$BC,\$B9,\$B3,\$A9, .BYTE \$9D,\$8F,\$B0,\$C0,\$C0,\$C0,\$C0,\$C0 .BYTE \$C0,\$C0,\$C0,\$C0,\$C0,\$C0,\$C0,\$C0</pre>

Figure 12.3: Examples of the x co-ordinates (highlighted) in dnaSpritesXPositionsArray used by the left and right chain when 'phase' is set to 5.

We said a new value for the top eye-ball in each chain is calculated every paint, but whether the existing values get propagated to the right in dnaSpritesXPositionsArray depends on the 'Speed' setting:

```
UpdateXPosArrays
DEC actualSpeed
BNE CalculateNewXPosForHead

; The speed setting determines how quickly
; the X pos values are propagated down the
chains.
LDA dnaCurrentSpeed
STA actualSpeed
JSR DNA_PropagatePreviousXPosToTheRight
```



Listing 12.3: dnaCurrentSpeed is set by the 'A' and 'S' keys.

The 'Speed' setting is effectively a counter that is decremented at every paint, once it reaches zero it gets reset back to its initial value (in this case '1') and the X Positions

are all shifted to the right in `dnaSpritesXPositionsArray`.

How is the new value for the eyeball at the head of each chain calculated? We get that it's updated at every screen paint but what determines it? The calculation is a two-phase process but both phases depend on the same array of values stored in `newXPosOffsetsArray`. This is a rattle-bag of X positions describing a more or less continuous curve:

```
newXPosOffsetsArray    .BYTE $40,$46,$4C,$53,$58,$5E,$63,$68
                      .BYTE $6D,$71,$75,$78,$7B,$7D,$7E,$7F
                      .BYTE $80,$7F,$7E,$7D,$7B,$78,$75,$71
                      .BYTE $6D,$68,$63,$5E,$58,$52,$4C,$46
                      .BYTE $40,$39,$33,$2D,$27,$21,$1C,$17
                      .BYTE $12,$0E,$0A,$07,$04,$02,$01,$00
                      .BYTE $00,$00,$01,$02,$04,$07,$0A,$0E
                      .BYTE $12,$17,$1C,$21,$27,$2D,$33,$39
                      .BYTE END_SENTINEL
```

Listing 12.4: Notice that the values start at \$40 rise gradually to \$80 back to \$00 and then back up to \$40 again.

The other factors in this calculation are the 'Frequency' selected for each chain, the left-hand chain (Wave 1) and the right-hand chain (Wave 2). The values that do the most work are `xPosOffsetForWave1` and `xPosOffsetForWave2`.

```
DNA_UpdateSettingsBasedOnFrequency
LDA dnaWave1Frequency
AND #$1F
TAX
LDA timesToNextUpdateForFrequency,X
STA initialTimeToNextUpdateForWave1
STA timeToNextUpdateCounterForWave1
LDA xPosOffsetsForFrequency,X
STA xPosOffsetForWave1

LDA dnaWave2Frequency
AND #$1F
TAX
LDA timesToNextUpdateForFrequency,X
STA initialTimeToNextUpdateForWave2
STA timeToNextUpdateCounterForWave2
LDA xPosOffsetsForFrequency,X
STA xPosOffsetForWave2
JMP DNA_UpdateDisplayedSettings
; Returns
```



Listing 12.5: The value selected for 'Frequency' for Wave 1 and 2 translates to settings used in calculating the next X position for each chain.

As we can see above these are populated from `xPosOffsetsForFrequency` using the 'Frequency' as an index.

```
..  
LDA xPosOffsetsForFrequency ,X  
STA xPosOffsetForWave1  
  
..  
LDA xPosOffsetsForFrequency ,X  
STA xPosOffsetForWave2
```

In each case the value populated here is added to a value selected from `newXPosOffsetsArray` to give the new X position for the eyeball at the head of the chain. Well, in a way that's only half true. Separate positions are indeed calculated for the left and right chain, but at the end of the day the position calculate for the right-hand chain (Wave 2) is purely notional. It's just used as an initial value that the value calculated for the left-hand chain is layered on to. This allows the two chains to 'interfere' with each other, as though the position of one affects the other - and reduces the chance of the two overlapping too much:

```
UpdateHeadOfWave  
; Take the new X Pos value we calculated for Wave 2, add the position  
; we calculated for Wave 1 and make that the new position for the  
; lead  
; sprite at the head of dnaSpritesXPositionsArray.  
LDA notionalNewXPosForWave2  
CLC  
ADC newXPosForWave1  
STA dnaSpritesXPositionsArray
```

Listing 12.6: From `CalculateValueOfNewXPosForWave1`

When we compare the routines responsible for calculating the X Pos for Wave 2 and Wave 1 side by side on the next page we can see many similarities between the two. They are both maintaining and updating an index into `newXPosOffsetsArray` and using that to come up with a new X position. They both have to deal towards the end of the routine with the need to ensure the index does not exceed \$40 and subtract \$40 from it if it does. The output of `CalculateValueOfNewXPosForWave2` is essentially the value stored in `notionalNewXPosForWave2` and this is what is used as the input above, in conjunction with `newXPosForWave1`, to come up with the final value for the X position for Wave 1. It is this value that gets added to the head of the working array `dnaXPosDataHeadArray` by `DNA_PropagatePreviousXPosToTheRight`.

```

CalculateValueOfNewXPosForWave2
    LDA dnaWave2Enabled
    BNE NewXPosWhenWave2Enabled

    ; If wave 2 is not enabled set $40
    ; as the initial X Pos (it will be
    ; incremented later).
    LDA #XPOS_OFFSETS_ARRAY_SIZE
    STA notionalNewXPosForWave2
ReturnFromNewXPos
    RTS

NewXPosWhenWave2Enabled
    DEC timeToNextUpdateCounterForWave2
    BNE ReturnFromNewXPos

    LDA initialTimeToNextUpdateForWave2
    STA timeToNextUpdateCounterForWave2

    ; If right hand chain is enabled
    ; get value from newXPosOffsetsArray,
    ; which has $40 (64) potential values.
    ; This logic uses
    ; indexToXPosdataArrayForWave2
    ; to get a value from it, halves it
    ; with ROR, and adds $40.
    LDX indexToXPosdataArrayForWave2
    LDA newXPosOffsetsArray,X
    CLC
    ROR      ; Halve it.
    CLC
    ADC #XPOS_OFFSETS_ARRAY_SIZE
    STA notionalNewXPosForWave2

    ; Add xPosOffsetForWave2 to
    ; indexToXPosdataArrayForWave2.
    ; Ensure it loops to start of array if
    ; greater than $40.
    TXA
    CLC
    ADC xPosOffsetForWave2
    CMP #XPOS_OFFSETS_ARRAY_SIZE
    BMI NoLoopingRequired

    SEC
    SBC #XPOS_OFFSETS_ARRAY_SIZE
NoLoopingRequired
    STA indexToXPosdataArrayForWave2
    RTS

```

Listing 12.7: Calculate a notional X Pos value for Wave 2.
This routine is called by the one on the right its return
value is `notionalNewXPosForWave2`.

```

CalculateValueOfNewXPosForWave1
    DEC actualSpeed
    BNE CalculateNewXPosForHead

    ; The speed setting determines how quickly
    ; the X pos values are propagated down the chains.
    LDA dnaCurrentSpeed
    STA actualSpeed
    JSR DNA_PropagatePreviousXPosToTheRight
CalculateNewXPosForHead
    ; Get a new value for Wave 2.
    JSR CalculateValueOfNewXPosForWave2
    DEC timeToNextUpdateCounterForWave1
    BNE ReturnFromUpdatingHead

    LDA initialTimeToNextUpdateForWave1
    STA timeToNextUpdateCounterForWave1

    ; Get newXPosForWave1, this will be added to
    ; notionalNewXPosForWave2. Both are sourced
    ; from the same array newXPosOffsetsArray.
    LDX indexToNextXPosForHead
    LDA newXPosOffsetsArray,X
    STA newXPosForWave1

    LDY dnaWave2Enabled
    BEQ UpdateHeadOfWave

    ; Halve the offset we will use if
    ; the right hand chain is enabled.
    CLC
    ROR
    STA newXPosForWave1
UpdateHeadOfWave
    ; Take the new X Pos value we calculated for Wave 2,
    ; add the position we calculated for Wave 1 and
    ; make that the new position for the lead
    ; sprite at the head of dnaSpritesXPositionsArray.
    LDA notionalNewXPosForWave2
    CLC
    ADC newXPosForWave1
    STA dnaSpritesXPositionsArray

    ; Get value for indexToNextXPosForHead for the next
    ; time around. Ensure it loops to start of array if
    ; greater than $40.
    TXA
    CLC
    ADC xPosOffsetForWave1
    TAX
    CPX #XPOS_OFFSETS_ARRAY_SIZE
    BMI UpdateNextXPos

    SEC
    SBC #XPOS_OFFSETS_ARRAY_SIZE
    TAX
UpdateNextXPos
    STX indexToNextXPosForHead
    ReturnFromUpdatingHead
    RTS

```

Listing 12.8: Calculate the new X Pos value for the head
sprites as well as propagate the values in
`dnaXPosdataArray` to the right when required.

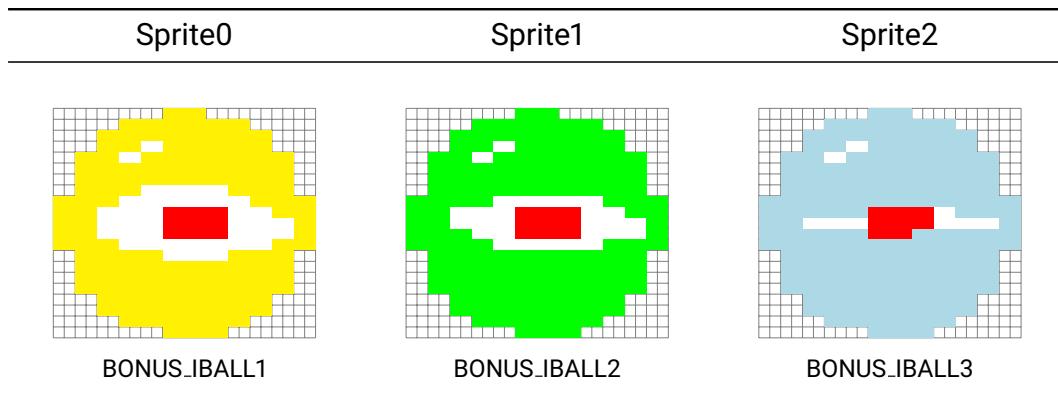


Figure 12.4: The eyeball sprites used by DNA.

An Oscillator in 4 Parts



Figure 13.1: The Torus oscillator animation and Iridis' bonus animation.

The Torus demo is also the laboratory where the elegant animation used when awarding a bonus was developed. The code handling each is identical and was only very lightly modified for the final game.

```

RunMainInterruptHandler
    LDY #$00
    LDA #$F0
    STA $D012 ; Raster Position
    DEC counterBetweenXPosUpdates
    BNE MaybeUpdateYPos

UpdateXPos
    LDA initialCounterBetweenXPosUpdates
    STA counterBetweenXPosUpdates

    LDA incrementForXPos
    CLC
    ADC indexForXPosInSpritePositionArray
    STA indexForXPosInSpritePositionArray

MaybeUpdateYPos
    DEC counterBetweenYPosUpdates
    BNE MaybeUpdateXPosOffset

    LDA initialCounterBetweenYPosUpdates
    STA counterBetweenYPosUpdates

    LDA indexForYPosInSpritePositionArray
    CLC
    ADC incrementForYPos
    STA indexForYPosInSpritePositionArray

MaybeUpdateXPosOffset
    DEC cyclesBetweenXPosOffsetUpdates
    BNE MaybeUpdateYPosOffset

    LDA oscillator3Value
    STA cyclesBetweenXPosOffsetUpdates
    INC indexForXPosOffsetInSpritePositionArray

MaybeUpdateYPosOffset
    DEC cyclesBetweenYPosOffsetUpdates
    BNE StoreInitialIndexValues

    LDA oscillator4Value
    STA cyclesBetweenYPosOffsetUpdates
    INC indexForYPosOffsetInSpritePositionArray

StoreInitialIndexValues
; Store the initial values for our indices
; on the stack:
    LDA indexForXPosInSpritePositionArray
    PHA
    LDA indexForYPosInSpritePositionArray
    PHA
    LDA indexForXPosOffsetInSpritePositionArray

```

```

    DEY
    BNE b64A0
    PLA
    TAY
    LDA (tempLoPtr1),Y
    SEC
    SBC #$01
    STA (tempLoPtr1),Y
    DEX
    BNE UpdateFinalScoreTally
    RTS

txtReasonGilbyDied .TEXT "
DEPLETED..OVERLOAD..ENTROPY...HIT SOMMAT"
;

; DrawReasonGilbyDied
;

DrawReasonGilbyDied
    LDA #$00
    LDY reasonGilbyDied
    BEQ b64F6
    CLC
    ADC #$0A
    DEY
    BNE b64F0
    TAX
    b64F7 LDA txtReasonGilbyDied,X
    AND #$3F
    STA SCREEN.RAM + LINE17.COL15,Y
    LDA #RED
    STA COLOR.RAM + LINE17.COL15,Y
    INY
    INX
    CPY #$0A
    BNE b64F7
    RTS

;

; JumpDisplayNewBonus
;

JumpDisplayNewBonus
    JMP DisplayNewBonus

bonusGilbinesPositionArray .BYTE $40,$46,$4C,$52,$58,
$5E,$63,$68 .BYTE $6D,$71,$75,$78,$7B,
$7D,$7E,$7F .BYTE $80,$7F,$7E,$7D,$7B,
$78,$75,$71 .BYTE $6D,$68,$63,$5E,$58,
$52,$4C,$46 .BYTE $40,$39,$33,$2D,$27,
$21,$1C,$17 .BYTE $12,$0E,$0A,$07,$04,
$02,$01,$00 .BYTE $00,$00,$01,$02,$04,
$07,$0A,$0E .BYTE $12,$17,$1C,$21,$27,
$2D,$33,$39 .BYTE $FF
bonusBountiesEarned .BYTE $00

```

Listing 13.1: Animation in Torus Demo

Listing 13.2: ... and Iridis Alpha

A Testing Hack

In the CheckKeyboardInGame Routine, we find the following:

```
b7722 ; Gilby is ground-based?
    LDA #$FF
    STA upperPlanetGilbyBulletMSBXPoSValue ,X
    JMP ResetUpperPlanetBullet
    ; Returns

; -----
; CheckBulletPositions
; -----
CheckBulletPositions
    LDA #$00
    STA currentMSBXPoSOffset
```

In the above the 'canAwardBonus' byte is the first letter in the name of the player with the top score in the Hi-Score table. By default this is 'YAK':

```
LDY currentGilbySpeed
BMI b7CF0
CLC
ADC currentGilbySpeed
```

But if we change 'Y' to \$1C like so, we can activate the hack:

hiScoreTablePtr	.TEXT "0068000"
canAwardBonus	.TEXT \$1C, "AK "

Note that \$1C is charset code for a bull's head symbol in Iridis Alpha, so it is also possible to enter this as the initial of a high scorer name if we get a score that puts us to the top of the table:

```
.BYTE $66,$C3,$7E,$5A,$7E,$7E,$3C,$00 ; .BYTE $66,$C3,$7E,$5A,$7E,$7E,$3C,$00
; CHARACTER $1c
; 01100110    **  **
; 11000011    **  **
; 01111110    *****
; 01011010    *  *  *
; 01111110    *****
; 01111110    *****
; 00111100    ****
; 00000000
```

I'm guessing this was used for testing the animation routine and left in as an Easter egg.

To start getting a handle on how the oscillation animation works, lets plot the first 24 animations that the Torus demo uses when left to its own devices. We get a variety of different trajectories, some relatively simple, some quite convoluted.

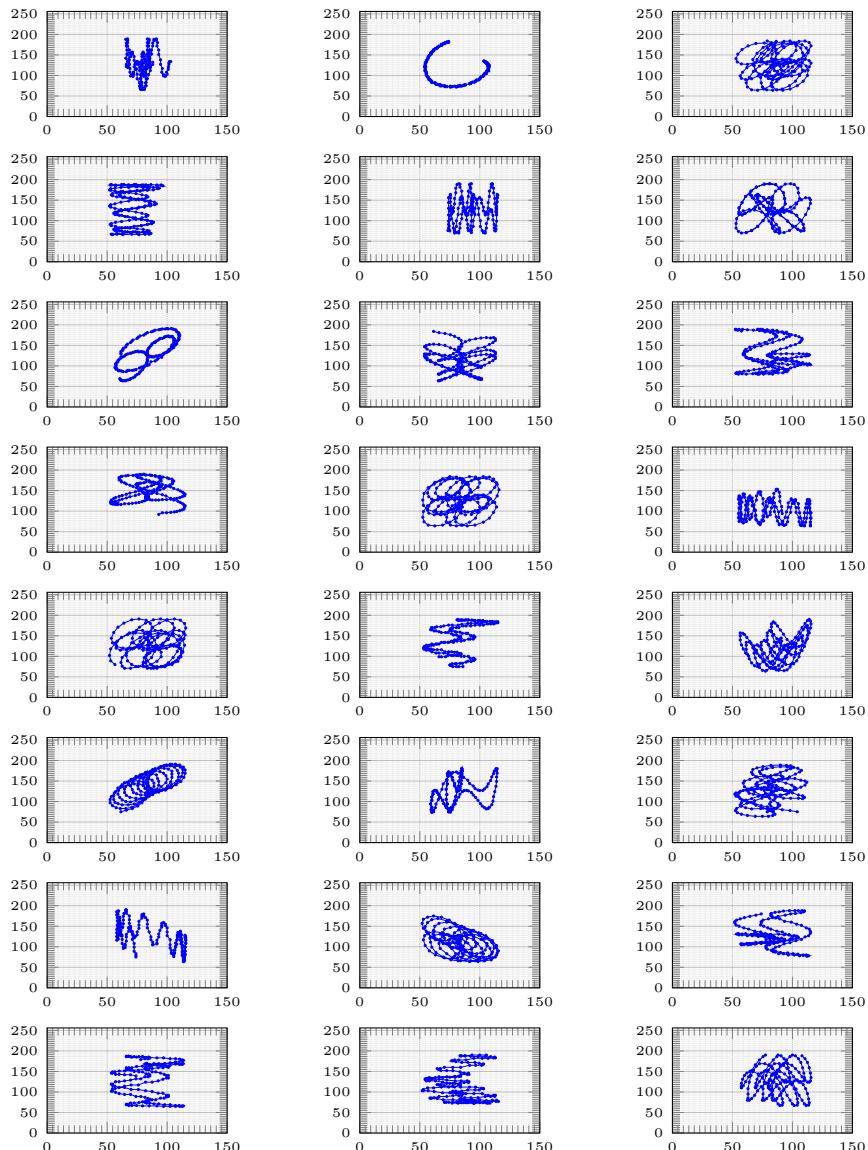


Figure 13.2: The first 24 oscillation patterns generated by the Torus demo.

When we look into the code we find this petting zoo of animations is principally driven by a simple sequence of bytes stored in `spritePositionArray`.

```
spritePositionArray .BYTE $40,$46,$4C,$52,$58,$5E,$63,$68
                   .BYTE $6D,$71,$75,$78,$7B,$7D,$7E,$7F
                   .BYTE $80,$7F,$7E,$7D,$7B,$78,$75,$71
                   .BYTE $6D,$68,$63,$5E,$58,$52,$4C,$46
                   .BYTE $40,$39,$33,$2D,$27,$21,$1C,$17
                   .BYTE $12,$0E,$0A,$07,$04,$02,$01,$00
                   .BYTE $00,$00,$01,$02,$04,$07,$0A,$0E
                   .BYTE $12,$17,$1C,$21,$27,$2D,$33,$39
                   .BYTE $FF
```

We can get a sense of how this rising and falling sequence of values can be used to plot a course across the screen if we treat each as an x and y value on a graph of cartesian co-ordinates. In the twenty four instances below we start by treating the value as providing both the x and y position. In each subsequent one we skip an increasing number of positions ahead in the sequence to get the y value, producing a variety of elliptical orbits around the screen.

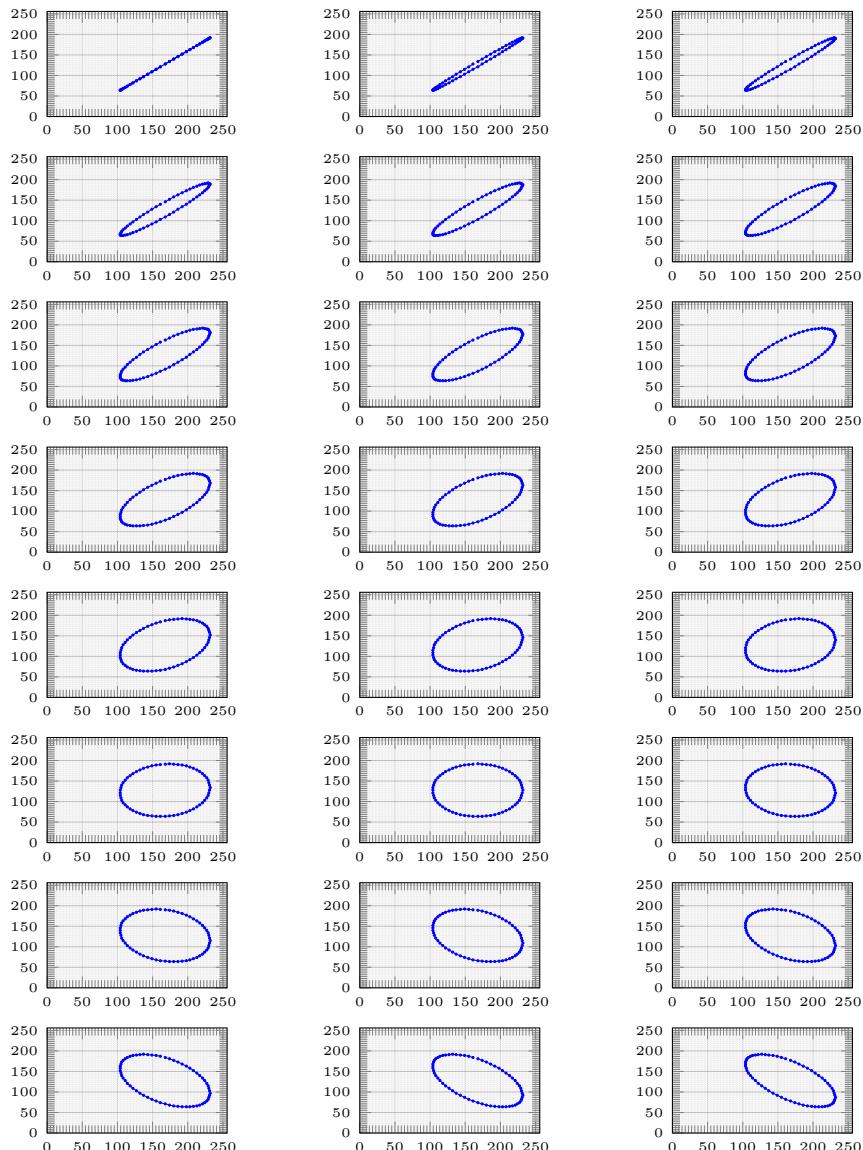


Figure 13.3: Using the x/y offset in spritePositionArray where y is the value after x in the array.

To get beyond simple ellipsoids we need to do more than pick a different value in the array for our x and y offsets. Here we experiment with something a little more involved. We update the x and y positions at different intervals and when skipping ahead in `spritePositionArray` for a new value for x and y we use a pre-selected, random number of bytes to skip past.

CHAPTER 13. AN OSCILLATOR IN 4 PARTS

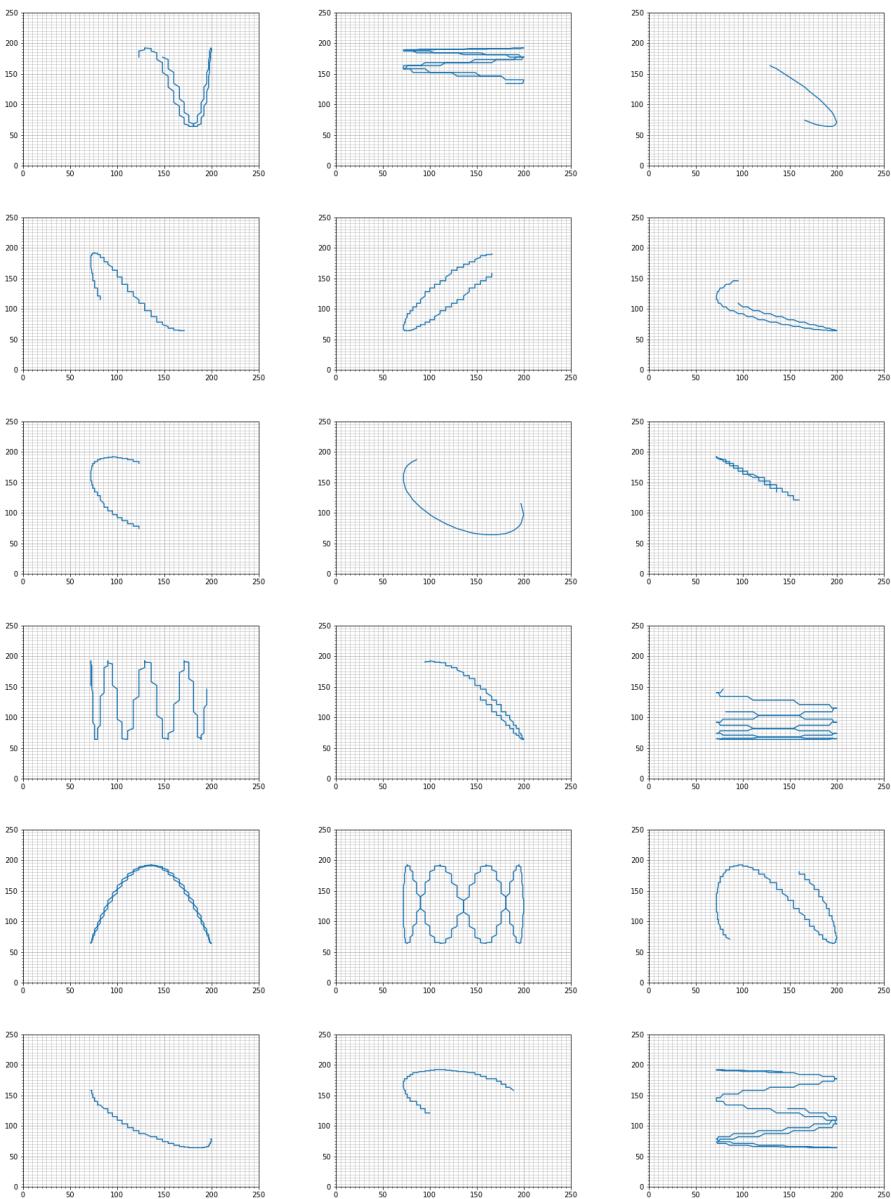


Figure 13.4: Testing different values of x and y

This is starting to look more like the actual results we observed and it is where the 4 values selectable by the player using keys z, x, c, and v in the Torus demo come in. In addition to controlling the music generation procedure, as we've already seen, they also determine the way the values in `spritePositionArray` are selected for position the sprite in each new frame. This is based on letting them determine the frequency with which the position of the x and y values of each sprite is changed and how far to skip ahead in `spritePositionArray` when selecting a new value from it for the x and y position.

Key	Name	Purpose	Code
Z	Oscillator 1	<ul style="list-style-type: none"> - Intervals between updating X position. - The amount to increment the index into <code>spritePositionArray</code> when getting the next X position. 	<pre> MaybeZKeyPressed CMP #\$0C BNE MaybeXKeyPressed ; Update Oscillator 1 LDA oscillator1Value CLC ADC #\$01 AND #\$0F STA oscillator1Value TAX LDA intervalBetweenPosUpdatesArray ,X STA initialCounterBetweenXPosUpdates LDA positionIncrementArray ,X STA incrementForXPos JMP ContinueCheckingForKeyPress </pre>
X	Oscillator 2	<ul style="list-style-type: none"> - Intervals between updating Y position. - The amount to increment the index into <code>spritePositionArray</code> when getting the next Y position. 	<pre> MaybeXKeyPressed CMP #\$17 BNE MaybeCKePressed ; Update Oscillator 2 LDA oscillator2Value CLC ADC #\$01 AND #\$0F STA oscillator2Value TAX LDA intervalBetweenPosUpdatesArray ,X STA initialCounterBetweenYPosUpdates LDA positionIncrementArray ,X STA incrementForYPos JMP ContinueCheckingForKeyPress </pre>
C	Oscillator 3	<ul style="list-style-type: none"> - How often to increase the index that seeks ahead to get a value from <code>spritePositionArray</code> for adding to the next X position. 	<pre> MaybeCKePressed CMP #\$14 BNE MaybeVKeyPressed ; Update Oscillator 3 LDA oscillator3Value CLC ADC #\$01 AND #\$0F STA oscillator3Value JMP ContinueCheckingForKeyPress </pre>
V	Oscillator 4	<ul style="list-style-type: none"> - How often to increase the index that seeks ahead to get a value from <code>spritePositionArray</code> for adding to the next Y position. 	<pre> MaybeVKeyPressed CMP #\$1F BNE MaybeF1Pressed ; Update Oscillator 4 LDA oscillator4Value CLC ADC #\$01 AND #\$0F STA oscillator4Value </pre>

Figure 13.5: The purpose of each of the oscillator values.

CHAPTER 13. AN OSCILLATOR IN 4 PARTS

In `RunMainInterruptHandler()` we can see how each of these values set by the player is used to maintain an accounting of the different sprite positions for each of the 8 sprites:

```
RunMainInterruptHandler
LDY #$00
LDA #$F0
STA $D012 ;Raster Position
DEC counterBetweenXPosUpdates
BNE MaybeUpdateYPos

UpdateXPos
LDA initialCounterBetweenXPosUpdates
STA counterBetweenXPosUpdates

LDA incrementForXPos
CLC
ADC indexForXPosInSpritePositionArray
STA indexForXPosInSpritePositionArray

MaybeUpdateYPos
DEC counterBetweenYPosUpdates
BNE MaybeUpdateXPosOffset

LDA initialCounterBetweenYPosUpdates
STA counterBetweenYPosUpdates

LDA indexForYPosInSpritePositionArray
CLC
ADC incrementForYPos
STA indexForYPosInSpritePositionArray

MaybeUpdateXPosOffset
DEC cyclesBetweenXPosOffsetUpdates
BNE MaybeUpdateYPosOffset

LDA oscillator3Value
STA cyclesBetweenXPosOffsetUpdates
INC indexForXPosOffsetInSpritePositionArray

MaybeUpdateYPosOffset
DEC cyclesBetweenYPosOffsetUpdates
BNE StoreInitialIndexValues

LDA oscillator4Value
STA cyclesBetweenYPosOffsetUpdates
INC indexForYPosOffsetInSpritePositionArray
```

Before animating each of the 8 sprites we use the values set by the player to prepare the variables that will be applied to positioning each sprite. For example the value selected with the Z key has been used to set `initialCounterBetweenXPosUpdates` and `incrementForXPos`. In the first lines above in `UpdateXPos` we use them to set up `indexForXPosInSpritePositionArray`. This is then used in `SpriteAnimationLoop` to select the X position of the current sprite:

```
SpriteAnimationLoop
LDA indexForXPosInSpritePositionArray
AND #$3F
TAX
LDA spritePositionArray,X
STA currSpriteXPos
```

You can follow the same lineage between the setting of each value in our table above with the rest of the `SpriteAnimationLoop` routine.

