









# EDITORIAL

New issue, new feature! Many sceners have been asking for this for a long time: we've finally switched to English so that more people can enjoy 64 NOPs. It's a lot of extra work, but it's in keeping with the international spirit of the demoscene. It also means that there are more potential contributors: Prodatron, rexbeng and Rhino have kindly agreed to take part in this issue.

The stunning cover, a nod to *GTA 3* (had you noticed?), was designed by Beb. This massive picture is the equivalent of 8 fullscreens, i.e. 394×1106 in Mode 0 with 25 colours. It took one month of erratic work to achieve it. Any resemblance to members of the editorial team or CPC references (some clues: *Crazy Cars 2*, *Open Space* and *Chapelle Sixteen*) is purely coincidental.

We would also like to extend our warmest thanks to Beb, who has enabled the editorial team to upgrade its hardware.

As usual, we've done our best to bring you interviews with the leading figures on the CPC scene, as well as making-of demos released this year. And of course, coders, graphic designers and musicians will be able to find their favourite sections.

Enjoy your reading!

THE 64 NOPs GANG

**4**  
**A FOURTH  
BENEDICTION PARTY TALE**

**6**  
**INTERVIEW: REXBENG /  
BITPLANE TECHNOMANTES**

**10**  
**MAKING-OF: GHOST NOP**

**14**  
**OH, MY DOTS!**

**19**  
**TOO CURVY TO TEACH**

**26**  
**INTERVIEW:  
RHINO / BATMAN GROUP**

**29**  
**EARNING  
A STACK WITH THE STACK**

**34**  
**MAKING-OF: J'AI PE-TELECRAN**

**40**  
**A SHARP PIPE  
STARTS WITH YOURSELF**

**45**  
**EVERYBODY LOVES TUNE**

**48**  
**A SIMPLE AND FAST HASH  
ALGORITHM IN Z80 ASSEMBLER**

**51**  
**I SEE YOUR TRUE  
COLORS SHINING THROUGH**

**56**  
**IN BED WITH FDC**





# A FOURTH BENEDICTION PARTY TALE

'Twas a long time ago, but still I remember,  
'Twas twenty-twenty-four, the first of November.

BY HWIKAA/PRALINE

The night before travel, my bags all askew,  
A question persisted: bring boxers or two?  
For hygiene and gaming, a fine line to tread,  
Four boxers were packed, plus a spare one instead.



The morning was crisp as I boarded my flight,  
With dreams of wide pixels and coders in sight.  
From Catalan soil to the grey Paris scene,  
Sid's car was awaiting—what a well-oiled machine!

We gathered, we greeted, with hugs and with cheer,  
AsT and Sid, companions sincere.  
Then AsT said something that really did hit,  
That went a bit like: "He's done with this shit!"

Speechless after such a mysterious phrase,  
I got in the car, ready for the next phase.  
Our journey commenced, to the West we did ride,  
Through highways and byways, with joy as our guide.





A ranch stop for fuel, an Elvis-tuned bite,  
No vegan Happy Meal in McDonald's sight.  
With hunger abated, we sped on our way,  
To Sourdeval-les-Bois, where our destiny lay.

Eliot's domain was a digital trove,  
Where talents and demos all freely did rove.  
I greeted the legends, my heart near to burst,  
From Prodatron to Zik, a fanboy well-versed.



With workstations humming, the hall came alive,  
Pixels and soundscapes began to arrive.  
Golem optimized, I made fire bright,  
Each coder and artist worked deep in the night.

Demos and games in progress shown,  
New friendships forged, old bonds grown.  
City Builder and *SonicGX* impressed,  
While Madram's *Ayane* was met with zest.



Targhan's *Arkos Tracker 3* received big applause,  
Krusty's *Bndbuild* earned a generous cause.  
Compote and rum fueled the late-night forays,  
While many projections set hearts ablaze.

iMPACT ravished us with demos on the Plus:  
*Happy Halloween, We Are*, and *IMPbus*.  
DBT's *Recycled Bug* brought cheer and delight,  
Pulpo Corrosivo's *Borderline* shone bright.

I timidly showed what I had in my sleeve:  
A preview of *Acquevives*, my game to achieve.  
Roudoudou's pizzas brought a culinary flair,  
As laughter and stories filled up the air.



But Sunday approached, with its bittersweet call,  
And farewells were shared in the festival hall.  
The Sidmobile sped me to Paris once more,  
With jazz and warm banter, a friendship to store.

Rain battered my home on Barcelona's shore,  
Yet my heart held the echoes of memories galore.  
With wife and child waiting, my journey was done,  
The coding adventure had been so much fun.

As I drifted to sleep, one last thought did dwell:  
What did AsT mean? That, I couldn't tell.  
A mystery wrapped in a coder's lament,  
An enigma of boxers, on which my mind bent.

Oh you, Benediction, great coding spree,  
A true Victorian ode to the CPC,  
Where passion for the eight-bit assembly and art,  
United the minds and touched every heart. ■







# REXBENG / BITPLANE TECHNOMANTES

“Along with Beb, rexbeng is the biggest graphics revolution on CPC”, SuperSly told me over a chicory drink at the last Benediction Coding Party. I couldn't disagree... So let's hear from the master of the scanline, the boss of abstract pixel art!

BY HICKS/VANITY (IN SEPTEMBER 2024)

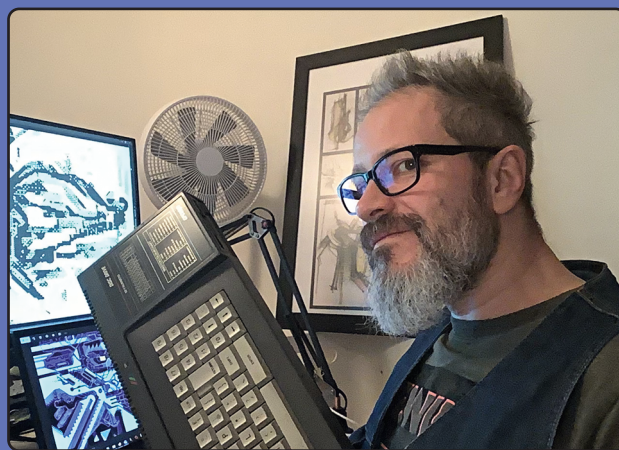
**Hello rexbeng. Can you start by introducing yourself and telling us when you got your CPC? How did you get into graphics?**

Hey, if you are looking for intimate info, you have to take me to dinner first. :P My choice of a CPC was a no-brainer as Amstrad was very well established in Greece; miles ahead in comparison to other 8-bit vendors. Every third kid had one and the big user base meant there would be more copied games to swap! Beyond the playing games part, I have been drawing since forever, so I naturally got interested in drawing using the CPC when I was around 14 years of age if memory serves.

**The Greek scene is still relatively unknown. How did it work at its beginning? Did you have easy access to the latest releases (games, demos, magazines, etc.)? Who were your contacts?**

Actually, for me, there's no such thing as a “Greek scene” these days. Back in the day there was a bit more contact between people, however that was mainly thanks to Catloc's efforts by having people over at his house every now and then. Eventually people grew (in both age and interests) out of that.

With regards to how things started, well, in the beginning of the 90s demos were largely unknown over here; scarcely available and circulated through local software pirates. Therefore “latest releases” wasn't a thing either; one only got to experience whatever little was being available from people who mostly cared about games. The first demo I saw was from Logon System; perhaps it was *The Demo*. Eventually I got interested enough to send out letters to people whose addresses I found in scrollers. To make a short list with the most important ones, I'd



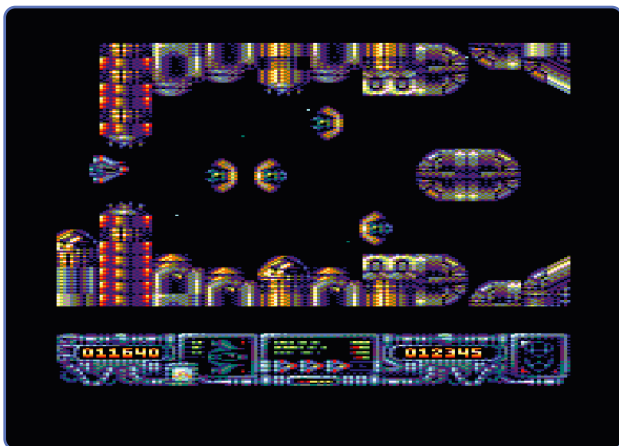
rexbeng



include NWC who will forever be my swapping god because he must have sent me every single demo that ever existed; and Elmsoft, who shared some insights and knowledge with me, and with whom I collaborated on a couple of things (and still owe him some money from back then, teehee). Then DSC, who offered me to join BENG! and in 1993 invited me to the meeting where I met BSC who has been a good friend since then. Finally, of course, Odiesoft! Together we made a piece of CPC history which is luckily well documented so I don't need to write anything about it.

After a few years' break, you came back to CPC with the game *Super Edge Grinder*, coded by Axelay. Since then, you've made a series of successful games: *Relentless*, *Dragon Attack*, *Corsair*, etc. What do you think makes your duo work? How do you work together?

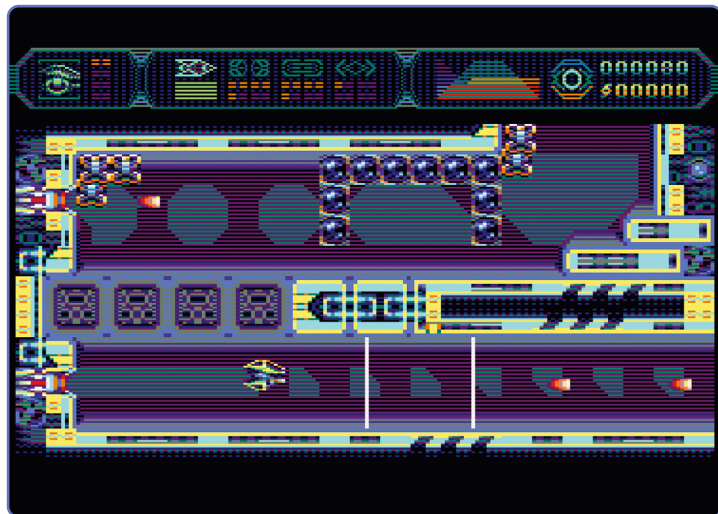
Axelay's methodology for creating games, I'd say. His vision isn't limited to the technical aspect alone. He cares about making games that are interesting, challenging to play, smooth and good to look at; and this was even before we started working together. I find much interest in his ideas, and especially his approach when setting constraints to achieve the best possible balance. The crafting process is very challenging, even a struggle at times, but the result is always satisfying. I enjoy challenges, so I give my best when making the graphics. I could say we have great chemistry and we've come to a point that we understand each other well.



*Super Edge Grinder (Project Argon)*

Your latest game with Axelay, *Hypernoid Zero*, was released at Revision this year. How did the project start? Did you have any technical constraints on the graphics?

You may read Axelay's interview at Retromaniac<sup>1</sup> magazine as he gives insight about how *HZ* came to be and I see no point in repeating that. With regards to technical constraints; of course there are! It wouldn't be interesting otherwise. Graphics are tailored to exploit CPC capabilities without being too expensive or consume resources planned for other areas. For example the bitmap data for level scenery graphics "weights" approximately 256 characters, or screen area of 64×128 Mode 0 pixels if you prefer.



*Hypernoid Zero (Bitplane Technomantes)*

What projects are you currently working on? Your shoot'em ups are magnificent, but are you planning to produce other types of games?

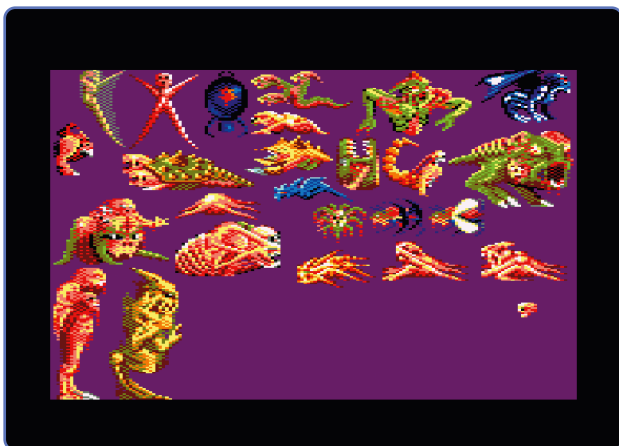
We experiment with various ideas and we have produced draft code and draft graphics for a few of them. We generally don't make plans, but rather end up deciding to go ahead with certain ideas as they evolve during our lengthy internal talks.

On the demomaking side, you mostly collaborated with Dirty Minds in your early days (Antitec), then with Pulpo Corrosivo (toms) more recently. Is your approach the same for a game as for a demo? What do you think makes a good demo?

A game and a demo are entirely different things, with different aims and approaches, then ultimately, goals. Have you ever seen a demogroup making teaser posts / videos of their upcoming demos? With regards to what makes a "good" demo. I believe code must always be first priority, but the coder should have the willpower and the patience to wrap good graphics and some design around it. I believe the CPC coders are mostly focused on technical achievement with very few exceptions. In an era of multiformat parties / compos, I find that platform specific challenges have not much meaning.

