Squeezing the ZX Spectrum

Download the code from GitHub: **wfmag.cc/wfmag19**

The Oliver Twins explain how they squeezed Fantasy World Dizzy into just 41kB of memory

AUTHORS
PHILIP AND ANDREW OLIVER

The Oliver Twins have been making games since the early eighties, and can now be found at their new consultancy firm, Game Dragons. **gamedragons.com**

We were relatively late getting our first ZX Spectrum. It was October 1986 when David Darling of Codemasters sent us one, and suggested we convert our games from the Amstrad versions we were creating at the time. Beginning with *Ghost Hunters*, we went on to create 17 Spectrum games over the next five years, many of them bestsellers. The most fondly remembered of those are the *Dizzy* games.

With *Dizzy*, we wanted to create a cartoon adventure where the player solved puzzles. Being fans of adventure games like *Zork*, we knew we needed an inventory and interesting, logical puzzles. These were functionally still 'key and door' puzzles, but dressed up with lots of themes: for example, the Mucky Grease Gun that enabled a rusted mine-cart to be moved in order to access a mine. There was a manhole cover blocking another route, which could be opened with a crowbar. At this time, Dizzy could only pick up and drop a single item at a time, leading to some additional gameplay of figuring out the optimum route and where best to leave items for collection later.

While *Dizzy* was slow to catch on, we eventually produced a sequel, 1988's *Treasure Island Dizzy*, with a new story and improved game mechanics. It was commercially successful, but there were lots of areas we wanted to improve; so in the summer of 1989 we designed *Dizzy 3: Fantasy World Dizzy*. We knew from the outset this would be a hit, whatever was in the box, but we wanted to make sure nobody was disappointed. We also wanted to stretch the Spectrum to its limits and make the best possible *Dizzy* game.

We enhanced the inventory by having a pop-up window selection system. We added names to every location and added 30 coins to collect; but most important was the story and the introduction of a bunch of new characters, and a lot of dialogue. There were 50 screens in all, and the game took over 30 minutes to complete if you knew what you were doing; but, like all games at that time, if you lost all your lives, you had to start over. Most players would play for over 50 hours to beat the game. That's quite a

ZX Spectrum games took ages to load, but colourful screens like this one kept players company for those lonely minutes.

› **The Spectrum's price point helped bring gaming to the eighties masses.**

lot of entertainment for £2.99, and squeezed into just 41kB.

Fantasy World Dizzy was met with critical and commercial success, and went on to sell over half a million copies, being converted to many different computers; years later, it still has a cult following. So how did we squeeze all that onto the Spectrum?

FIRST-GEN BEDROOM CODERS

We weren't taught to code or make games; we had to work everything out for ourselves. There was no internet, and few books on the subject. We got ideas and inspiration from other people's games, either in arcades or on home computers. We'd challenge ourselves to achieve as much as possible, but also gave ourselves self-imposed, tight deadlines: we set ourselves the challenge of creating one game a month.

Fantasy World Dizzy took about six weeks, and we developed it first on the Amstrad and then converted graphics and some routines to the Spectrum. Since the Spectrum was similar in many ways to the Amstrad and also based on Z80, most of the game code was identical. The only differences were in the input routine (reading keys) and output routines (display and sound). This was our 17th (and last) game on the Spectrum; so, reusing code from other games, the Spectrum version only took a day or two.

RAM

We ensured our games all ran on the 48K Spectrum. The screen was 256 pixels (32 bytes) × 192 pixels, so a total of 6144 bytes for the bitmap data, plus another 768 bytes for the colour attributes, making a total of 6.6kB. After a few other system memory reserves, there was only 41kB of RAM left for the game.

HEXADECIMAL AND BINARY

You'll notice there's an awful lot of counting in hexadecimal and working out in binary. The fact is the 8-bit language requires you to be constantly thinking in these numbers. Everything relies on it, and if you start thinking in these terms, you *will* write better code.