Iridis Oops!

14.1 The Byte that Broke

We must apologise to some of the earliest purchasers of IA, as well... it seems that some data was corrupted during the production phase of the data duplication, and the first few hundred copies of the game used to bug out if you lost at the Bonus Phase All the buggy copies known of have been replaced, and all current tapes are fine, but if you find you have a buggy version you should send it back to those awfully nice Hewsons people and they'll give you an unglitchified one,

— Jeff Minter's Newsletter 'The Nature of the Beast' August 1986 [?]

What happened is that the master tape had dropped the very last byte in the second section of game data on the tape.

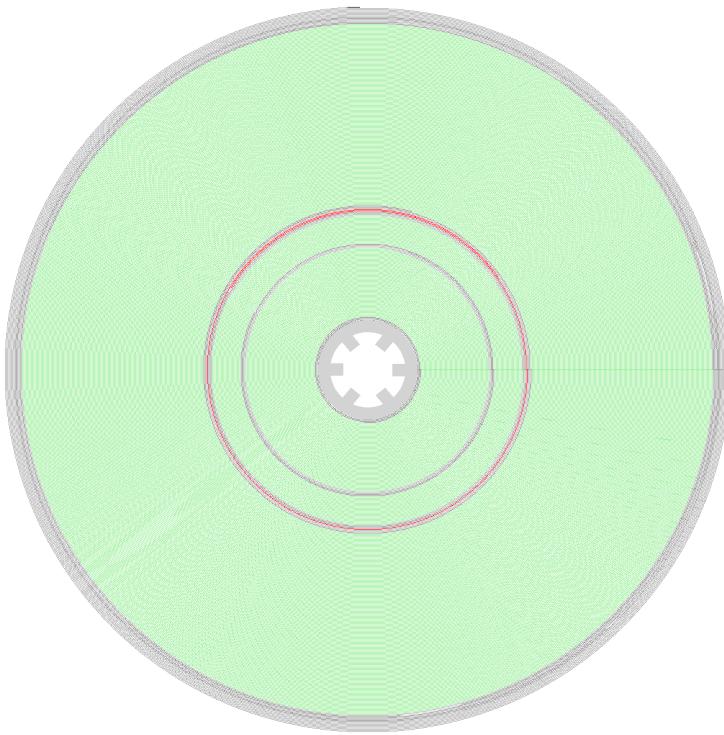


Figure 14.1: The problematic section of data highlighted in red.

Start Address	End Address	Note
0800	BFFE	.
BF00	BFFF	.
C000	CFE	.
E000	F7FF	.

Figure 14.2: Chunk BF00 is missing its last byte.

```
$BFFF A2 07      LDX #$07
$C001 A9 08      LDA #$08
$C003 8D F8 BF    STA $BFF8
```

Listing 14.1: The data segment as it should be with \$A2 at \$BFFF

```
$BFFF 00          BRK
$C000 07 A9        SLO $A9
```

```
$C002 08      PHP
$C003 8D F8 BF STA $BFFF8
```

Listing 14.2: The corrupt byte with \$00 at \$BFFF

14.2 Reappearing Enemies

When you start a new game, enemies from the previous game show up in the first wave. For most people starting out, this will take the form of a few residual 'licker ships' zapping them just as they're getting started.

This bug happens because the 'wave' data isn't cleared down when a new game starts. So whatever is in there from the previous game gets used until they're flushed out by being killed and replaced with the level's proper enemy data.

This isn't a problem for the first game after Iridis Alpha is loaded because the first level's data is hardcoded in there.

The fix is simple enough, we initialize the active wave data stored in 'activeShipsWaveDataLoPtrArray' and 'activeShipsWaveDataHiPtrArray' with the first level's data whenever we start a new game.

```
LDA #$0F
STA $D418 ; Select Filter Mode and Volume
JSR ClearPlanetTextureCharsets
JSR InitializeActiveShipArray ; Added to fix the bug.
JMP PrepareToLaunchIridisAlpha

; -----
; InitializeActiveShipArray
; -----
InitializeActiveShipArray
LDX #$00
InitializeActiveShipLoop
LDA <planet1Level1Data
STA activeShipsWaveDataLoPtrArray ,X
LDA >planet1Level1Data
STA activeShipsWaveDataHiPtrArray ,X
INX
CPX #$10
BNE InitializeActiveShipLoop
RTS
```

Listing 14.3: Fixing the reappearing enemy bug

14.3 A Sort of Cheat

After a minute or two in the title screen, the game enters 'Attract Mode' and plays a random level on autopilot for a few seconds. If you press F1 during this play you enter the 'Made in France' pause-mode mini game. If you press F1 again you can now start playing the level 'Attract Mode' selected at random.

This is because the 'CheckKeyboardInGame' routine doesn't try to prevent you from entering 'Pause Mode' while Attract Mode is running:

```
CheckKeyboardInGame
    LDA lastKeyPressed
    CMP #$40 ; $40 means no key was pressed
    BNE KeyWasPressed
    LDA #$00
    STA f1WasPressed
ReturnEarlyFromKeyboardCheck
    RTS

KeyWasPressed
    LDY f1WasPressed
    BNE ReturnEarlyFromKeyboardCheck
    LDY attractModeCountdown
    BEQ b787C
    ; If a key is pressed during attract mode, accelerate the
    ; countdown so that it exits it nearly immediately.
    LDY #$02
    STY attractModeCountdown
```

Listing 14.4: Fixing the reappearing enemy bug

Sprite Atlas

APPENDIX A. SPRITE ATLAS

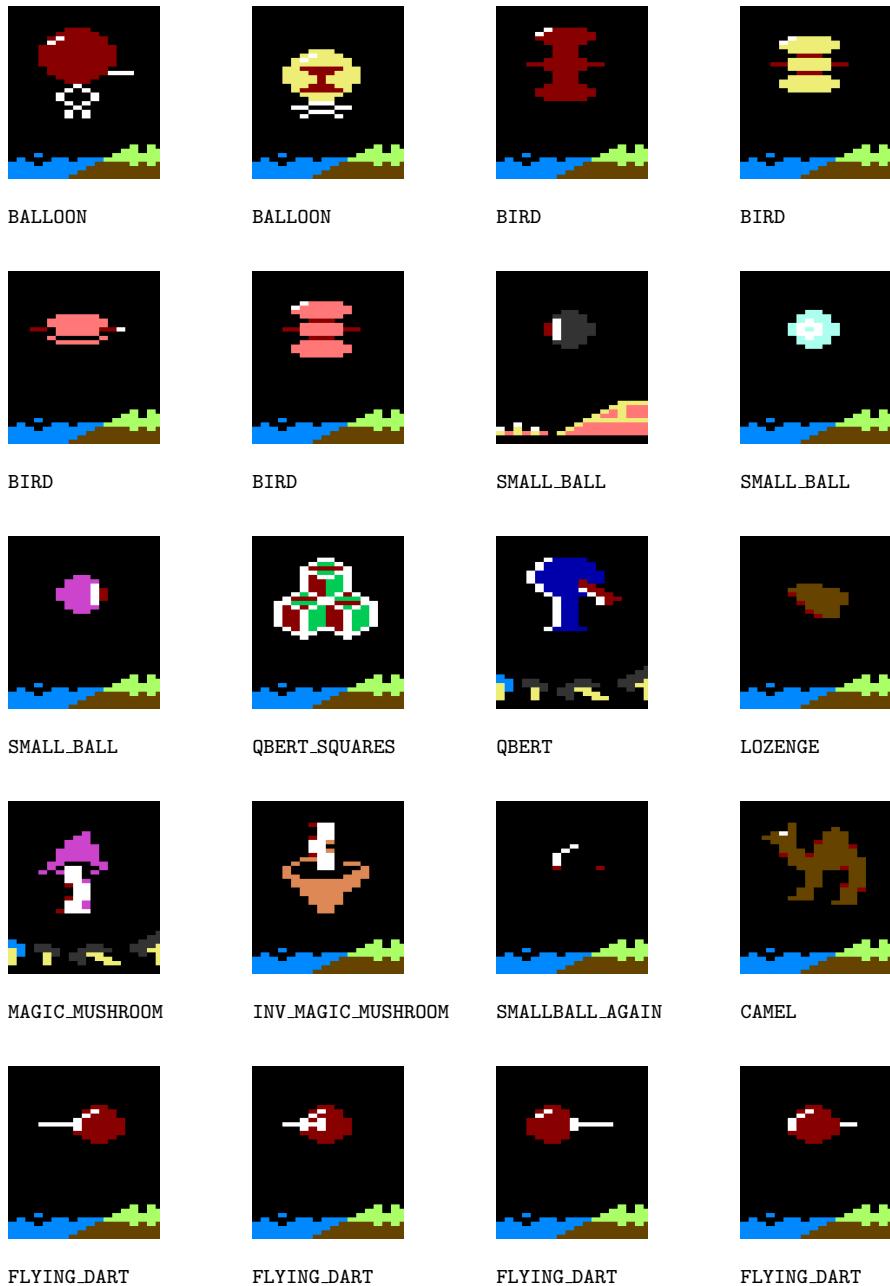


Figure A.1: Sprites

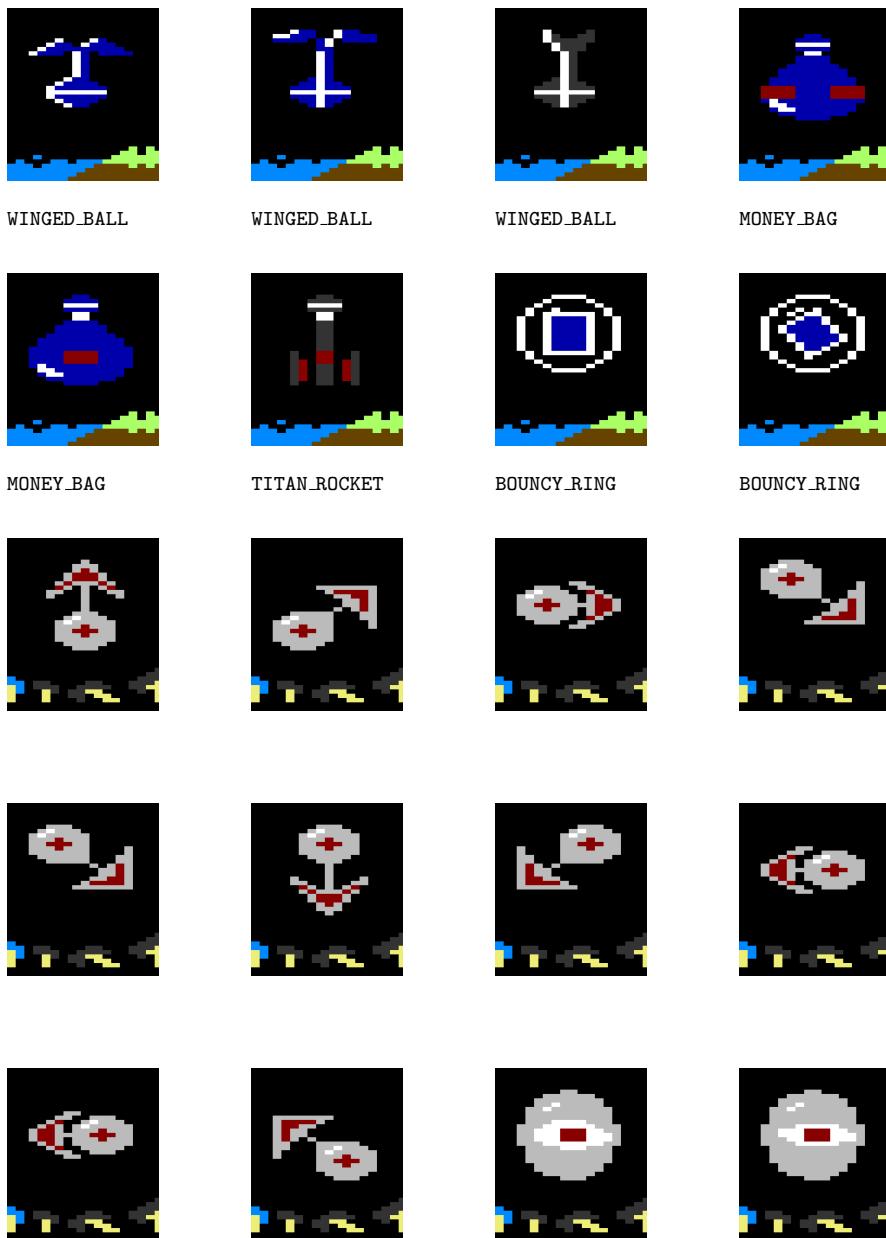


Figure A.2: Sprites

APPENDIX A. SPRITE ATLAS

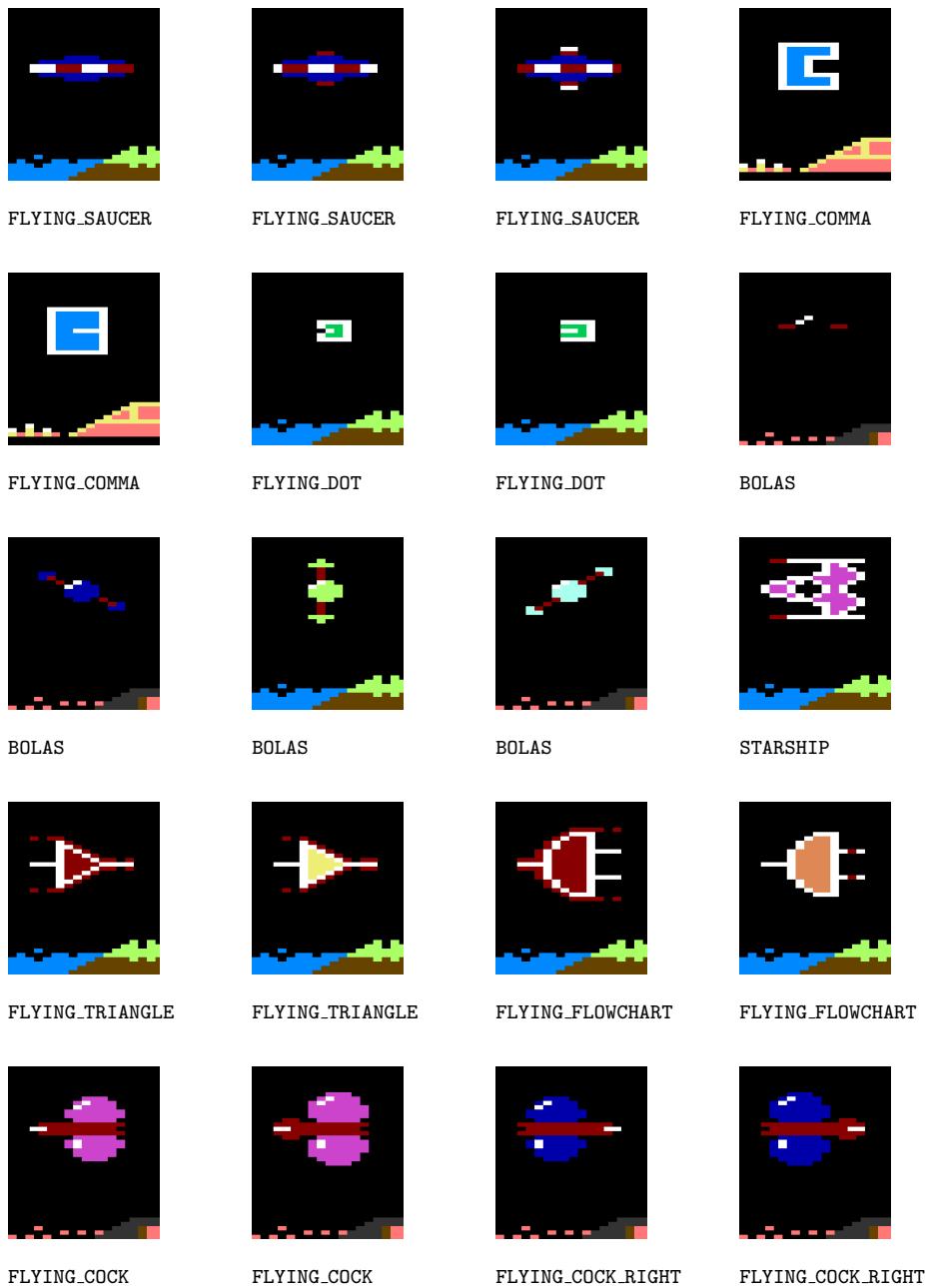


Figure A.3: Sprites

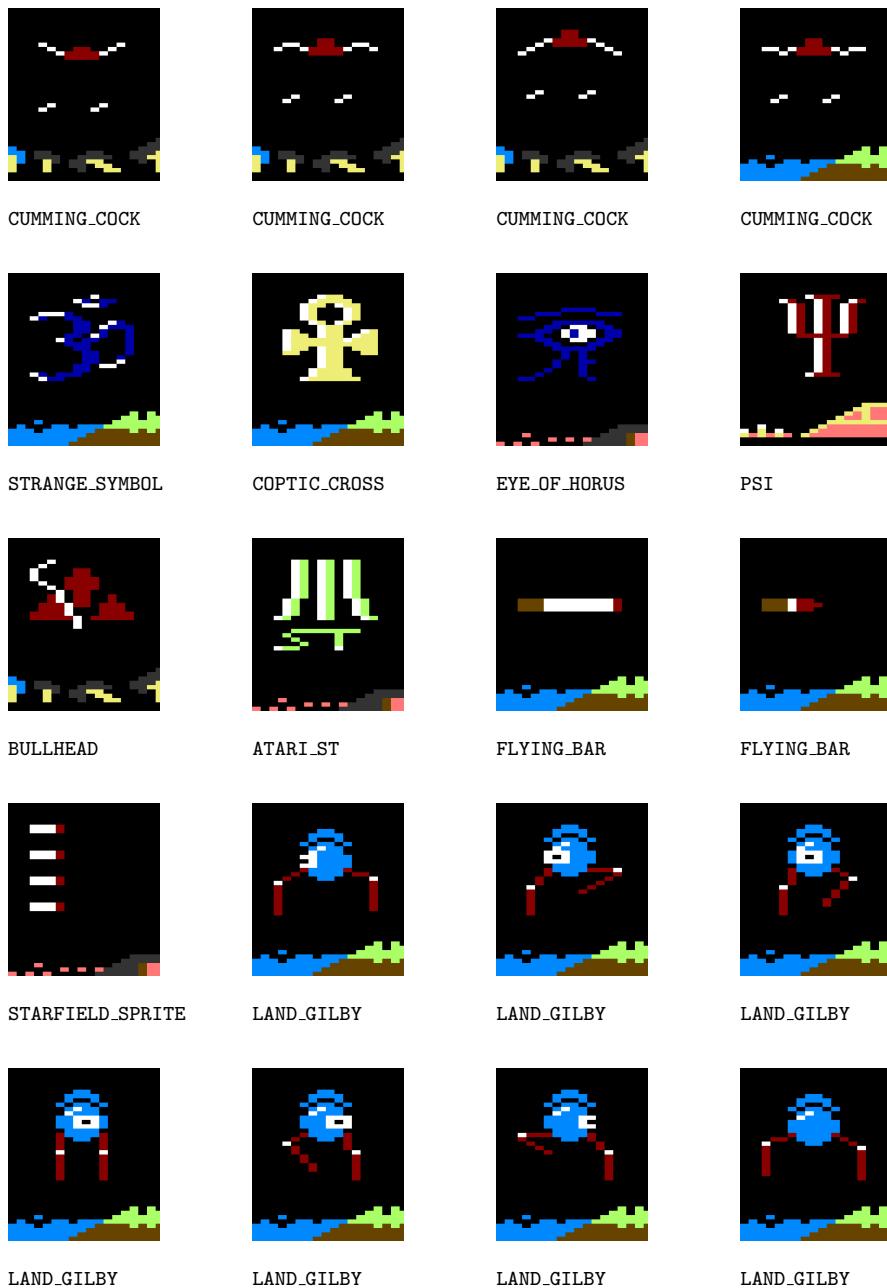


Figure A.4: Sprites

APPENDIX A. SPRITE ATLAS



LAND_GILBY



LAND_GILBY



LAND_GILBY



LAND_GILBY



GILBY_TAKING_OFF



GILBY_TAKING_OFF



GILBY_TAKING_OFF



GILBY_TAKING_OFF



GILBY_TAKING_OFF



GILBY_AIRBORNE_LEFT



GILBY_AIRBORNE_TURNING



GILBY_AIRBORNE_RIGHT



LAND_GILBY_LOWERPLANET



LAND_GILBY_LOWERPLANET



LAND_GILBY_LOWERPLANET



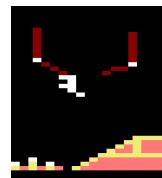
LAND_GILBY_LOWERPLANET



LAND_GILBY_LOWERPLANET



LAND_GILBY_LOWERPLANET



LAND_GILBY_LOWERPLANET



LAND_GILBY_LOWERPLANET

Figure A.5: Sprites



LAND_GILBY_LOWERPLANET LAND_GILBY_LOWERPLANET LAND_GILBY_LOWERPLANET GILBY_TAKING_OFF_LOWERPLANET



GILBY_TAKING_OFF_LOWERPLANET TAKING_OFF_LOWERPLANET TAKING_OFF_LOWERPLANET TAKING_OFF_LOWERPLANET TAKING_OFF_LOWERPLANET



GILBY_AIRBORNE_LOWERPLANET GILBY_AIRBORNE_TURNING GILBY_AIRBORNE_LOWERPLANET GILBY_AIRBORNE_LOWERPLANET GILBY_AIRBORNE_LOWERPLANET



SPINNING_RING

SPINNING_RING

SPINNING_RING

SPINNING_RING



LASER_BULLET

EXPLOSION_START

EXPLOSION_MIDDLE

EXPLOSION_END

Figure A.6: Sprites

APPENDIX A. SPRITE ATLAS

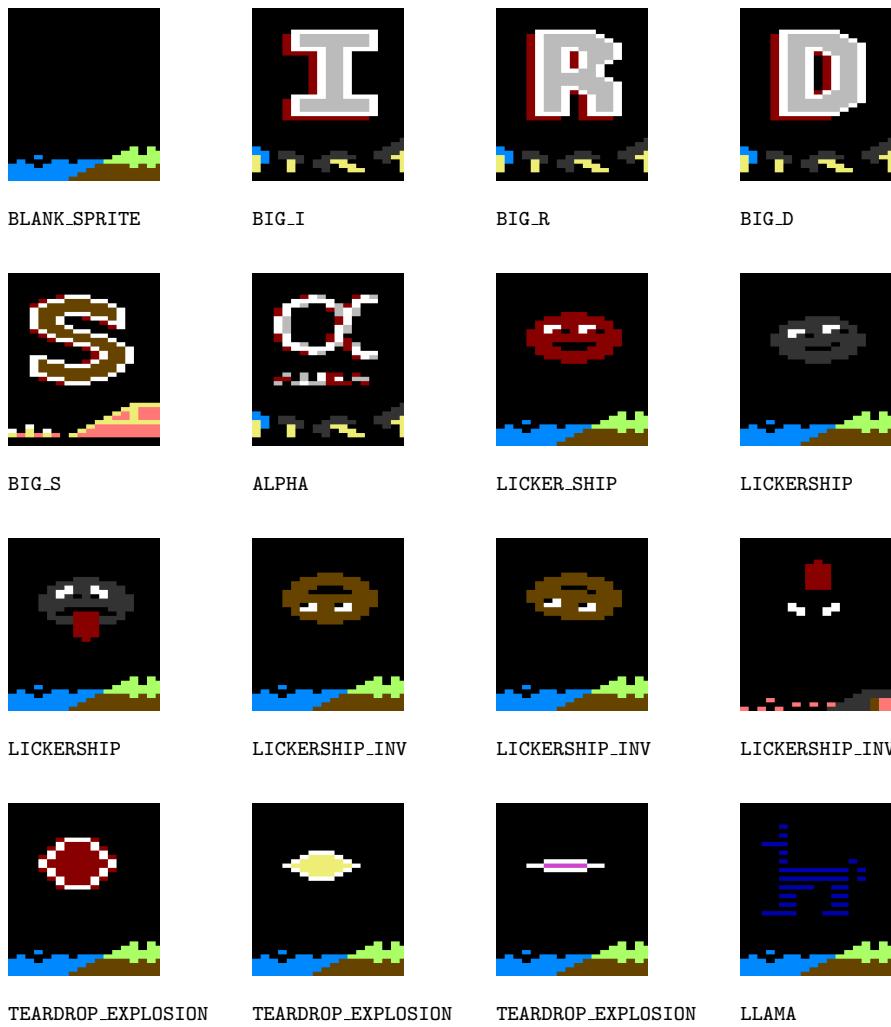
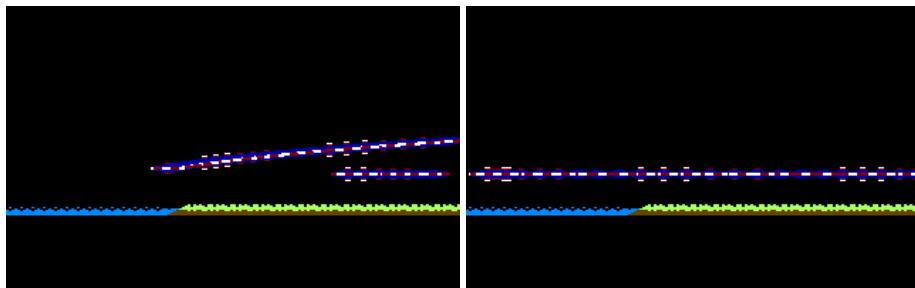


Figure A.7: Sprites

Enemy Data

This section provides the level data for each wave of enemies in each planet. Figures are provided to indicate the appearance and movement pattern of the enemy.

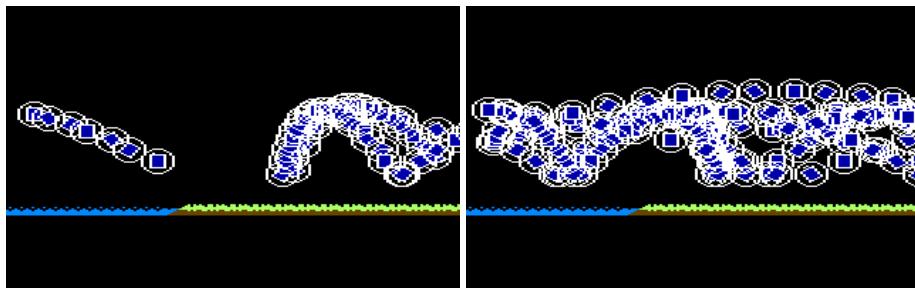
Sheep Planet - Level 1 Data

Sample enemy movement for Sheep planet, level 1.

Byte	Value	Description
Byte 1	\$06	Index into array for sprite color
Byte 3	FLYING_SAUCER1	Sprite value for the attack ship on the upper planet
Byte 4	\$03	The animation frame rate for the attack ship.
Byte 6	FLYING_SAUCER1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$40	Update rate for attack wave
Byte 18	planet1Level1Data2ndStage	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$06	X Pos movement for attack ship.
Byte 20	\$01	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$01	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$00	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	spinningRings	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$02	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$02	Does colliding with this enemy decrease the gilby's energy?.
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?.
Byte 38	\$04	Number of waves in data.
Byte 39	\$18	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Sheep Planet - Level 1 .

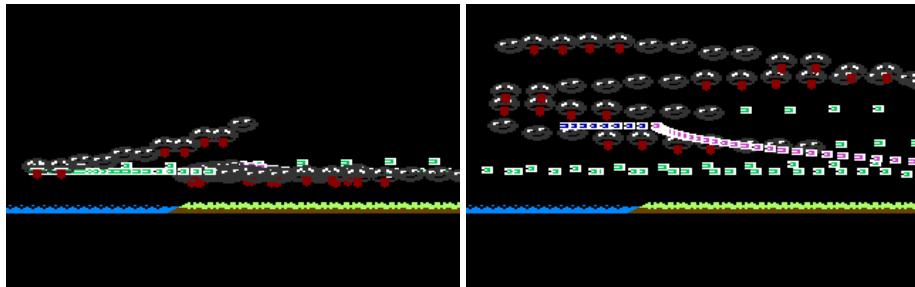
Sheep Planet - Level 2 Data



Sample enemy movement for Sheep planet, level 2.

Byte	Value	Description
Byte 1	\$06	Index into array for sprite color
Byte 3	BOUNCY_RING	Sprite value for the attack ship on the upper planet
Byte 4	\$01	The animation frame rate for the attack ship.
Byte 6	BOUNCY_RING	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$24	Y Pos movement pattern for attack ship.
Byte 21	\$02	X Pos Frame Rate for Attack ship.
Byte 22	\$01	Y Pos Frame Rate for Attack ship.
Byte 23	\$01	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	planet1Level2Data	Hi Ptr for another set of wave data.
Byte 30	spinningRings	Hi Ptr for Explosion animation.
Byte 32	planet1Level2Data	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$01	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$01	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Sheep Planet - Level 2 .

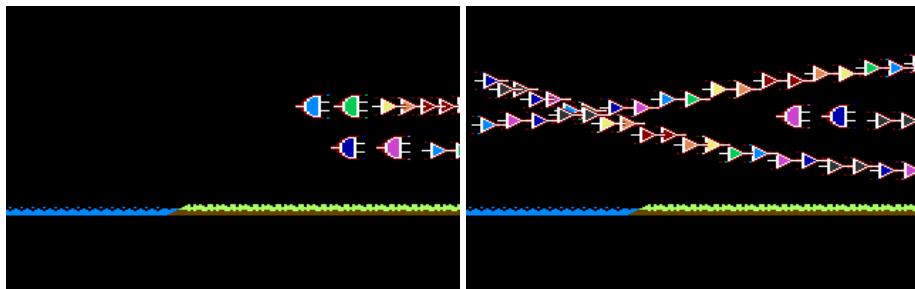
Sheep Planet - Level 3 Data

Sample enemy movement for Sheep planet, level 3.