<sup>1</sup> <https://www.retromaniacmagazine.com/2024/06/yo-queria-hacer-desde-el-principio-algo.html>





*Rigor Mortis*

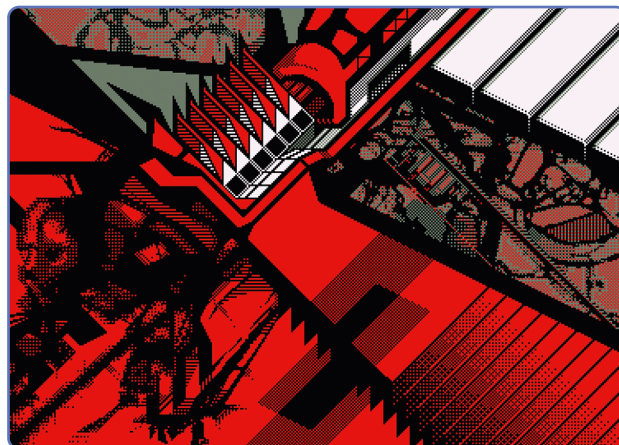
**Your style evolved considerably since you started on CPC. What are the influences (on all platforms) that led you to this very distinctive style that is immediately recognized as “rexheng”? For example, your graphics have a pronounced abstract dimension, which is quite rare.**

It might come as a surprise but during the 90s and early 00s I didn't care about the broader demoscene and hadn't been watching demos on other platforms. So, when I got interested in drawing with the CPC there were no influences other than from games on the CPC, then later games on bigger systems, and of course demos on CPC. Also, for some reason I never got interested about pixeling on the Amiga, which I only played games on. Pixeling on the CPC was fascinating at first, but I was soon hitting walls with the *Advanced Art Studio*. Making full screen images (or even more so overscan images which couldn't be previewed properly), was way more laborious than fun. To make my life easier I used to draw on my screen with a non-permanent marker and then trace the drawing using the “continuous lines” tool to create individual figures / elements that I then tried to combine into compositions. This method became too limited after a while and this is evident in my only released slideshow where elements in some images appear as “glued” one on top of the other rather than being parts of a coherent image. Generally, I was feeling more comfortable with making smaller graphics (“sprites”) because the tool limitations didn't feel as heavy as with big images. Small objects were easier to repeat & modify within the same screen area which resulted in better compositions. That's the reason I felt much more creative when making *Megablasters* and even more so when working on *Rigor Mortis*. I feel the experimentation in both the usage of pixels and the colours is evident in the latter. *Rigor Mortis* was my true first step of departure from the “traditional” way of pixeling. But with that project frozen, I stopped pixeling on the CPC in the mid 90s and came back

effectively in the late 00s. That's about 15 years. The scarce releases that occurred in the time between were mostly recycling of older graphics and not signs of real activity. I'd say there was some point in time when, with the use of proper tools on PC (hurray for layers and unlimited saves!) and through a lot of experimenting, I came to figuring out how to create pixel images that did actually please myself and did satisfy my fixations. At heart I am an abstract expressionist, therefore my relation to actual painting is rather physical, instinctive and experimental. So, up until that certain point I wasn't seeing any reason to bring that practice over to a tiny computer screen where there has never been much going on beyond the traditional stylized, figurative images. Yet, I managed it. I'm still developing my methods and have fun while at it.

**Can you tell us how you go about creating graphics? Beyond technique, what makes an image have an impact in your eyes?**

As an abstract expressionist I rarely have a preconception of what I want to do, so one could even argue I'm not really creating graphics. Most of the time I draw relentlessly until I see things that “speak themselves out”. So, ultimately, even if a drawing ends up being stylized, it never started with that as the goal. There's things I've abandoned because they didn't seem to go anywhere, and there's abandoned things I've picked up after years and managed to bring to some conclusion (say, the compo entry in last year's Benediction Coding Party).



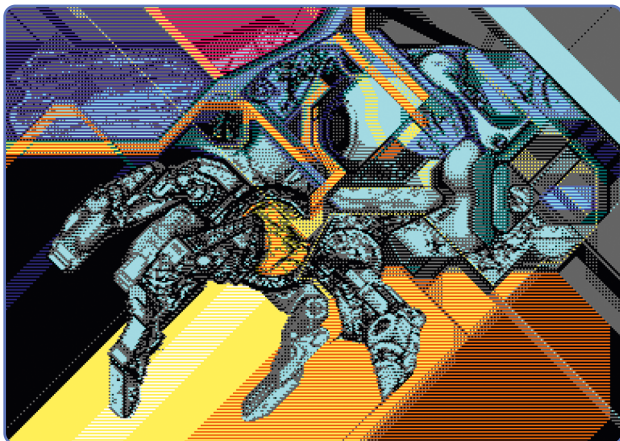
*Enter Dragon*, released at Benediction Coding Party #3

**There's regular controversy about plagiarism and the use of AI to create an image. What's your opinion on these two phenomena?**

Strictly speaking about the pixel graphics side of the demoscene, having participated in relevant discussions has left me with mixed feelings. First of all plagiarism is one and the same. I don't see a need to



diferenciate the two; when converting existing images to pixel images, the crafting process is exactly the same for images sourced from the internet drawn by other people and for images generated with AI. Depending on the abilities of the scener making the conversion, it may end as well crafted or as not well crafted. Then, I generally don't mind that conversions exist in the demoscene as they do serve the need for having high volumes of images when used in demos. But I am getting a bit cringey with all the people who are not honest about the origins of pictures they publish. Of course this thing isn't new; has been happening in the demoscene since the 90s and I've come to realize that many of the people who created pixel images this way back then (and even participated in compos) still do the same today (and even participate in compos). Ultimately, if I had to make a conclusion, I gather that, when looking at an image, people in the demoscene are mostly interested in the "craft" behind the handing of pixels (which is indeed a thing on its own) rather than the "art" in the creative process of drawing something from scratch. I guess the demoscene is indeed about the art of coding with everything else having a secondary, supplemental role. Which, in the end, is totally fine.



*Chronovoros, released at Syntax 2024 (using EGX1)*

**Have you ever experimented with advanced graphics modes (EGX, flipping, R mode, etc.)? Your use of tiles in demos to create images (*Octopus Pocus*, *Ghost NOP*) is very unusual. Does this come from your experience in games?**

The control panel in *Hypernoid Zero* is in EGX, so, yeah I have. I don't really like flipping and to be honest the difficulty involved means that most likely flipping images are going to be conversions, so of no much interest to me. I have used split rasters in a few published instances and in a few unpublished ones. I was intrigued about creating images using standardized elements and restrictions when I became curious about the C64.

**You've produced graphics for both the CPC and Commodore 64. Is your approach the same for both platforms? Do you find the challenge greater on one than the other?**

Both machines have their bright and dark sides and I go back and forth bringing things from one platform to the other. But the approach is more or less the same, so the "challenges" are mostly mental. On the one hand the C64 is the more interesting platform for me when it comes to pixeling and sharing images. There's many people creating original things and even more so, people who develop styles, come up with interesting designs and approaches that go beyond the typical "nicely done figurative stuff" that the demoscene is bloated with. On the other hand the CPC is littered with all those "can't do that" tags, so making "impossible" games for it feels like the greatest turn-on in the world.

**I understand you graduated from an art school. Has this background changed your approach to pixel art? On the other hand, has your pixel art practice influenced your approach to art in general?**

I am not sure how to answer those questions as I'm wondering if you are expecting to read something practical, as for example, "my knowledge of Architecture is the reason behind how I'm laying the graphics for games I make". Surely, having been taught things related to colour and form and so on plays a significant part, however that applies to everything I do out of pure personal interest, not just to pixeling. I always liked aggressive strokes, contradictions, thick lines and combining different materials; those are evident to my work regardless of the medium I am using. Then, every medium comes with its own distinctive characteristics and I find much interest in carrying methods, practices and approaches between different mediums. Naturally, the characteristics of my "pixeling on obsolete platforms" have found their way into that big pool of materials; for example, I love toying with the CPC's aggressive palette even outside the CPC realm. To sum-up my background, I have studied Fine Art, Applied Arts and have been taught Art History and Architecture. My profession is in the broader Graphics Design sector.

**Thank you for your answers, and congratulations once again on your work. Any final words?**

Thank you. I am only looking forward to the moment when my 7 years old son will start getting it and stop asking "what exactly is it that you are drawing". Up to this point my answer has been "frankly, I'm not sure".



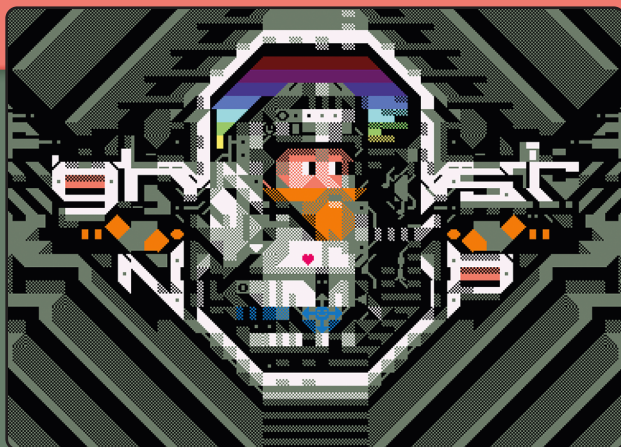




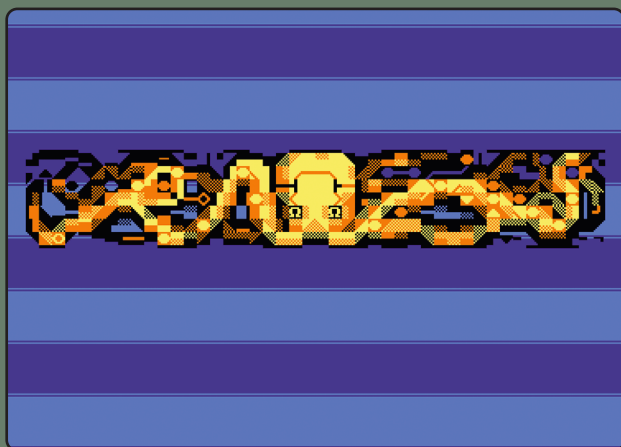
# GHOST NOP

It had been around 3 years since a major demo had been released for the CPC. Far too long. Being one of the main demoscene events of the year, the Revision party was the ideal place to rectify this anomaly!

BY TOMS/PULPO CORROSIVO



*Ghost NOP*



*Opening sequence*

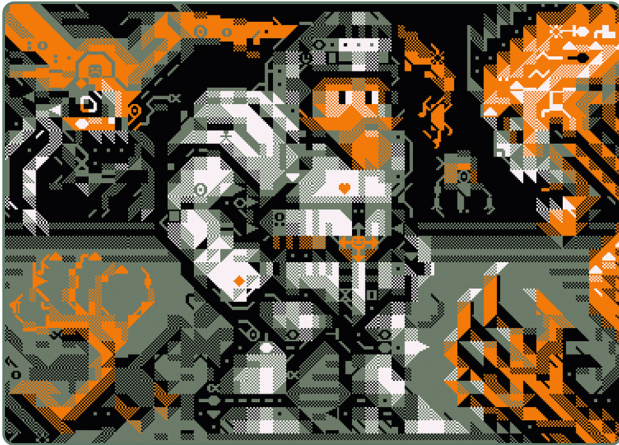
*Ghost NOP* is a demo published by Pulpo Corrosivo and Futurs' at Revision 2024 which had the honour of winning the prestigious Oldskool Demo competition (a first for an Amstrad CPC demo!). The whole stuff runs full screen at 50 Hz on any standard CPC 6128 with a CRTC 0 or 1. No extension is required.

## THE THING THAT SHOULD NOT BE

The story of the demo unofficially began in September 2023 when I coded the infinite zoom effect which was originally planned for another project that was eventually abandoned. At the same time, I was aware that OffseT had a stunning effect in his drawers. To prevent these 2 effects from being lost, I suggested him that we could make a demo together and submit it at Revision. Luckily, he accepted! I then contacted Hwikaa and rexbeng for the visuals, and Zik for the music. The team was complete and the project was officially under way. There were only a few months left to make the demo. Don't worry, everything's going to be fine.

We quickly decided that OffseT's effect should represent an iconic CPC game character. As Hwikaa had been working on the Amstrad Plus version of *Ghosts'n Goblins* for several years (see the





Fullscreen picture displayed while loading part 2

interview with Golem13 in the previous issue), the choice of the game's hero (named Arthur) was an obvious one. This oriented the visual direction of the whole demo and suggested the title: *Ghost NOP*. In reference to the game of course, but also to the tight machine time for some effects (we had to find some ghost NOPs to make them possible). Music lovers will also find a reference to ghost notes, a nod to Zik's funky music. Yes, I like when titles have multiple meanings.

As for the music, I asked Zik for something funky and I gave him an approximation of the duration of each effect. We had a lot of discussions to adjust the synchronisation between the visuals and the music.

#### LOADING SYSTEM

To make life easier, we decided not to make a single-load demo. As the voxel sprite effect was the most memory-intensive, we decided that it would be loaded from disk at the end of the demo. This made it easy to switch between the fullscreen picture and the effect (see later for more details).

OffseT suggested using the Amsdos loading system which, of course, has a few drawbacks, such as being slow and not being able to play music while a file is loading. However, as this file is not very large (23 kb), the loading time is not that long. By using a trick on the hard envelopes, the music doesn't cut out and the loading goes almost unnoticed!

As you can imagine, the first part of the demo wipes out the firmware, rendering the Amsdos unable to load anything when it should. So, at the start of the demo, we have to save a few areas of it (it's not necessary to save it completely) which we'll restore just before loading the second part. That's 2703 bytes.

To save loading time, the file is opened when we start the demo, then we switch off the drive motor. It is restarted just before the image is displayed and the actual loading takes place.

One last important thing: we need to check whether the demo is launched from an Albireo or an M4 by reading the byte in &BE5F. If so, we'll need to add a small delay to get the same loading time as from a disk. Otherwise the image won't be displayed long enough.

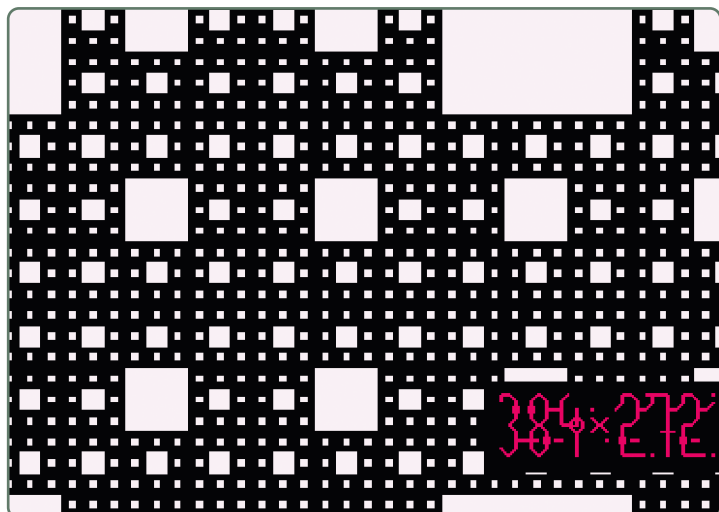
#### ASCII GRAPHICS

To save memory, rexbeng and I decided that all logos and images would be made from ASCII characters available in ROM. So there's no need to store them in memory, unlike custom characters like we did in *Octopus Pocus*. All we need to store for a picture (or logo) is the character index and its colours (pen and paper), which makes graphics much lighter than bitmap ones!

As the characters are stored in Mode 2, I had to create a routine to display them in Mode 1 with the right colours (1 byte for the character index and 1 byte for the pen / paper combination). Not as easy as it sounds! When you see the very stylised result, you still wonder why there are so few ASCII graphics on the CPC...

#### SIERPINSKI CARPET'S INFINITE ZOOM

As I've already written, I started by coding the infinite zoom effect. My routine is partly based on some of the principles used in *Checkmate* (see the making-of in *64 NOPs #2*), namely storing in memory the different lines making up the Sierpinski carpet for each zoom step. However, this time the lines shifted by 1 pixel (or more) are not stored in memory as this would be far too greedy (in Mode 1, there are 4 shifts per byte). The shift is made when the lines to be displayed are generated on the fly.