GRAPHICS

We created the sprites (all the 2D graphic assets) in our own 'Panda Sprites' utility, which we'd written and published on the Amstrad when we were at school. We figured every hobbyist would want to create games with animated sprites, and went to great efforts to give the editor plenty of features, even including rotation of sprites for people making top-down games. This feature led to Dizzy doing his spinning jump. We spent time creating flexible sprite printing routines that people could put in their own games. We even allowed them to write their games in BASIC, which was really easy to use. It was possibly the first middleware for game creation.

Printing sprites on a Spectrum was interesting. As with the BBC B and Amstrad, the way the ➔

THE ZX SPECTRUM

The ZX Spectrum was an affordable computer with just 48kB of RAM and a 'rubber' keyboard. It had blocky, low-resolution graphics, but it could plug into a regular TV and used a cassette player for storage. Its main processor was an 8-bit Z80 running at 3.5MHz. The screen resolution was 256×192 pixels, and supported eight colours, with 'colour attributes' limited to one foreground and one background colour per 8×8 character. This severely limited what colours could be displayed and where, but it was efficient on memory and improved the speed of printing graphics to the screen; it also led to the distinctive appearance of the computer's games.

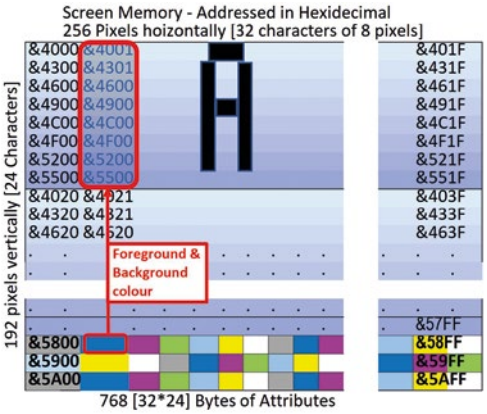


Figure 1: A diagram showing the ZX Spectrum's screen memory in hexadecimal.

screen memory is laid out is odd (see **Figure 1**). If you watch a Spectrum game's title screen loading, it starts top-left, goes across to the right, jumps down one character line (eight pixels) and works all the way down the screen. Finally, it fills in all the colour attributes to display the finished picture.

CODE

There isn't space here to explain assembler coding on the Z80 in detail, but here's a basic overview. The 8-bit chip had a main A register, three multi-purpose paired registers: BC, DE, and HL, and then two further paired registers: IX and IY. These were duplicated for use by interrupt routines. The paired registers gave 16-bit pointers to address up to 64kB RAM/ROM.

There were standard operations: LD (load), ADD, AND, XOR, SLA, and SRA (shift bits left or right within a byte), along with conditional operations – compare and jump to a new section of code – but it didn't have multiply or divide operations, and many things had to be done by creating loops and lookup tables.

Not all operations would work on all registers: everything would work with register A, but only subsets of operations worked with the other registers, so there was an awful lot of register swapping needed to write Z80 code.

The simplest 'load' (LD) instruction took four clock cycles and, depending how it was used, could take up to 19 (see **Figure 2**).

Coding assembler efficiently was essential, given the limitations of the 3.5MHz Z80 processor. This was where being a 'Code Master' really came into play. We were literally obsessed by writing fast, efficient code, counting every byte and clock cycle (time to execute an instruction), but also coming up with advanced techniques to reduce the runtime logic required.

For example, there's a common routine you'd need to get the memory location of an X and Y coordinate on the screen. The conventional method was as shown in **Figure 3** (we recently pulled this from the internet – a luxury we didn't have back then).

We'd literally write random bytes to memory to see what appeared where, then reverse-engineer the screen memory configuration and develop code to write to it! But we knew everything printed on to the screen went via this, so if we could speed it up, our games could have far more things on the screen (see **Figure 4**).

This required two tables each of 192 bytes to be created, each on consecutive page boundaries – all the Low screen addresses in the first table, then 256 bytes later all the High screen addresses.