Byte	Value	Description
Byte 1	\$05	Index into array for sprite color
Byte 3	FLYING_DOT1	Sprite value for the attack ship on the upper planet
Byte 4	\$04	The animation frame rate for the attack ship.
Byte 6	FLYING_DOT1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$30	Update rate for attack wave
Byte 18	planet1Level3Data2ndStage	Hi Ptr to the wave data we switch to when first hit.
Byte 19	SFA	X Pos movement for attack ship.
Byte 20	\$01	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$00	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	lickerShipWaveData	Hi Ptr for Explosion animation.
Byte 32	lickerShipWaveData	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$02	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$01	Does colliding with this enemy decrease the gilby's energy?.
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?.
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Sheep Planet - Level 3 .

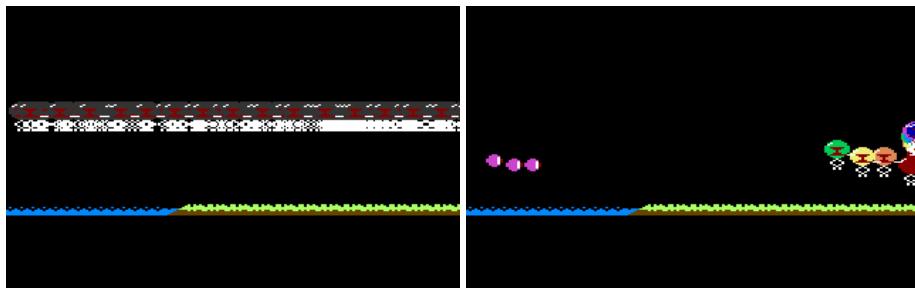
Sheep Planet - Level 4 Data



Sample enemy movement for Sheep planet, level 4.

Byte	Value	Description
Byte 1	\$11	Index into array for sprite color
Byte 3	FLYING_TRIANGLE1	Sprite value for the attack ship on the upper planet.
Byte 4	\$03	The animation frame rate for the attack ship.
Byte 6	FLYING_TRIANGLE1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$60	Update rate for attack wave
Byte 18	planet1Level4Data2ndStage	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$07	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet1Level4Data2ndStage	Hi Ptr for Explosion animation.
Byte 32	planet1Level4Data2ndStage	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$04	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$02	Does colliding with this enemy decrease the gilby's energy?.
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?.
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Sheep Planet - Level 4 .

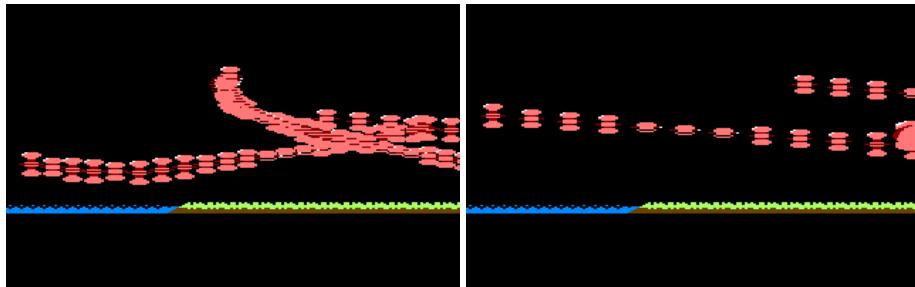
Sheep Planet - Level 5 Data

Sample enemy movement for Sheep planet, level 5.

Byte	Value	Description
Byte 1	\$11	Index into array for sprite color
Byte 3	BALLOON	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	BIRD1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	SFC	X Pos movement for attack ship.
Byte 20	\$23	Y Pos movement pattern for attack ship.
Byte 21	\$02	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	planet1Level5Data2ndStage	Hi Ptr for another set of wave data.
Byte 30	planet1Level5Data3rdStage	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$05	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Sheep Planet - Level 5 .

Sheep Planet - Level 6 Data



Sample enemy movement for Sheep planet, level 6.

Byte	Value	Description
Byte 1	\$0A	Index into array for sprite color
Byte 3	BIRD1	Sprite value for the attack ship on the upper planet.
Byte 4	\$03	The animation frame rate for the attack ship.
Byte 6	BIRD1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$03	Update rate for attack wave
Byte 18	planet1Level6Data2ndStage	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$01	Y Pos Frame Rate for Attack ship.
Byte 23	\$01	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	spinningRings2ndType	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$01	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$04	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$10	Number of ships in wave.
Byte 40	\$00	Unused bytes.

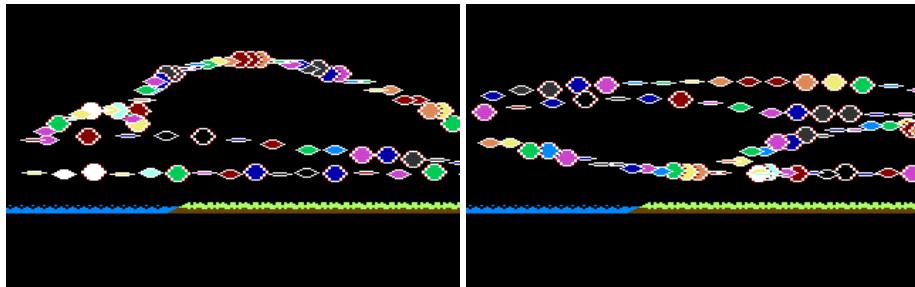
Sheep Planet - Level 6 .

Sheep Planet - Level 7 Data

Byte	Value	Description
Byte 1	\$09	Index into array for sprite color
Byte 3	FLAG_BAR	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	FLAG_BAR	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$50	Update rate for attack wave
Byte 18	planet1Level7Data2ndStage	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$07	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet1Level7Data2ndStage	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$03	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$28	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Sheep Planet - Level 7 .

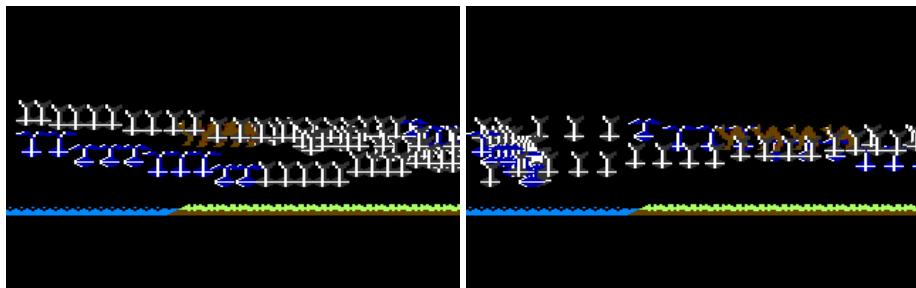
Sheep Planet - Level 8 Data



Sample enemy movement for Sheep planet, level 8.

Byte	Value	Description
Byte 1	\$11	Index into array for sprite color
Byte 3	TEARDROP_EXPLOSION1	Sprite value for the attack ship on the upper planet.
Byte 4	\$03	The animation frame rate for the attack ship.
Byte 6	TEARDROP_EXPLOSION1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$05	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet1Level8Data2ndStage	Hi Ptr for Explosion animation.
Byte 32	planet1Level8Data2ndStage	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$02	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$02	Does colliding with this enemy decrease the gilby's energy?.
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?.
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Sheep Planet - Level 8 .

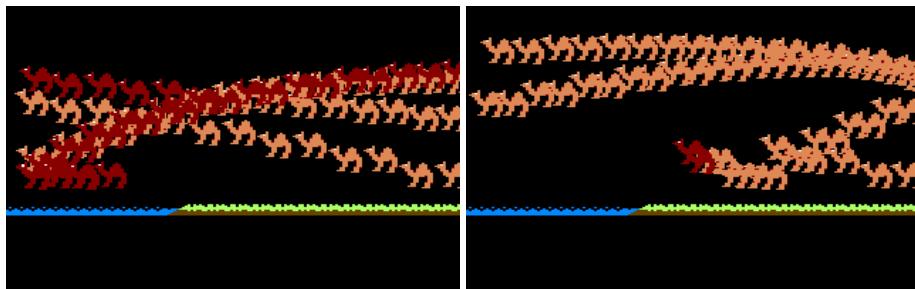
Sheep Planet - Level 9 Data

Sample enemy movement for Sheep planet, level 9.

Byte	Value	Description
Byte 1	\$06	Index into array for sprite color
Byte 3	WINGBALL	Sprite value for the attack ship on the upper planet
Byte 4	\$03	The animation frame rate for the attack ship.
Byte 6	MONEY.BAG	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$0C	Update rate for attack wave
Byte 18	planet1Level9DataSecondStage	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$FC	X Pos movement for attack ship.
Byte 20	\$23	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$03	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$00	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	planet1Level9DataSecondStage	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	defaultExplosion	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$08	Does colliding with this enemy decrease the gilby's energy?.
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Sheep Planet - Level 9 .

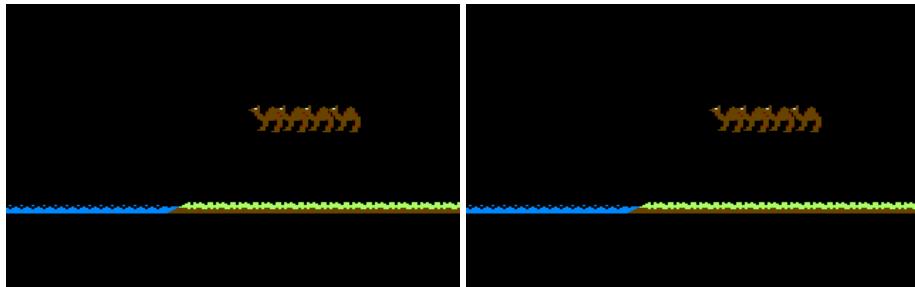
Sheep Planet - Level 10 Data



Sample enemy movement for Sheep planet, level 10.

Byte	Value	Description
Byte 1	\$08	Index into array for sprite color
Byte 3	CAMEL	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	INV_MAGIC_MUSHROOM	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$80	Update rate for attack wave
Byte 18	planet1Level10Data2ndStage	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$25	Y Pos movement pattern for attack ship.
Byte 21	\$00	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	planet1Level10Data	Hi Ptr for another set of wave data.
Byte 30	planet1Level10Data2ndStage	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$06	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$18	Number of ships in wave.
Byte 40	\$00	Unused bytes.

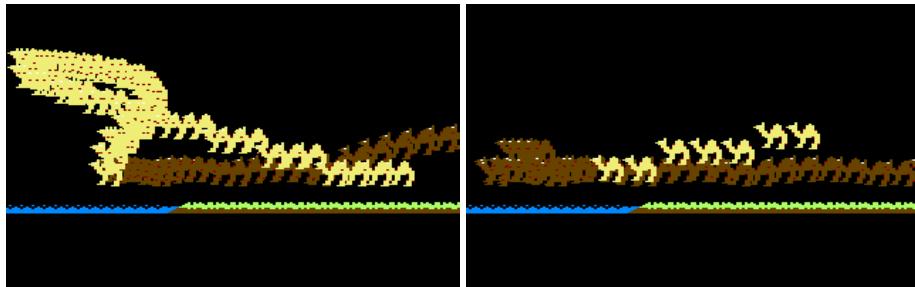
Sheep Planet - Level 10 .

Sheep Planet - Level 11 Data

Sample enemy movement for Sheep planet, level 11.

Byte	Value	Description
Byte 1	\$0E	Index into array for sprite color
Byte 3	GILBY_AIRBORNE_LEFT	Sprite value for the attack ship on the upper planet
Byte 4	\$06	The animation frame rate for the attack ship.
Byte 6	GILBY_AIRBORNE_LOWERPLANET_RIGHT	Sprite value for the attack ship on lower planet
Byte 7	\$03	Whether a specific attack behaviour is used.
Byte 9	smallDotWaveData	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$02	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	secondExplosionAnimation	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$04	Does destroying this enemy increase the gilby's energy?
Byte 36	\$05	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$10	Number of ships in wave.
Byte 40	\$00	Unused bytes.

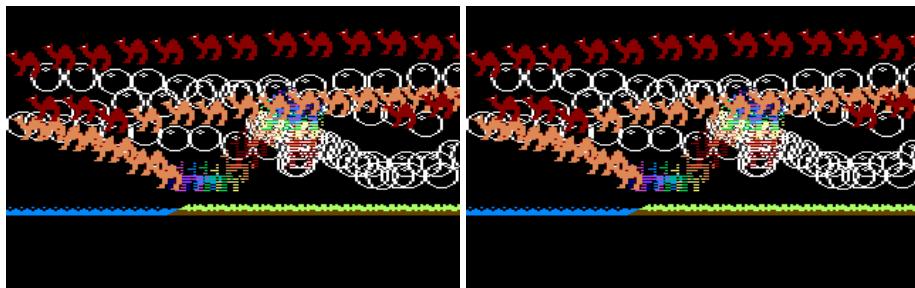
Sheep Planet - Level 11 .

Sheep Planet - Level 12 Data

Sample enemy movement for Sheep planet, level 12.

Byte	Value	Description
Byte 1	\$09	Index into array for sprite color
Byte 3	CAMEL	Sprite value for the attack ship on the upper planet
Byte 4	\$02	The animation frame rate for the attack ship.
Byte 6	LICKERSHIP_INV1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$FC	X Pos movement for attack ship.
Byte 20	\$21	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$01	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	planet1Level12Data	Hi Ptr for another set of wave data.
Byte 30	planet1Level2Data2ndStage	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$03	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$02	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Sheep Planet - Level 12 .

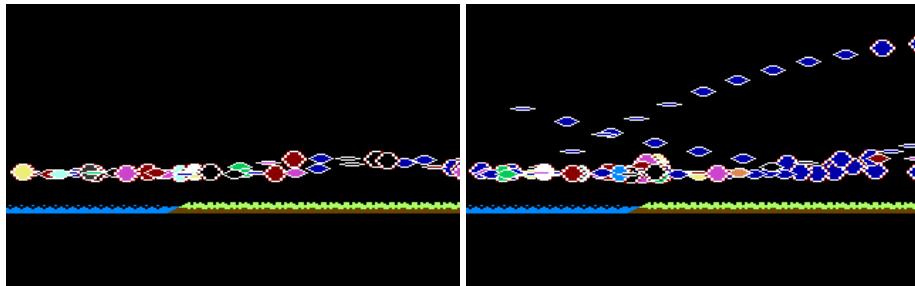
Sheep Planet - Level 13 Data

Sample enemy movement for Sheep planet, level 13.

Byte	Value	Description
Byte 1	\$0B	Index into array for sprite color
Byte 3	BUBBLE	Sprite value for the attack ship on the upper planet
Byte 4	\$04	The animation frame rate for the attack ship.
Byte 6	BUBBLE	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$24	Y Pos movement pattern for attack ship.
Byte 21	\$02	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$01	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	planet1Level13Data	Hi Ptr for another set of wave data.
Byte 30	planet1Level13Data2ndStage	Hi Ptr for Explosion animation.
Byte 32	planet1Level13Data2ndStage	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$04	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$02	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Sheep Planet - Level 13 .

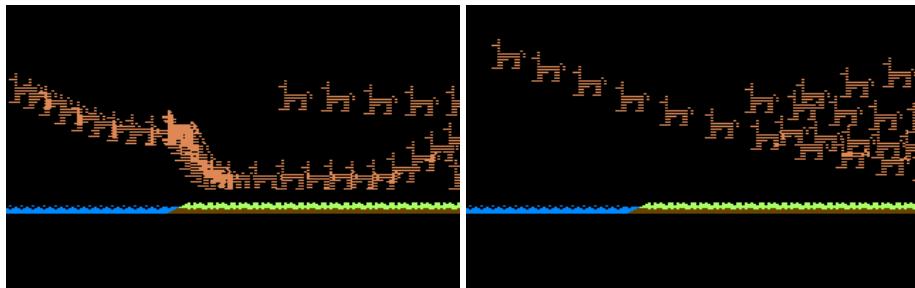
Sheep Planet - Level 14 Data



Sample enemy movement for Sheep planet, level 14.

Byte	Value	Description
Byte 1	\$06	Index into array for sprite color
Byte 3	TEARDROP_EXPLOSION1	Sprite value for the attack ship on the upper planet
Byte 4	\$05	The animation frame rate for the attack ship.
Byte 6	TEARDROP_EXPLOSION1	Sprite value for the attack ship on lower planet
Byte 7	\$03	Whether a specific attack behaviour is used.
Byte 9	planet1Level8Data	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$FA	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$01	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet1Level8Data	Hi Ptr for Explosion animation.
Byte 32	planet1Level8Data	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$08	Does colliding with this enemy decrease the gilby's energy?.
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$10	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Sheep Planet - Level 14 .

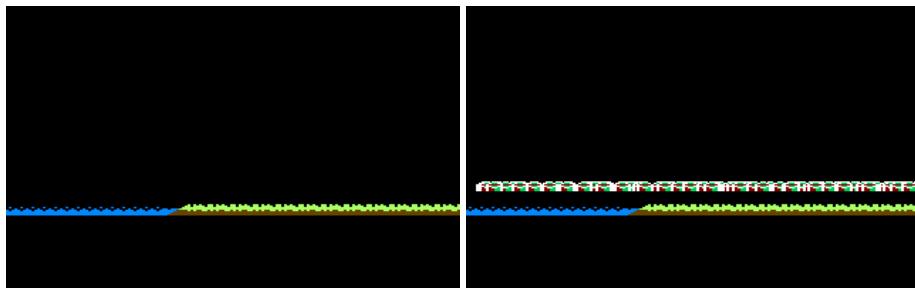
Sheep Planet - Level 15 Data

Sample enemy movement for Sheep planet, level 15.

Byte	Value	Description
Byte 1	\$08	Index into array for sprite color
Byte 3	LLAMA	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	LLAMA	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$10	Update rate for attack wave
Byte 18	planet1Level15Data	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$01	Y Pos Frame Rate for Attack ship.
Byte 23	\$01	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	teardropExplosion	Hi Ptr for Explosion animation.
Byte 32	lickerShipWaveData	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$03	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$03	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Sheep Planet - Level 15 .

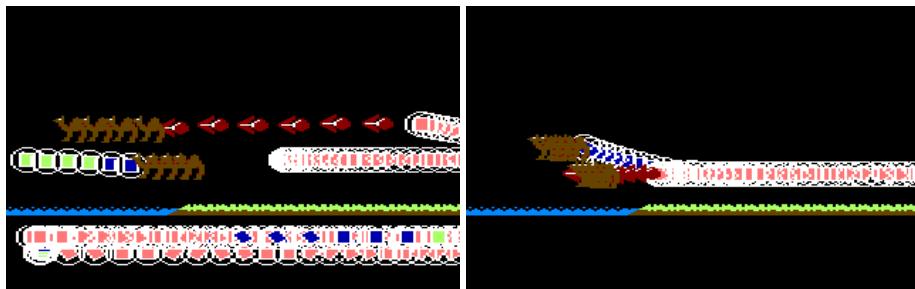
Sheep Planet - Level 16 Data



Sample enemy movement for Sheep planet, level 16.

Byte	Value	Description
Byte 1	\$05	Index into array for sprite color
Byte 3	QBERT_SQUARES	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	QBERT_SQUARES	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$02	X Pos Frame Rate for Attack ship.
Byte 22	\$00	Y Pos Frame Rate for Attack ship.
Byte 23	\$01	Stickiness factor, does the enemy stick to the player
Byte 24	\$00	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet4Level19Data	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$06	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$18	Number of ships in wave.
Byte 40	\$00	Unused bytes.

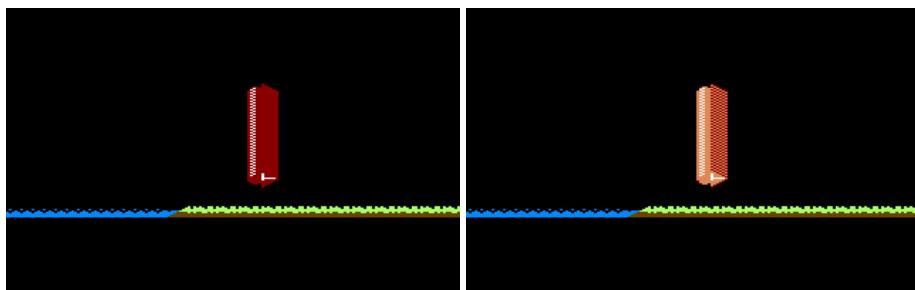
Sheep Planet - Level 16 .

Sheep Planet - Level 17 Data

Sample enemy movement for Sheep planet, level 17.

Byte	Value	Description
Byte 1	\$0A	Index into array for sprite color
Byte 3	BOUNCY_RING	Sprite value for the attack ship on the upper planet.
Byte 4	\$02	The animation frame rate for the attack ship.
Byte 6	BOUNCY_RING	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$40	Update rate for attack wave
Byte 18	planet1Level17Data2ndStage	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$80	X Pos movement for attack ship.
Byte 20	\$80	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$01	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$00	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	gilbyLookingLeft	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$05	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$0C	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Sheep Planet - Level 17 .

Sheep Planet - Level 18 Data

Sample enemy movement for Sheep planet, level 18.

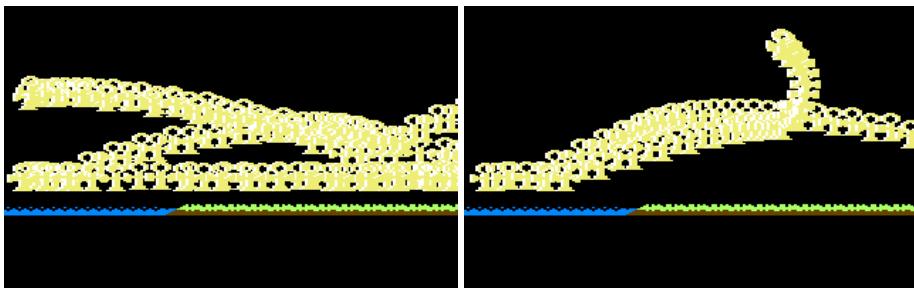
Byte	Value	Description
Byte 1	\$05	Index into array for sprite color
Byte 3	GILBY_AIRBORNE_RIGHT	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	GILBY_AIRBORNE_RIGHT	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$06	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet1Level18Data2ndStage	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$03	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Sheep Planet - Level 18 .

Sheep Planet - Level 19 Data

Byte	Value	Description
Byte 1	\$04	Index into array for sprite color
Byte 3	STARSHIP	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	STARSHIP	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$20	Update rate for attack wave
Byte 18	planet1Level19Data	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$02	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$80	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet5Level6Data	Hi Ptr for Explosion animation.
Byte 32	planet5Level6Data	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$04	Does colliding with this enemy decrease the gilby's energy?.
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?.
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Sheep Planet - Level 19 .

Sheep Planet - Level 20 Data

Sample enemy movement for Sheep planet, level 20.

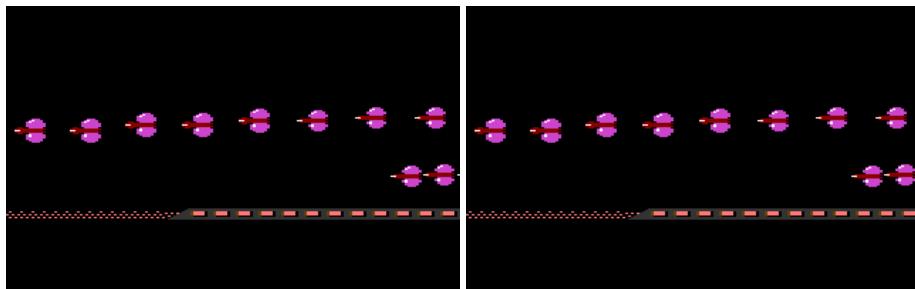
Byte	Value	Description
Byte 1	\$07	Index into array for sprite color
Byte 3	COPTIC_CROSS	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	COPTIC_CROSS	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$04	X Pos movement for attack ship.
Byte 20	\$24	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	copticExplosion	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet1Level20Data	Hi Ptr for Explosion animation.
Byte 32	planet1Level20Data	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$01	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$01	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$40	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Sheep Planet - Level 20 .

Tech Planet - Level 1 Data

Byte	Value	Description
Byte 1	\$55	Index into array for sprite color
Byte 3	LITTLE_DART	Sprite value for the attack ship on the upper planet
Byte 4	\$01	The animation frame rate for the attack ship.
Byte 6	LITTLE_DART	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$08	Update rate for attack wave
Byte 18	planet2Level1Data	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$01	Stickiness factor, does the enemy stick to the player
Byte 24	\$10	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	pinAsExplosion	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$01	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$02	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$18	Number of ships in wave.
Byte 40	\$00	Unused bytes.

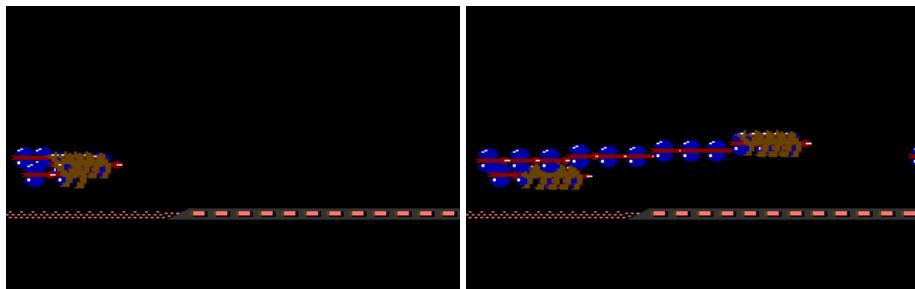
Tech Planet - Level 1 .

Tech Planet - Level 2 Data

Sample enemy movement for Tech planet, level 2.

Byte	Value	Description
Byte 1	\$04	Index into array for sprite color
Byte 3	FLYING_COCK1	Sprite value for the attack ship on the upper planet.
Byte 4	\$05	The animation frame rate for the attack ship.
Byte 6	FLYING_COCK1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$E9	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	secondExplosionAnimation	Hi Ptr for Explosion animation.
Byte 32	lickerShipWaveData	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$02	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$02	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$18	Number of ships in wave.
Byte 40	\$00	Unused bytes.

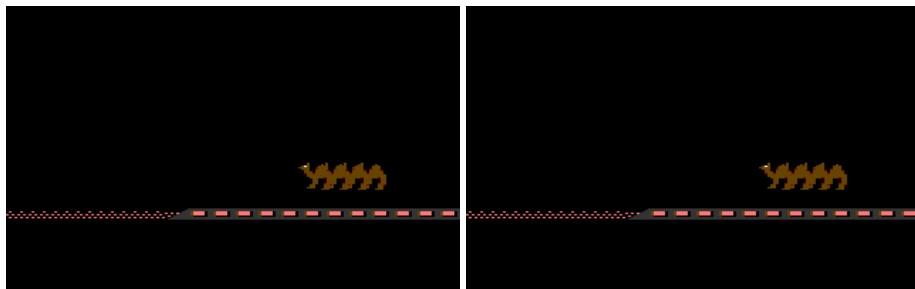
Tech Planet - Level 2 .

Tech Planet - Level 3 Data

Sample enemy movement for Tech planet, level 3.

Byte	Value	Description
Byte 1	\$06	Index into array for sprite color
Byte 3	FLYING_COCK_RIGHT1	Sprite value for the attack ship on the upper planet.
Byte 4	\$05	The animation frame rate for the attack ship.
Byte 6	FLYING_COCK_RIGHT1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$17	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$03	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	secondExplosionAnimation	Hi Ptr for Explosion animation.
Byte 32	lickerShipWaveData	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$02	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$02	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$18	Number of ships in wave.
Byte 40	\$00	Unused bytes.

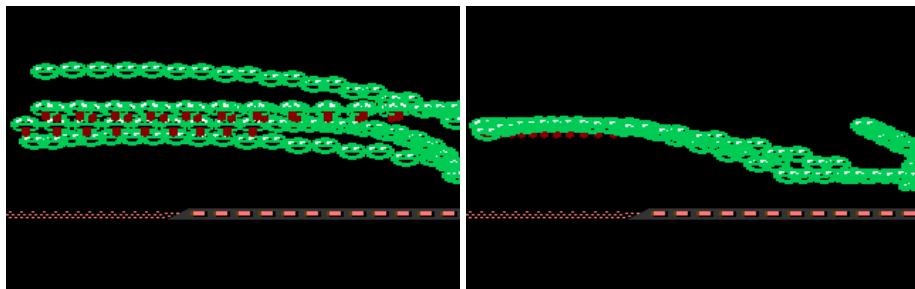
Tech Planet - Level 3 .

Tech Planet - Level 4 Data

Sample enemy movement for Tech planet, level 4.

Byte	Value	Description
Byte 1	\$05	Index into array for sprite color
Byte 3	TEARDROP_EXPLOSION1	Sprite value for the attack ship on the upper planet
Byte 4	\$01	The animation frame rate for the attack ship.
Byte 6	TEARDROP_EXPLOSION1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$FC	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$02	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	secondExplosionAnimation	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$06	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$03	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$18	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Tech Planet - Level 4 .

Tech Planet - Level 5 Data

Sample enemy movement for Tech planet, level 5.

Byte	Value	Description
Byte 1	\$05	Index into array for sprite color
Byte 3	LICKER_SHIP1	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	LICKERSHIP_INV1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$06	X Pos movement for attack ship.
Byte 20	\$24	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	planet2Level5Data2ndStage	Hi Ptr for another set of wave data.
Byte 30	planet2Level5Data3rdStage	Hi Ptr for Explosion animation.
Byte 32	planet2Level5Data2ndStage	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$02	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$01	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$30	Number of ships in wave.
Byte 40	\$00	Unused bytes.

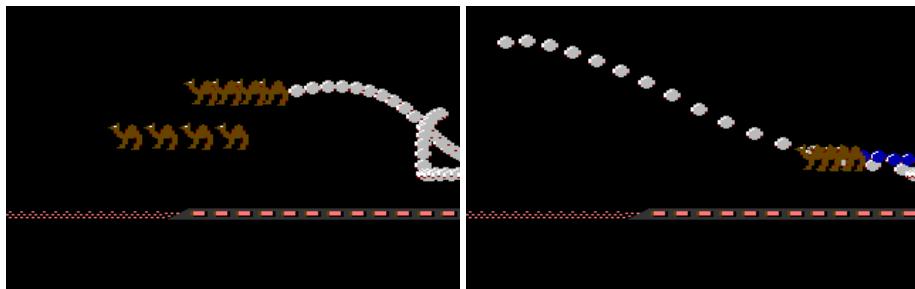
Tech Planet - Level 5 .

Tech Planet - Level 6 Data

Sample enemy movement for Tech planet, level 6.