Sierpinski carpet's infinite zoom



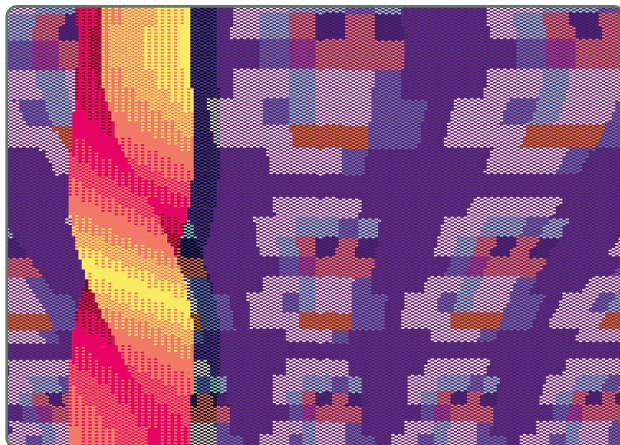
As I've chosen to display the carpet with 4 iterations, we have 16 different lines 128 bytes wide to store for each zoom step. You may have noticed that, when zooming in on the carpet, the animation loops at a certain point, when the smaller squares reach the size of the larger ones. So we only need to store this part of the animation. To get a smooth one, I've chosen to store 32 steps (less is too fast). Er, wait... storing 32 steps of 16 lines 128 bytes wide takes 64 kb! That's way too much!

What would happen if the smallest squares were always displayed but their colour changed between black and white? By doing this, we only need to store 8 lines and the data will occupy 32 kb. That's not too bad. We can easily manage this by addressing the CRTC or the Gate Array depending on whether we want to display another line or change the colour of the smallest squares.

As for the "kaleidoscope" effect on the carpet, I just display the lines by mirroring them. The "384x272" logo is displayed on top of the effect thanks to vertical splitting, and the equalizers are achieved by adjusting register 1 of the CRTC and changing the border colours.

#### WAVE EFFECT

I really liked this effect when I saw it in *Unique's Coco* on the Amiga. This kind of thing can be achieved on the CPC by mixing two well-known techniques: wobbler and magic rasters.



*Run Arthur, run!*

To start with, I had to create a line with a repeated pattern of 9 inks in a row in such a way that if you repeat this line vertically, you get columns with these inks. I then stored this line in memory at different zoom steps so that I could display a classic wobbler effect by modifying the offset along a curve.

The knight sprite is displayed using magic rasters, in other words, you simply set the wobbler inks to build

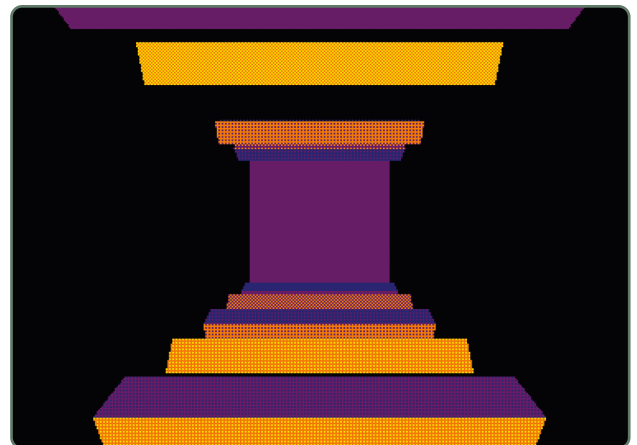
the sprite. Eliot wrote an excellent article on this technique in the previous issue of *64 NOPS*, you should definitely read it! Of course, changing 9 inks can't be done instantly and some visual glitches can occur as some inks are set in the middle of the line. But with everything moving on the screen, it's almost imperceptible.

The twisting bar (and its shadow) is created by drawing its rotation steps on the wobbler lines stored in memory. The drawback is that the animation of the twist is linked to the wobbler curve. But have you noticed?

Finally, horizontal scrolling is done simply by rotating the inks of the sprite. Thanks to Beb for suggesting this (he also sent me some palettes that were not included in the end). A version with a slower scrolling exists but it means a smaller sprite.

#### X-ROTATOR

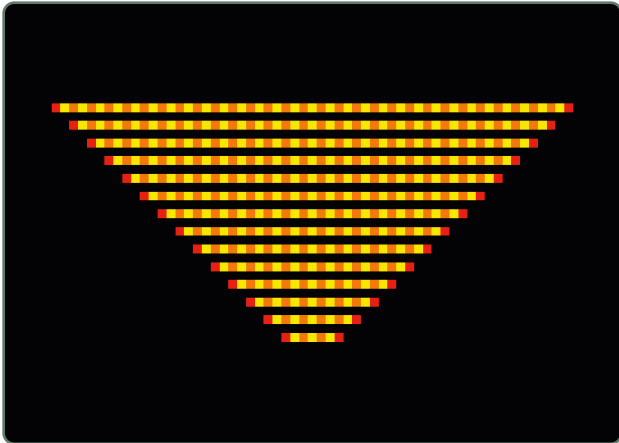
X-rotator is a common effect on various platforms but rarely done on CPC while it lends itself rather well to this thanks to line splitting. The few existing ones are only simple objects rotating on the screen. For this demo, I wanted to do something more complex and the *Ghost'n Goblins* theme gave us the idea of exploring the trapped corridors of a castle. Hwikaa gave me precious advice when I was designing the various scenes. He also suggested the nice colours.



*Watch out for the traps!*

The effect being managed by using line splitting, the available VRAM is quite limited (a total of 84 lines). It works by storing lines with different width in memory, one being 2 pixels wider than the previous one. Unfortunately, as we work in Mode 1, we don't have enough lines to display something close and something else far at the same time. It was a limitation in *From Scratch* or *Debris*. Note that there would have been no problem if we had worked in Mode 0.





Additional pixels (in red on the image) are set in orange or black, depending on the width required.

As I wanted to display a corridor with traps, I needed to have more lines available to have more depth. In order to do this, I put 2 additional pixels in a free ink (1 on the left and 1 on the right) on each of the 84 lines (now one is 4 pixels wider than the previous one). By switching on or off these 2 additional pixels by changing the ink's colour, you can access to 2 different widths with the same line. As a result, we now have 168 lines! Good enough for what I wanted to do!

```
ld a,(hl)
inc l
out (&ff),a ; set ink 0 colour

ld a,(hl)
out (c),h ; select ink 1
out (&ff),a ; set colour

jr nc,$+3 ; Carry = 0 if width is odd
ld a,b ; b = &74 = black
out (c),c ; select ink 2
out (&ff),a ; a = &74 or ink 1 colour
```

The effect manages lighting by using dithering, so 2 colours need to be updated per scanline. Taking into account a third one to adjust the line width as explained before, 3 inks have to be set per scanline, which is quite demanding in machine time. The main drawback here is that only one ink is left for the background. To save a few NOPs, OUT (&FF),A is used to set the colours. See the article *These colours don't run* in the first issue of 64 NOPS to find out how this works.

The different scenes are based on 5 animation loops, each 120 frames (168 for the downhill one). The number of frames has partly been chosen so that the doors open in synchronization with the drums.

For each prerendered frame, 5 bytes are stored per visible face: its height, its hue, the width of its first line, and the slope between its first and last lines (on 16 bits). Over the whole height, I add the slope data to the first line's width at each scanline, which gives a new width that lets me know which line to display. The additional pixels we talked about before are enabled if the line width is odd, otherwise they are disabled (by testing the Carry flag which was updated earlier with RRCA). The face colours are selected from a table (pointed by HL in the nearby source) using the hue data.

#### FULLSCREEN PICTURE DURING LOADING

The picture had to be located in the &8000-&FFFF area because of some constraints on the voxel sprite effect. As this is exactly the area where the firmware was to be restored and we wanted the picture to remain displayed during loading, I had to use the Gate Array's &7FC2 configuration (this means linearising the banks). In this way, the firmware was restored in the banks while the CRTC still addresses the main memory to display the picture.

#### VOXEL SPRITE

As I'm not the coder of this effect, I won't go into much details. All the stuff relies on a classic line splitting which addresses only 2 lines. While you display one repeated 8 times, you have some time to build the next one by decrunching data and calculating the translation, wave and twist (using pre-computed index tables). Then you display it while working on the other, and so on. Not forgetting to manage the rasters in the background, of course!

Obviously, all these effects deserve to be described in more detail, but the article would be far too long! Last but not least, this demo puts into practice a number of topics covered in previous issues of 64 NOPS. Conclusion: subscribe now! ■



Voxel sprite coded by OffseT and designed by Hwika

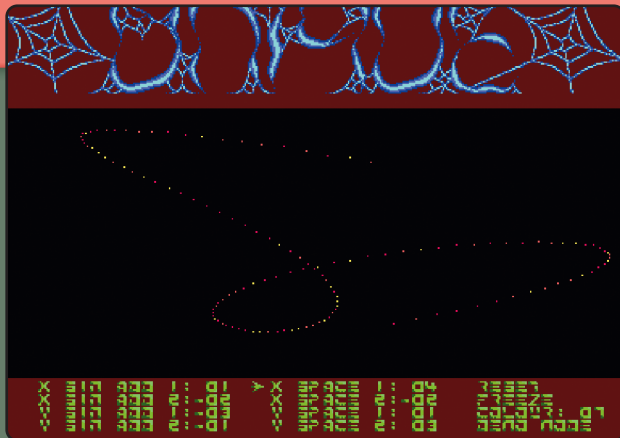




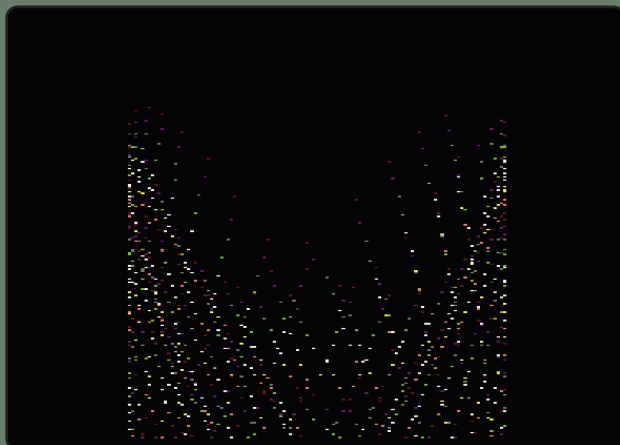
# OH, MY DOTS!

In this issue, I suggest to look at an effect that, I assure you, still has a lot of potential on our CPC screens: “dots”, sinus or not, I'm not a sectarian! Indeed, when I was looking for a topic for this article, I noticed that dots effects are not very common, often with a modest number of dots and a design reduced to the simplest form!

BY ELIOT/BENEDICTION AND MADRAM



*The Plot, Unique Megademo, 128 dots (NWC)*



*Réchauffé (Madram)*

Surprising, since in issue 45 of *Amstrad Cent Pour Cent* (October 1992), Pict from the Logon System group had paved the way with an optimised routine displaying 320 dots. To get in the mood, run NWC's *The Plot* in *Unique MD*, Targhan's *Gates To Delirium* in *DemolzArt*, Overlanders' *MiniMeeting 1*, Overlanders' famous *Tire Au Flan* and its 256-byte sequel *Réchauffé* by Madram, the huge *Batman Forever* or Grim's minimalist *Sintimus* in 256 bytes!

Our first objective will be to get a classic routine of dots whose X and Y coordinates follow a sine curve, and then we'll move on to more visually advanced things... As this article is aimed more at beginners, we have to admit that it's not an easy task, as there are a lot of concepts to tackle! Our tools: some BASIC programs to generate data, Z80 assembler, a brain (or what's left of it since the last issue...) and a coffee machine to accompany those long, pleasant nights of democoding!

## CHECKPOINT LIST

First of all, let's make life as easy as possible for ourselves like in Pict's routine and optimise the code using 3 tricks that have become (almost) commonplace:



- a 16 KB screen page with a format of 64 bytes wide (register 1 of the CRTC to 32) by 256 lines (register 6 of the CRTC to 32), which allows simplified calculation of operations on screen addresses: each line start is then a multiple of 64 (&40) and a single addition to the less significant byte of the screen address will suffice to position us on one of the 64 bytes of the line! It's a trick used in many games...

```
Lines 0 to 7:  &4000, &4800, &5000, &5800, &6000, &6800, &7000, &7800
Lines 8 to 15: &4040, &4840, &5040, &5840, &6040, &6840, &7040, &7840
Lines 16 to 23: &4080, &4880, &5080, &5880, &6080, &6880, &7080, &7880
Lines 24 to 31: &40C0, &48C0, &50C0, &58C0, &60C0, &68C0, &70C0, &78C0
Lines 32 to 39: &4100, &4900, &5100, &5900, &6100, &6900, &7100, &7900...
```

- page flipping (alternating between 2 screen pages to avoid screen scanning) is made easier by using the Gate Array's &7FC3 configuration, which switches the &C000-&FFFF page to &4000-&7FFF in central RAM (see the Gate Array connection of banks on *Quasar CPC* or *Grimware*). This way our display addresses will always be between &4000 and &7FFF, so there's no need to manage 2 different addresses tables or make an addition to each dot to switch between 2 screen pages.
- a clever choice of ink for the dots. If you look at the coding of pixels in a byte<sup>1</sup>, in Mode 0 or Mode 1, you will see that a pixel in colour 15 or 3 has all its bits set to 1, which makes life easier during logical operations. Our pixels will therefore be in colour 3 for Mode 1 or colour 15 for Mode 0.

### NO MORE RAMBLING!

It's time to get on with it! To enable our dots to move to ANY point on the screen, we need to create the X table containing ALL the X positions (from 0 to 63 bytes), the Y table for the screen addresses at the start of the 256 lines, and of course the sine curves that will allow us to select from the X and Y tables! For a subject such as dots, it is obvious that the precision of our calculations and our display must be that of the pixel.

To move a pixel in Mode 1, there are 4 steps in a single byte. In Mode 0 there are only 2 steps! These graphic steps are coded with the X table.

Let's start with Mode 1, 4 pixels per byte (see the table beside):

X position of the dot (in pixels)	X Position (in bytes) TABLEX.BIN	On-screen drawing of the byte (X means the dot pixel)	Byte to be displayed (Mode 1) TABLEGFX.BIN
0	0	X000	&88 = %10001000
1	0	0X00	&44 = %01000100
2	0	00X0	&22 = %00100010
3	0	000X	&11 = %00010001
4	1	X000	&88 = %10001000
...	...	...	...
254	63	00X0	&22 = %00100010
255	63	000X	&11 = %00010001

We also need to generate sine tables, 2 to start with in our first routine.