Because the code was now very short, we'd paste it in-line, when needed, to avoid the extra 'call' and 'return' from a routine.

The upside-down levels (**Figure 5**) were created as normal; we'd just flip the values in the lookup tables.

The Oliver Twins learned Z80 from a £2.99 Pocket Guide.

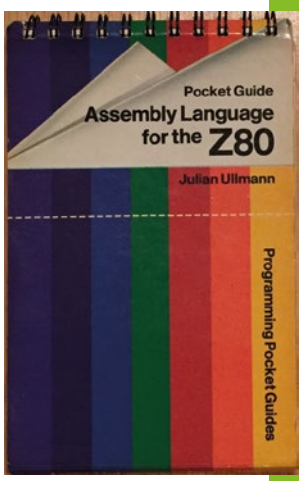


Figure 3: The conventional method to locate the screen memory from a coordinate.

```
                                ; B = Y pixel position
                                ; C = X pixel position
                                ; Returns address in HL
   Get_Pixel_Address_Conventional
4      LD A,B                   ;Calculate Y2,Y1,Y0
7      AND %00000111            ;Mask out unwanted bits
7      OR %01000000             ;Set base address of screen
4      LD H,A                   ;Store in H
4      LD A,B                   ;Calculate
4      RRA                      ;Shift to position
4      RRA                      ;Shift to position
4      RRA                      ;Shift to position
7      AND %00011000            ;Mask out unwanted bits
4      OR H                     ;OR with Y2,Y1,Y0
4      LD H,A                   ;Store in H
4      LD A,B                   ;Calculate Y5,Y4,Y3
4      RLA                      ;Shift to position
4      RLA                      ;Shifting bits left
7      AND %11100000            ;Mask out unwanted bits
4      LD L,A                   ;Store in L
4      LD A,C                   ;Calculate X4,X3,X2,X1,X0
4      RRA                      ;Shift into position
4      RRA                      ;Shifting bits right
4      RRA                      ;
7      AND %00011111            ;Mask out unwanted bits
4      OR L                     ;OR with Y5,Y4,Y3
4      LD L,A                   ;Store in L
10     RET                      ;Return from routine
117 Total Clock Cycles          ;28 Bytes
```

Instruction	eg.	Comment	Clock cycles
LD r,r	LD A,B	Load single 8 Bit register, with another	4
LD r,n	LD A,10	Load a fixed number	7
LD r,(rr)	LD A,(HL)	Load 8-bit register with contents of pointer	7
LD rr,nn	LD HL,&4000	Set of 16-bit register	10
LD rr,rr	LD HL,DE	Paired register operations	11
LD r,(ir+n)	LD A,(IX+5)	Load register with contents of pointer+offset	19

Figure 2: An example of register use on the ZX Spectrum, and how many clock cycles an operation could take.

The Olivers showing off some of their earliest Codemasters titles, including *Super Robin Hood*.

```
                        ; in A = X (0-255), L = Y (0-191)
                        ; out HL = screen address
   Get_Screen_Address_Oliver_Method:
4      RRA              ;Rotate Right A
4      RRA              ;Rotate Right A
4      RRA              ;X coord has been divded by 8
7      AND #1F          ;Mask possible bits rotated in
7      LD H,high LKUP_TAB ;Load High Byte of lookup table
7      ADD A,(HL)       ;add X(0-31) to Low Screen Address
4      INC H            ;Move Lookup table pointer up 1 page (256)
7      LD H,(HL)        ;Load the high byte of screen address
4      LD L,A           ;Move calculated low byte pointer to L
48 Total Clock Cycles   11 Bytes
```

Figure 4: The improved method to locate the screen memory from a coordinate.

DIZZY SPRITE PRINTING

The Dizzy sprite was 24 pixels wide by 20 pixels high – put another way, he was 3 × 2.5 characters, with each character being 8 by 8 pixels.