Byte	Value	Description
Byte 1	\$04	Index into array for sprite color
Byte 3	SPINNING_RING1	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	SPINNING_RING1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$07	X Pos movement for attack ship.
Byte 20	\$24	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$01	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$00	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	planet2Level6Data2ndStage	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	secondExplosionAnimation	Hi Ptr for Explosion animation.
Byte 32	lickerShipWaveData	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$01	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$02	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

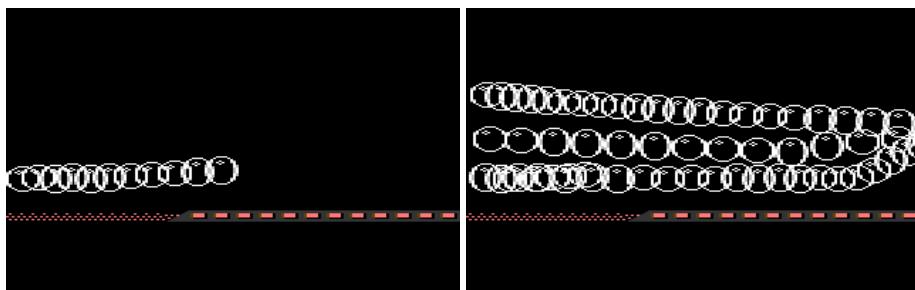
Tech Planet - Level 6 .

Tech Planet - Level 7 Data

Sample enemy movement for Tech planet, level 7.

Byte	Value	Description
Byte 1	\$0F	Index into array for sprite color
Byte 3	SMALLBALL AGAIN	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	SMALLBALL AGAIN	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$40	Update rate for attack wave
Byte 18	planet2Level7Data2ndStage	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$04	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$01	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	secondExplosionAnimation	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$03	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$02	Does colliding with this enemy decrease the gilby's energy?.
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?.
Byte 38	\$04	Number of waves in data.
Byte 39	\$10	Number of ships in wave.
Byte 40	\$00	Unused bytes.

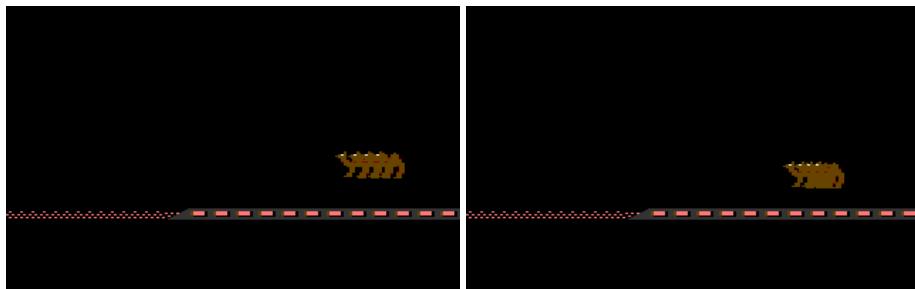
Tech Planet - Level 7 .

Tech Planet - Level 8 Data

Sample enemy movement for Tech planet, level 8.

Byte	Value	Description
Byte 1	\$0C	Index into array for sprite color
Byte 3	BUBBLE	Sprite value for the attack ship on the upper planet
Byte 4	\$04	The animation frame rate for the attack ship.
Byte 6	BUBBLE	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$60	Update rate for attack wave
Byte 18	planet2Level8Data2ndStage	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$00	X Pos Frame Rate for Attack ship.
Byte 22	\$01	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet2Level8Data2ndStage	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$20	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$10	Number of ships in wave.
Byte 40	\$00	Unused bytes.

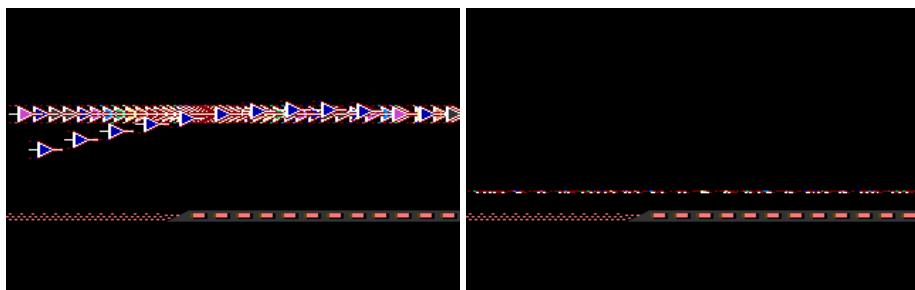
Tech Planet - Level 8 .

Tech Planet - Level 9 Data

Sample enemy movement for Tech planet, level 9.

Byte	Value	Description
Byte 1	\$04	Index into array for sprite color
Byte 3	LAND_GILBY1	Sprite value for the attack ship on the upper planet
Byte 4	\$03	The animation frame rate for the attack ship.
Byte 6	LAND_GILBY_LOWERPLANET1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$04	X Pos movement for attack ship.
Byte 20	\$24	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	planet2Level9Data	Hi Ptr for another set of wave data.
Byte 30	gilbyTakingOffAsExplosion	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$04	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$10	Number of ships in wave.
Byte 40	\$00	Unused bytes.

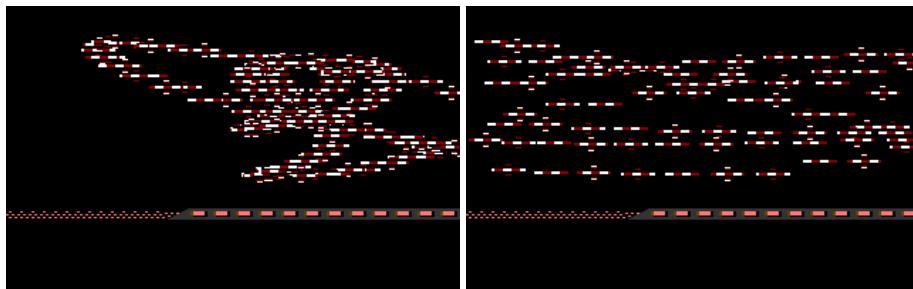
Tech Planet - Level 9 .

Tech Planet - Level 10 Data

Sample enemy movement for Tech planet, level 10.

Byte	Value	Description
Byte 1	\$11	Index into array for sprite color
Byte 3	FLYING_TRIANGLE1	Sprite value for the attack ship on the upper planet.
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	FLYING_TRIANGLE1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$06	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$00	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$00	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	flowchartArrowAsExplosion	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$06	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$18	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Tech Planet - Level 10 .

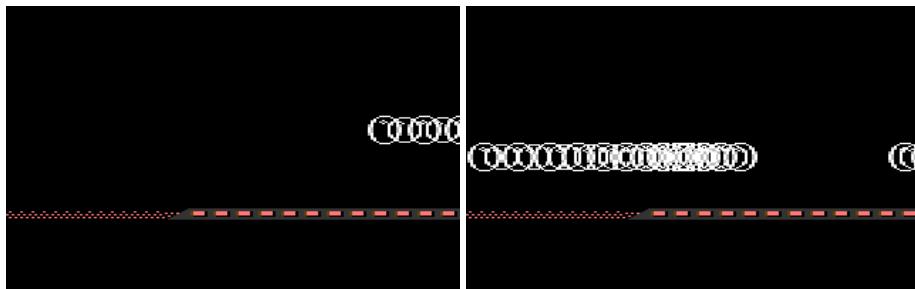
Tech Planet - Level 11 Data

Sample enemy movement for Tech planet, level 11.

Byte	Value	Description
Byte 1	\$00	Index into array for sprite color
Byte 3	FLYING_SAUCER1	Sprite value for the attack ship on the upper planet
Byte 4	\$01	The animation frame rate for the attack ship.
Byte 6	FLYING_SAUCER1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$10	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	nullPtr	Hi Ptr for Explosion animation.
Byte 32	planet2Level11Data2ndStage	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$00	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$10	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Tech Planet - Level 11 .

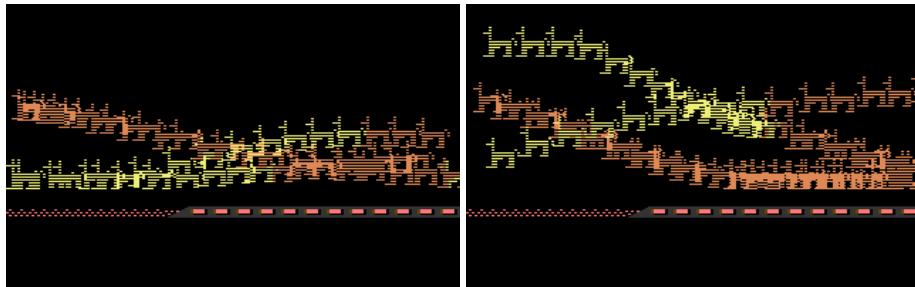
Tech Planet - Level 12 Data



Sample enemy movement for Tech planet, level 12.

Byte	Value	Description
Byte 1	\$0C	Index into array for sprite color
Byte 3	BUBBLE	Sprite value for the attack ship on the upper planet
Byte 4	\$01	The animation frame rate for the attack ship.
Byte 6	BUBBLE	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$03	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$00	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$00	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet2Level1Data	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	CAMEL	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$30	Number of ships in wave.
Byte 40	\$00	Unused bytes.

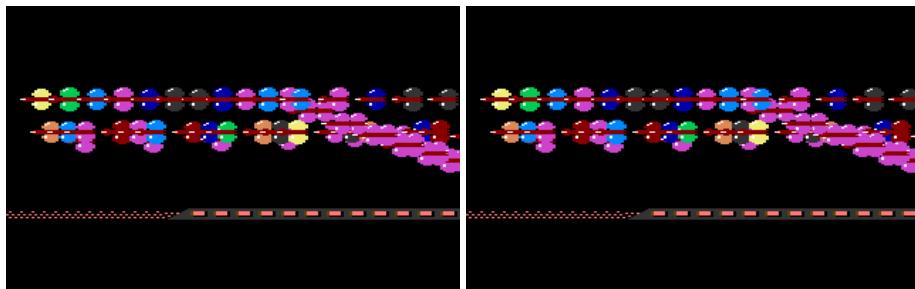
Tech Planet - Level 12 .

Tech Planet - Level 13 Data

Sample enemy movement for Tech planet, level 13.

Byte	Value	Description
Byte 1	\$08	Index into array for sprite color
Byte 3	LLAMA	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	LLAMA	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$02	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$01	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet2Level13Data2ndStage	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$02	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$01	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$40	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Tech Planet - Level 13 .

Tech Planet - Level 14 Data

Sample enemy movement for Tech planet, level 14.

Byte	Value	Description
Byte 1	\$04	Index into array for sprite color
Byte 3	FLYING_COCK1	Sprite value for the attack ship on the upper planet
Byte 4	\$05	The animation frame rate for the attack ship.
Byte 6	FLYING_COCK1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$E9	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet2Level14Data2ndStage	Hi Ptr for Explosion animation.
Byte 32	lickerShipWaveData	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$02	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$02	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$18	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Tech Planet - Level 14 .

Tech Planet - Level 15 Data

Byte	Value	Description
Byte 1	\$08	Index into array for sprite color
Byte 3	FLAG.BAR	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	FLAG.BAR	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$03	X Pos movement for attack ship.
Byte 20	\$22	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$01	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	planet2Level15Data	Hi Ptr for another set of wave data.
Byte 30	planet2Level15Data2ndStage	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$03	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$02	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

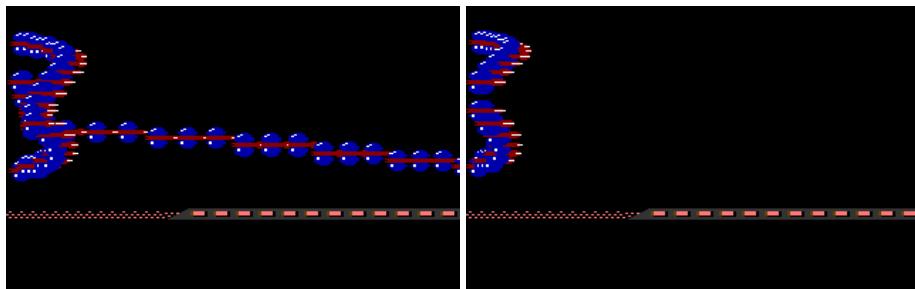
Tech Planet - Level 15 .

Tech Planet - Level 16 Data

Sample enemy movement for Tech planet, level 16.

Byte	Value	Description
Byte 1	\$10	Index into array for sprite color
Byte 3	WINGBALL	Sprite value for the attack ship on the upper planet
Byte 4	\$04	The animation frame rate for the attack ship.
Byte 6	MONEY_BAG	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$FC	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$00	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$00	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet1Level9Data	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$10	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$30	Number of ships in wave.
Byte 40	\$00	Unused bytes.

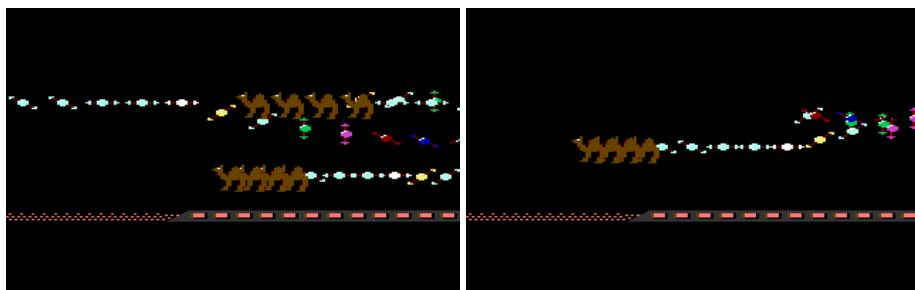
Tech Planet - Level 16 .

Tech Planet - Level 17 Data

Sample enemy movement for Tech planet, level 17.

Byte	Value	Description
Byte 1	\$06	Index into array for sprite color
Byte 3	FLYING_COCK_RIGHT1	Sprite value for the attack ship on the upper planet
Byte 4	\$05	The animation frame rate for the attack ship.
Byte 6	FLYING_COCK_RIGHT1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$17	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$03	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet2Level17Data2ndStage	Hi Ptr for Explosion animation.
Byte 32	lickerShipWaveData	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$02	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$02	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$18	Number of ships in wave.
Byte 40	\$00	Unused bytes.

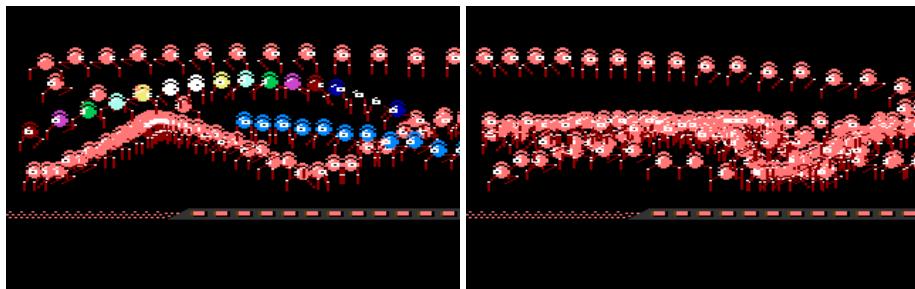
Tech Planet - Level 17 .

Tech Planet - Level 18 Data

Sample enemy movement for Tech planet, level 18.

Byte	Value	Description
Byte 1	\$10	Index into array for sprite color
Byte 3	BOLAS1	Sprite value for the attack ship on the upper planet
Byte 4	\$02	The animation frame rate for the attack ship.
Byte 6	BOLAS1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$08	Update rate for attack wave
Byte 18	planet2Level18Data2ndStage	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$01	Y Pos Frame Rate for Attack ship.
Byte 23	\$01	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	defaultExplosion	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$06	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

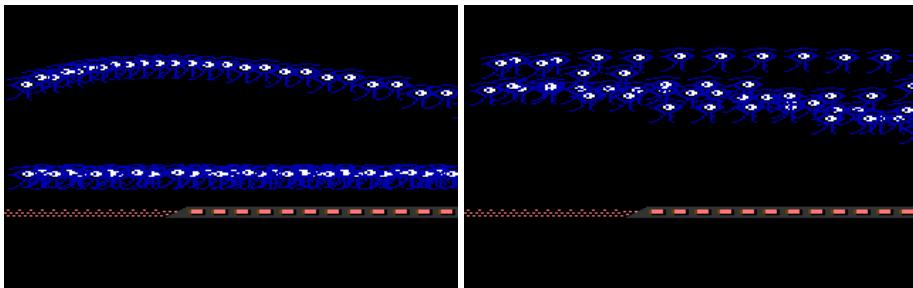
Tech Planet - Level 18 .

Tech Planet - Level 19 Data

Sample enemy movement for Tech planet, level 19.

Byte	Value	Description
Byte 1	\$0E	Index into array for sprite color
Byte 3	LAND_GILBY1	Sprite value for the attack ship on the upper planet
Byte 4	\$04	The animation frame rate for the attack ship.
Byte 6	LAND_GILBY_LOWERPLANET1	Sprite value for the attack ship on lower planet
Byte 7	\$0C	Whether a specific attack behaviour is used.
Byte 9	landGilbyAsEnemy	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$05	X Pos movement for attack ship.
Byte 20	\$24	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	planet2Level19Data	Hi Ptr for another set of wave data.
Byte 30	planet2Level19Data2ndStage	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$04	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$02	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$38	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Tech Planet - Level 19 .

Tech Planet - Level 20 Data

Sample enemy movement for Tech planet, level 20.

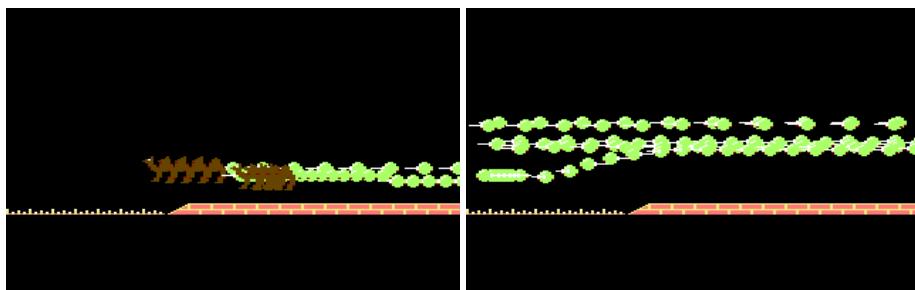
Byte	Value	Description
Byte 1	\$06	Index into array for sprite color
Byte 3	EYE_OF_HORUS	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	EYE_OF_HORUS	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$FC	X Pos movement for attack ship.
Byte 20	\$24	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	copticExplosion	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet2Level20Data	Hi Ptr for Explosion animation.
Byte 32	planet2Level20Data	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$01	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$01	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$40	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Tech Planet - Level 20 .

Brick Planet - Level 1 Data

Byte	Value	Description
Byte 1	\$10	Index into array for sprite color
Byte 3	\$FC	Sprite value for the attack ship on the upper planet
Byte 4	\$02	The animation frame rate for the attack ship.
Byte 6	\$FC	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$20	Update rate for attack wave
Byte 18	planet3Level1Data	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$02	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$01	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	\$50	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$01	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$01	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$53	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$41	Number of waves in data.
Byte 39	\$56	Number of ships in wave.
Byte 40	\$2A	Unused bytes.

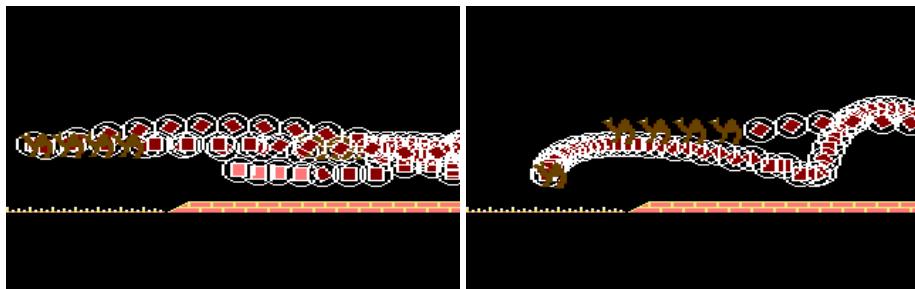
Brick Planet - Level 1 .

Brick Planet - Level 2 Data

Sample enemy movement for Brick planet, level 2.

Byte	Value	Description
Byte 1	\$0D	Index into array for sprite color
Byte 3	LITTLE.DART	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	LITTLE.DART	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$50	Update rate for attack wave
Byte 18	planet3Level2Data2ndStage	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$F8	X Pos movement for attack ship.
Byte 20	\$01	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$0C	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$00	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	secondExplosionAnimation	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$02	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$05	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$18	Number of ships in wave.
Byte 40	\$00	Unused bytes.

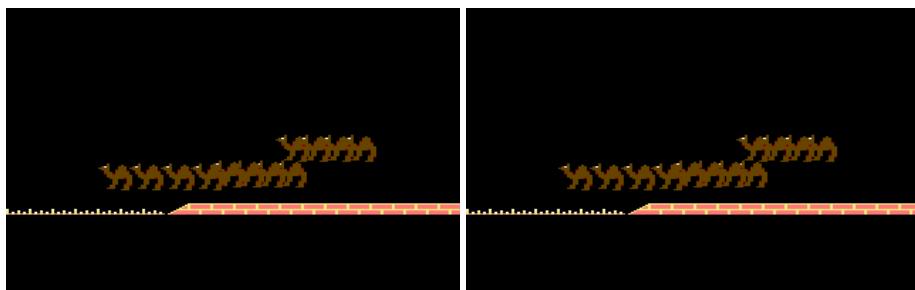
Brick Planet - Level 2 .

Brick Planet - Level 3 Data

Sample enemy movement for Brick planet, level 3.

Byte	Value	Description
Byte 1	\$02	Index into array for sprite color
Byte 3	BOUNCY_RING	Sprite value for the attack ship on the upper planet
Byte 4	\$04	The animation frame rate for the attack ship.
Byte 6	BOUNCY_RING	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$03	X Pos movement for attack ship.
Byte 20	\$23	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$01	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	planet3Level3Data2ndStage	Hi Ptr for another set of wave data.
Byte 30	secondExplosionAnimation	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$02	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$04	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$10	Number of ships in wave.
Byte 40	\$00	Unused bytes.

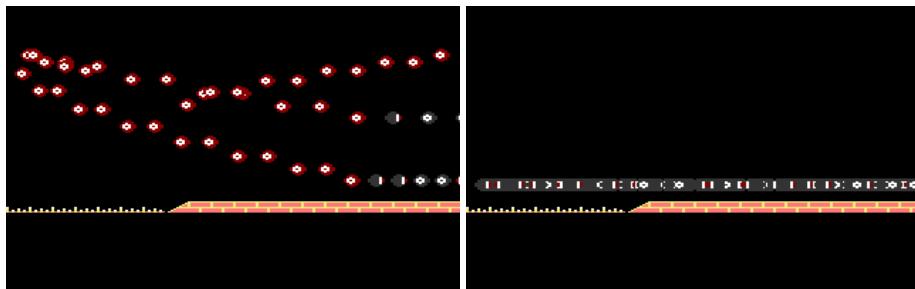
Brick Planet - Level 3 .

Brick Planet - Level 4 Data

Sample enemy movement for Brick planet, level 4.

Byte	Value	Description
Byte 1	\$06	Index into array for sprite color
Byte 3	GILBY_AIRBORNE_RIGHT	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	GILBY_AIRBORNE_RIGHT	Sprite value for the attack ship on lower planet
Byte 7	\$04	Whether a specific attack behaviour is used.
Byte 9	gilbyLookingLeft	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$08	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$03	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	secondExplosionAnimation	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$04	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$08	Does colliding with this enemy decrease the gilby's energy?.
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?.
Byte 38	\$04	Number of waves in data.
Byte 39	\$18	Number of ships in wave.
Byte 40	\$00	Unused bytes.

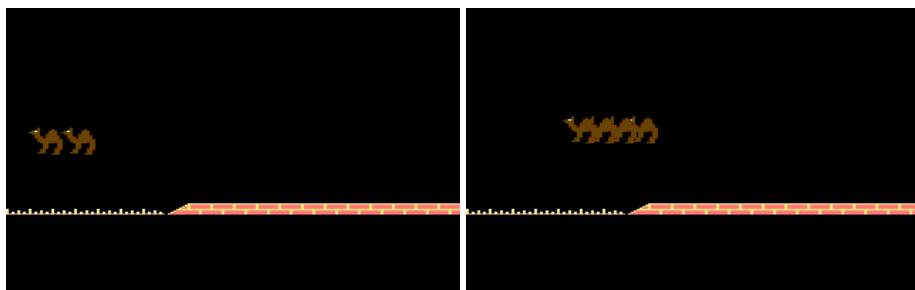
Brick Planet - Level 4 .

Brick Planet - Level 5 Data

Sample enemy movement for Brick planet, level 5.

Byte	Value	Description
Byte 1	\$0B	Index into array for sprite color
Byte 3	SMALL_BALL1	Sprite value for the attack ship on the upper planet
Byte 4	\$02	The animation frame rate for the attack ship.
Byte 6	SMALL_BALL1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$00	X Pos Frame Rate for Attack ship.
Byte 22	\$00	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$00	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	stickyGlobeExplosion	Hi Ptr for Explosion animation.
Byte 32	planet3Level5Data	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$02	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$03	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

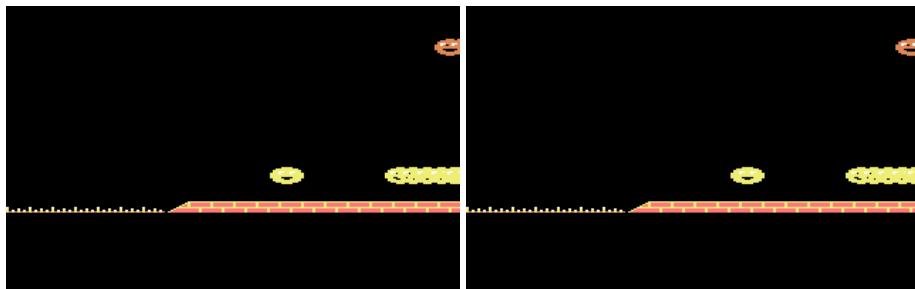
Brick Planet - Level 5 .

Brick Planet - Level 6 Data

Sample enemy movement for Brick planet, level 6.

Byte	Value	Description
Byte 1	\$00	Index into array for sprite color
Byte 3	LAND_GILBY1	Sprite value for the attack ship on the upper planet
Byte 4	\$01	The animation frame rate for the attack ship.
Byte 6	LAND_GILBY_LOWERPLANET1	Sprite value for the attack ship on lower planet
Byte 7	\$04	Whether a specific attack behaviour is used.
Byte 9	planet3Level6Additional	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$F9	X Pos movement for attack ship.
Byte 20	\$23	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$07	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	planet3Level6Data	Hi Ptr for another set of wave data.
Byte 30	planet2Level9Data	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$03	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$04	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$10	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Brick Planet - Level 6 .

Brick Planet - Level 7 Data

Sample enemy movement for Brick planet, level 7.

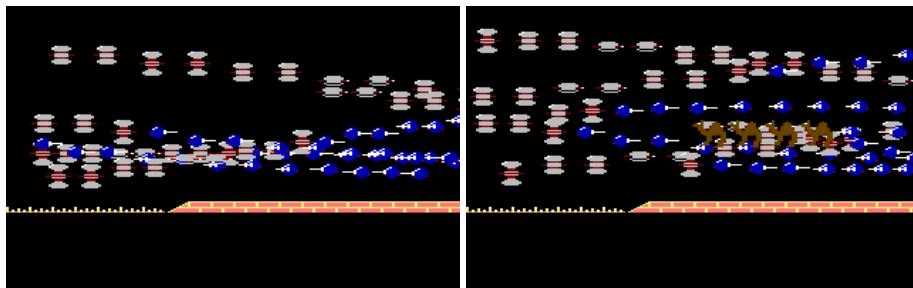
Byte	Value	Description
Byte 1	\$07	Index into array for sprite color
Byte 3	LICKER_SHIP1	Sprite value for the attack ship on the upper planet
Byte 4	\$07	The animation frame rate for the attack ship.
Byte 6	LICKERSHIP_INV1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$03	X Pos movement for attack ship.
Byte 20	\$23	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$00	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	planet3Level7Data2ndStage	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	spinningRings	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$01	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$02	Does colliding with this enemy decrease the gilby's energy?.
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?.
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Brick Planet - Level 7 .

Brick Planet - Level 8 Data

Byte	Value	Description
Byte 1	\$0C	Index into array for sprite color
Byte 3	BUBBLE	Sprite value for the attack ship on the upper planet
Byte 4	\$03	The animation frame rate for the attack ship.
Byte 6	BUBBLE	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$00	X Pos Frame Rate for Attack ship.
Byte 22	\$00	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$00	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	bubbleExplosion	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$0C	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$10	Number of ships in wave.
Byte 40	\$00	Unused bytes.

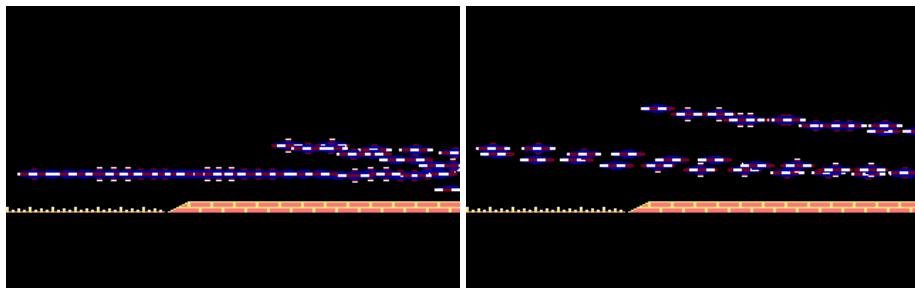
Brick Planet - Level 8 .

Brick Planet - Level 9 Data

Sample enemy movement for Brick planet, level 9.

Byte	Value	Description
Byte 1	\$06	Index into array for sprite color
Byte 3	FLYING_DART1	Sprite value for the attack ship on the upper planet
Byte 4	\$05	The animation frame rate for the attack ship.
Byte 6	FLYING_DART1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$0C	Update rate for attack wave
Byte 18	planet3Level9Data2ndStage	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$01	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	secondExplosionAnimation	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$03	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$05	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$18	Number of ships in wave.
Byte 40	\$00	Unused bytes.

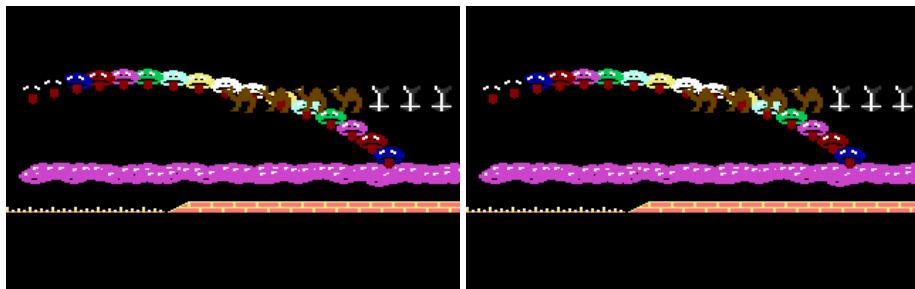
Brick Planet - Level 9 .