With these sine curves (2SINUS.BIN) and tables of 256 X coordinates (TABLEX.BIN), X bytes (TABLEGFX.BIN) and 256 Y coordinates (TABLEY.BIN) generated by 2 very simple Basic programs, we can calculate the position of the dot and then display the corresponding byte. But not directly,

because we have to consider the case where the dot is displayed on a byte (4 pixels possible) that already contains one or more other dots!

This is where selecting

ink 3 in Mode 1 (or 15 in Mode 0) comes in handy, as a simple OR between the background and the byte to be displayed will return the resulting byte.

Once the byte is displayed, we store its screen address in a TAB\_STOCK\_ADDRESS buffer with an economical PUSH (4 NOPs only to save 2 bytes!), so that we can clear it later before redrawing in that screen! Since we're dependent on page flipping, we'll alternate between 2 buffers.

For each VBL, we move along the 2 X and Y curves by auto-modifying MOVE\_SINX and MOVE\_SINY (in our example with a simple INC L) and then for each dot, STEP\_SINX and STEP\_SINY are the steps along their respective curves. There's a lot of fun to be had in varying these parameters!

So here, in less than 40 bytes (!!!), is our main loop, which is deliberately very generic and can be greatly improved!

The A, HL, DE and BC registers and the stack are all we need. The secondary registers (EXX) and the more time-consuming IX and IY are not (yet) part of the game! Everything is in dots\_routine1.asm<sup>2</sup>. In terms of visuals, we have pretty much the same result as Pict, so now we're going to have some fun!

<sup>1</sup> <https://cpctech.cpcwiki.de/docs/graphics.html>

<sup>2</sup> Download all the sources of this article at: <https://64nops.memoryfull.net/files/issue-3/oh-my-dots/>



```

    ld b, TABLEY/256 ; most significant byte of Y addresses table
    ld c, NB_DOTS      ; number of dots

MAIN_LOOP_DOTS
; point to Y sine table
POSY ld hl, SINUS2
    ld a, STEP_SINY
; step forward X values in the sine table
    add l
; and this position is saved for the NEXT DOT
    ld (POSY+1), a
; get the Y line corresponding to the sine in L
    ld l, (hl)
    ld h, b ; B holds the most significant byte of the Y addresses table
; get the screen address of the start of the line in DE
    ld e, (hl)
    inc h
    ld d, (hl)
; point to the X sine table
POSX ld hl, SINUS1
    ld a, STEP_SINX
; step forward X values in the sine table
    add l
; and this position is saved for the NEXT DOT
    ld (POSX+1), a
; and the X position in pixels corresponding to the sine is copied to L
    ld l, (hl)
    ld h, TABLEX_H
; get the X address (in bytes from 0 to 63)
    ld a, (hl)
; and add it to the less significant byte of the screen address (which is in E)
    add e
    ld e, a ; DE = DOT screen address
; H points to the table of bytes to be displayed
    inc h
; read the byte from the screen
    ld a, (de)
; interfere with the byte to be displayed
    or (hl)
; and display the result on the screen!!!
    ld (de), a
; don't forget to save the screen address in the buffer pointed to by SP
    push de
; move on to the next DOT
    dec c
    jr nz, MAIN_LOOP_DOTS

```

### MOVE YOUR SINUS!

We were satisfied with one curve for each axis. Adding 2 curves for one of the axes - or even better, for both - gives much more dynamic and original movements. This is where our real precision of 256 points by 256 lines in our tables comes into play. With a single sine curve, it is possible to optimise by storing only the X or Y coordinates relating to that curve. When adding 2 sine curves, the result of the addition is not known in advance, so all 256 possible positions, in X or Y, must be calculated. Let's use registers IX

and IY to manage the additional curves. The instructions LD IX, nnnn, ADD A, (IX+0), INC IXL are a little more demanding in terms of machine time than their equivalent in conventional registers, but you save a lot by not having to save IX or IY during the main loop, and there is still room for optimisation... Have a look at the source dots\_routine2.asm, the changes are relatively minor but the effect really improves its visual impact! It's up to you to calculate some original curves and vary the parameters, 8 in our example, between the curves and the steps!



```

MOVE_SINY  ld hl,SINUS1
           inc l
           ld (MOVE_SINY+1),hl
           ld (POSY+1),hl

MOVE_SINY2 ld IY,SINUS2
           ld a,IYL
STEP_MAIN_SINY2 add 255
           ld (MOVE_SINY2+2),a

```

### DON'T TOUCH THE (BACK)GROUND!

I'd like to add some depth to our effect. At the moment our sinus dots are wandering around on a monochrome background, which we restore by displaying an empty byte at the screen address stored for each dot. If we have a background image, we can also save the byte affected by the dot in just a few Z80 instructions. Don't laugh, this has only been done very rarely...



*dots\_routine3.asm, 256 dots*

Saving:

```

; DE holds the screen address
; read the byte from the screen
ld a,(de)
; save the screen address DE and byte A
push de
push af

```

Restoring:

```

RESTORE_BACKGROUND
repeat NB_DOTS
pop af
pop hl
ld (hl),a
rend

```

The buffer size is now doubled. PUSH AF and POP AF are used to save a single byte. What a waste! It would be a good idea to reorganise the main loop to save 2 bytes at once, for example by managing 2 dots per iteration...

### CHEAP SCROLLING

```

ld a,(hl) ; read a first sine value
add (IY+0) ; add the second Y sine to it
dec IYL

```

```

SCROLL_DELAY add 0 ; add the scroll offset

```

```

ld l,a ; L holds the Y line corresponding
; to the calculated sine

```

With just 1 instruction and 2 NOPs for each point, it will be possible to manage a vertical hardware scrolling! All you have to do is add an offset to the value obtained by adding our 2 Y sine curves. Nothing complicated really, but once again it changes the visual impact of the effect! SCROLL\_DELAY can be varied according to the desired scrolling. You can use CRTC register 5 to achieve line precision. A simple example with an 8 lines scrolling can be found in *dots\_routine4.asm* which, in 78 NOPs and about 50 bytes, manages to display and clear a dot set up with 4 curves on a vertically scrolling background. 78 NOPs? Can you hear the sarcastic laughter echoing through Semilecaenta's cave?

### NON-FINAL DOT

This is a good basis for more advanced effects, on a larger screen and with options not mentioned here: interaction with other dots or the background using the shade-bobs technique, dots that leave a trace by storing multiple buffers for saving addresses and the background, use of colour cycling, display of dots larger than a single pixel, etc. With a few tricks, *Tire Au Flan* is within reach of your code! Optimising a dots routine is an interesting task, the very essence of demomaking: with each NOP gained, the number of dots displayed increases rapidly! What's more, it's in keeping with the times: dots lend themselves well to size coding! While we're on the subject of optimisation, I'll leave you to enjoy the thoughts of the experienced Madram. Enjoy it!



## CLAUDE POINT'S TIPS

### DIVIDE AND CONQUER

The principle is as simple as can be. Separate the screen address as follows:

- The most significant byte defines the Y position.
- The less significant byte defines the X position, independently.

For a simple lissajous (without adding tables), the routine (provided by Roudoudou, revised by Promana) looks like this:

```
; DE = Y curve followed by X curve in +&100
ld h,convtables/&100 ; Conversion y -> Y offset
ld a,(de) : ld l,a    ; L = doty
ld b,(hl)      ; B = Y offset
inc d
ld a,(de) : ld l,a    ; L = dotx
dec d : inc e      ; See note below
inc h           ; Conversion x -> X offset
                ; (x/4 in Mode 1)
ld c,(hl)      ; BC = full offset
inc h          ; Conversion x -> mask
ld a,(hl)      ; A = pixel as it should be
```

No need to do any calculations - it's all good.

A general tip for this type of code is to process the odd dots in the opposite direction (X then Y) to save on register updates. This technique is known as “zigzag”, “pingpong” or “gloplop”, depending on the region.

How do you set up such an easiness? The last issue of your favourite publication after “Grosses Cylindrées” suggested a solution: RVI for linear addressing. Far too greedy for my taste, I put the idea back to the cellar to refine it like a good vegan Roquefort. Don't be surprised if I come up with something mouldy in 2 years.

For now, let me suggest a more subtle technique. Split by character lines to get the following sequence of offsets: C000, C800, ..., C100, C900, ..., C200, ... then you have to use other pages, it spreads out quickly. On average, it only requires 1.25 OUT for every 8 raster lines, when reordering addresses a little more cleverly.

If you're not completely stunned by 2 hours on the phone, you'll have noticed that these most significant bytes are not linear. No problem! If a single table is used for the Y positions, it will be converted beforehand.

### HARD FITS BETTER

The important thing is to shift the Y curve with respect to the X curve. Hence we're going to hard code the positions of the latter! The code now looks like this:

```
; DE = Y curve
ld a,(de) : ld h,a ; H = Y offset
inc e
ld l,n      ; Hard-coded X offset
set n,(hl) ; Hard-coded pixel displaying
```

We're down to 9 NOPs per displayed dot, and it becomes easier to handle curves with more than 256 values. Optionally, shifting of the X curve can be done using a sliding window on the generated code (update the entry jump and move the RET one dot later). Or you can simply replace the positions incrementally with a completely different curve.

### MOVED BY SUCH INGENUITY

Clearing the displayed dots usually requires you to save the addresses you have scanned. With the previous trick, you only need to save the Y positions, as the X ones will be hard-coded in the erase routine. It is quicker not to save them at all and to read them directly from the table. You can get even better results by deleting the old dots while displaying the new ones. Consider the following typical scenario:

```
Frame 0: X0Y0 X1Y1 X2Y2...
Frame 1: X0Y1 X1Y2 X2Y3...
```

So, in frame 1:

```
Erase X0Y0
Display X0Y1
Delete X1Y1
Display X1Y2
...
```

X0 can be reused, then Y1, and so on. Of course, this is especially useful when X and Y are separated by the first trick. There remains one big problem, which has at least two solutions. It is left as an exercise for the reader.

### DELTA FOR ALPHA MALE

Again, based on Y / X separation: incremental update of the X offset (INC E / DEC E / NOP if you want to remain stable in machine time).

### MISCELLANEOUS

Use the same X twice in a row. Or even Y, staggered. Especially useful when juggling more than 256 dots! Drink more water. ■





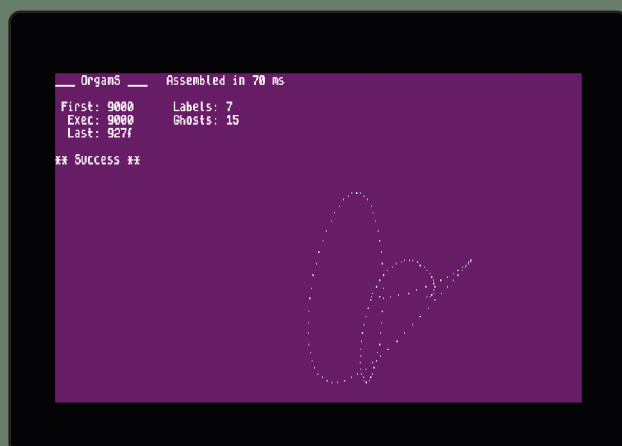
# TOO CURVY TO TEACH

How to generate a simple sine, less simple waves, and controlled curves in 2D (in no particular order). All while sipping hibiscus tea. Oh, and as a bonus, minimalist user interface. Life is full of surprises, enjoy them.

BY MADRAM



Orgams graphical dump



Plotting in 2D

## VISUALISATION

We are going to build curves, each crazier than the other, without alcohol. It's good to have quick graphical feedback.

## ORGAMS

Orgams' memory dump allows you to quickly visualize a 1D curve stored in a table.

```
256 ** BYTE SIN(5*#)/&2000/&1000+&80
```

By the way, I recommend that you isolate your ripple tables in a dedicated source. While adjusting, you can import it. Once crystallised, you can load the generated binary instead to save assembly time.

## PLOTTING

The firmware is fast enough (see source next page). For 1D version, simply take  $E = L$  (abscissa = index).

## THE KEY TO HAPPINESS. IN-DEMO EDITOR (IDE, A WORTHWILE INTERLUDE)

The best thing to do remains to modify and test the parameters on your own effect. Think of the *Plasma Demo* control screen, both fun and educational. Thank you Gozeur.



```

gra_plot_absolute = &BBEA
km_wait_char = &BB06

dots# = 128

MACRO ALIGN n:SKIP -$ MOD n:ENDM

; Plot 2d
    ld hl,wave0
    ld de,wave1
.lp2d
    push de
    push hl
    ld a,(de):ld e,a:ld d,1 ; x (shifted)
    ld l,(hl):ld h,0 ; y
    call gra_plot_absolute
    pop hl
    pop de
    inc e
    inc l
    ld a,l
    cp dots#
    jr nz,.lp2d

    jp km_wait_char

ALIGN(&0100)
wave0 dots# ** BYTE SIN(5*#)
        /&2000/&1000+&80

ALIGN(&0100)
wave1 dots# ** BYTE SIN(3*#)+SIN(5*#)
        /&0100/&0400+&80

```

Of course, a full GUI may be overkill, but a rustic key management is an excellent investment. An esteemed member of the cordial people community wrote:

"Spend a little time to save time. The "a little" is important. Rewriting yet another editor to "cross-develop" is not saving time, it is procrastination to avoid Z80 coding (I take the observation from Krusty, who applies a similar technique). At the end of the day, we don't care about productivity, a real fool's trap (see Alan Watts). It's about making the creative process as pleasant and serendipitous as possible."

With such tools, the speed of assembly is not an issue, as there is no need to reassemble to test a new combination. This interactive customisation will also be appreciated by the pixel artist or director of the project. Even the chef can get his hands dirty.

The following routines assume keymap filled with the values returned by the AY's port A for each row, aka "PPI keyboard scanning".

#### key\_pressed

```

; In: A = key code (schema on your keyboard)
;     Array keymap prefilled with all
;     keyboard state
; Out: HL = pointer in keymap
;      C = mask
;      Z = 1 if 'C AND (HL)' is 0
;      (e.g. key currently pressed)
;      B = 0
;      A trashed. DE preserved.

```

```

; First compute
; C = keycode / 8 -> keyboard line
; A = 2 ^ (keycode mod 8) -> mask

```

```

    ld c,a
    ld a,1
    srl c:jr nc,.ok1
    add a
.ok1 srl c:jr nc,.ok2
    2 ** add a
.ok2 srl c:jr nc,.ok3
    4 ** add a
.ok3 ld b,0
    ld hl,keymap
    add hl,bc
    ld c,a
    and (hl)
    ret

```

#### key\_down

```

; Like key_pressed, but:
; - keymap0 must contain previous
;   content of keymap.
; - return Z only when key is pressed down.

```

```

    call key_pressed
    ret nz ; NZ: not pressed

```

```

; Now check whether key was previously
; pressed, as we want to detect
; "not pressed" to "pressed" transition.