As shown in **Figure 6**, the first *Dizzy* game used our general-purpose sprite system. He was printed onto the screen using XOR [swap bits with 1s], so he would be printed once to appear and a second time to make him disappear. (There was no drawing full screens to back buffers in those days!) This worked easily, but when he walked in front of other graphics, he'd appear a little messy, and could be difficult to see. Our solution was to create two versions, with body and hands moving up and down in opposite motion, so he was always moving and this made him much easier to see. Most sprites in our early games worked this way.

The images in **Figure 6** show Dizzy in the open, and then in front of a tree. As you can see, when he's standing in front of background graphics it doesn't work so well.

By the time we created *Fantasy World Dizzy,* we used a new method involving masked sprites. It was slower and more complex code, but gave a much better result, as you can see in **Figure 7**.

Dizzy could clearly be seen in front of a complex background using the advanced masked sprite system. This meant that whilst Dizzy still picked up on the attribute colours of the background – like a chameleon – he could be seen more easily when standing over background graphics. To have moved his colour around with him would have seen a character-aligned white box following him, which would have looked awful.

In this system, we created a mask image, slightly bigger than Dizzy, and copied the 24 × 20 pixels from the screen memory into a temporary buffer. We then printed the mask directly to the screen memory, creating a black Dizzy-shaped hole, and then printed the regular Dizzy sprite into this as before. To remove Dizzy as he moved on, we printed back the temporary buffer to the screen, replacing whatever he'd rubbed out (see **Figure 8**). We didn't need to look up areas of the map, or what objects had been placed there – we simply copied back the rubbed-out screen area. It was very fast and effective.

"We used a new method involving masked sprites"

This worked well, except when two sprites crossed over each other. We enhanced the method by working out what we wanted to print to the screen, combined the screen with the mask using the AND function, then stored that result XOR-ed with the screen. We then wrote the byte onto the screen. This way, to rub out we used XOR to rewrite the buffer back to the screen, so now multiple sprites could go over each other without leaving corruption on the screen.

Each sprite was 24 (3 bytes) × 20 high = 60 bytes × 2 for the mask, so 120 bytes per frame.

There were 39 Dizzy frames of animation, so he took a total of 4680 bytes or ~4.5kB.

Figure 5: *Fantasy World Dizzy*'s market square, which flipped the sprites by adjusting the values in lookup tables.

Figure 6: Fast, easy, XOR sprite printing as used in the original *Dizzy* game.

Figure 7: The improved, masked Dizzy sprite we used in *Fantasy World Dizzy*.

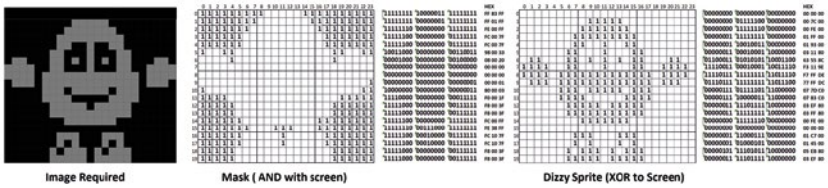
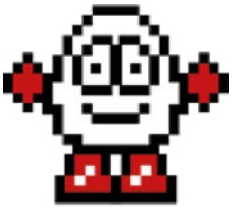


Figure 8: Dizzy with his mask sprite.



This explains why the other characters in the game don't animate or move around!

The code to print sprites had been iterated and iterated to make it very efficient, with great results.

We don't have space in this article for the sprite code listing, but it involved some unconventional tricks to improve speed. We'd disable interrupts, store the stack, and use one set of registers for source sprite data, pulling [POP] double byte register pairs from the stack; then, using the alternate registers, we set this to the screen memory address to write [PUSH] the data back fast. We even used some self-modifying code, to patch the code it was about to execute. Naughty coding, but fast!

"We even used some self-modifying code. Naughty, but fast!"