Brick Planet - Level 10 Data

Sample enemy movement for Brick planet, level 10.

Byte	Value	Description
Byte 1	\$06	Index into array for sprite color
Byte 3	FLYING_SAUCER1	Sprite value for the attack ship on the upper planet
Byte 4	\$03	The animation frame rate for the attack ship.
Byte 6	FLYING_SAUCER1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$0A	Update rate for attack wave
Byte 18	planet3Level10Data2ndStage	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$01	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	spinningRings	Hi Ptr for Explosion animation.
Byte 32	planet3Level10Data	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$03	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$03	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

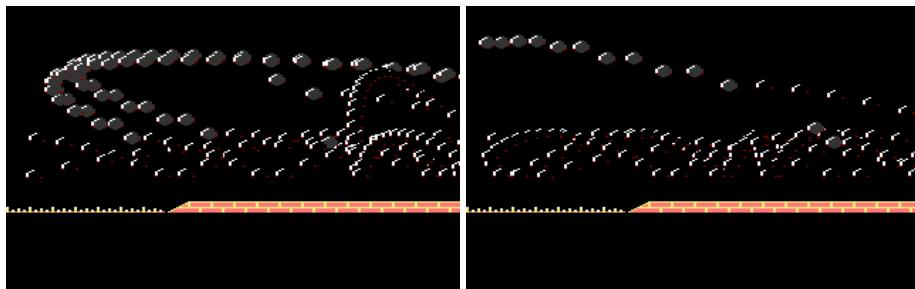
Brick Planet - Level 10 .

Brick Planet - Level 11 Data

Sample enemy movement for Brick planet, level 11.

Byte	Value	Description
Byte 1	\$04	Index into array for sprite color
Byte 3	LICKER_SHIP1	Sprite value for the attack ship on the upper planet
Byte 4	\$05	The animation frame rate for the attack ship.
Byte 6	LICKERSHIP_INV1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$FD	X Pos movement for attack ship.
Byte 20	\$21	Y Pos movement pattern for attack ship.
Byte 21	\$02	X Pos Frame Rate for Attack ship.
Byte 22	\$01	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	planet3Level11Data	Hi Ptr for another set of wave data.
Byte 30	planet3Level11Data2ndStage	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$08	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

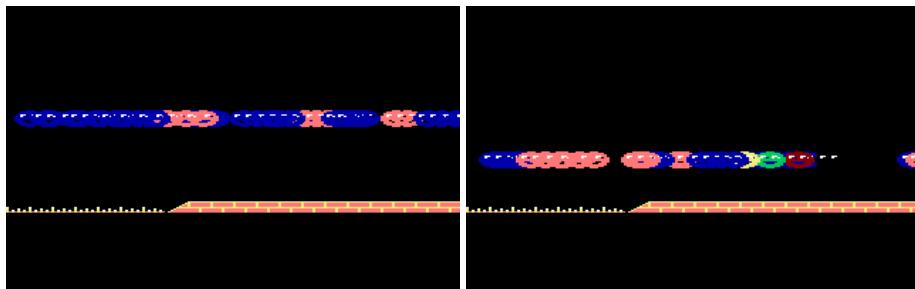
Brick Planet - Level 11 .

Brick Planet - Level 12 Data

Sample enemy movement for Brick planet, level 12.

Byte	Value	Description
Byte 1	\$00	Index into array for sprite color
Byte 3	SMALLBALL AGAIN	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	SMALLBALL AGAIN	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$23	Y Pos movement pattern for attack ship.
Byte 21	\$00	X Pos Frame Rate for Attack ship.
Byte 22	\$01	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	planet3Level12Data	Hi Ptr for another set of wave data.
Byte 30	planet3Level12Data2ndStage	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$01	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$02	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$28	Number of ships in wave.
Byte 40	\$00	Unused bytes.

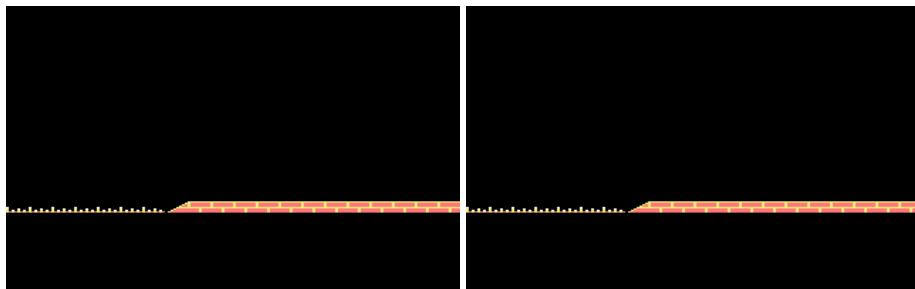
Brick Planet - Level 12 .

Brick Planet - Level 13 Data

Sample enemy movement for Brick planet, level 13.

Byte	Value	Description
Byte 1	\$06	Index into array for sprite color
Byte 3	LICKER.SHIP1	Sprite value for the attack ship on the upper planet
Byte 4	\$05	The animation frame rate for the attack ship.
Byte 6	LICKERSHIP_INV1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$00	X Pos Frame Rate for Attack ship.
Byte 22	\$00	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$00	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	lickerShipAsExplosion	Hi Ptr for Explosion animation.
Byte 32	defaultExpllosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?
Byte 36	\$05	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$18	Number of ships in wave.
Byte 40	\$00	Unused bytes.

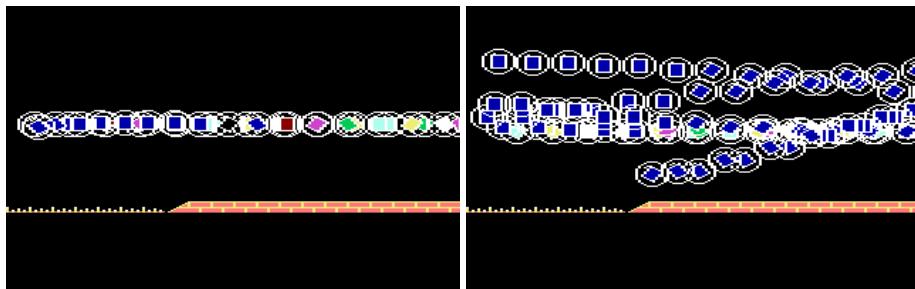
Brick Planet - Level 13 .

Brick Planet - Level 14 Data

Sample enemy movement for Brick planet, level 14.

Byte	Value	Description
Byte 1	\$08	Index into array for sprite color
Byte 3	CAMEL	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	CAMEL	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$F0	Update rate for attack wave
Byte 18	planet1Level12Data	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$00	X Pos Frame Rate for Attack ship.
Byte 22	\$00	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$00	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet3Level14Data2ndStage	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$08	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

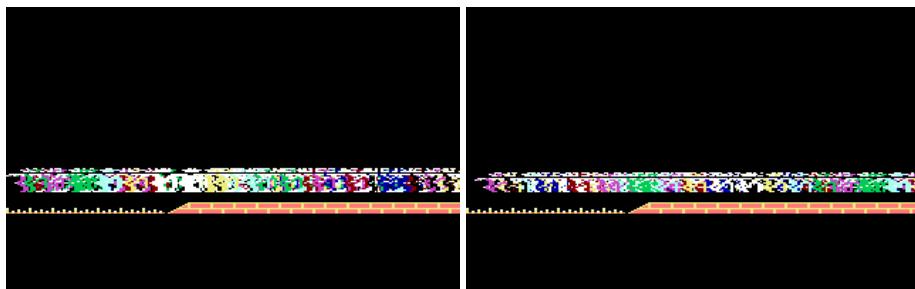
Brick Planet - Level 14 .

Brick Planet - Level 15 Data

Sample enemy movement for Brick planet, level 15.

Byte	Value	Description
Byte 1	\$10	Index into array for sprite color
Byte 3	BOUNCY_RING	Sprite value for the attack ship on the upper planet
Byte 4	\$01	The animation frame rate for the attack ship.
Byte 6	BOUNCY_RING	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$00	X Pos Frame Rate for Attack ship.
Byte 22	\$00	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$00	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet3Level15Data2ndStage	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$03	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$02	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$28	Number of ships in wave.
Byte 40	\$00	Unused bytes.

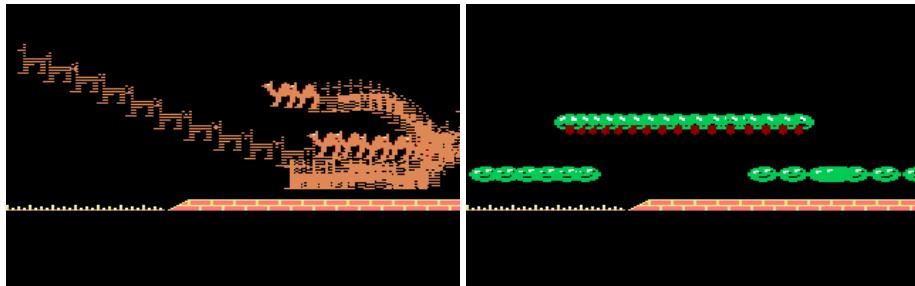
Brick Planet - Level 15 .

Brick Planet - Level 16 Data

Sample enemy movement for Brick planet, level 16.

Byte	Value	Description
Byte 1	\$10	Index into array for sprite color
Byte 3	STRANGE_SYMBOL	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	STRANGE_SYMBOL	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$C0	Update rate for attack wave
Byte 18	planet2Level5Data	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$00	X Pos Frame Rate for Attack ship.
Byte 22	\$00	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$00	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet3Level16Data	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$10	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$10	Number of ships in wave.
Byte 40	\$00	Unused bytes.

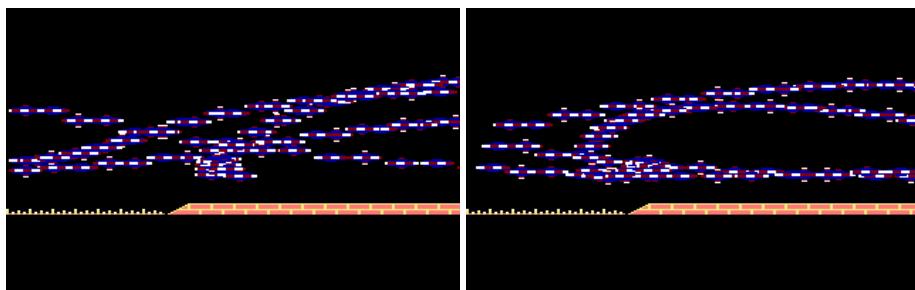
Brick Planet - Level 16 .

Brick Planet - Level 17 Data

Sample enemy movement for Brick planet, level 17.

Byte	Value	Description
Byte 1	\$08	Index into array for sprite color
Byte 3	LLAMA	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	LLAMA	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$03	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$01	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet5Level14Data	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$0C	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$30	Number of ships in wave.
Byte 40	\$00	Unused bytes.

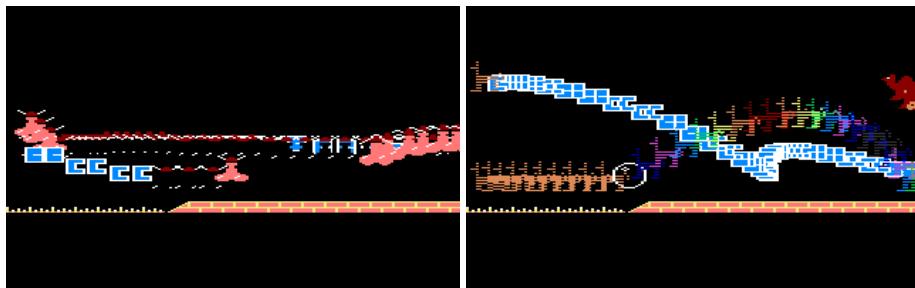
Brick Planet - Level 17 .

Brick Planet - Level 18 Data

Sample enemy movement for Brick planet, level 18.

Byte	Value	Description
Byte 1	\$06	Index into array for sprite color
Byte 3	FLYING_SAUCER1	Sprite value for the attack ship on the upper planet
Byte 4	\$01	The animation frame rate for the attack ship.
Byte 6	FLYING_SAUCER1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$02	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$01	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet3Level18Data2ndStage	Hi Ptr for Explosion animation.
Byte 32	planet3Level18Data2ndStage	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$04	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$02	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

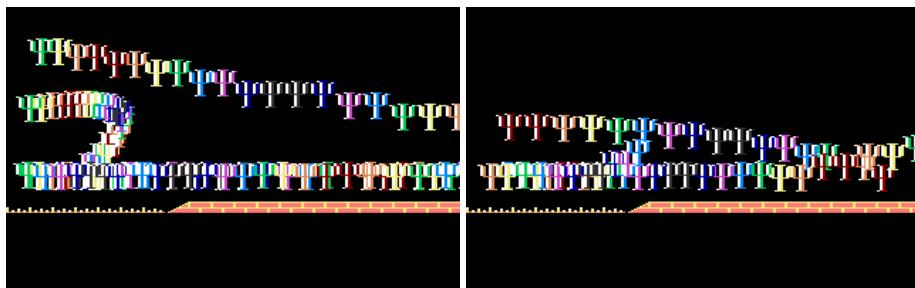
Brick Planet - Level 18 .

Brick Planet - Level 19 Data

Sample enemy movement for Brick planet, level 19.

Byte	Value	Description
Byte 1	\$0E	Index into array for sprite color
Byte 3	FLYING_COMMAS1	Sprite value for the attack ship on the upper planet
Byte 4	\$04	The animation frame rate for the attack ship.
Byte 6	FLYING_COMMAS1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$40	Update rate for attack wave
Byte 18	planet3Level19Data2ndStage	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$05	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet4Level17Data	Hi Ptr for Explosion animation.
Byte 32	planet4Level17Data	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$04	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

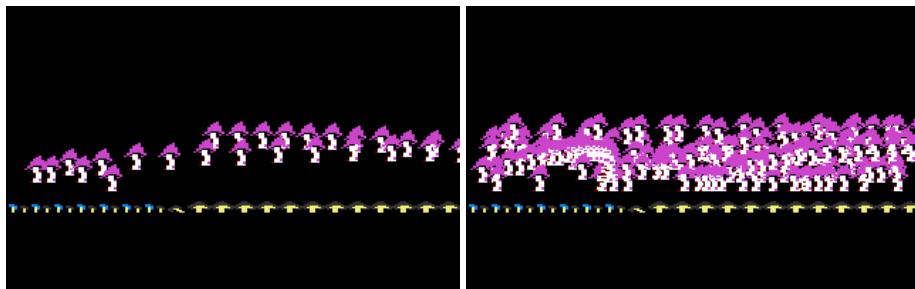
Brick Planet - Level 19 .

Brick Planet - Level 20 Data

Sample enemy movement for Brick planet, level 20.

Byte	Value	Description
Byte 1	\$11	Index into array for sprite color
Byte 3	PSI	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	PSI	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$06	X Pos movement for attack ship.
Byte 20	\$24	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	copticExplosion	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet3Level20Data	Hi Ptr for Explosion animation.
Byte 32	planet3Level20Data	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$01	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$01	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$40	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Brick Planet - Level 20 .

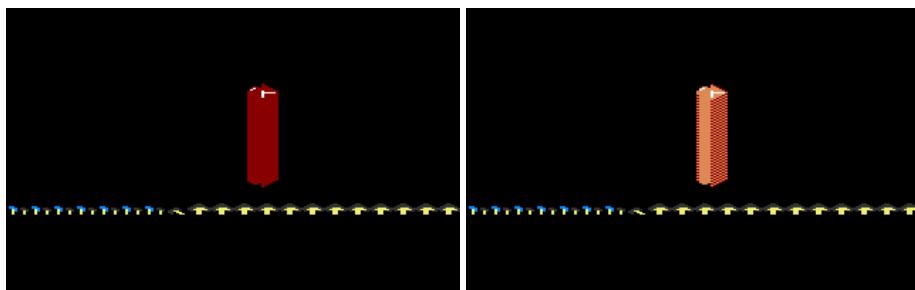
Mushroom Planet - Level 1 Data

Sample enemy movement for Mushroom planet, level 1.

Byte	Value	Description
Byte 1	\$04	Index into array for sprite color
Byte 3	MAGIC_MUSHROOM	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	INV_MAGIC_MUSHROOM	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$04	X Pos movement for attack ship.
Byte 20	\$23	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	planet4Level1Data2ndStage	Hi Ptr for another set of wave data.
Byte 30	spinningRings	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$02	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$02	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Mushroom Planet - Level 1 .

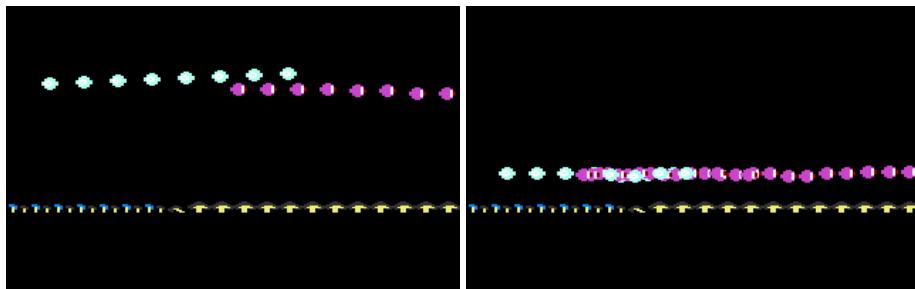
Mushroom Planet - Level 2 Data



Sample enemy movement for Mushroom planet, level 2.

Byte	Value	Description
Byte 1	\$0E	Index into array for sprite color
Byte 3	GILBY_AIRBORNE_RIGHT	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	GILBY_AIRBORNE_RIGHT	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$0C	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet4Leve2Data2ndStage	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$04	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$01	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$10	Number of ships in wave.
Byte 40	\$00	Unused bytes.

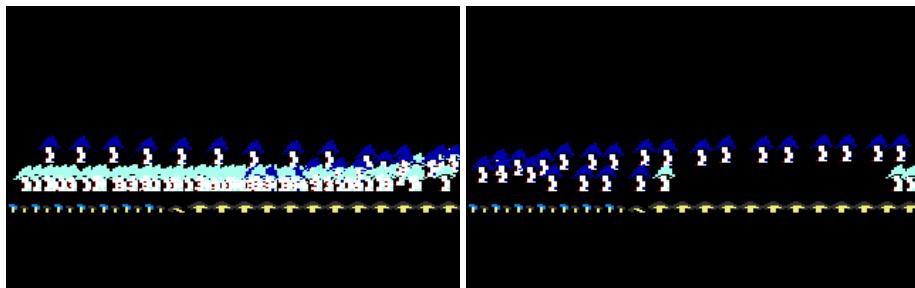
Mushroom Planet - Level 2 .

Mushroom Planet - Level 3 Data

Sample enemy movement for Mushroom planet, level 3.

Byte	Value	Description
Byte 1	\$02	Index into array for sprite color
Byte 3	LITTLE.DART	Sprite value for the attack ship on the upper planet
Byte 4	\$03	The animation frame rate for the attack ship.
Byte 6	LITTLE.DART	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$60	Update rate for attack wave
Byte 18	planet4Level2Data2ndStage	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$03	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$03	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$02	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet1Level5Data3rdStage	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$03	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

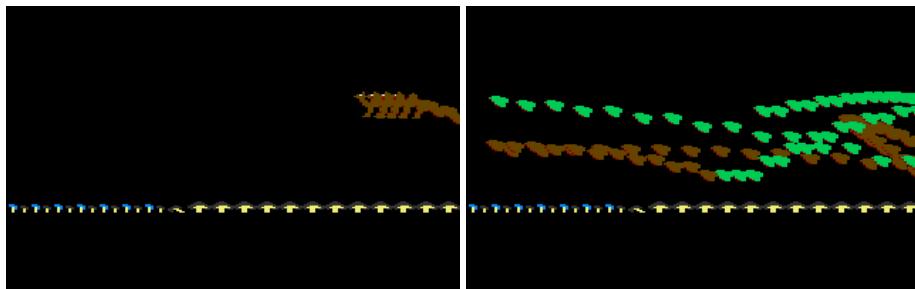
Mushroom Planet - Level 3 .

Mushroom Planet - Level 4 Data

Sample enemy movement for Mushroom planet, level 4.

Byte	Value	Description
Byte 1	\$03	Index into array for sprite color
Byte 3	MAGIC_MUSHROOM	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	INV_MAGIC_MUSHROOM	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$40	Update rate for attack wave
Byte 18	planet4Level4Data2ndStage	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$00	X Pos Frame Rate for Attack ship.
Byte 22	\$01	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	spinningRings	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$02	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$04	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$18	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Mushroom Planet - Level 4 .

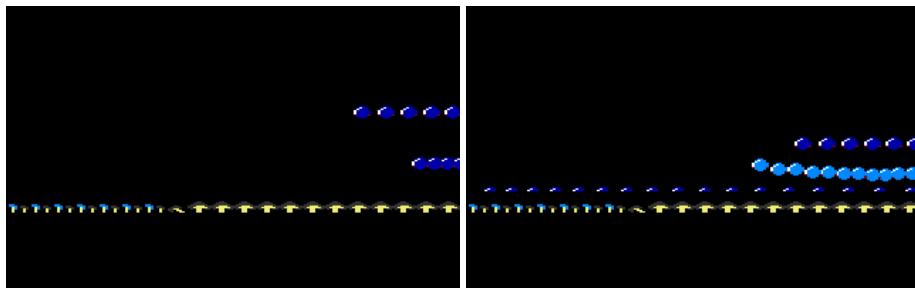
Mushroom Planet - Level 5 Data

Sample enemy movement for Mushroom planet, level 5.

Byte	Value	Description
Byte 1	\$09	Index into array for sprite color
Byte 3	LOZENGE	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	LOZENGE	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$04	X Pos movement for attack ship.
Byte 20	\$23	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	planet4Level5Data2ndStage	Hi Ptr for another set of wave data.
Byte 30	secondExplosionAnimation	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$02	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$04	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$18	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Mushroom Planet - Level 5 .

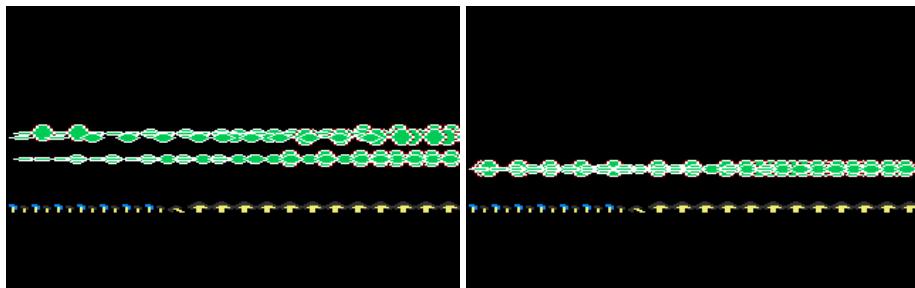
Mushroom Planet - Level 6 Data



Sample enemy movement for Mushroom planet, level 6.

Byte	Value	Description
Byte 1	\$06	Index into array for sprite color
Byte 3	SMALLBALL AGAIN	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	SMALLBALL AGAIN	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$20	Update rate for attack wave
Byte 18	planet4Level6Data2ndStage	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$00	X Pos Frame Rate for Attack ship.
Byte 22	\$00	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$00	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	spinningRings	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$01	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$04	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

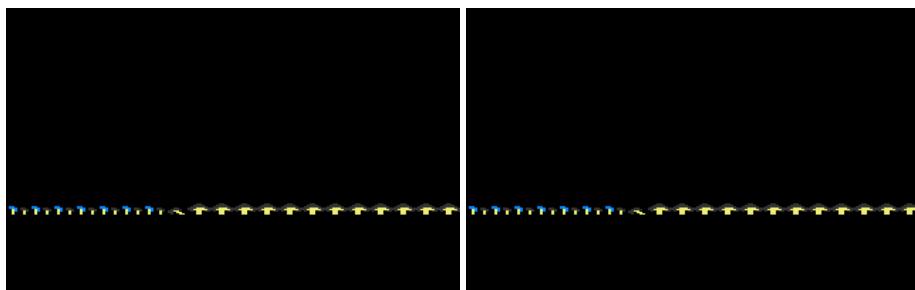
Mushroom Planet - Level 6 .

Mushroom Planet - Level 7 Data

Sample enemy movement for Mushroom planet, level 7.

Byte	Value	Description
Byte 1	\$05	Index into array for sprite color
Byte 3	TEARDROP_EXPLOSION1	Sprite value for the attack ship on the upper planet
Byte 4	\$06	The animation frame rate for the attack ship.
Byte 6	TEARDROP_EXPLOSION1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$E0	Update rate for attack wave
Byte 18	planet1Level14Data	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$00	X Pos Frame Rate for Attack ship.
Byte 22	\$00	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$00	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	defaultExplosion	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$08	Does colliding with this enemy decrease the gilby's energy?.
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$08	Number of ships in wave.
Byte 40	\$00	Unused bytes.

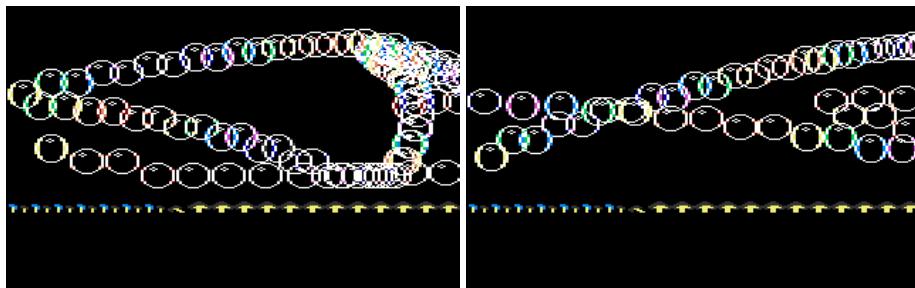
Mushroom Planet - Level 7 .

Mushroom Planet - Level 8 Data

Sample enemy movement for Mushroom planet, level 8.

Byte	Value	Description
Byte 1	\$00	Index into array for sprite color
Byte 3	LLAMA	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	LLAMA	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$00	X Pos Frame Rate for Attack ship.
Byte 22	\$00	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$00	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet4Level8Data	Hi Ptr for Explosion animation.
Byte 32	planet4Level8Data2ndStage	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$00	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Mushroom Planet - Level 8 .

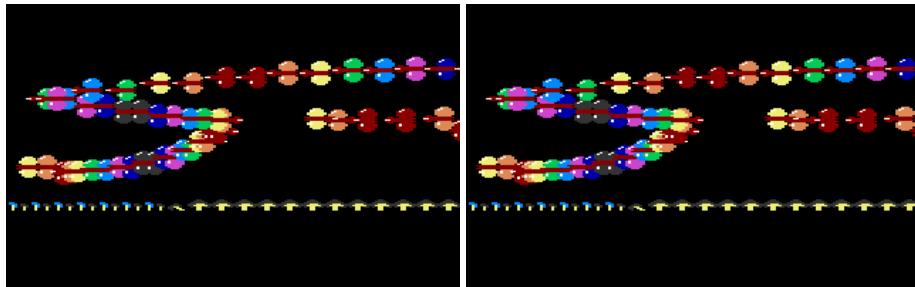
Mushroom Planet - Level 9 Data

Sample enemy movement for Mushroom planet, level 9.

Byte	Value	Description
Byte 1	\$11	Index into array for sprite color
Byte 3	BUBBLE	Sprite value for the attack ship on the upper planet
Byte 4	\$04	The animation frame rate for the attack ship.
Byte 6	BUBBLE	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$80	X Pos movement for attack ship.
Byte 20	\$25	Y Pos movement pattern for attack ship.
Byte 21	\$80	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	planet4Level9Data2ndStage	Hi Ptr for another set of wave data.
Byte 30	spinningRings	Hi Ptr for Explosion animation.
Byte 32	lickerShipWaveData	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$04	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$01	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Mushroom Planet - Level 9 .

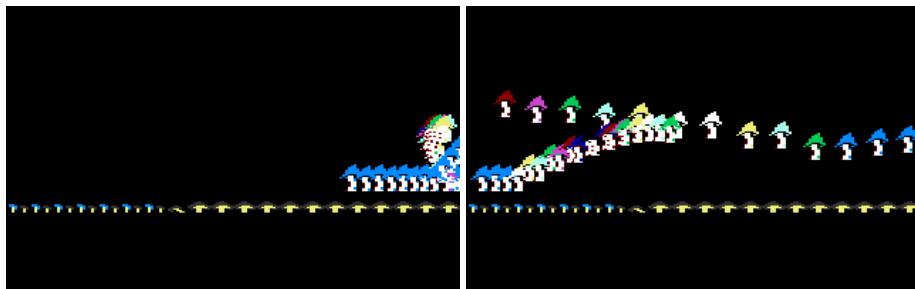
Mushroom Planet - Level 10 Data



Sample enemy movement for Mushroom planet, level 10.

Byte	Value	Description
Byte 1	\$11	Index into array for sprite color
Byte 3	FLYING_COCK_RIGHT1	Sprite value for the attack ship on the upper planet
Byte 4	\$05	The animation frame rate for the attack ship.
Byte 6	FLYING_COCK_RIGHT1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$10	Update rate for attack wave
Byte 18	planet4Level10Data2ndStage	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$0A	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	spinningRings	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$08	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Mushroom Planet - Level 10 .

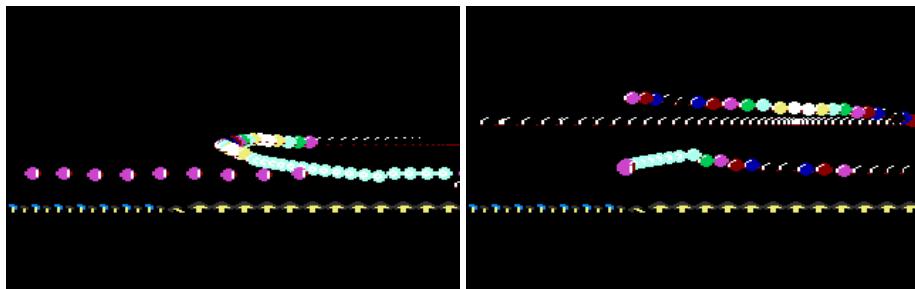
Mushroom Planet - Level 11 Data

Sample enemy movement for Mushroom planet, level 11.

Byte	Value	Description
Byte 1	\$0E	Index into array for sprite color
Byte 3	MAGIC_MUSHROOM	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	INV_MAGIC_MUSHROOM	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$E0	Update rate for attack wave
Byte 18	planet4Level11Data2ndStage	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$00	X Pos Frame Rate for Attack ship.
Byte 22	\$01	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet4Level11Data2ndStage	Hi Ptr for Explosion animation.
Byte 32	planet4Level11Data2ndStage	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$02	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$02	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Mushroom Planet - Level 11 .