```

```

    ld a,c
    ld bc,keymap0 - keymap
    add hl,bc ; pointer in previous state
    ld c,a
    and (hl)
    xor c ; flip so we get Z when bit was 1
    ret

```

```

keymap0 FILL 10,&FF
keymap FILL 10,&FF

```



We can now introduce the main engine :

```

handle_keys
    ld hl,keymap
    ld de,keymap0
    ld bc,10
    ldir          ; backup previous content
    ld hl,keymap
    call scan_keys ; fill keymap via PPI
                    ; read for each key line
    ld de,shortcuts
.search
    ld a,(de):inc de
    inc a
    ret z
    dec a
    call key_down
    ex de,hl
    ld e,(hl):inc hl
    ld d,(hl):inc hl
    ex de,hl
    jr nz,.search
    jp hl ; jump to associated routine

shortcuts
; Key code, followed
    BYTE 66:WORD exit
    BYTE 1:WORD go_right
    BYTE 8:WORD go_left
; ...
    BYTE -1 ; Sentinelle

exit
    BRK

go_right
    ld a,23
    call key_pressed
    jr z,control_right
; Here... just right

```

Such a user interface can be activated in place of the music to suit the various machine-time or memory constraints.

### GENERATING CURVES

You may be familiar with the notion of the midpoint of two points A and B, which is cheerfully denoted  $(A + B) / 2$ . In practice, we mean (ah!) taking the average of each coordinate  $(x_A + x_B) / 2$  and  $(y_A + y_B) / 2$ . You'll agree much easier to use points which are agnostic in regards to the number of coordinates (3 in 3D, 11 in astral travel). It also works with colours to obtain an intermediate hue, when applied to each isolated component (red, green, blue) or rather (hue, saturation, luminance) to be more accurate.

Another way of writing it is  $1 / 2 \times A + 1 / 2 \times B$ .

Let's generalise with  $P(t) = (1 - t) \times A + t \times B$ . P is the point obtained as a function of a parameter t, which justifies the notation P(t). If  $t = 0$ , we simply have the point A. If  $t = 1$ , we arrive at B. If  $t = 1 / 2$ , we're back to the midpoint. Through the intermediate values from 0 to 1, we travel along the segment [AB]. You can think of t as a potentiometer, if you like. This is called parameterisation. When t represents time, P(t) represents the trajectory along the segment. If this doesn't give the fastest algorithm for tracing such a segment, this naive method is still an excellent first step for plotting hairline-controlled curves.

### BEZIER. THE AUTOMOTIVE INDUSTRY AT THE SERVICE OF COMPUTER GRAPHICS.

We now want to go from point A to point C, moving momentarily towards point B (Figure 1). We introduce a point M going from A to B and a point N going from B to C. Our parameterisation gives:

- $M(t) = (1 - t) \times A + t \times B$
- $N(t) = (1 - t) \times B + t \times C$

Hence,

- $M(0) = A, M(1) = B$
- $N(0) = B, N(1) = C$

The point P on our curve should gradually travel from moving source M(t) to moving target N(t):

- $P(t) = (1 - t) \times M(t) + t \times N(t)$

We verify that:

- $P(0) = M(0) = A$  and  $P(1) = N(1) = C$

Note that the principle is limpid, and can be generalised to multiple control points. Do not hesitate to carry out the calculation by replacing M and N with their definitions:

$$P(t) = (1 - t)^2 \times A + 2 \times (1 - t) \times t \times B + t^2 \times C$$

Grouping by powers of t gives:

$$P(t) = t^2 \times (A - 2 \times B + C) + 2 \times t \times (B - A) + A$$

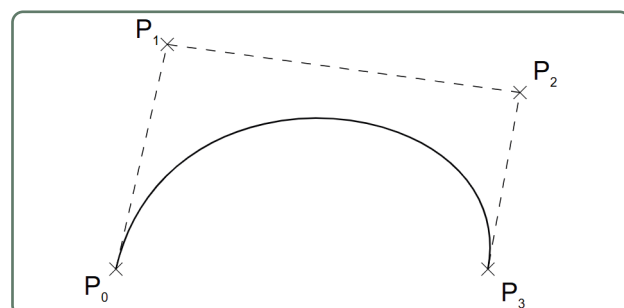


Figure 1: a Bézier curve of degree three with four control points in two space dimensions



### INCREMENTAL VERSION. THE IMPORTANCE OF STAYING DISCRETE

We are going to run through the different values of  $t$ , incrementing it by a constant value  $h$ , depending on the desired resolution, whether you want a “continuous” curve or dotted line. A word about the continuous version. The gap  $h$  must be small enough not to leave a hole. This can cause either accuracy problems (see dedicated paragraph) or performance problems. The reason is as follows: the points are not evenly distributed. The spacing at the ends is proportional to the lengths of the corresponding control segments, which typically differ. To fill in the gaps on one side, you have to repeat the same points several times on the other side. If your aim is to draw continuous curves as quickly as possible, a variant of Bresenham's algorithm would be a better fit.

We pose:

- $D = A - 2 \times B + C$
- $E = B - A$

We can now write:

- $P(t) = t^2 \times D + 2 \times t \times E + A$

In practice, this means precalculating these factors, eliminating the need to retain  $B$  and  $C$ . Rather than recalculating everything for each new value of  $t$ , consider:

$$P(t+h) = (t+h)^2 \times D + 2 \times (t+h) \times E + A$$

Where  $h$  is the little increment applied to  $t$ , hence advancing to the next point. Assuming we already have computed  $P(t)$ , we wish to know how much to add in order to obtain  $P(t+h)$ :

$$\text{delta}(t) = P(t+h) - P(t) = (2 \times h \times t + h^2) \times D + 2 \times h \times E$$

Notice that the term involving  $t^2$  was cancelled out and only one more term depends on  $t$ , which still requires a multiplication. Let's repeat the process:

$$\text{sigma}(t) = \text{delta}(t+h) - \text{delta}(t) = 2 \times h^2 \times D$$

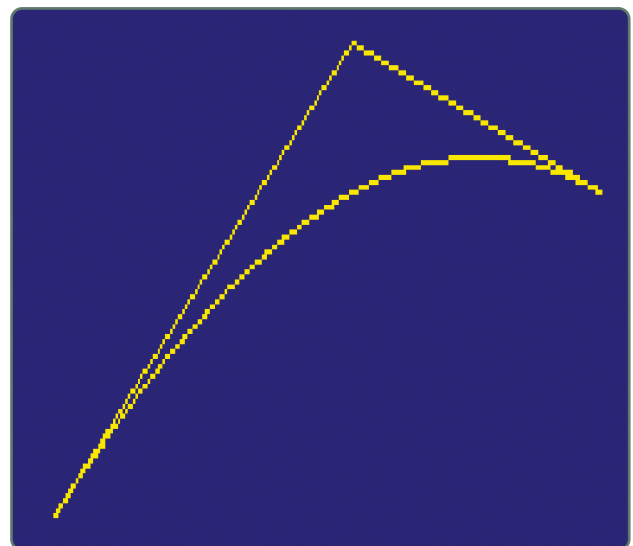
It's a constant, there is no need to parametrize it by  $t$ ! Don't hesitate to carry out the calculation on paper to convince yourself and practise. The initial value  $\text{delta}(f_0)$  is  $h^2 \times D + 2 \times h \times E$ , and we update it by adding  $\text{sigma}$ . The initial value  $P(0)$  is  $A$ , and we update it by adding  $\text{delta}(t)$ . We have casually applied a method called “forward finite differences”. And we haven't even finished our tea. In short, by adding a constant difference, we obtain a linear progression. A free analogy: when the speed is

constant, the distance travelled is proportional to the time passed. By adding a linearly increasing distance, we obtain what is called a quadratic progression. When the speed *increases* in a constant manner, we are accelerating and the distance travelled is proportional to the square of the time passed. C.f. the 3 towers on September 11, falling at the speed of free fall.

We can also observe this phenomenon with the sequence of squares (0 1 4 9 16 25 36 49...), if we take the trouble to calculate their successive differences (exercise 3<sup>1</sup>).

Let's put all this together in the form of a small BASIC program.

```
10 ' Parameters -----
20 xa=10:ya=20
30 xb=130:yb=210
40 xc=230:yc=150
50 iter = 128
60 ' Show control polygon -----
70 MOVE xa,ya:DRAW xb,yb:DRAW xc,yc
100 ' Setup -----
110 h = 1/iter
120 dx = xa + xc - 2*xb
130 dy = ya + yc - 2*yb
140 deltax = h^2*dx + 2*h*(xb-xa):
    sigmax = 2*h^2*dx
150 deltay = h^2*dy + 2*h*(yb-ya):
    sigmay = 2*h^2*dy
160 x = xa
170 y = ya
200 ' (Midline) Process -----
210 FOR i=0 TO iter
220 PLOT x,y
230 x=x+deltax:deltax=deltax+sigmax
240 y=y+deltay:deltay=deltay+sigmay
250 NEXT
```



1 Solution: 1, 3, 5, 7, 9, 11, 13...



Notice that when C is the midpoint of [AB],  $D = 0$ , so  $\sigma = 0$ ,  $\delta$  is constant, so we get back our linear segment. Practical application: if the abscissa of the control point is in the middle of the extreme abscissas, simply apply the algorithm to the ordinates. Or vice versa, it doesn't matter, in either case the calculation is simplified.

#### FIXED DECIMAL POINT

The firmware offers support for floating-point “reals”, with a precision that covers all your needs, from calculating your taxes to your ray-tracing engine. In assembler, performance concerns complicate things. On the face of it, registers represent integers. Well, it's all a matter of perspective, you can also think of H as the integer part and L as the decimal part. So HL represents  $1/256^{\text{th}}$ s and we approximate any number  $x$  by  $x' / 256$  where  $x'$  is an integer. Since  $x \approx x' / 256$ , then  $x' \approx x \times 256$ .

In other words, we're changing units. Just as 1m96 is 196 cm ( $\times 100$  for the conversion), multiplying all our working variables by a constant (called “precision” below) allows us to talk in integers (integrity is important to me) while maintaining a decent precision. For the display, we perform the inverse conversion (a division) to get the expected unit. Now dividing a 16-bit register by 256 simply means taking the most significant byte (MSB). Only the strong survive (and all those who are curvy enough).

Validating that in our BASIC program amounts to adding the following lines:

```
55 precision = 256
57 DEF FNfixed(x) = CINT(x*precision)
```

And modifying:

```
140 deltax = FNfixed(h^2*dx + 2*h*(xb-xa)):
sigmax = FNfixed(2*h^2*dx)
150 deltax = FNfixed(h^2*dy + 2*h*(yb-ya)):
sigmay = FNfixed(2*h^2*dy)
160 x = FNfixed(xa)
170 y = FNfixed(ya)

220 PLOT x/precision,y/precision
```

Tip by the way: the program cheats by performing the delta initialisation with full precision before the final conversion via `cint`. For the assembler version, it is best to evaluate it as follows:  $h \times (h \times D + 2 \times E)$ . This is equivalent on paper, but in practice it avoids some loss of precision. It's also simpler.

Let's come back to the fixed-point principle. For illustration, we take  $x = 1 / 3$ . Then  $x' = \text{cint}(256 / 3)$  gives

85 (`cint` rounds to the nearest whole number). In hexadecimal, the successive multiples of 85 are `&55 &aa &ff &15a &1a9 &1fe &253...` We note with unabashed satisfaction that the MSB increments every third time, as does the integer part of the multiples of  $1 / 3$  (please translate in French for the rhyme). Observe that we get `&ff` instead of `&100` (which would represent 1.00 when we reach  $3 \times 1 / 3$ ), due to the loss of precision, but that doesn't bother us that much.

#### TOO APPROXIMATE?

Oh, maybe we're going to be a little annoyed after all. When a number is rounded to the closest integer, the maximum alteration introduced is 0.5. Similarly, our accuracy of  $1 / 256$  leaves a possible error of  $1 / 512$ . Accumulated 128 times (the number of iterations), such an error would still be undetectable. Remember, however, that we are working at second order. As a matter of fact, to P we successively add  $\delta(0)$ ,  $\delta(0) + \sigma$ ,  $\delta(0) + 2 \times \sigma$ , ... up to  $\delta(0) + 127 \times \sigma$ . In total we have added  $128 \times \delta(0) + 127 \times 64 \times \sigma$ . The cumulative error can therefore be as high as  $(128 + 127 \times 64) / 512$ , i.e. around 16 pixels!

What's more, if you run the second version of the program, you'll notice that the curve no longer ends where it should. What a disappointment! At the same time, it's great to be able to test so quickly in BASIC. The solution is simple, increase the precision. If you don't need more than 64 pixels, you can split HL into 6 bits for the integer part and 10 bits for the decimal part. In the BASIC program, this is equivalent to setting `precision = 1024 ( $2^{10}$ )`. However, it might be quicker and easier to switch to 24 bits.

Hang on a minute! That's 64 NOPs you're holding in your grimy hands. One would expect more ingenuity from such a prestigious magazine. Let's imagine for a moment that in our BASIC program  $dx$  is a multiple of 32, then it is written as  $32 \times kx$  where  $kx$  is an integer, and we have  $\text{sigmax} = \text{cint}(2 \times 32 \times kx \times 256 / 128^2)$ , i.e.  $\text{cint}(kx)$ , i.e.  $kx$  itself, without loss of precision. The same principle applies to  $dy$ . All we have to do is slightly modify the control point to respect this constraint. In our example,  $xb = 136$  and  $xy = 213$  solves the problem without betraying the initial curve. Só alegria! (in Portuguese in the text.)

#### INTERPOLATION

A Bezier curve does not pass through the control point, which only provides the direction of curvature. It may be preferable to hit this point. Fortunately, through three non-aligned points pass an arc of a parabola! In fact, an infinite number of parabolas. This gives you the opportunity to play with an extra



parameter. If you don't wish to play, we can stop at a one criterion, such as the shortest possible arc. Incidentally, a quadratic Bezier curve is an arc of a parabola that satisfies the variation diminishing property. The calculations are left as an exercise, and may be corrected in the next issue. The production asked me to leave some plot points (ah!) vague enough to encourage you to subscribe.

### MORE CONTROL FOR MORE FLEXIBILITY!

Only 3 control points won't allow for more complex curves, such as a S shape. Each new control point offers more plasticity while increasing the degree of the curve, meaning more computations. With 4 control points (two extremities and two intermediates), the same technic involves 3 variables instead of 2:

$$\begin{aligned} P(t+h) &= P(t) + \text{delta}(t) \\ \text{delta}(t+h) &= \text{delta}(t) + \text{sigma}(t) \\ \text{sigma}(t+h) &= \text{sigma}(t) + \text{constant} \end{aligned}$$

This update is still doable easily on the Z80:

```
; HL = P
; DE = delta
; BC = sigma
  add hl,de    ; Update P
  ex de,hl
  add hl,bc    ; Update delta
  ex de,hl
  ld a,c:add constant:ld c,a ; Update sigma
  jr nz,.okcorral
  inc b
.okcorral
```

Yet a simpler approach is to build the curve piecewise.