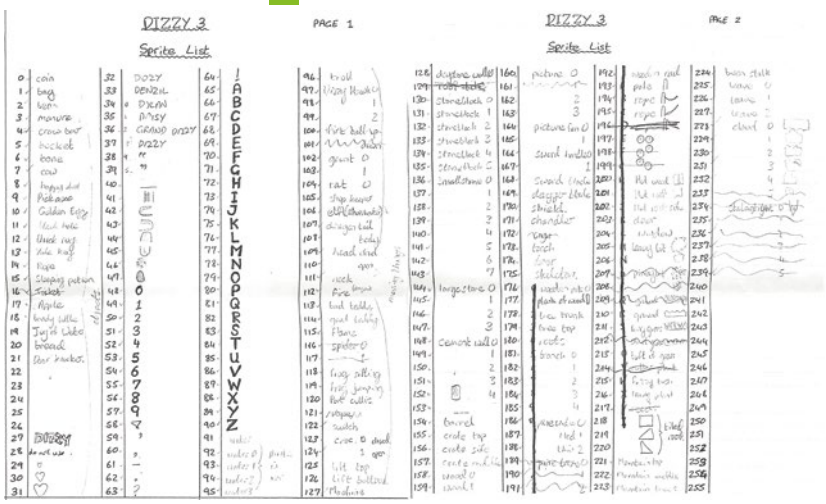
Printing Dizzy was very specific fast code. It couldn't be clipped on the side of the screen and didn't change colour attributes. Our general-purpose, flexible sprite routine could print sprites of any size, anywhere on the screen, with clipping and with colour, and even had the ability to flip sprites left to right.

This involved a lot of rotating bytes to get them on any pixel, which was much slower, but we just used this for printing the location backgrounds and the inanimate sprites that appeared. It didn't matter if it was a little slow as it would only have to do this once as the player entered a new screen or moved an item.

When moving to a new screen, we'd print the whole screen with black foreground and background colour attributes so it would disappear instantly; clear the screen memory and print all the sprites making up the background screen, with a secondary buffer for the colour attributes; then copy them all at the end, so the screen appeared instantly.

Being an 8-bit computer, we gave ourselves 256 sprites so each could be referenced with a single byte. 48 through to 90 were used for the alphanumeric characters, as per ASCII standard. While Dizzy's animations had already been included into this sprite list from previous games, we freed up more space for other graphics once we could remove a lot of his sprites when we introduced the masking system (see **Figure 9**). This final graphic file took just over 7kB.

Figure 9: The original list of sprites used in the game.



BACKGROUND LOCATIONS

We planned out the overall map on a large piece of card (see **Figure 10**), with individual hand-drawn locations (or screens) glued on. If a location needed to be altered, we'd remove the first, draw another, and glue it back in the slot. When planning out the position of puzzles, where to find the items to solve them, and places for coins, we used cut-out pictures and words and placed them on the cardboard on the floor – being careful not to sneeze, slam a door, or open a window. On later *Dizzy* games, we used tracing paper overlays, but that was an advanced technique we'd not yet thought of.

We created each location on the Spectrum screen by having a list of sprites, with the coordinates of where to print them and the colour and attributes to assign to them. Each entry took four bytes: X ,Y, sprite, attribute. Whilst the attribute byte was primarily to assign the colour to the sprite, colour only required three bits, so we could encode other functionality into the spare bits: solid, background, or 'sinkable' for the clouds. This enabled a collision system. With more bits still available, we added the ability to flip sprites left to right, to give more variety, and change the method by which the sprite was printed – either Direct (erasing previous pixels) or XOR (toggling previous printed pixels).

We wanted all the screens to link together, so that they each represented a small corner of a fantasy cartoon world.

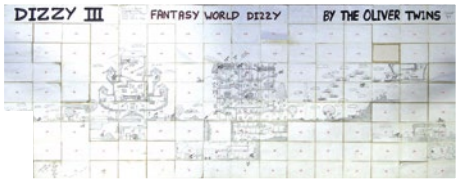


Figure 10: The Olivers' original hand-drawn *Fantasy World Dizzy* map.