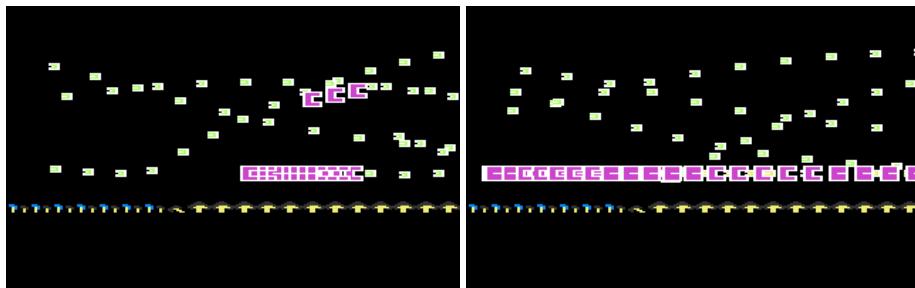
Mushroom Planet - Level 12 Data



Sample enemy movement for Mushroom planet, level 12.

Byte	Value	Description
Byte 1	\$00	Index into array for sprite color
Byte 3	SMALLBALL AGAIN	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	SMALLBALL AGAIN	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$00	X Pos Frame Rate for Attack ship.
Byte 22	\$00	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$00	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet4Level12Data2ndStage	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$04	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$30	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Mushroom Planet - Level 12 .

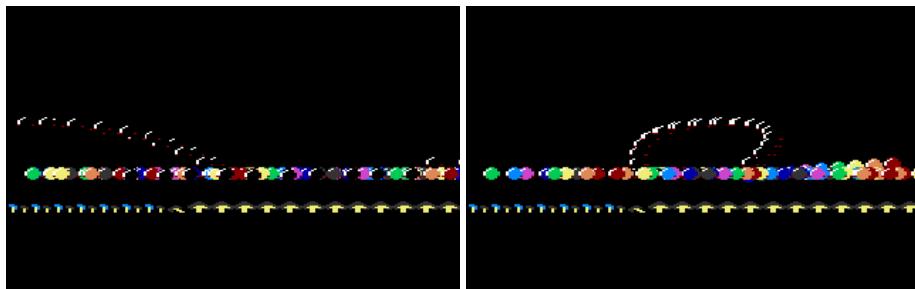
Mushroom Planet - Level 13 Data

Sample enemy movement for Mushroom planet, level 13.

Byte	Value	Description
Byte 1	\$0D	Index into array for sprite color
Byte 3	FLYING_DOT1	Sprite value for the attack ship on the upper planet
Byte 4	\$03	The animation frame rate for the attack ship.
Byte 6	FLYING_DOT1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$F9	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$01	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet5Level5Data	Hi Ptr for Explosion animation.
Byte 32	planet5Level5Data	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$0C	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$40	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Mushroom Planet - Level 13 .

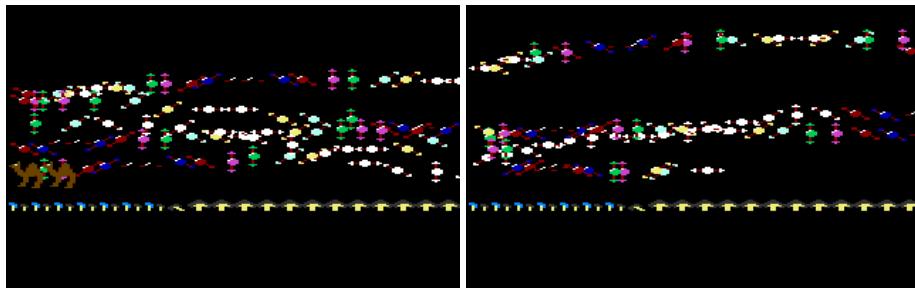
Mushroom Planet - Level 14 Data



Sample enemy movement for Mushroom planet, level 14.

Byte	Value	Description
Byte 1	\$11	Index into array for sprite color
Byte 3	SMALLBALL AGAIN	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	SMALLBALL AGAIN	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$80	X Pos movement for attack ship.
Byte 20	\$80	Y Pos movement pattern for attack ship.
Byte 21	\$80	X Pos Frame Rate for Attack ship.
Byte 22	\$80	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$00	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet4Level14Data2ndStage	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$10	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$18	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Mushroom Planet - Level 14 .

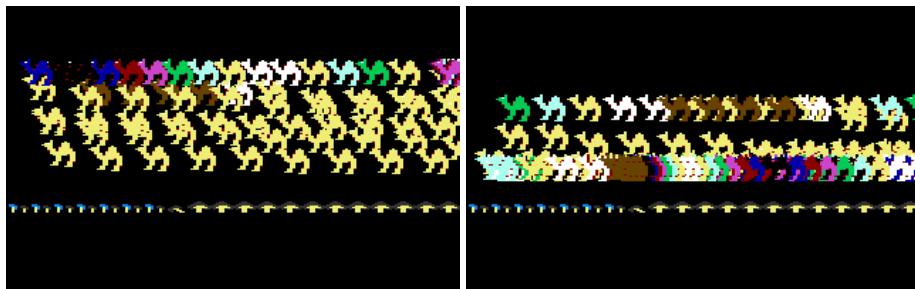
Mushroom Planet - Level 15 Data

Sample enemy movement for Mushroom planet, level 15.

Byte	Value	Description
Byte 1	\$10	Index into array for sprite color
Byte 3	BOLAS1	Sprite value for the attack ship on the upper planet
Byte 4	\$02	The animation frame rate for the attack ship.
Byte 6	BOLAS1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$02	X Pos Frame Rate for Attack ship.
Byte 22	\$03	Y Pos Frame Rate for Attack ship.
Byte 23	\$01	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	spinnerAsExplosion	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$03	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$03	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$28	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Mushroom Planet - Level 15 .

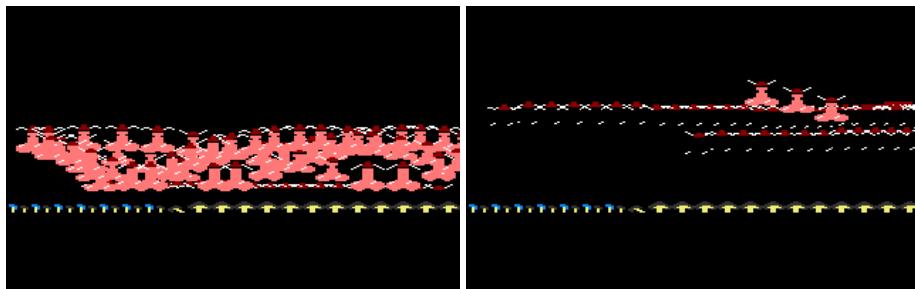
Mushroom Planet - Level 16 Data



Sample enemy movement for Mushroom planet, level 16.

Byte	Value	Description
Byte 1	\$07	Index into array for sprite color
Byte 3	CAMEL	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	CAMEL	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$F8	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$04	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet4Level16Data2ndStage	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$0C	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$30	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Mushroom Planet - Level 16 .

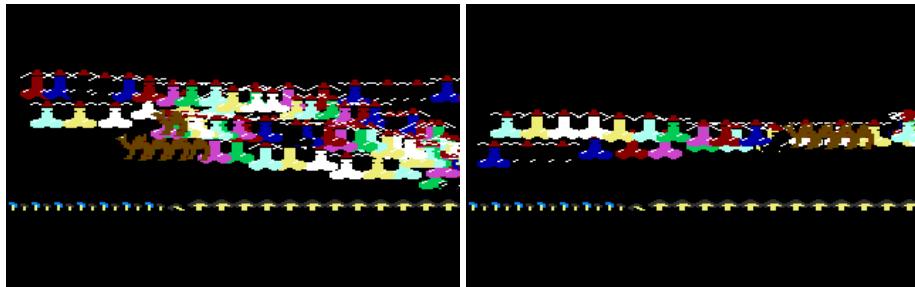
Mushroom Planet - Level 17 Data

Sample enemy movement for Mushroom planet, level 17.

Byte	Value	Description
Byte 1	\$00	Index into array for sprite color
Byte 3	CUMMING_COCK1	Sprite value for the attack ship on the upper planet
Byte 4	\$06	The animation frame rate for the attack ship.
Byte 6	BOLAS1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$04	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$00	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$00	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	cummingCock	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$0C	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Mushroom Planet - Level 17 .

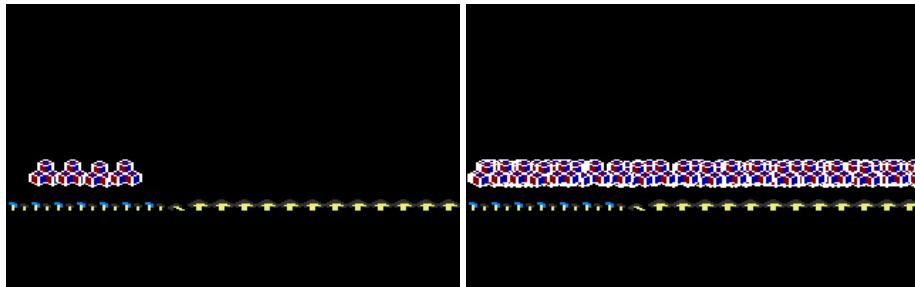
Mushroom Planet - Level 18 Data



Sample enemy movement for Mushroom planet, level 18.

Byte	Value	Description
Byte 1	\$10	Index into array for sprite color
Byte 3	CUMMING_COCK1	Sprite value for the attack ship on the upper planet
Byte 4	\$05	The animation frame rate for the attack ship.
Byte 6	LICKERSHIP_INV1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$24	Y Pos movement pattern for attack ship.
Byte 21	\$02	X Pos Frame Rate for Attack ship.
Byte 22	\$03	Y Pos Frame Rate for Attack ship.
Byte 23	\$01	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	planet4Level18Data	Hi Ptr for another set of wave data.
Byte 30	secondExplosionAnimation	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$03	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$02	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Mushroom Planet - Level 18 .

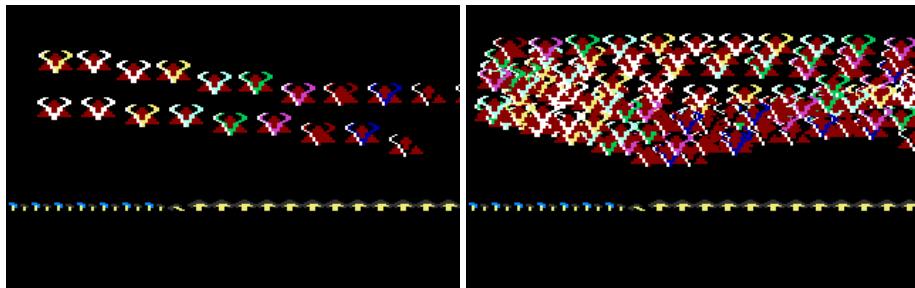
Mushroom Planet - Level 19 Data

Sample enemy movement for Mushroom planet, level 19.

Byte	Value	Description
Byte 1	\$06	Index into array for sprite color
Byte 3	QBERT_SQUARES	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	QBERT	Sprite value for the attack ship on lower planet
Byte 7	\$01	Whether a specific attack behaviour is used.
Byte 9	planet4Level19Additional	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$01	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$01	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	planet4Level19Data	Hi Ptr for another set of wave data.
Byte 30	spinningRings	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$02	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$08	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$0C	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Mushroom Planet - Level 19 .

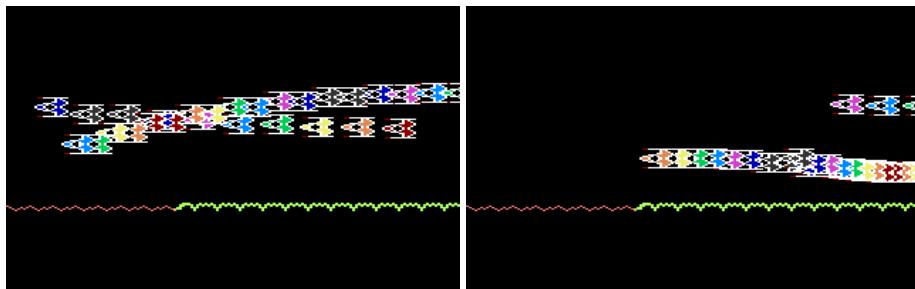
Mushroom Planet - Level 20 Data



Sample enemy movement for Mushroom planet, level 20.

Byte	Value	Description
Byte 1	\$10	Index into array for sprite color
Byte 3	BULLHEAD	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	BULLHEAD	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$FA	X Pos movement for attack ship.
Byte 20	\$24	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	copticExplosion	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet4Level20Data	Hi Ptr for Explosion animation.
Byte 32	planet4Level20Data	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$01	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$01	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$05	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$05	Number of waves in data.
Byte 39	\$05	Number of ships in wave.
Byte 40	\$05	Unused bytes.

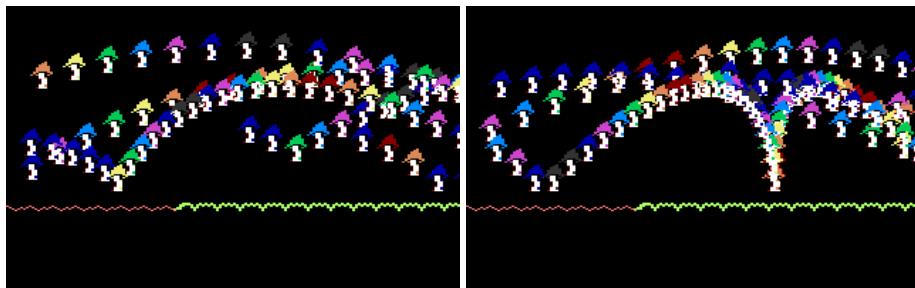
Mushroom Planet - Level 20 .

Om Planet - Level 1 Data

Sample enemy movement for Om planet, level 1.

Byte	Value	Description
Byte 1	\$11	Index into array for sprite color
Byte 3	STARSHIP	Sprite value for the attack ship on the upper planet.
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	STARSHIP	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$60	Update rate for attack wave
Byte 18	planet5Level1Data2ndStage	Hi Ptr to the wave data we switch to when first hit.
Byte 19	SFC	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	spinningRings	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$01	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$01	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$18	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Om Planet - Level 1 .

Om Planet - Level 2 Data

Sample enemy movement for Om planet, level 2.

Byte	Value	Description
Byte 1	\$11	Index into array for sprite color
Byte 3	MAGIC_MUSHROOM	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	INV_MAGIC_MUSHROOM	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$25	Y Pos movement pattern for attack ship.
Byte 21	\$00	X Pos Frame Rate for Attack ship.
Byte 22	\$01	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	planet5Level2Data	Hi Ptr for another set of wave data.
Byte 30	planet5Level2Explosion	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$08	Does colliding with this enemy decrease the gilby's energy?.
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$18	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Om Planet - Level 2 .

Om Planet - Level 3 Data

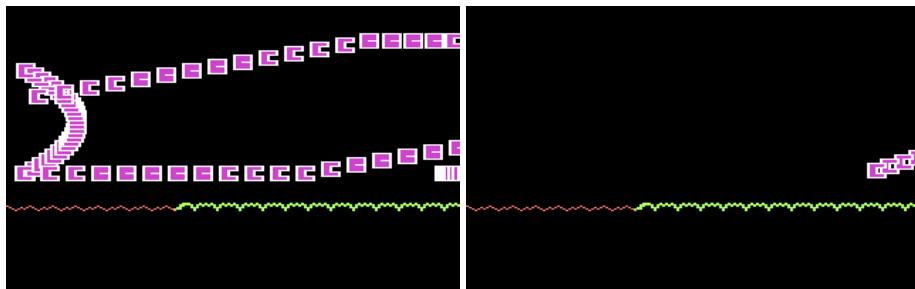
Byte	Value	Description
Byte 1	\$02	Index into array for sprite color
Byte 3	LAND_GILBY1	Sprite value for the attack ship on the upper planet
Byte 4	\$04	The animation frame rate for the attack ship.
Byte 6	LAND_GILBY_LOWERPLANET1	Sprite value for the attack ship on lower planet
Byte 7	\$01	Whether a specific attack behaviour is used.
Byte 9	planet5Level3Additional	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$FD	X Pos movement for attack ship.
Byte 20	\$24	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	planet5Level3Data	Hi Ptr for another set of wave data.
Byte 30	planet5Level3Data2ndStage	Hi Ptr for Explosion animation.
Byte 32	lickerShipWaveData	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$02	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$01	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$30	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Om Planet - Level 3 .

Om Planet - Level 4 Data

Byte	Value	Description
Byte 1	\$0E	Index into array for sprite color
Byte 3	TEARDROP_EXPLOSION1	Sprite value for the attack ship on the upper planet
Byte 4	\$04	The animation frame rate for the attack ship.
Byte 6	TEARDROP_EXPLOSION1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$10	Update rate for attack wave
Byte 18	planet5Level5Data2ndStage	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$00	Y Pos Frame Rate for Attack ship.
Byte 23	\$01	Stickiness factor, does the enemy stick to the player
Byte 24	\$00	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	spinningRings	Hi Ptr for Explosion animation.
Byte 32	lickerShipWaveData	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$02	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$02	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

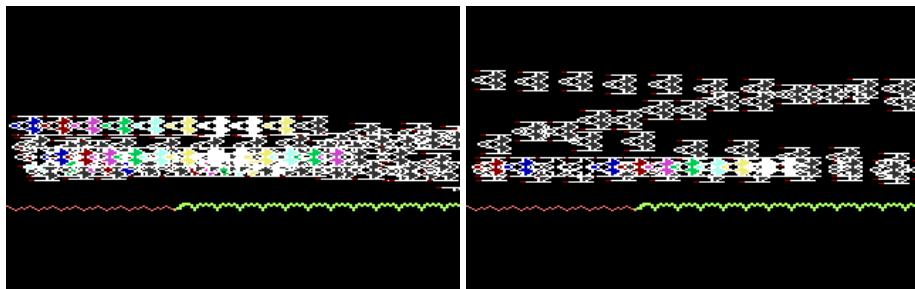
Om Planet - Level 4 .

Om Planet - Level 5 Data

Sample enemy movement for Om planet, level 5.

Byte	Value	Description
Byte 1	\$04	Index into array for sprite color
Byte 3	FLYING_COMMAS1	Sprite value for the attack ship on the upper planet
Byte 4	\$05	The animation frame rate for the attack ship.
Byte 6	FLYING_COMMAS1	Sprite value for the attack ship on lower planet
Byte 7	\$05	Whether a specific attack behaviour is used.
Byte 9	planet5Level5Additional	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$30	Update rate for attack wave
Byte 18	planet5Level5Data2ndStage	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$07	X Pos movement for attack ship.
Byte 20	\$03	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$01	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$00	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	spinningRings	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$02	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$02	Does colliding with this enemy decrease the gilby's energy?.
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?.
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Om Planet - Level 5 .

Om Planet - Level 6 Data

Sample enemy movement for Om planet, level 6.

Byte	Value	Description
Byte 1	\$0B	Index into array for sprite color
Byte 3	STARSHIP	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	STARSHIP	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$F4	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	fighterShipAsExplosion	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?
Byte 36	\$05	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

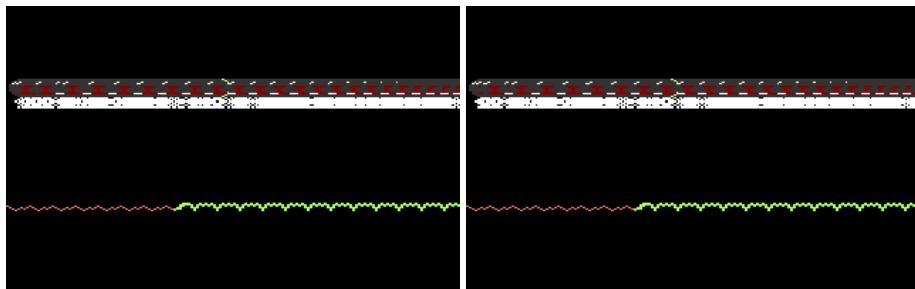
Om Planet - Level 6 .

Om Planet - Level 7 Data

Sample enemy movement for Om planet, level 7.

Byte	Value	Description
Byte 1	\$10	Index into array for sprite color
Byte 3	FLYING_FLOWCHART1	Sprite value for the attack ship on the upper planet
Byte 4	\$01	The animation frame rate for the attack ship.
Byte 6	FLYING_FLOWCHART1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$FE	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$01	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet5Level7Data2ndStage	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$01	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$03	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$20	Number of ships in wave.
Byte 40	\$00	Unused bytes.

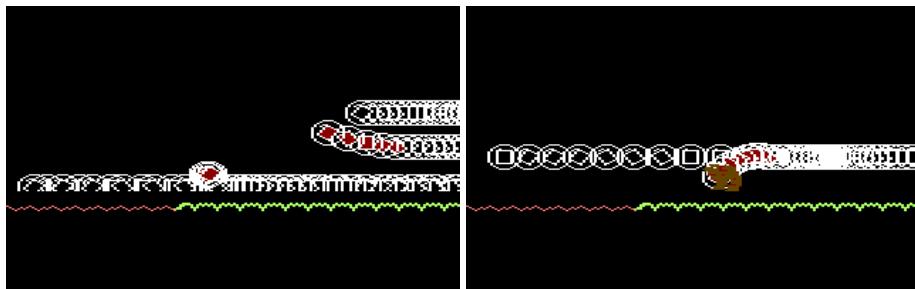
Om Planet - Level 7 .

Om Planet - Level 8 Data

Sample enemy movement for Om planet, level 8.

Byte	Value	Description
Byte 1	\$10	Index into array for sprite color
Byte 3	BALLOON	Sprite value for the attack ship on the upper planet
Byte 4	\$01	The animation frame rate for the attack ship.
Byte 6	BOUNCY_RING	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$04	X Pos movement for attack ship.
Byte 20	\$24	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	planet5Level8Data	Hi Ptr for another set of wave data.
Byte 30	planet1Level5Data	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$10	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$30	Number of ships in wave.
Byte 40	\$00	Unused bytes.

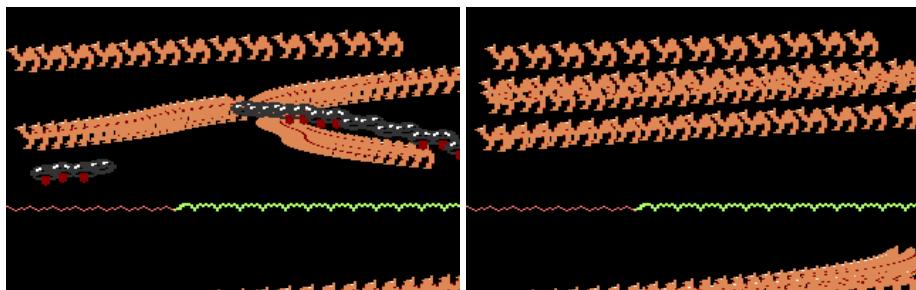
Om Planet - Level 8 .

Om Planet - Level 9 Data

Sample enemy movement for Om planet, level 9.

Byte	Value	Description
Byte 1	\$00	Index into array for sprite color
Byte 3	BOUNCY_RING	Sprite value for the attack ship on the upper planet
Byte 4	\$03	The animation frame rate for the attack ship.
Byte 6	BOUNCY_RING	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$E0	Update rate for attack wave
Byte 18	planet5Level9Data2ndStage	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$00	X Pos Frame Rate for Attack ship.
Byte 22	\$00	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$00	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet5Level9Data2ndStage	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$02	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$01	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$08	Number of ships in wave.
Byte 40	\$00	Unused bytes.

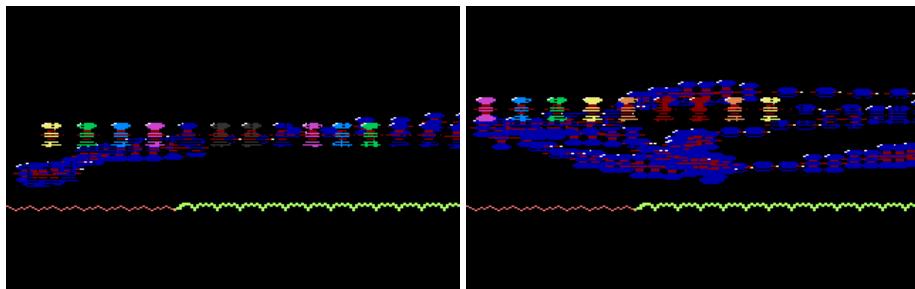
Om Planet - Level 9 .

Om Planet - Level 10 Data

Sample enemy movement for Om planet, level 10.

Byte	Value	Description
Byte 1	\$08	Index into array for sprite color
Byte 3	CAMEL	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	CAMEL	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$20	Y Pos movement pattern for attack ship.
Byte 21	\$00	X Pos Frame Rate for Attack ship.
Byte 22	\$01	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$00	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	defaultExplosion	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	lickerShipWaveData	Hi Ptr for Explosion animation.
Byte 32	lickerShipWaveData	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$03	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$03	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$18	Number of ships in wave.
Byte 40	\$00	Unused bytes.

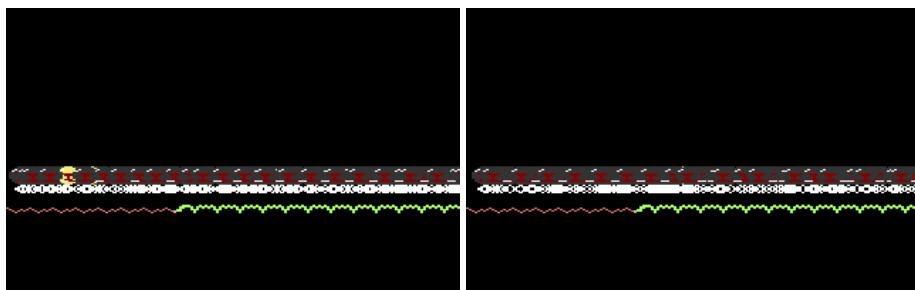
Om Planet - Level 10 .

Om Planet - Level 11 Data

Sample enemy movement for Om planet, level 11.

Byte	Value	Description
Byte 1	\$06	Index into array for sprite color
Byte 3	BIRD1	Sprite value for the attack ship on the upper planet
Byte 4	\$04	The animation frame rate for the attack ship.
Byte 6	BIRD1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$0C	Update rate for attack wave
Byte 18	planet5Level11Data	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$10	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet5Level11Data2ndStage	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$03	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$02	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$18	Number of ships in wave.
Byte 40	\$00	Unused bytes.

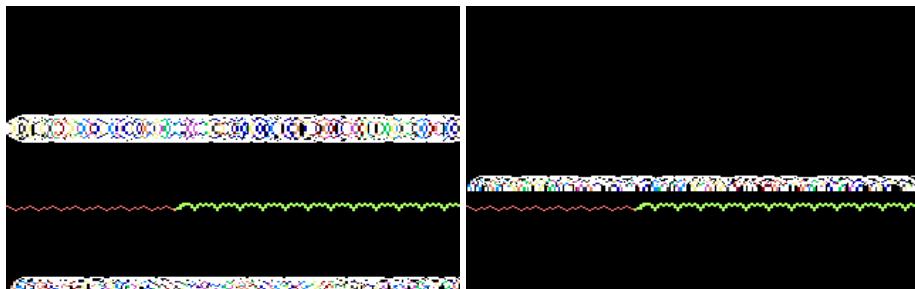
Om Planet - Level 11 .

Om Planet - Level 12 Data

Sample enemy movement for Om planet, level 12.

Byte	Value	Description
Byte 1	\$07	Index into array for sprite color
Byte 3	BALLOON	Sprite value for the attack ship on the upper planet
Byte 4	\$03	The animation frame rate for the attack ship.
Byte 6	LAND.GILBY.LOWERPLANET8	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$23	Y Pos movement pattern for attack ship.
Byte 21	\$02	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$18	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	planet1Level5Data	Hi Ptr for another set of wave data.
Byte 28	planet5Level12Data	Hi Ptr for another set of wave data.
Byte 30	nullPtr	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$88	Unused.
Byte 34	\$14	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$04	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$08	Number of ships in wave.
Byte 40	\$00	Unused bytes.

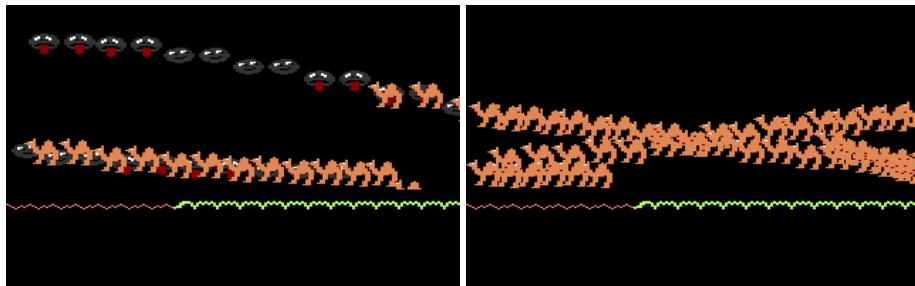
Om Planet - Level 12 .

Om Planet - Level 13 Data

Sample enemy movement for Om planet, level 13.

Byte	Value	Description
Byte 1	\$11	Index into array for sprite color
Byte 3	BUBBLE	Sprite value for the attack ship on the upper planet
Byte 4	\$01	The animation frame rate for the attack ship.
Byte 6	BUBBLE	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$00	X Pos Frame Rate for Attack ship.
Byte 22	\$00	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$00	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet5Level13Data2ndStage	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$20	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$C0	Number of ships in wave.
Byte 40	\$00	Unused bytes.

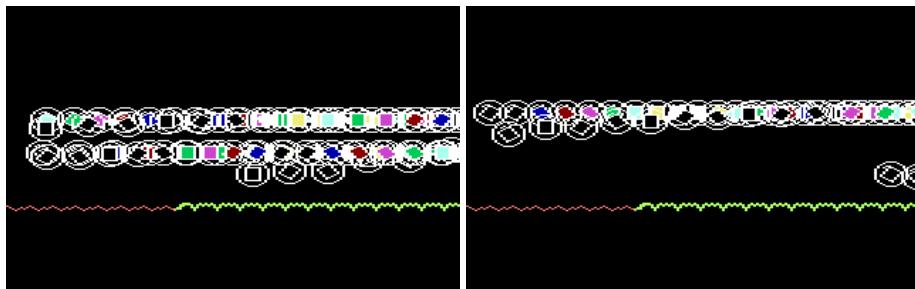
Om Planet - Level 13 .

Om Planet - Level 14 Data

Sample enemy movement for Om planet, level 14.