### SINUS

Great practical application! A sine can also be approximated by a parabola. As Taylor used to say:

$$\cos(x) = 1 - x^2 / 2 + x^4 / 4! - x^6 / 6 \dots$$

For non-trigonometric purposes, the first two terms are enough. We'll tinker with the coefficients so that we land on our feet. We ask for:

- $\text{coco}(0) = H$ , where H as in Height, Half or Hu-guette represents the amplitude.
- $\text{coco}(64) = 0$ , since 64 represents  $\pi / 2$  reached at a quarter of a period 256.

Let's write  $\text{coco}(x) = H - k \times x^2$ . Hence  $k = H / 4096$ , if I'm not mistaken.

Let's apply the principle of finite differences once again, with an increment of 1 (shift x in our table).

$$\begin{aligned} \text{coco}(x+1) &= H - k \times (x^2 + 2 \times x + 1) \\ \text{coco}(x) &= H - k \times x^2 \end{aligned}$$

The difference is  $-2 \times k \times x - k$ . It is zero at 0 (top of the cosine, for which the tangent is horizontal). It then becomes negative, indicating that the ordinate is decreasing. The difference of the differences is  $-2 \times k$ , i.e.  $-H / (8 \times 256)$ . It is negative, which means that the ordinate is decreasing faster and faster.

With a fixed point resolution of  $1 / 256$ , we will be careful to choose a multiple of 8 for H so that k is whole, avoiding any problem of error accumulation.

### SYMMETRY 1

Instead of generating our parabola from 0 to 63, let's apply the formula in one go from -64 to 63, filling in the first 128 values of our table. That is, we generate a sine instead, using with good honesty  $\sin(x) = \cos(x - \pi / 2)$ . At -64, the difference is  $2 \times k \times 64 - k$ . We will therefore initialise it at this value.

### SYMMETRY 2

The second half of the table is obtained by symmetry. Choose one of the following:

- $\sin(x + \pi) = -\sin(x)$  i.e. in +128 we poke the opposite value.
- $\sin(2 \times \pi - x) = -\sin(x)$  if we fill in the second half of the table from its end.

### SUMMARY

In most cases, you don't want the sine to be centred around 0, which would imply you have to work with signed values. However, we can start from any value, as we are working incrementally. The result is the following routine, which has been used with mixed success for over twenty years. It's not the most concise, but it's easy to study and allows the generation of sinusoids of different amplitudes.

Nothing prevents setting BC and DE to arbitrary values. This exposes you to discontinuities in the curve, which can have the best effect.

The record size for generating an approximate sine is 16 bytes (neon/darklite + Baze/3SC) and is based on the same principle<sup>2</sup>.

<sup>2</sup> <https://github.com/neonz80/sine/tree/main>



```

table = &a000 ; Destination
mid = 128     ; Midpoint in y
amp = 14      ; Amplitude/8

IF table AND &FF
    !! must be aligned
END

IF amp AND 1
    !! should be even (otherwise there is a
        little discontinuity)
END

    ld ix,table+&80
    ld a,mid:ld h,a:ld l,0
    add a:ld iyh,a
    ld de,amp*&3F + amp/2
    ld bc,-amp

.gen_sine
    ld a,iyh
    sub h
    ld (ix+0),a
    ld (ix-&80),h
    add hl,de
    ex de,hl
    add hl,bc
    ex de,hl
    inc ixl
    jr nz,.gen_sine

```

### COMBINING PARTS CHEERFULLY

Starting with this primal undulation (twerk), let's look at the transformations we can apply.

#### AMPLITUDE CHANGE

If the sine generation routine doesn't produce the desired amplitude, we can create a sine of arbitrary amplitude by adding it to a phase-shifted version. Taking the average improves the resolution, but prohibits any amplification (amplitude greater than the original one).

```

shift = 30      ; 0: no change, 128: flat

    ld hl,sine ; Already existing sine
    ld de,sine+shift
    ld bc,dest
.mean
    ld a,(de):inc e
    add (hl):inc l
    rra
    ld (bc),a:inc c
    jr nz,.mean

```

#### AMORTIZED (DAMPED)

We apply the previous principle and gradually increase the phase shift. To go from full amplitude to

full amplitude, the phase shift goes from 0 to 128. It is therefore incremented every second time.

```

    ld hl,sine ; Already existing sine
    ld de,sine
    ld bc,dest
.amor
    ld a,(de):inc e
    add (hl):inc l
    rra
    bit 0,c:jr z,.noinc
    inc l
.noinc
    ld (bc),a:inc c
    jr nz,.amor

```

For better crunchability, you may unroll two iterations instead of conditionally jumping to .noinc.

#### SPIRAL

As a reminder, a circle of radius  $r$  is drawn by setting:

- $x = \cos(t) \times r$
- $y = \sin(t) \times r$

Where  $t$  varies from 0 to  $\pi / 2$  (from 0 to 255 if we use a table). If  $r$  increases (or decreases, who cares?) with  $t$ , a spiral emerges. All you have to do is combine a damped sine with a damped cosine.

#### ACCELERATION

To double the frequency, simply go through the sine table with a step of 2. More interestingly, let's say you want to progressively accelerate from a single speed to a triple speed. The step, initially set at 1, must therefore increase by  $2 / 256$  at each iteration. Fixed decimal point will once again do the job.

```

    ld h,sine/&0100 ; Already existing sine
    ld ix,0          ; pos*256
    ld de,&0100      ; step*256
    ld bc,dest
.accelerate
    ld a,ixh:ld l,a
    ld a,(hl)
    add ix,de
    inc de:inc de
    ld (bc),a:inc c
    jr nz,.accelerate

```

Thanks to Krusty for proofreading and suggestions. Any remaining errors are his. ■





# RHINO / BATMAN GROUP

After a pit stop to refuel the Amiga scene with *Batman Rises*, Rhino is back on CPC to tackle the final stretch of the eagerly-awaited *Vespertino*. He kindly agreed to answer our questions before crossing the finishing line.

BY TOMS/PULPO CORROSIVO (IN JULY 2024)



*Vespertino*

Hello Rhino! You've been working for several years on *Vespertino*, a car racing game, can you tell us about it? Who is in the development team?

We started the game in early 2019 and although we announced it that same year with a video where we showed the engine we had then, the truth is that several factors happened that made us keep it on pause for a long time. First of all, that same spring there was a resurgence of the Amiga demo scene, with the mythical Revision party compo of that year where demos like *Eon* or *Brutalism* were released. That provoked a majority feeling in the group to return to that platform, which together with the fact that we had given our word to return to the Amiga after the development of *Pinball Dreams* and a motivational video that the spanish Amiga scene made for us titled "llamando a Rhino", made us get fully into the development of *Batman Rises*, pausing the development of *Vespertino*. Then came the coronavirus crisis and it wasn't until last year that we resumed the CPC game. Currently it has been on pause for 3 months due to family reasons, but we are going to resume it soon. I hope this does not sound like an excuse, but as a testimony of how complicated it can



sometimes be to finish a development for a retro platform for pure hobby.

As for the people involved, Toni Gálvez, Mac and I are the most involved so far. There are areas of development such as music or testing that we have not yet entered and for which we will have more people. TotO is also collaborating.



Toni Galvez, Batman, Mac, Rhino, Reset and Dredd at Christmas 2023 Batman Group meeting

### How far has the project progressed?

We are just at the most fun point, which is when the gameplay of the game is established, so we could say that the hardest and most tedious work with tools, engine, etc. is already finished.

### Will the game run on an original CPC 6128, or will it require a memory extension to run?

The game will work on the original Amstrad models thanks to the cartridge format, which allows us to have the necessary memory for a game like this. A similar approach to that Abalore and TotO did with the fantastic *Alcon 2020*.

### What were the major challenges you faced during development, and how did you overcome them? It seems that you're using a rather clever hardware technique for the road management...

The engine development was a great challenge. Curiously, we designed and developed it almost completely in 2 or 3 months in early 2019. The engine design includes a specific format for the game sprites, which is an adapted and more efficient version than conventional compiled sprites. For that we recently had to make a tool that does a serious compression and code optimization work. At the time of launching the video we still did not have this tool, so we used a more rudimentary tool that required to optimize sprites code by hand, which we could afford for what was shown then, but it was not a reasonable option for the amount of sprites finally required and that are counted by thousands!

### You've used some rather intriguing names to communicate the game's technical features like "3D CRT-C-FX Engine", "PRI Technology" or "UFD Sprites". Is this a marketing play or a genuine technical find?

Both things, hehe. Without a doubt, these bombastic names have a marketing effect, but there are also special techniques behind. For example, with "UFD Sprites" I mean the special sprites format that I commented before, or PRI (programmable raster interrupt) refers to the little explored capability of classic CPC to generate interrupts on any screen line, as opposed to the 6 fixed interrupts that are classically used. I use this so that the dynamic splitscreen between scoreboard and play area adapts to changes in road relief. "3D CRT-C-FX Engine" refers to the engine I use to render 3D elements using the CRT-C capabilities.

### What tools did you use throughout the development process? How does the team work?

I am using *WinAPE* and *Notepad++* for the code, the graphists use *Promotion*, *Photoshop* and 3D programs such as *LightWave*, among others. Then I also use a good amount of own tools for technical issues, and to communicate we use *Telegram*, *Discord* and *Google Drive* mainly.

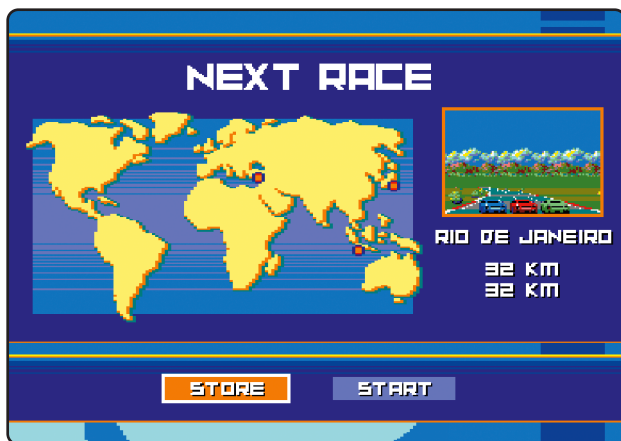


Vespertino - Garage

### What are your favourite racing games? Will Vespertino bring something new to the genre?

Amiga's *Lotus* is being a great influence as well as the classic arcade *Outrun* and on aesthetic, *Outrun 2*. It is very difficult to innovate in a game like this and for which so many and such good titles have been developed for more than 40 years, but we will try to give it our touch, hehe.





Vespertino - Next race

**Development of the game was temporarily put on hold to create *Batman Rises*, an absolutely fantastic demo on the Amiga 500. Can you tell us about the genesis of this demo?**

Thanks! As I mentioned before, it was a conjunction of factors. I think that if you search on Google "llamando a Rhino" it is still possible to find the motivational video that the spanish Amiga scene made us inviting us to develop again for the platform that saw us born as a group and after having given our word that we would return, something we had to do. The other important factor was that after Revision 2019, we saw the Amiga scene almost as lively as in the old days and I think it was in the summer of that year that we made a meeting and decided that our next great release should be a demo for the classic Amiga.

*Batman Rises* is a different demo than others we've done because we've tried to keep a coherent narrative throughout the demo. For this, the first thing we did was a script that defined the most relevant narrative elements and from there we started to develop the tools, engines and effects that we needed.

**How do you work when you take on a new demo project? Do you draw up the storyboard yourself? Do you use effects that you already have in your drawers? Or does the storyboard decide on the effects?**

It depends on the demo. For *Batman Rises* we had a very clear idea about the narrative of the demo and we made the effects as we saw that they could fit in some way. At other times, when there is not such a clear story or theme behind a demo you can just do a collection of effects, but this is an approach that I like less and less, even though the coder has more freedom to make any effect on them.

**From my point of view, one of the great successes of *Batman Forever* and *Batman Rises* is their rhythm and the perfect synchronisation with the music. How did you go about working with the musician?**

Thanks! I think the key is to give the musician enough freedom (besides they are quite good, hehe). What I usually do is show them the demo effects and tell them the demo background, as well as everything that until then is defined from the storyboard. Then they send me a first version of the music with which I try to get an idea of how the effects could fit. Here it is usually necessary to make some music and / or effects adjustment so that all fit well. In short, the important thing is to give the importance that this aspect deserves.

**Do you plan to release new demos on the CPC? What other challenges would you like to tackle on this platform?**

CPC is still one of my favorite platforms although I have yet to get the motivation to do another great demo. I have several ideas floating around in my mind that I'd like to do for CPC but, for now, I have to focus on finishing the racing game and we'll see what happens next.

**Any chance of running into you one day at a CPC meeting or at Revision?**

I hadn't been to any demo party for a long time but lately I've become a regular at local parties like Posadas. I wouldn't like to go to a big party like Revision without presenting something good, so if one day we see each other there it will probably be because I'm bringing that new CPC demo you were asking about before, hehe.

**Thank you very much for your time. In keeping with tradition, the final words are yours...**

Well, thank you for giving me the opportunity to do this interview and for keeping the CPC scene alive. ■



Batman, Narcisound, Dredd, Rhino and Mac at Amstrad Eterno 2023





# EARNING A STACK WITH THE STACK

Almost all programs use the stack, at least to save registers (PUSH) or call routines (CALL). But this November, I've put on my slippers and mittens to explain how to make advanced use of it and impress your brother-in-law at the next Christmas Eve.

By Hicks/VANITY

## A FEW BASIC PRINCIPLES

Before looking at advanced usage, a few remarks on how the stack works on the CPC.

- The “stack” refers to an area of memory which, in standard use, allows data to be temporarily saved (stacked) and then retrieved (unstacked). The address of this zone can be freely defined using the z80's 16-bit SP register (Stack Pointer).
- The z80 stack operates on the LIFO principle (Last In, First Out). The last address stacked will be the first address unstacked. So when a POP is encountered, it always unpacks the address stacked by the last PUSH.
- Don't forget that the CPC works with a Little Endian encoding: when a word (two bytes) is stacked, its LSB (Least Significant Byte) is stored before its MSB (Most Significant Byte).

```
ld sp,&4000
ld hl,&1122
push hl
```

SP	(SP)
&3FFE	&22 (L)
&3FFF	&11 (H)

- The system makes heavy use of the stack. You must therefore be careful if you modify SP when interrupts are enabled (EI instruction). At start-up, the system sets  $SP = \&C000$ , then writes below this zone for firmware management. Care should be taken when using the stack in conjunction with interrupts, or disabling them (DI instruction).