Figure 11: Instructions for the editor, which explain how the data was stored for the rooms.

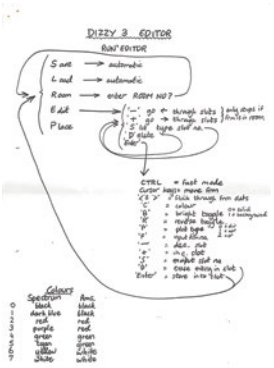


Figure 12: A Spectrum memory map for *Fantasy World Dizzy*.

The locations had 60 sprites each on average, therefore 240 bytes. With 50 locations, this took about 12kB of the RAM. Most games at that time were character mapped, meaning they held one byte per character cell on the screen. That's 768 bytes (32 × 24 characters) if they didn't use compression. Our method enabled *Dizzy* to have much bigger maps than many other games.

We wrote the map editor inside the game. This was useful, because we could run Dizzy around and then press other keys on the keyboard to add or edit the background sprites while the game was running live (**Figure 11**). This made it fun to edit and fast to iterate. There was no recompiling of code when drawing and checking elements of the map and puzzles.

GAMEPLAY CODE

Players took Dizzy through 50 locations, meeting lots of characters, and solving over 20 puzzles along the way. Gameplay code didn't need to focus on speed, but it did need to be clear, easily adaptable, and efficient on the memory it used.

Each location and item was given a name, and there were conversations with characters and additional text explaining parts of the story. These were displayed at the relevant times with a neat conversation panel system. Dialogue was important for creating atmosphere, and driving the story forward, but it didn't come cheap: there were around 250 lines of dialogue, taking about 6kB of memory. Game code was around 4000 lines, taking about 8kB of RAM.

COMPRESSION

Interestingly, we didn't use any compression algorithms in this game. A few tricks would have allowed us to have squeezed in even more – for example, there's a very simple method to reduce text memory use by half.

AUDIO

While the game had great music and sound effects, we certainly can't take any credit for this: we contracted David Whittaker to provide it. He produced the music and sound effects on a cassette with code built in to play the music. The whole file was 4143 bytes, so just over 4kB.

We used the speech routine we'd developed four years earlier to play a recording of one of us saying, "Fantasy World Dizzy." This was fitted into a buffer space as soon as the game ran and deleted as the memory was used. You can see how all the game elements were squeezed into the Spectrum's memory in **Figure 12**.

A CRACKING ADVENTURE

So, that's how we squeezed a pretty large, fun game into just 41kB. We're always shocked by file sizes these days, and how inefficient they are; when we take pictures on a regular smartphone they average 4MB. We could write 100 games in that memory! As for modern games, *Red Dead Redemption 2* is over 100GB on PS4 – that's 2.5 million games' worth of memory! Here's the funny thing – in another 30 years' time, there's a good chance that 100GB for a big game will be considered small. How times change. We enjoyed the challenge and the art of squeezing so much fun out of so little, and remember those times with great fondness and pride.

&F800	to	&FFFF	4 k	Music
&F000	to	&F7FF	0.5 k	Speech Data -> Buffer memory
&D000	to	&E7FF	6 k	Game text
&A800	to	&CFFF	12 k	Map data
&9300	to	&A7FF	5.5 k	Background sprites & font
&8000	to	&92FF	4.5 k	Dizzy sprites
&6000	to	&7FFF	8 k	Code [Around 4000 lines]
&5F00	to	&5FFF	0.5 k	Data/lookup Tables
		RAM for Game	41 k	
&5B00	to	&5CFF	0.5 k	System reserved RAM
&5800	to	&5AFF	0.8 k	Screen attributes
&4000	to	&57FF	6 k	Screen memory
		Total RAM	48 k	
&0000	to	&3FFF	16 k	System ROM (inc BASIC)