Byte	Value	Description
Byte 1	\$08	Index into array for sprite color
Byte 3	CAMEL	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	CAMEL	Sprite value for the attack ship on lower planet
Byte 7	\$06	Whether a specific attack behaviour is used.
Byte 9	llamaWaveData	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$FC	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	spinningRings	Hi Ptr for Explosion animation.
Byte 32	lickerShipWaveData	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$03	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$01	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$60	Number of ships in wave.
Byte 40	\$00	Unused bytes.

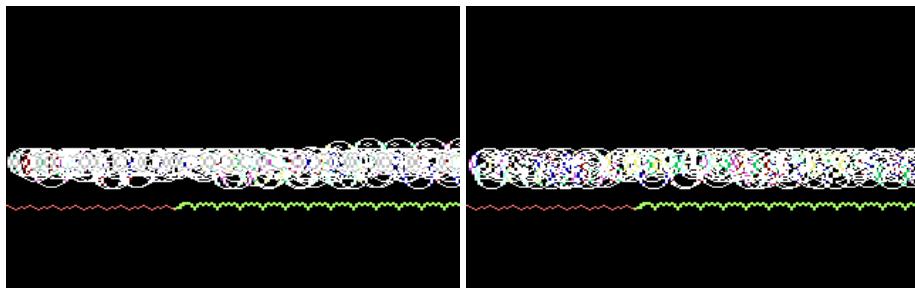
Om Planet - Level 14 .

Om Planet - Level 15 Data

Sample enemy movement for Om planet, level 15.

Byte	Value	Description
Byte 1	\$10	Index into array for sprite color
Byte 3	BOUNCY_RING	Sprite value for the attack ship on the upper planet
Byte 4	\$04	The animation frame rate for the attack ship.
Byte 6	BOUNCY_RING	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$00	X Pos Frame Rate for Attack ship.
Byte 22	\$00	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$00	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet5Level15Data2ndStage	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$06	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$10	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$10	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Om Planet - Level 15 .

Om Planet - Level 17 Data

Sample enemy movement for Om planet, level 17.

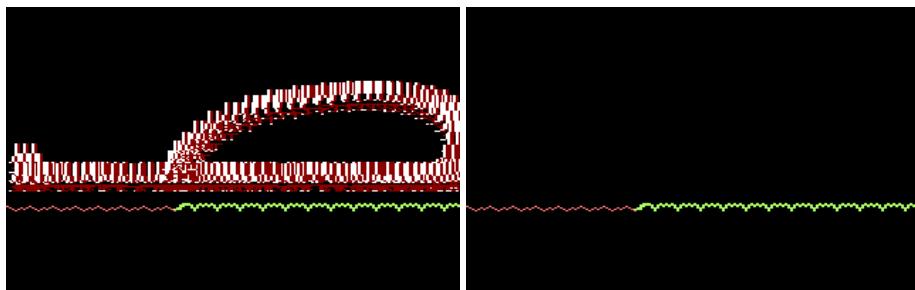
Byte	Value	Description
Byte 1	\$10	Index into array for sprite color
Byte 3	BUBBLE	Sprite value for the attack ship on the upper planet
Byte 4	\$02	The animation frame rate for the attack ship.
Byte 6	BUBBLE	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$02	X Pos movement for attack ship.
Byte 20	\$22	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$01	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	planet5Level17Data	Hi Ptr for another set of wave data.
Byte 30	planet3Level8Data	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$0C	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$30	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Om Planet - Level 17 .

Om Planet - Level 18 Data

Byte	Value	Description
Byte 1	\$10	Index into array for sprite color
Byte 3	LITTLE_OTHER_EYEBALL	Sprite value for the attack ship on the upper planet
Byte 4	\$01	The animation frame rate for the attack ship.
Byte 6	SMALL_BALL1	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$30	Update rate for attack wave
Byte 18	planet5Level18Data	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$00	X Pos movement for attack ship.
Byte 20	\$00	Y Pos movement pattern for attack ship.
Byte 21	\$02	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$01	Stickiness factor, does the enemy stick to the player
Byte 24	\$01	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	nullPtr	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet1Level5Data3rdStage	Hi Ptr for Explosion animation.
Byte 32	defaultExplosion	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$00	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$0C	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$40	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Om Planet - Level 18 .

Om Planet - Level 20 Data

Sample enemy movement for Om planet, level 20.

Byte	Value	Description
Byte 1	\$02	Index into array for sprite color
Byte 3	ATARI_ST	Sprite value for the attack ship on the upper planet
Byte 4	\$00	The animation frame rate for the attack ship.
Byte 6	ATARI_ST	Sprite value for the attack ship on lower planet
Byte 7	\$00	Whether a specific attack behaviour is used.
Byte 9	nullPtr	Lo and Hi Ptr for alternate enemy mode
Byte 11	nullPtr	Hi Ptr for an animation effect (Doesn't seem to be used?)?
Byte 12	\$00	some kind of rate limiting for attack wave
Byte 14	nullPtr	Hi Ptr for a stage in wave data (never used).
Byte 15	\$00	Unused, see GetNewShipDataFromDataStore
Byte 16	\$00	Update rate for attack wave
Byte 18	nullPtr	Hi Ptr to the wave data we switch to when first hit.
Byte 19	\$0C	X Pos movement for attack ship.
Byte 20	\$24	Y Pos movement pattern for attack ship.
Byte 21	\$01	X Pos Frame Rate for Attack ship.
Byte 22	\$02	Y Pos Frame Rate for Attack ship.
Byte 23	\$00	Stickiness factor, does the enemy stick to the player
Byte 24	\$23	Does the enemy gravitate quickly toward the player when its hit?
Byte 26	copticExplosion	Hi Ptr for another set of wave data.
Byte 28	nullPtr	Hi Ptr for another set of wave data.
Byte 30	planet5Level20Data	Hi Ptr for Explosion animation.
Byte 32	planet5Level20Data	Hi Ptr for another set of wave data for this level.
Byte 33	\$00	Unused.
Byte 34	\$00	Unused, see GetNewShipDataFromDataStore.
Byte 35	\$01	Does destroying this enemy increase the gilby's energy?.
Byte 36	\$01	Does colliding with this enemy decrease the gilby's energy?
Byte 37	\$00	Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38	\$04	Number of waves in data.
Byte 39	\$40	Number of ships in wave.
Byte 40	\$00	Unused bytes.

Om Planet - Level 20 .

Planet Data

APPENDIX C. PLANET DATA

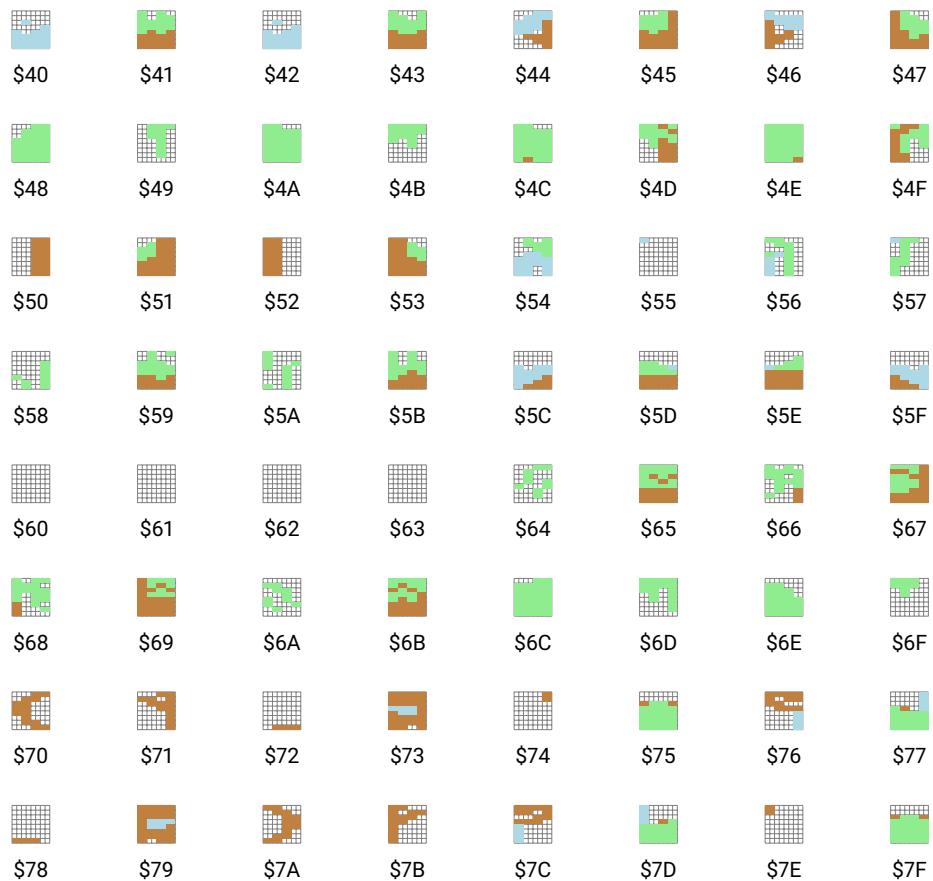


Figure C.1: Tilesheet: planet1Charset

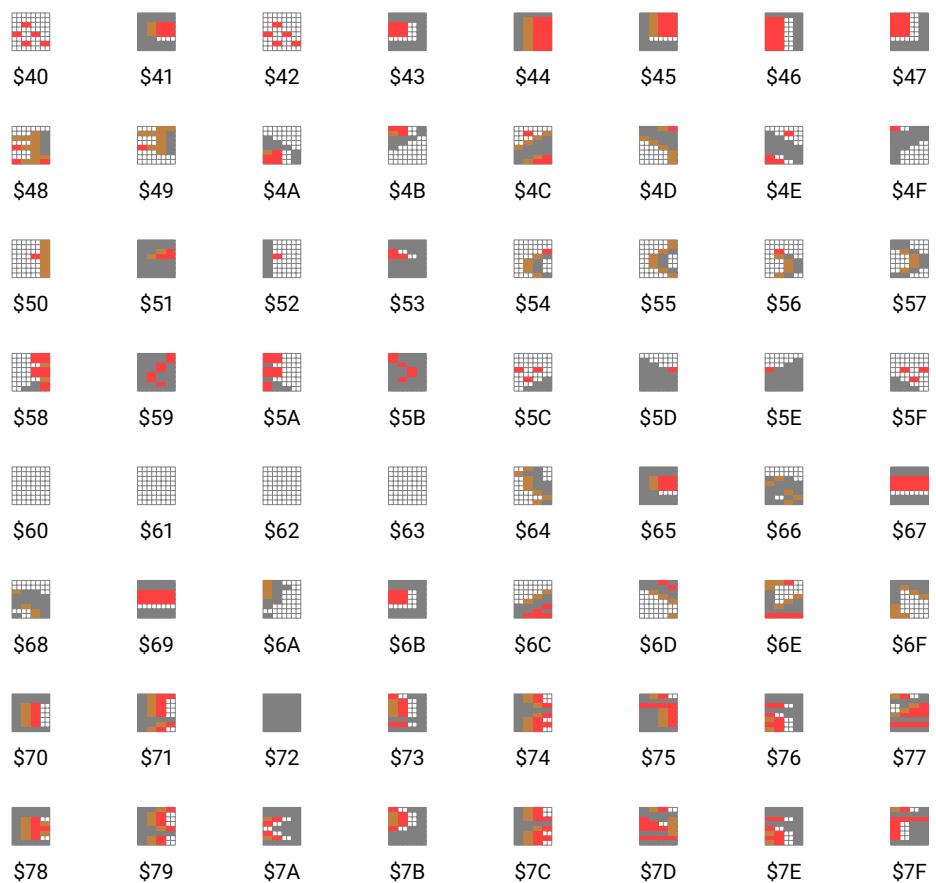


Figure C.2: Tilesheet: planet2CharSet

APPENDIX C. PLANET DATA



Figure C.3: Tilesheet: planet3Charset



Figure C.4: Tilesheet: planet4Charset

APPENDIX C. PLANET DATA

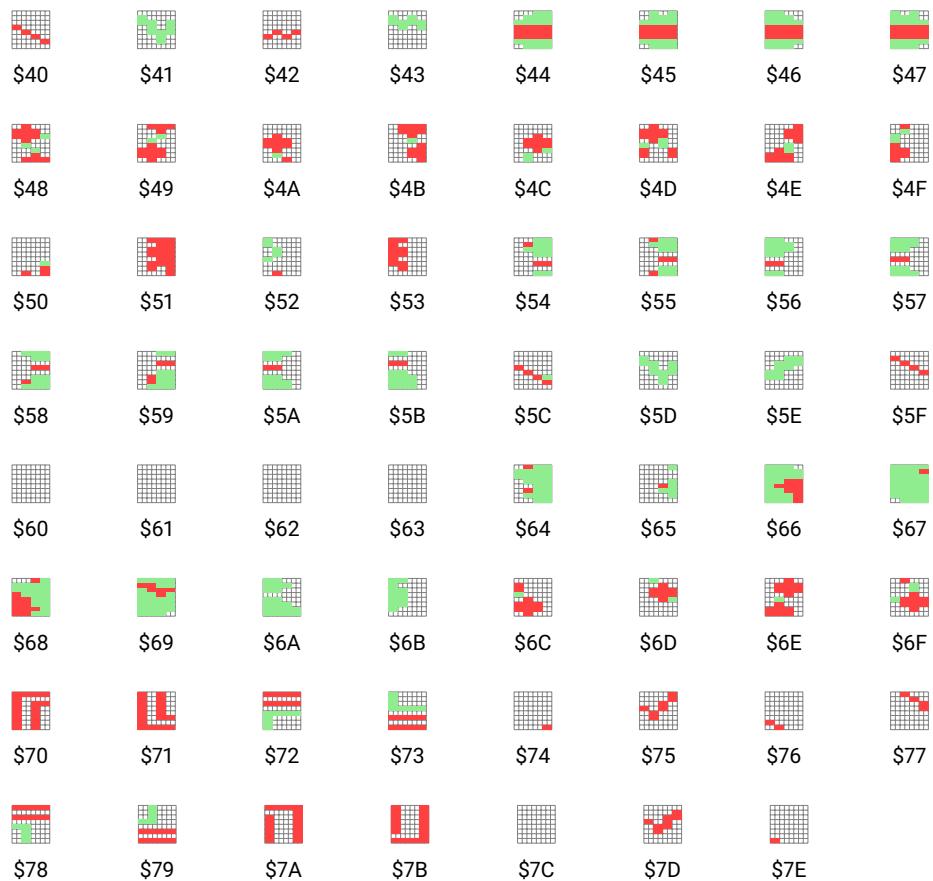
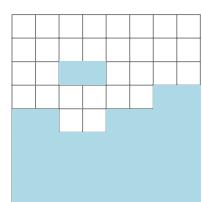
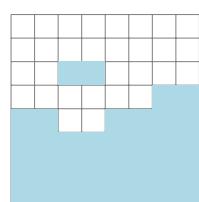


Figure C.5: Tilesheet: planet5Charset



planet1Charset \$40

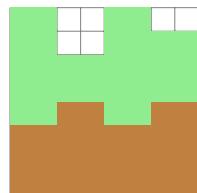


planet1Charset \$42

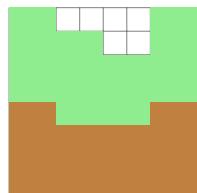
Figure C.6: Tilesheet: Planet 1 Sea.



Figure C.7: planet1Charset Sea



planet1Charset \$41

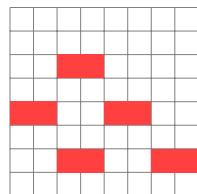


planet1Charset \$43

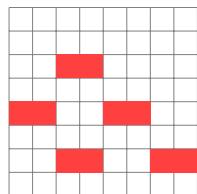
Figure C.8: Tilesheet: Planet 1 Land.



Figure C.9: planet1Charset Land



planet2Charset \$40

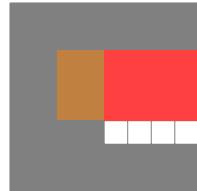


planet2Charset \$42

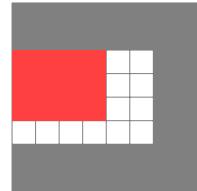
Figure C.10: Tilesheet: Planet 2 Sea.



Figure C.11: planet2Charset Sea



planet2Charset \$41

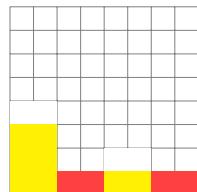


planet2Charset \$43

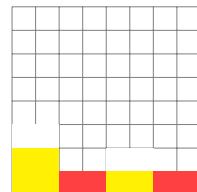
Figure C.12: Tilesheet: Planet 2 Land.



Figure C.13: planet2Charset Land



planet3Charset \$40



planet3Charset \$42

Figure C.14: Tilesheet: Planet 3 Sea.

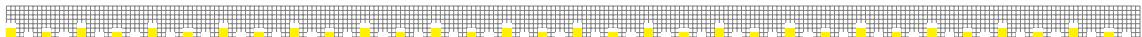
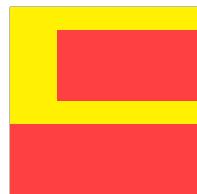
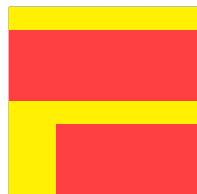


Figure C.15: planet3Charset Sea



planet3Charset \$41

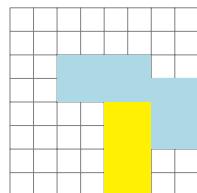


planet3Charset \$43

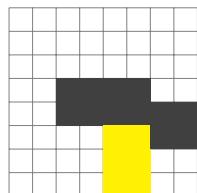
Figure C.16: Tilesheet: Planet 3 Land.



Figure C.17: planet3Charset Land



planet4Charset \$40

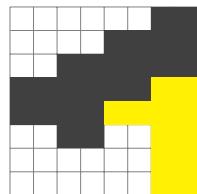


planet4Charset \$42

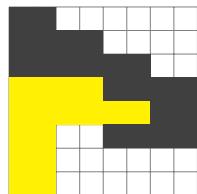
Figure C.18: Tilesheet: Planet 4 Sea.



Figure C.19: planet4Charset Sea



planet4Charset \$41

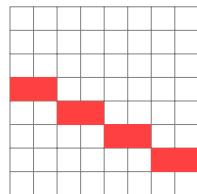


planet4Charset \$43

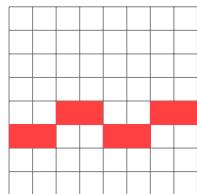
Figure C.20: Tilesheet: Planet 4 Land.



Figure C.21: planet4Charset Land



planet5Charset \$40



planet5Charset \$42

Figure C.22: Tilesheet: Planet 5 Sea.

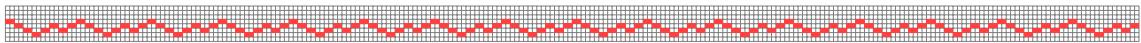
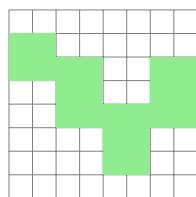
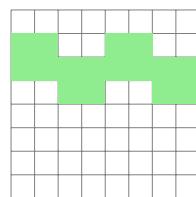


Figure C.23: planet5Charset Sea



planet5Charset \$41



planet5Charset \$43

Figure C.24: Tilesheet: Planet 5 Land.

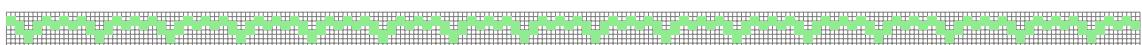


Figure C.25: planet5Charset Land

18/100,000,000,000,000 Theme Tunes

Preciously Generated

Iridis Alpha Title Theme

Jeff Miesner

(Sax)

Preciously Generated

Iridis Alpha Title Theme

Jeff Miesner

(Sax)

APPENDIX D. 18/100,000,000,000,000 THEME TUNES

Previously Generated

Iridis Alpha Title Theme

5 of 100,000,000,000,000

Art Music

Previously Generated

Iridis Alpha Title Theme

4 of 100,000,000,000,000

Art Music

Previously Generated

Iridis Alpha Title Theme

5 of 100,000,000,000,000

Art Music

Previously Generated

Iridis Alpha Title Theme

6 of 100,000,000,000,000

Art Music

APPENDIX D. 18/100,000,000,000,000 THEME TUNES

Previously Generated *Iridis Alpha Title Theme* *(Std)*
8 of 100,000,000,000,000 *All Music*

Previously Generated *Iridis Alpha Title Theme* *(Std)*
7 of 100,000,000,000,000 *All Music*

Previously Generated *Iridis Alpha Title Theme* *(Std)*
9 of 100,000,000,000,000 *All Music*

Previously Generated *Iridis Alpha Title Theme* *(Std)*
10 of 100,000,000,000,000 *All Music*

APPENDIX D. 18/100,000,000,000,000 THEME TUNES

Promised Ground

Iridis Alpha Title Theme

12 of 100/000/000/000/000

**

Precisely Generous

Iridio Alpha Title Theme
13 of 100(000000000000)

(b) *Af House*

APPENDIX D. 18/100,000,000,000,000 THEME TUNES

Previously Generated *Iridis Alpha Title Theme* *(Std.)*

15 of 100,000,000,000,000

Previously Generated *Iridis Alpha Title Theme* *(Std.)*

16 of 100,000,000,000,000

Previously Generated *Iridis Alpha Title Theme* *(Std.)*

17 of 100,000,000,000,000

APPENDIX D. 18/100,000,000,000,000 THEME TUNES

Previously Generated *Iridis Alpha Title Theme* *(Not)* *Ad Hoc*

Previously Generated *Iridis Alpha Title Theme* *(Not)* *Ad Hoc*

Previously Generated *Iridis Alpha Title Theme* *(Not)* *Ad Hoc*

Previously Generated *Iridis Alpha Title Theme* *(Not)* *Ad Hoc*

APPENDIX D. 18/100,000,000,000,000 THEME TUNES

Precisely Generalized Iridis Alpha Title Theme
24 of 100,000,000,000,000

Iridis Alpha Title Theme
26 of 100,000,000,000,000

(See All Measures)

APPENDIX D. 18/100,000,000,000,000 THEME TUNES

Previously Generated

Iridis Alpha Title Theme

28 of 100,000,000,000,000

(Std.)

Jeff Miesner

Previously Generated

Iridis Alpha Title Theme

29 of 100,000,000,000,000

(Std.)

Jeff Miesner

Previously Generated

Iridis Alpha Title Theme

30 of 100,000,000,000,000

(Std.)

Jeff Miesner

APPENDIX D. 18/100,000,000,000,000 THEME TUNES

Previously Generated *Iridis Alpha Title Theme* *(Std.)*
 31 of 100/100,000,000,000,000
All Major

Previously Generated *Iridis Alpha Title Theme* *(Std.)*
 32 of 100/100,000,000,000,000
All Major

Previously Generated *Iridis Alpha Title Theme* *(Std.)*
 33 of 100/100,000,000,000,000
All Major

Previously Generated *Iridis Alpha Title Theme* *(Std.)*
 34 of 100/100,000,000,000,000
All Major

APPENDIX D. 18/100,000,000,000,000 THEME TUNES

Previously Generated Iris Alpha Title Theme *(Std.)*
35 of 100,000,000,000,000
Ad lib.

Previously Generated Iris Alpha Title Theme *(Std.)*
36 of 100,000,000,000,000
Ad lib.

Previously Generated Iris Alpha Title Theme *(Std.)*
37 of 100,000,000,000,000
Ad lib.

Previously Generated Iris Alpha Title Theme *(Std.)*
38 of 100,000,000,000,000
Ad lib.

APPENDIX D. 18/100,000,000,000,000 THEME TUNES

Precariously Generated

Iridis Alpha Title Theme
99¢ 100,000,000,000,000

(for the
old house)

Precipitously Generated

Iridis Alpha Title Theme

40 of 100,000,000,000,000

Previously Generated

Iridis Alpha Title Theme
41 of 100,000,000,000,000

Percussively Generous

Iridis Alpha Title Theme

42 / 100,000,000 / 000,000

Agitato

APPENDIX D. 18/100,000,000,000,000 THEME TUNES

Previously Generated

Iridis Alpha Title Theme

45 of 100,000,000,000,000

(Not)

Alpha Miner

Previously Generated

Iridis Alpha Title Theme

46 of 100,000,000,000,000

(Not)

Alpha Miner

Previously Generated

Iridis Alpha Title Theme

45 of 100,000,000,000,000

(Not)

Alpha Miner

Previously Generated

Iridis Alpha Title Theme

46 of 100,000,000,000,000

(Not)

Alpha Miner

APPENDIX D. 18/100,000,000,000,000 THEME TUNES

Precociously Gave

Iridis Alpha Title Theme
48 of 100,000,000,000,000

(b)1
Act Three

Iridis Alpha Title Theme
49 of 100,000,000,000,000

Precariously Grounded

Iridis Alpha Title Theme

50 | 100,000,000,000,000

(S)
 Key Change

APPENDIX D. 18/100,000,000,000,000 THEME TUNES

Preciously Generated

Iridis Alpha Title Theme

51 of 100,000,000,000,000

(Std.)

Alt Metric

Preciously Generated

Iridis Alpha Title Theme

52 of 100,000,000,000,000

(Std.)

Alt Metric

Preciously Generated

Iridis Alpha Title Theme

53 of 100,000,000,000,000

(Std.)

Alt Metric

Preciously Generated

Iridis Alpha Title Theme

54 of 100,000,000,000,000

(Std.)

Alt Metric

APPENDIX D. 18/100,000,000,000,000 THEME TUNES

Previously Generated Iris Alpha Title Theme *(Std)*
55 of 100/100,000,000,000,000
All Music

Previously Generated Iris Alpha Title Theme *(Std)*
56 of 100/100,000,000,000,000
All Music

Previously Generated Iris Alpha Title Theme *(Std)*
57 of 100/100,000,000,000,000
All Music

Previously Generated Iris Alpha Title Theme *(Std)*
58 of 100/100,000,000,000,000
All Music

APPENDIX D. 18/100,000,000,000,000 THEME TUNES

Preciously Gained

Iridis Alpha Title Theme
59 of 100,000,000,000,000

Ad Hoc

Preciously Gained

Iridis Alpha Title Theme
60 of 100,000,000,000,000

Ad Hoc

Preciously Gained

Iridis Alpha Title Theme
61 of 100,000,000,000,000

Ad Hoc

APPENDIX D. 18/100,000,000,000,000 THEME TUNES

Previously Generated

Iridis Alpha Title Theme

65 of 100,000,000,000,000

Left Hand

Iridis Alpha Title Theme

66 of 100,000,000,000,000

Left Hand

Previously Generated

Iridis Alpha Title Theme

67 of 100,000,000,000,000

Left Hand

Iridis Alpha Title Theme

68 of 100,000,000,000,000

Left Hand

APPENDIX D. 18/100,000,000,000,000 THEME TUNES

Iridis Alpha Title Theme
67 of 100,000,000,000,000

(Std) *Adagio*

Previously Generated

Iridis Alpha Title Theme
67 of 100,000,000,000,000

(Std) *Adagio*

Previously Generated

Iridis Alpha Title Theme
68 of 100,000,000,000,000

(Std) *Adagio*

Previously Generated

Iridis Alpha Title Theme
68 of 100,000,000,000,000

(Std) *Adagio*

Previously Generated

APPENDIX D. 18/100,000,000,000,000 THEME TUNES

Preciously Generated

Iridis Alpha Title Theme
71 of 100,000,000,000,000,000

Precisely Ground

Irids Alpha Title Theme

72 of 100,000,000 (000,000)

Precisely Generated Iris Alpha Title Theme (35.000.000.000.000.000)

(Sax
Alg Harp)

Precociously Genuinely

Iridis Alpha Title Theme

74 of 100,000,000,000,000

APPENDIX D. 18/100,000,000,000,000 THEME TUNES

Previously Generated

Iridis Alpha Title Theme

77 of 100,000,000,000,000

Art Music

This musical score page shows the first ten measures of the Iridis Alpha Title Theme. The title is at the top center, and the subtitle '77 of 100,000,000,000,000' is just below it. The key signature is A major (no sharps or flats). The time signature is common time (indicated by 'C'). The score consists of five staves, each labeled 'Vocal 1', 'Vocal 2', 'Vocal 3', 'Vocal 4', and 'Vocal 5'. The vocal parts are primarily composed of eighth-note patterns, with some sixteenth-note figures and rests. Measure 1 starts with a forte dynamic. Measures 2-3 show a transition with more complex rhythms. Measures 4-5 continue the melodic line. Measures 6-7 feature a more sustained harmonic pattern. Measures 8-9 show a return to the earlier rhythmic style. Measure 10 concludes the section.

Previously Generated

Iridis Alpha Title Theme

78 of 100,000,000,000,000

Art Music

This musical score page shows measures 11 through 20 of the Iridis Alpha Title Theme. The title is at the top center, and the subtitle '78 of 100,000,000,000,000' is just below it. The key signature is A major. The time signature is common time. The vocal parts continue their eighth-note patterns. Measure 11 begins with a forte dynamic. Measures 12-13 show a continuation of the melodic line. Measures 14-15 feature a more sustained harmonic pattern. Measures 16-17 show a return to the earlier rhythmic style. Measure 18 concludes the section.

Previously Generated

Iridis Alpha Title Theme

77 of 100,000,000,000,000

Art Music

This musical score page shows measures 11 through 20 of the Iridis Alpha Title Theme. The title is at the top center, and the subtitle '77 of 100,000,000,000,000' is just below it. The key signature is A major. The time signature is common time. The vocal parts continue their eighth-note patterns. Measure 11 begins with a forte dynamic. Measures 12-13 show a continuation of the melodic line. Measures 14-15 feature a more sustained harmonic pattern. Measures 16-17 show a return to the earlier rhythmic style. Measure 18 concludes the section.

Previously Generated

Iridis Alpha Title Theme

78 of 100,000,000,000,000

Art Music

This musical score page shows measures 11 through 20 of the Iridis Alpha Title Theme. The title is at the top center, and the subtitle '78 of 100,000,000,000,000' is just below it. The key signature is A major. The time signature is common time. The vocal parts continue their eighth-note patterns. Measure 11 begins with a forte dynamic. Measures 12-13 show a continuation of the melodic line. Measures 14-15 feature a more sustained harmonic pattern. Measures 16-17 show a return to the earlier rhythmic style. Measure 18 concludes the section.