## THE INSTRUCTION SET

Insert 1 lists the z80 instructions involving the stack. I've added some pseudo-code, indicating what effects these instructions have. To make it easier to read, I've only mentioned IX, but all the instructions that exist with IX also exist with IY (only the prefix differs).

Some remarks on these instructions:

- Speed: the PUSH / POP pair takes 4 and 3 NOPs respectively, which is very fast. There's great potential here for optimising routines.
- Size: many instructions are very compact, which could be useful in a sizecoding context. For example: LD SP,HL (1), EX (SP),HL (1), PUSH / POP (1), RET (1), RST (1).



### SET INSTRUCTION INVOLVING THE STACK

(Based on the Organs notice, thanks to Madram!)

ld sp,d16	M	3b	3 us	sp = d16
ld sp,h1	M	1b	2 us	sp = h1
ld sp,ix	M	2b	3 us	sp = ix
ld sp,(adr)	M	?	6 us	sp = (adr)
ld (adr),sp	/	?	6 us	(adr) = sp
push qq	M/W	1b	4 us	dec sp : dec sp : (sp),qq
push ix	M/W	2b	5 us	dec sp : dec sp : (sp),ix
pop qq	M/R	1b	3 us	qq = (sp) : inc sp : inc sp
pop ix	M/R	2b	4 us	ix = (sp) : inc sp : inc sp
ex (sp),h1	R/W	1b	6 us	pop tmp : push h1 : ld h1,tmp
ex (sp),ix	R/W	2b	7 us	pop tmp : push ix : ld ix,tmp
add h1,sp	/	2b	3 us	h1 = h1 + sp
add ix,sp	/	3b	4 us	ix = ix + sp
adc h1,sp	/	3b	4 us	h1 = h1 + sp + C
sbc h1,sp	/	3b	4 us	h1 = h1 - sp - C
call nn	M/W	3b	5 us	push pc+3 : JP nn
call ccc,nn	M/W	3b	5/3 us	push pc+3 : JP nn if ccc
rst ttt	W	1b	4 us	push pc+1 : JP ttt
ret	R	1b	3 us	pop pc
ret ccc	R	1b	4/2 us	pop pc if ccc

Legend:

- R: read (SP)
- W: write (SP)
- M: modify SP

### A STACK OF NEW INSTRUCTIONS?!

By combining or hijacking the standard instructions, we can imagine new ones and encapsulate them in macros to extend the possibilities offered by the stack. Some examples!

LD SP,HL is available, but not the reverse operation. No problem!

```
macro LD_HL_SP
  ld h1,0
  add h1,sp
endm
```

Cost: 6 NOPs, 4 bytes. We can try it with IX and IY registers.

Unfortunately we can't write the address of SP to the address pointed to by SP... Just kidding!

```
macro PUSH_SP
  LD_HL_SP()
  push h1
endm
```

Cost: 10 NOPs, 5 bytes. Arg, we are not kidding around anymore. Thanks to roudoudou for the tip, and his apples cooked in calvados at BND #4.

Between two demos, toms sometimes likes to make POP SP:

```
macro POP_SP
  pop h1
  ld sp,h1
endm
```

Cost: 5 NOPs, 2 bytes. A great deal.

If RET is equal to a POP PC, how can we make a PUSH PC? Golem13 answered:

```
macro PUSH_PC
  call $+3
endm
```

Cost: 5 NOPs, 3 bytes. "\$" symbol is equal to PC address. The start address of the next line is saved.



### ADVANCED USE OF THE PUSH INSTRUCTION

The PUSH instruction writes 2 bytes to memory in just 4 NOPs and 1 byte. The equivalent of a PUSH DE would cost 8 NOPs and 4 bytes:

```
dec hl
ld (hl),d
dec hl
ld (hl),e
```

So it's twice as fast and 4 times as compact! We will try to take advantage of this power.

### EXAMPLE 1: FAST SCREEN CLEANING (FSC)

Do you know the fastest way to clean the screen?

```
ld hl,0
ld sp,hl
ld b,h ; 256 iterations
fast_clean_screen
64 ** push hl
djnz fast_clean_screen
```

SP must be initialised with &0000 address since the PUSH instruction will decrement it before writing (&FFFF, then &FFFE, etc., up to &C000).  $256 \times 64 = &4000$ , everything is ok.

### EXAMPLE 2: SOFTWARE MAGIC RASTERS (SMR)

The *Crownland* game preview makes clever use of the stack to scroll a repeating landscape strip. The principle is quite simple: if you preload 3 16-bit registers (BC, DE, HL), for example, you can build each line of the landscape with a 6-byte pattern repetition. You just have to write, from right to left (PUSH decrements SP), a 6-byte wide pattern that you can repeat:

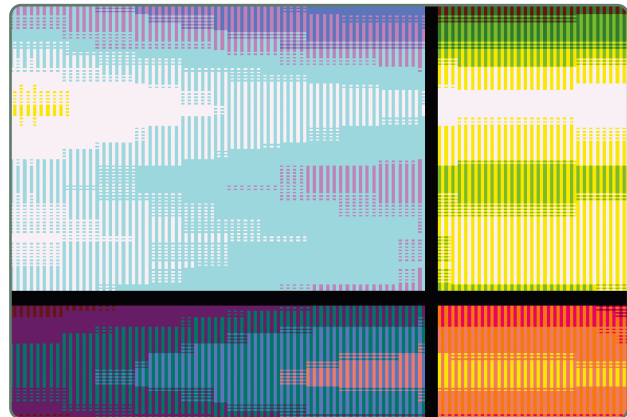
```
push hl
push de
push bc
push hl
push de
push bc
...
```

To display the next lines of the bitmap, simply reload the registers and use the same code. To make this landscape scroll by pixel, you also just need to reload the registers, taking the shift into account. You can extend the technique with index (IX, IY) and secondary (BC', DE', HL') registers. In this way, a 16-byte pattern can be created.

About CPU, to fill a 32-word wide line, 32 PUSH are needed, i.e.  $32 \times 4 = 128$  NOPs. A 16-line pattern could then be completely re-displayed and shifted in just over 32 lines of machine time. In a way, this is a software alternative to magic rasters (see *64 NOPs #2!*).

### EXAMPLE 3: DOUBLE POINTER (DP)

In a demo context, you often need to manage two pointers in parallel: one to browse data or perform calculations, the other to write the result to the screen. The stack can therefore be a useful replacement for one of these pointers. Here we look at the case where it is used to write with a PUSH.

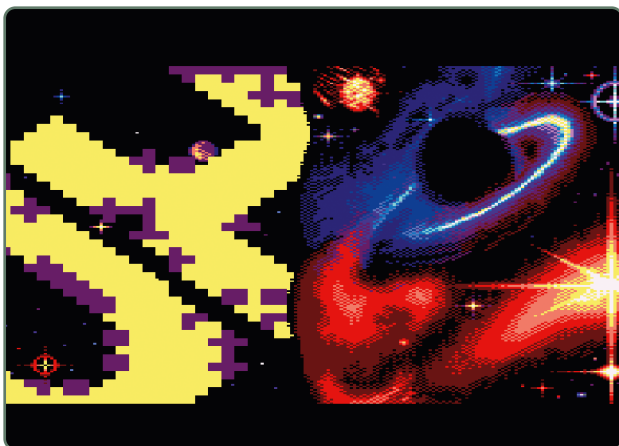


*Still Rising (Vanity)*

This is how the rotozoom works in *Still Rising* (Vanity). A first pointer (HL) moves through a texture according to a slope (INC L to move in X, INC H to move in Y) and a zoom level (INC more or less frequent). DE is loaded by the contents of the texture, and written to the screen with a PUSH, pointed by SP:

```
ld e,(hl)
inc l / inc h
ld d,(hl)
inc l / inc h
push de
```





*Pheelone (Condense)*

Each byte of the texture is thus read in 3 NOPs, and written in 2 NOPs. This byte is then duplicated 4 times with line splitting, in order to display 4 bytes in 5 NOPs.

The *Pheelone (Condense)* demo takes an even more radical approach. The display code, generated upstream, is simply a succession of PUSH. So if, for example, we assign the colour yellow to HL and purple to DE, each line to be displayed will be a combination of PUSH HL and PUSH DE. However, the resolution will be lower (per word), and the type of texture more limited.

#### EXAMPLE 4: ULTRA-FAST DRAWING (UFD)

A few years ago, Rhino introduced a way of using the stack for display that he called UFD<sup>1</sup>. Many coders had been using the stack to display sprites for a long time, but his technique offers something new. Until now, displaying on the stack meant that the width of the sprite was a multiple of a number of PUSH (i.e. 2 bytes), which is rather restrictive for displaying ordinary sprites in a variety of shapes.

To display a sprite in this way, there are three operations: filling the right edge of the sprite, the centre, and the left edge. If we assume that HL is pointing at the screen:

- Center: the fastest way is to load 2 bytes into a 16-bit register (BC for example) and write it via a PUSH (7 NOPs in all). Each time you do this, you only reload what changes in BC (BC, B, C, or nothing):

```
ld sp,hl
ld bc,#1122
push bc      ; write #1122
ld c,#33
push bc      ; write #1133
...
```

- Right edge: as the address is initially in HL, we can use it before starting our series of PUSH:

```
ld a,(hl)
and mask1
or byte1
ld (hl),a    ; first pixel
ld sp,hl     ; push can begin
```

- Left edge: rather than displaying it at the end, since HL already contains the start-of-line address, why not subtract the sprite width and display this edge directly?

```
ld sp,hl
ld bc,-width
add hl,bc
ld a,(hl)
and mask2
or byte2
ld (hl),a    ; then, push can begin
```



*Pinball Dreams (Batman Group)*

In *Pinball Dreams* (Batman Group), Rhino uses it mainly to display the big lights. Indeed, as he explained, the larger the sprites, the greater the gap between the UFD technique and a conventional auto-generated display technique without the stack. For example:

```
ld (hl),#11
inc l
ld (hl),#22
inc l
```

This code takes 8 NOPs per 2 bytes, compared with 7 for the UFD. So you save one NOP per word. Conventional auto-generated code requires us to work with a screen address of constant MSB (H register), which often means a screen format of 32 words wide, or losing the benefit of INC L. UFD, on the other hand, is compatible with any screen format.

<sup>1</sup> Rhino explains his technique on the CPC Wiki forum, in a post entitled “UFD Technology (Ultra-Fast Drawing)” opened on 31 March 2019.



The line change can be managed with:

```
ld hl,delta_next_line ; &800 or &c800
add hl,sp
```

You can adjust the `delta_next_line` value so that you are positioned exactly in the right place on the next line.

#### ADVANCED USE OF THE POP INSTRUCTION

The POP instruction reads 2 bytes from memory in 3 NOPs and 1 byte, which is very fast. The equivalent of a POP DE would consume 8 NOPs and 4 bytes:

```
ld e,(hl)
inc hl
ld d,(hl)
inc hl
```

So it's more than twice as fast. And since it's faster than a PUSH, if you have the choice between reading data with a POP or writing data with a PUSH, choose the POP whenever possible.

#### EXAMPLE 1: MAKING DOTS LIKE ELIOT (MDLE)

In this issue of *64 NOPS*, the great Eliot offers an introduction to manage and display dots for your beautiful demos. In this context, the stack is useful not for displaying the dots, but for deleting them: by building a table of addresses where you display them with a simple PUSH, it will be easy to find them again later and delete them using a simple routine like (A=0):

```
nb_dots ** [
    pop hl
    ld (hl),a
]
```

If you want to restore the background, it's a little more complex. If you want to go further, read his article!

#### EXAMPLE 2: FASTER LDI (FLDI)

How do you copy a zone from one place to another as quickly as possible? If you perform 64 LDI, at 5 NOPs per LDI, this takes 320 NOPs. A better solution, with a small constraint (destination address must be hardcoded):

```
64/2 ** [
    pop hl
    ld (adr+ #*2),hl
]
```

"#" is the internal loop counter (in *Orgams* assembler). This routine takes 8 NOPs per word, so 4 per byte instead of 5. 20% faster!

#### ADVANCED USE OF THE RET INSTRUCTION

When the PC encounters a RET, it takes the address pointed by SP. The RET instruction is therefore the equivalent of a POP PC. This makes it a very powerful instruction (3 NOPs, 1 byte) for building jump tables.

#### EXAMPLE 1: FAST JUMP TABLE (FJT)

Sometimes you need to execute a series of routines in a variable order. In this case, you can use the stack to fix this order by building an address table to which SP points, and jump from one routine to another elegantly via a simple RET.

In *École Buissonnière* (Overlanders), several effects work on this principle: retrieve the address of the data and the screen via 2 POP, then a RET jumps to the execution address of the next routine:

```
pop hl    ; screen address
pop de    ; data address
ret       ; next routine address
```

In particular, this allows the same routine to be run several times, without it jumping to the same address each time when it finishes.



*École Buissonnière* (Overlanders)

#### EXAMPLE 2: CONDITIONAL RST (CRST)

In the first issue of *64 NOPS*, I gave you a tip from Golem13 on how to produce a conditional RST. Take a look at it, or order it (– this is product placement –).

#### STACK OVERFLOW

I had planned to tell you about the possibility of combining advanced stack usage without disabling interrupts, but I've run out of space. This could be the subject of a future article! ■





# J'AI PÉ-TÉLÉCRAN

*J'AI PÉ-TÉLÉCRAN* is a 4K released at Revision 2024 demoparty. I coded<sup>1</sup> and organized this 4k, while Targhan/Arkos wrote the music and Exocet/Dentifrice did the graphics. It was ranked 4th in the Oldskool Intro competition. The aim of this article is to give an overview of the method used to create it.

BY KRUSTY/BENEDICTION

Initially, I wanted to make an abstract demo that simulated a plotter to display algorithmically generated drawings (something pretty close to a logo interpreter, in fact). Believing that I wouldn't have the time to code the mathematical routines, the virtual machine, and search for interesting programs, I ended up doing something less abstract by displaying line-based images drawn by a graphic artist. The rendering simulates a “telescreen” (named “télécran” in French, hence the title of the demo).

Giving the graphic artist control over the display order of the pixels would have been tedious (for me by creating a dedicated tool, for him by drawing with such a tool), and not controlling the display order is totally out of the question (either the display would have made no sense, or the amount of data would have been too large). I have selected an alternative where a program tries to choose a display order that's not too bad (this is an optimization problem for which it's not possible to algorithmically find the best choice in a reasonable time, so only approximately good orders can be found). In hindsight, it's more complicated than it should be, but with the time available to work on the project, I'm pretty satisfied with the solution.