APPENDIX D. 18/100,000,000,000,000 THEME TUNES

Precisely Generated

Iridis Alpha Title Theme

(MIDI File)

79 of 100/100,000,000,000,000

Precisely Generated

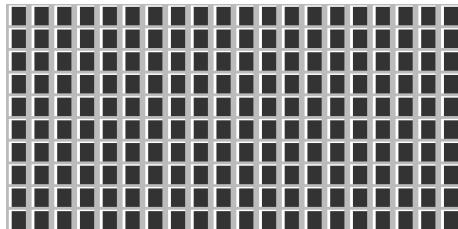
Iridis Alpha Title Theme

(MIDI File)

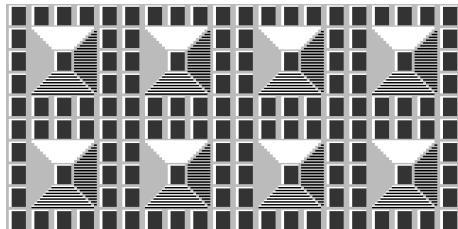
80 of 100/100,000,000,000,000

Bonus Phase - Map Segments

APPENDIX E. BONUS PHASE - MAP SEGMENTS



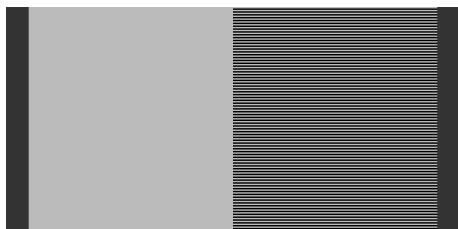
\$00,\$00,\$00,\$00,\$00,\$00,\$00,\$00,\$00,\$00



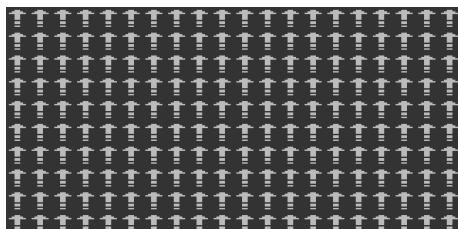
\$00,\$15,\$16,\$17,\$00,\$00,\$15,\$16,\$17,\$00



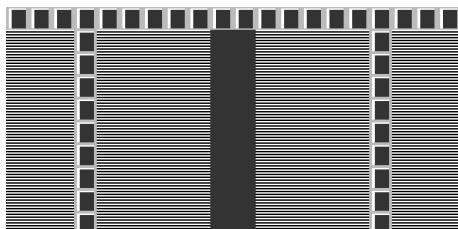
\$00,\$14,\$14,\$14,\$14,\$14,\$14,\$14,\$14,\$00



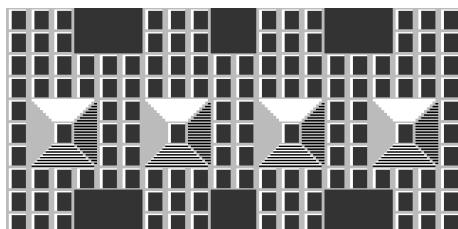
\$11,\$11,\$11,\$11,\$11,\$11,\$11,\$11,\$11,\$11



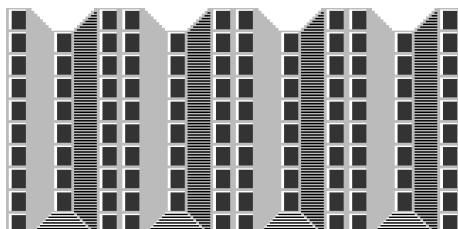
\$13,\$13,\$13,\$13,\$13,\$13,\$13,\$13,\$13,\$13



\$12,\$12,\$12,\$12,\$12,\$12,\$12,\$12,\$12,\$00



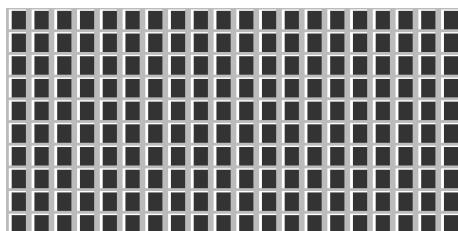
\$14,\$14,\$00,\$15,\$16,\$17,\$00,\$00,\$14,\$14



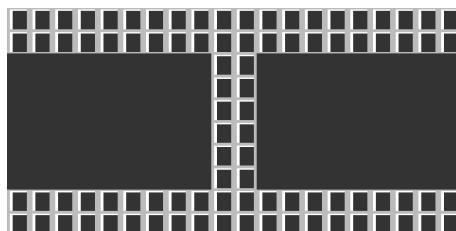
\$15,\$16,\$16,\$16,\$16,\$16,\$16,\$16,\$16,\$17

Figure E.1: Bonus Phase Map Segments.

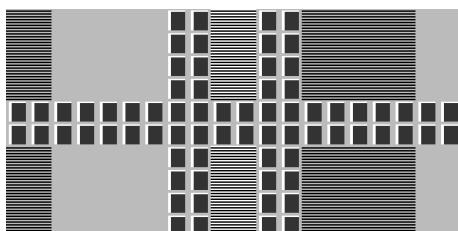
APPENDIX E. BONUS PHASE - MAP SEGMENTS



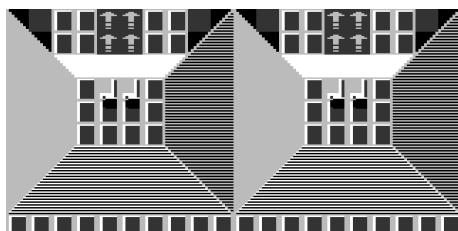
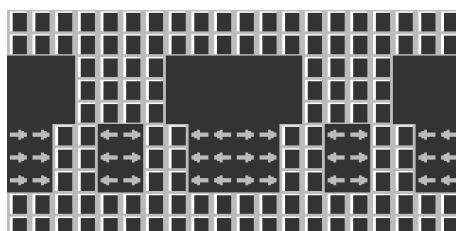
\$00,\$00,\$00,\$00,\$00,\$00,\$00,\$00,\$00,\$00



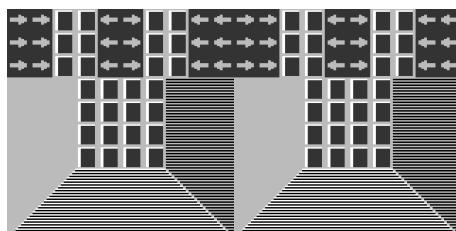
\$00,\$00,\$0F,\$0F,\$0F,\$0F,\$0F,\$0F,\$00,\$00



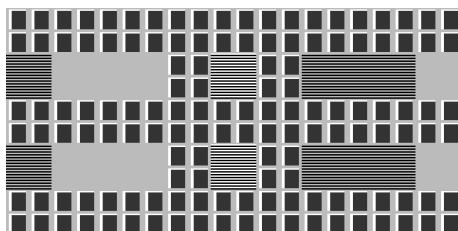
\$01,\$01,\$01,\$01,\$00,\$00,\$01,\$01,\$01,\$01



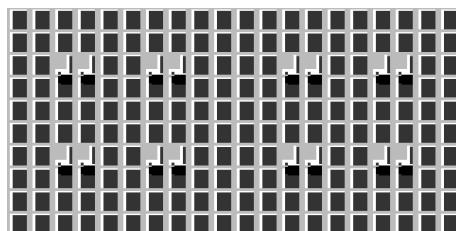
\$00,\$00,\$0B,\$0B,\$0B,\$0C,\$0C,\$0C,\$00,\$00



\$00,\$02,\$03,\$04,\$05,\$06,\$07,\$08,\$09,\$0A



\$02,\$03,\$04,\$05,\$05,\$05,\$0B,\$0B,\$0B



\$00,\$00,\$01,\$01,\$00,\$00,\$01,\$01,\$00,\$00

Figure E.2: Bonus Phase Map Segments.

APPENDIX E. BONUS PHASE - MAP SEGMENTS

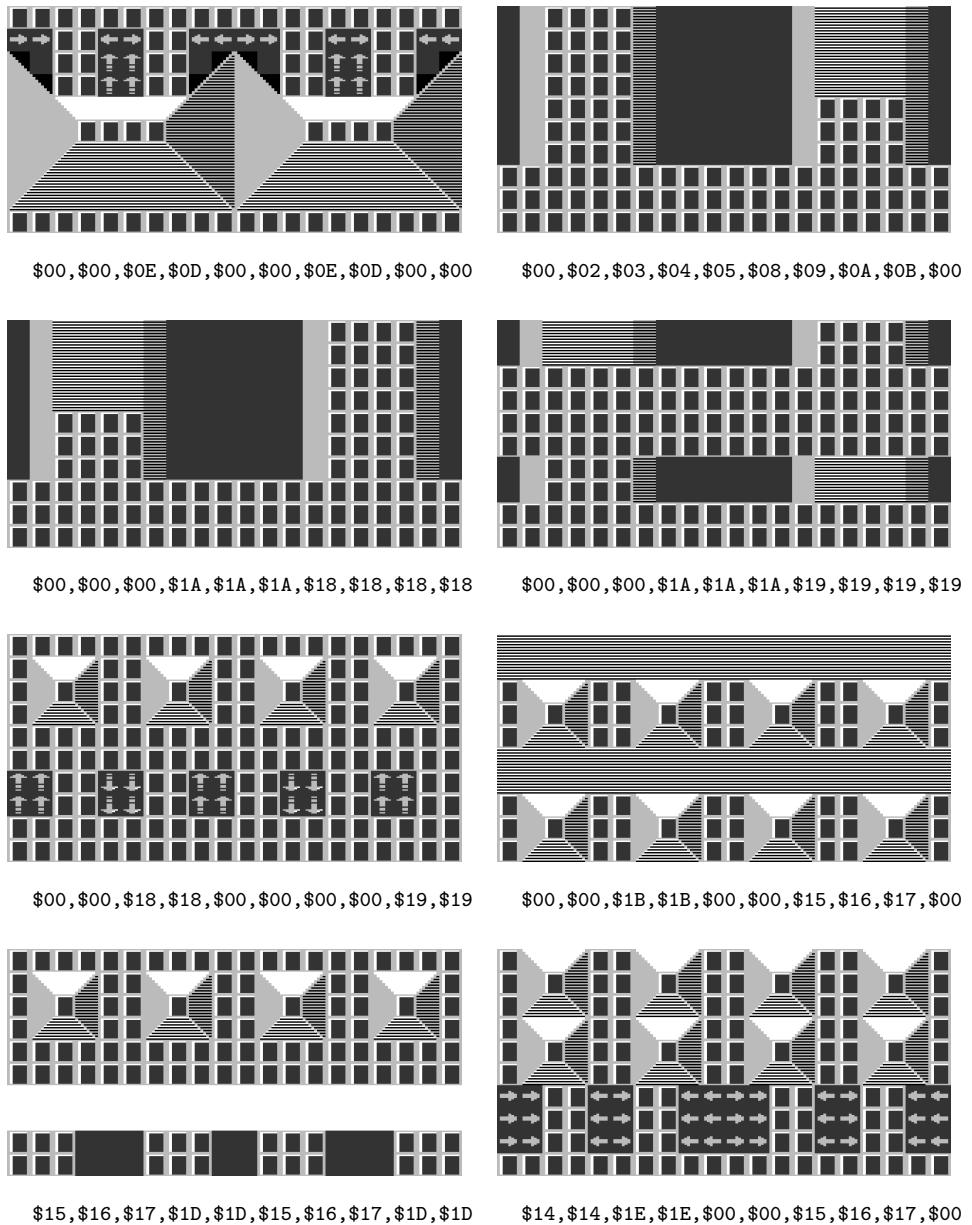


Figure E.3: Bonus Phase Map Segments.

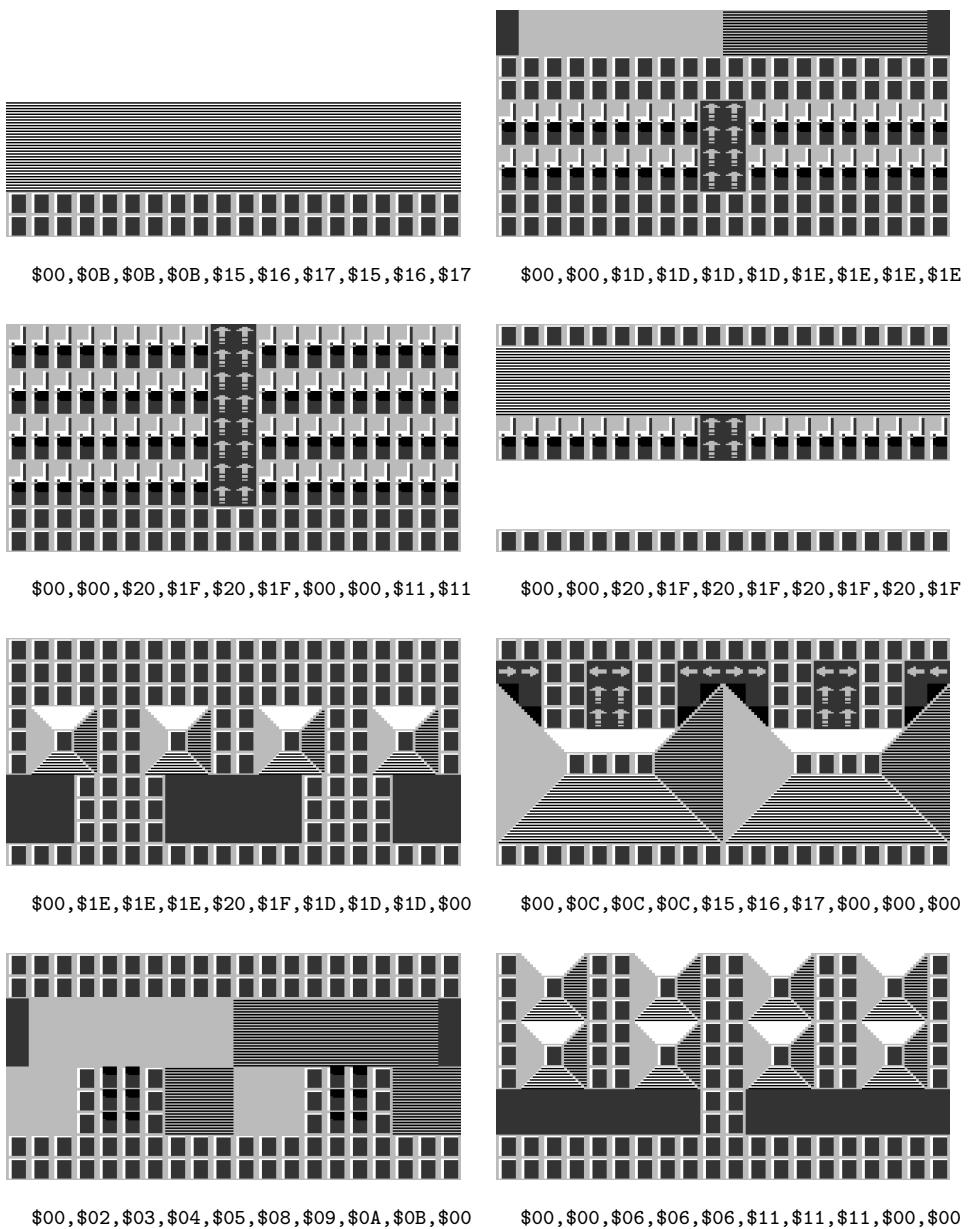


Figure E.4: Bonus Phase Map Segments.

Bonus Phase - Map Rows

APPENDIX F. BONUS PHASE - MAP ROWS

Index	Image
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	

Figure F.1: The first 16 of the 32 row definitions.

APPENDIX F. BONUS PHASE - MAP ROWS

Index	Image
16	
17	
18	
19	
20	
21	
22	
23	
24	
25	
26	
27	
28	
29	
30	
31	

Figure F.2: The second 16 of the 32 row definitions.

Bonus Phase - Tilesheet

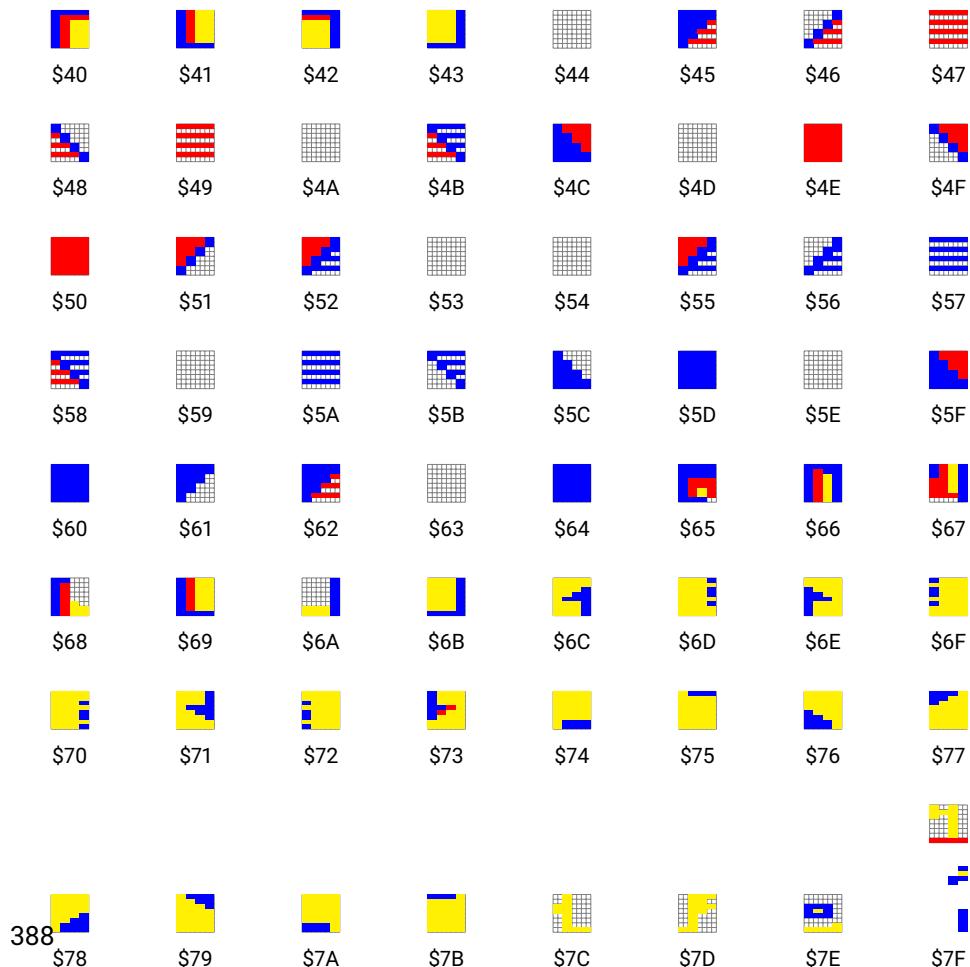


Figure G.1: Tilesheet: Bonus Phase

Bumph

Cover

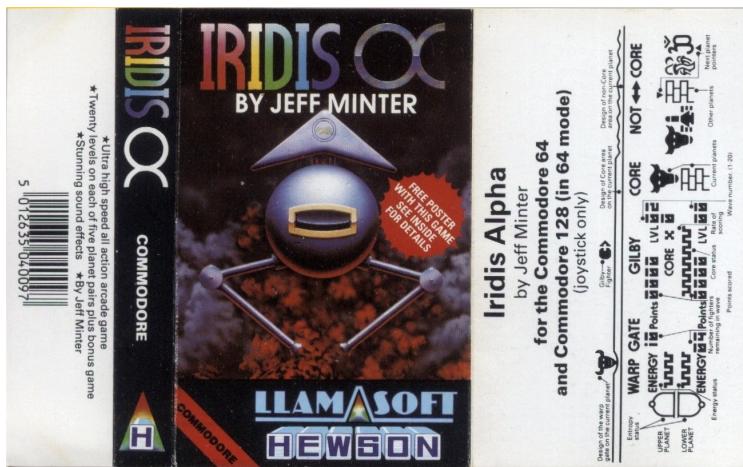


Figure H.1: Front Cover

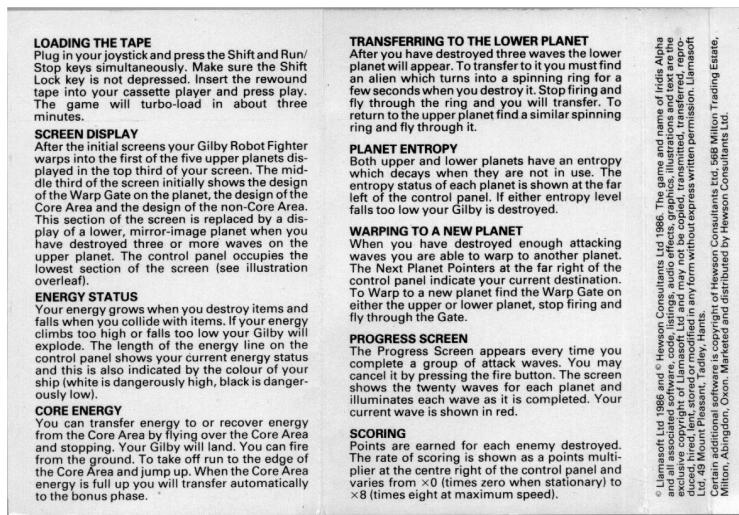


Figure H.2: Back Cover

Manual

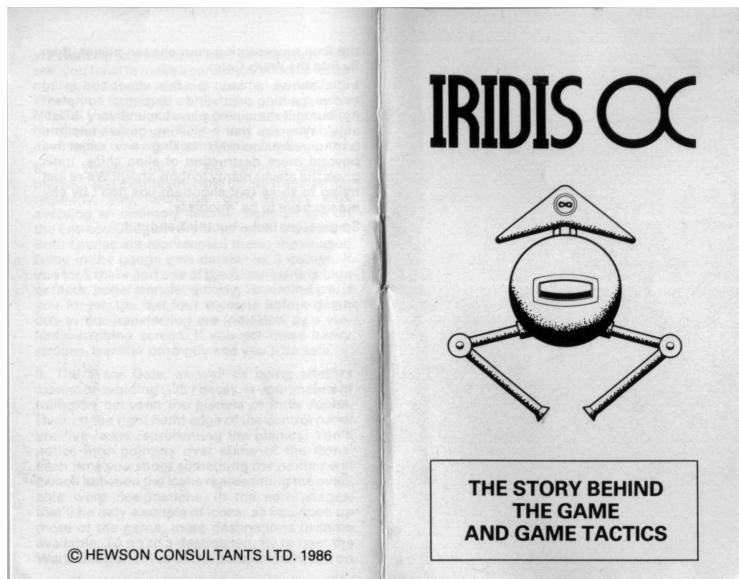


Figure H.3: Back Cover

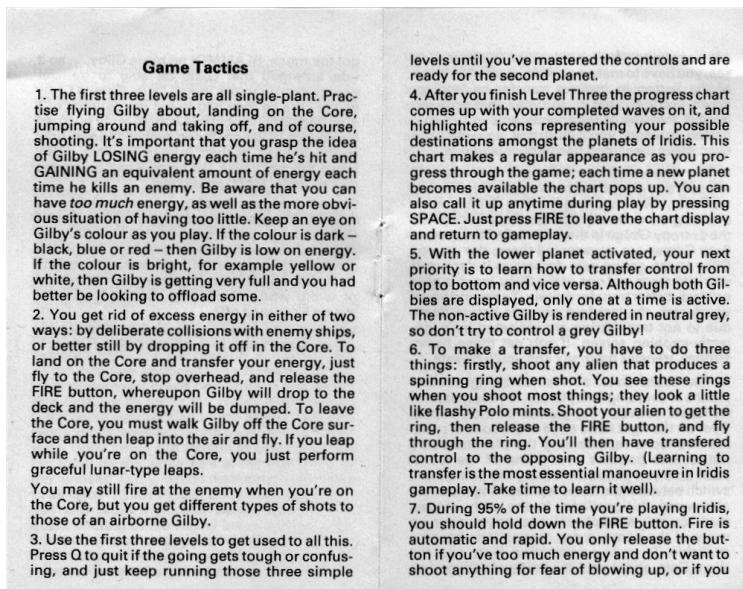


Figure H.4: Back Cover

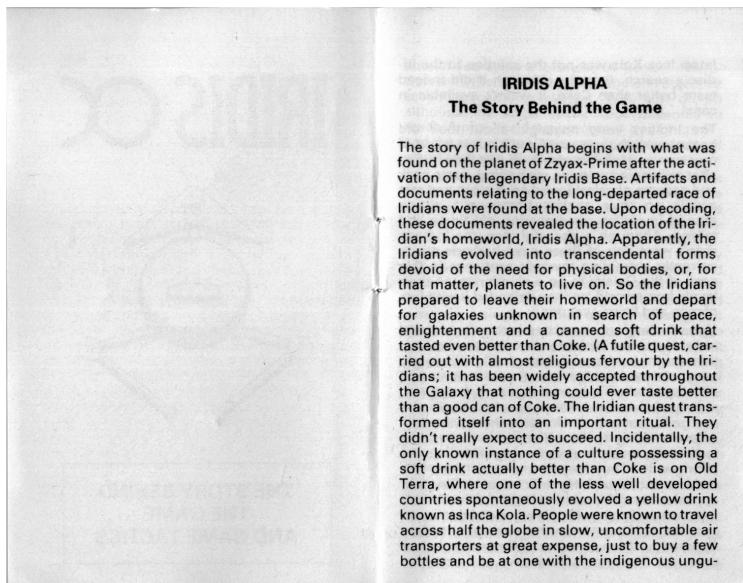


Figure H.5: Back Cover

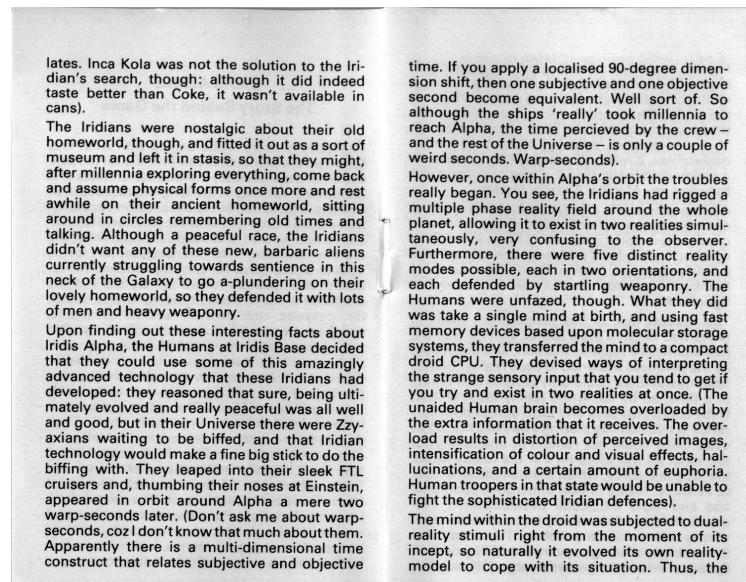


Figure H.6: Back Cover

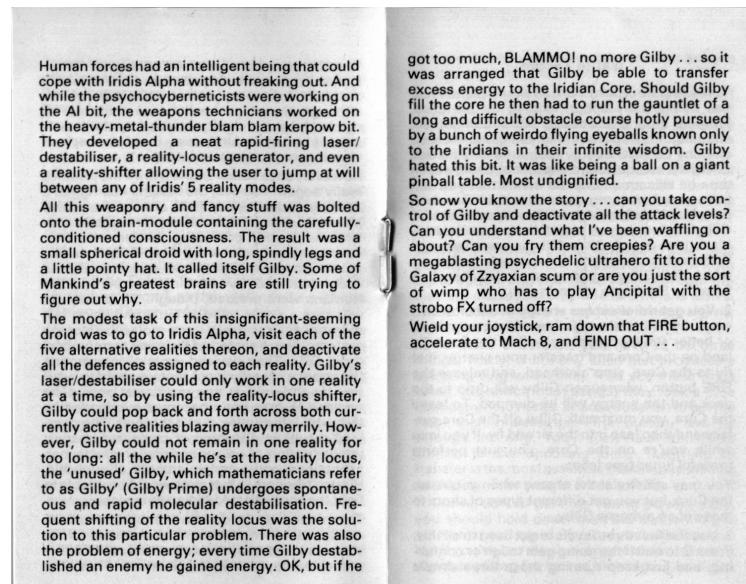


Figure H.7: Back Cover

are wanting to transfer or land on the Core. You see, you have to make a conscious effort to let go of the button to make a transfer or landing. When you get used to the idea that you can NEVER transfer or land while you've got that button pressed, you'll find that you rarely, if ever, make unwanted transfers or landings.

8. The unused Gilby of the pair will decay if unused, and eventually blow up. By transferring regularly, you 'recharge' both Gilbies thus avoiding an untimely demise. Keep an eye on the Entropy Gauge in the lower left of the screen. Both Gilbies are represented there; the unused Gilby in the gauge gets darkest as it decays. If you look there and one of the Gilby-icons is blue or black, better transfer quickly. To remind you if you forget, the last four seconds before death due to not transferring are indicated by a violently-strobing screen. If you get these heavy strobos, transfer promptly and you'll be safe.

9. The Warp Gate, as well as being another means of avoiding Gilby decay, is your means of transport between the planets of Iridis Alpha. Over on the right hand edge of the control panel are five icons representing the planets. You'll notice little pointers over some of the icons. Each time you shoot something the pointer will switch between the icons representing the available warp designations. In the early stages, that'll be only a couple of icons; as you open up more of the game, more destinations become available. To go to a destination, fly to near the Warp Gate, shoot until the pointer is aligned on

the icon representing your chosen planet, then fly into the Warp Gate.

Iridis shows a way for the shoot-em-up to evolve, gaining depth and a degree of complexity, but still remaining playable and very 'blastable'. Whereas many blasting games become boring very quickly, lacking any objective beyond mere destruction of alien ships, Iridis gives the player plenty to think about. We're just trying to show that shoot-em-ups don't by any means have to be 'mindless' . . .

So give 'em hell – but think about it . . .

Figure H.8: Back Cover

Notes & References

- [1] **TAP Format**, A description of the TAP Format, C64 Wiki.
- [2] **The Megasave Loader**, Luigi Di Fraia's Mega SaveLoader page.
- [3] **A History of Llamasoft**, Jeff Minter's memoir of the early years of Llamasoft.
- [4] **C64 Programmer's Reference Guide**, The bible for C64 programmers.
- [5] **Programming the 6502**, Rodnay Zaks The book used by Jeff Minter to learn assembly. "Learning Machine Code ... I used PROGRAMMING THE 6502 by Rodney Zaks as a good 6502 book, and used the VIC and 64 Programmers Reference Guides for each machine. The VIC REVEALED is also good for VIC 20 owners."
- [6] **Iridis Alpha Source Code**, Source code for Iridis Alpha, reverse-engineered by the author of this book.
- [7] **Iridis Alpha: Tape Download**, A tape copy of Iridis Alpha.

From the preface:

This book describes the inner workings of a relatively obscure video game created by an eccentric Englishman in 1986 for the Commodore 64.

If you are curious about old computers such as the Commodore 64 or the detailed mechanics of making a glorified digital breadboard produce something on a screen that flashes, bleeps, and fascinates then this book is hopefully for you.