Figure 1: The first picture drawn

## USED TOOLS

Too “young” to have any affinity or sentimental relationship with the Amstrad CPC, I'm only interested in watching demos running on it, not manipulating it to craft my demos or play games. I couldn't care less about not working within the constraints of the original machine; my carpal tunnel is just fine. The last project published and natively coded is probably my part in the *232425 Meeting Demo* released in 2002, coded at the same time as *Come Join Us* with DAMS and a Ramcard.

<sup>1</sup> <https://github.com/rgiot/demo.revision2024.etchy>



Since *Can Robots Take Control?*, I've been using crossdev's Benediction suite of tools for Amstrad CPC<sup>2</sup>: my assembler *basm*, my project construction and dependency management tool *bndbuild*, various image conversion and transfer tools for DSK, HFE, SNA, M4. I test in real time on my Amstrad CPC by sending a snapshot to the M4, and debug with *ACE-DL* or *WinAPE*.

*basm* was initially written to be compatible with *rasm*, but I think this will become less and less the case as it evolves. Its parser is unfortunately slower (up to a factor of 10 in some test files with macros!), but assembly is still fast, and I control all the available directives and their syntax (many of which don't exist on *rasm*, such as those related to section management that have been intensively used for *Come Join Us* to ease its finalization).

The aim of *bndbuild* is to: embed in a single executable all my crossdev tools, automate project construction and execution (both on emulators and the real machine), and allow non-coders to easily test the impact of their modifications on project data in real time. I only use the command-line flavor, but I also provide a graphical version. A click on a button of the graphical version builds a project and sends it to the CPC's M4 or an emulator. It can also monitor modified files on the PC and automatically launch builds without requesting any user interaction. The various versions of *Stand Up!* (originally written for *WinAPE* but easily assembled by *basm*) were tested very quickly in this way.

Build rules are defined in a YAML file. Figure 2 depicts an extract of this file displayed in the graphical version of *bndbuild*. Clicking on M4 builds a snapshot and sends it to the CPC. The file *GPT\_LECRAN.DSK* is built if it doesn't exist, or if one of the files *src/etch4k.asm*, *bootstrap.o*, *etch\_header.o*, *etch\_a\_sketch.sym* or *etch.dsk* is newer. Three commands are executed to build it: the first builds the Amsdos file, the second builds a DSK, and the third copies it in. *bootstrap.o* is a dependency that needs to be built using *basm* if it doesn't exist, or if the files *src/bootstrap.asm*, *src/deshrink.asm*, *etch.o* and *etch\_a\_sketch.sym* are newer. *bndbuild* calculates the dependency graph so that it knows when to build a file based on the state of the files on the PC disk. This is nothing new for those familiar with make.

Since the release of *J'AI PÉ-TÉLÉCRAN*, *bndbuild* has strongly evolved. Syntax can be less verbose by using a Jinja-based template mechanism, variables can be provided at launch time to adapt the behavior of the rules (for example to specify the IP address of a M4 and replace the hardcoded value). It is able to download and to launch the mainstream emulators.



Figure 2: Graphical *bndbuild* after requesting to clean generated files

The *src/deshrink.asm* file can be seen as a dependency. Indeed, *Shrinkler* compresses this 4k file. Like in *rasm*, its cruncher is embedded within *basm*. So it's “free” to build a compressed version of the 4k: thumb up for crossdev ;)

*Arkos Tracker 2* is used for the music with its AKG player. An additional tool has been written to generate the data of the 4k and is presented later in this article.

Of all the tools used, *Shrinkler* is by far the most important. It's all about the compression ratio. Blueberry/Loonies is therefore the most important contributor to this 4k ;)

## ONLY SIZE MATTERS

4k corresponds to 4,096 bytes including the 128-byte Amsdos header. I think the Amstrad CPC is at a disadvantage compared with other platforms that have smaller headers (e.g., on the C64, there are 2 bytes of header for the loading address, then 16 bytes of content needed for the BASIC program that launches the binary). This is all the more frustrating as most of these bytes are not used by header<sup>3</sup>!

Thanks to the experience of having released other 4k intros on Amstrad CPC (*Glory Holes*, *Still the bests!*, *Stop that nyan cat!*, and *Wunderbar*), I have few reflexes that help reduce the size of the final executable. The size of the compressed executable should be optimized, not the size of the binary before compression: these two objectives can be contradictory! In no particular order, here are some aspects to test:

- Test different pairs of crunchers/uncrunchers routines (if *Shrinkler* is not selected because of its slow uncrunching, otherwise it's almost certain to be the winner). And above all, compress each

<sup>2</sup> <https://github.com/cpcsdk/rust.cpllib>

<sup>3</sup> [https://www.cpcwiki.eu/index.php?title=AMSDOS\\_Header](https://www.cpcwiki.eu/index.php?title=AMSDOS_Header)



version to keep track of progress and avoid unpleasant surprises. Such checks are facilitated by cross-assemblers which integrate compression directives.

- Include precomputed data rather than their data generation code. It's counterintuitive, but sometimes data compress better than their generator. Select the version conditionally.
- Make closer similar areas of code/data in memory. This can help the cruncher to detect identical patterns.
- Duplicate code instead of looping and conditional version selection.
- Rewrite functions in different ways to increase their similarity, even if this means adding unnecessary instructions.
- When a function is called only once, integrate it directly at the call point (saving a call and a ret). When a function is called several times, it can also be useful to duplicate the code (using macros helps in this case).
- Everything possible and unimaginable to generate similar byte sequences (whether code and / or data).

There are so many possible variants that it becomes complicated to test them all. This is problematic, especially knowing that choices relevant at a given moment may no longer be relevant several versions later. Using conditional assembly based on the value of certain assembly constants allows you to have a single code, which can be assembled/crunched into different versions to help you select the smallest.

For this 4k, I used an additional trick described by CheshireCat in her blog: put code in the Amsdos header. This way, you can save a few bytes by including code or data in the header. Unfortunately, the number of bytes usable for a 4k file is lower than the amount usable in a 1k file; that's still a gain. I have been told that using code and data is not safe when copying files. I have to admit that I have not checked at which point it is a solid trick.

*bas*m builds the Amsdos header (see *src/etch4k.asm*) on its own, calculating the checksum to be integrated using its loop management assembly directives as well as functions for reading already assembled memory zones.

We can see several things: the value of using tools such as *bndbuild*, as this type of project needs to be built in several stages, and the ability of *bas*m to calculate the header checksum in real time. Once again, thumbs up for crossdev.

In my projects, music has always been the most memory-hungry element. I think it systematically takes

up more than 1k after compression. Unlike Vanity's 4k, I haven't yet felt the need to specifically optimize the size of this part, even if it means no longer using a tracker.

## RENDERING ENGINE AND DATA STORAGE

In the end, there's nothing extraordinary about this 4k's rendering engine: it reads and decodes the coordinates of a pixel and displays it. Possibly several times per frame, so as not to be too slow, and in different colors to make a trace.

The interesting part is how to store and decode the data.

I tested different variants, accessible by changing an assembly constant (*SELECTED\_DATA\_ENCODING*). They encode plotter movements differently (which impacts the way data is stored) and decode them differently (which impacts the way data is read). Each of these variants has a different compression ratio, and I've naturally chosen the one that compresses the best: unsurprisingly, the stupidest variant (*DATA\_ENCODING3*) which uses the most uncompressed memory space, but contains the most repetitions on the data side and also uses fewer instructions on the code side! For example, we can clearly see here the absence of decoding when reading the plotter's movement in the figure 3 (after rereading the article, I realize that there are even 2 unnecessary instructions in the version selected :() ).

```
; store the movement
ld b,a
if SELECTED_DATA_ENCODING==DATA_ENCODING1
    and %1111
else if SELECTED_DATA_ENCODING==DATA_ENCODING2
    and %111
else if SELECTED_DATA_ENCODING==DATA_ENCODING3
    ; there is stricly nothing to do
else
    fail "unhandled case"
endif
ld (.movement),a

; store the amount
ld a,b
if SELECTED_DATA_ENCODING==DATA_ENCODING1
    srl a : srl a : srl a : srl a
else if SELECTED_DATA_ENCODING==DATA_ENCODING2
    srl a : srl a : srl a
else if SELECTED_DATA_ENCODING==DATA_ENCODING3
    ; no concept of repetition
else
    fail "unhandled case"
endif
```

Figure 3: Data deciphering differences among versions



Data encoding is done at assembly time by *basm*, and not during data generation by the tool presented in the next section. This makes it possible to convert an image once and generate different variants of the executable. This was made possible by the use of macros and assembly variables, which enabled me to create a state machine that “reads” data (i.e., executes macros that describe an image), encodes it according to the chosen configuration, and stores it by generating the appropriate DB directives according to the state of the automata. Figure 4 shows an extract of the code generated to describe an image. `src/commands.asm` is the file that defines the macros used to describe the plotter's movements (U, D, DR, ...) and the actions at the start (START, which initializes the automata) and end (STOP, which ensures that all encoded data has been output) of the image. This is surely one of the most interesting parts of this project, but I don't have enough space to provide more details. I encourage you to take a look at the code and let me know what you think.

```
include once "commands.asm"
START 144, 96
  U 7
  D 5
  DR 1
  UR 3
  ...
  DR 1
  U 1
STOP (void)
```

Figure 4: Image description with dedicated macros

I think this is the first time I've generated data using this kind of state machine on the assembler side, and I imagine I'll be using such a method later on to generate code (rather than using tools outside the assembler).

The most important thing to remember about this section is that the display part is rather simple and naive, and that several versions can be built according to assembly constants. The big negative is assembly speed; off the top of my head, I think a large image takes 1 min to assemble, due to the code generation part of the macros followed by their parsing. This is overcome by assembling the images individually and including their binary in the main program (once again, *bndbuild* makes this trivial).

#### DATA GENERATION

We have seen that figure 4 shows an extract of the code/macro representing an image to be displayed (the list of plotter motion commands). This representation is generated by a specifically created tool (see source `convert/src/main.rs`) whose behavior is described as follows.



Figure 5: A nice test made by Voxfreax that has not been integrated in the final 4k

The principle is as follows: given a black-and-white image (such as the one in Figure 5), choose the starting point of a drawing, and iteratively, the following contiguous pixels in such a way that the plotter passes through all the black pixels in the image, minimizing the number of passes per pixel (for obvious reasons of display and storage time; ideally, 1 pass per pixel). It turns out that there are no standard algorithms for solving this problem efficiently. So we have to be creative to obtain an acceptable approximation.

The figure 6 illustrates the general principle of image conversion on a simple example: a T in a 5x4 pixel grid.

The first step is to abstract the image with a graph where each node represents a pixel, and two nodes are connected if the two related pixels are neighbors in all 8 directions. Here the graph is represented graphically with a node-link view (each pixel is represented by a disk, 2 adjacent pixels are connected by a line), and numerically with its adjacency matrix (each row/column represents a pixel, when 2 pixels are neighbors, there is 1 in the corresponding cell).

In a second step, the path length (the number of edges to be traversed) between each pair of nodes is calculated. By definition, the graph is connected (otherwise the image is not compatible with the concept of the 4k): there is a finite distance between each pixel of the image to be drawn. Here, I've represented the distance matrix between each pair of nodes and a node-link view of the complete graph, which displays the size of an edge as a function of the path length between the two nodes.

In a third step, a method for solving the traveling salesman problem is applied: the aim is to find a path that passes through all the nodes once and only once, and returns to the starting point. The cumulative distance (i.e., number of pixels) between all the



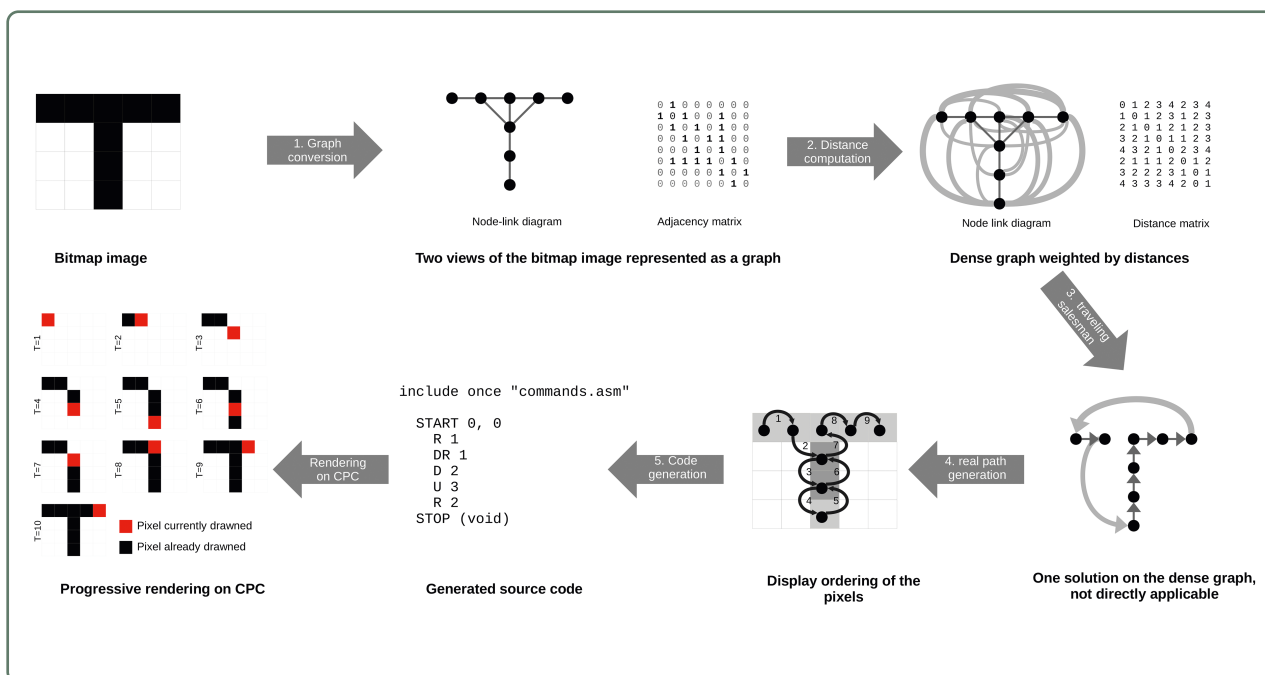


Figure 6: Illustration of conversion steps for a single image

nodes of the best path must be as low as possible. A solution exists, since the graph is complete; however, this best solution is not necessarily achievable by solving methods; approximations will be collected.

A fourth step transforms this path on the complete graph into one compatible with our constraints on the image graph. We can see that the first move is correct: the plotter can go from (0,0) to (1,0). However, the second is not: the plotter cannot go from (1,0) to (2,3), as these pixels are not adjacent; in this case, this displacement is replaced by the shortest path connecting these two nodes in the graph representing the image: [(1,0), (2,1), (2,2), (2,3)]. Because of these replacements, some nodes/pixels may be crossed several times by the plotter. We do not care of the portion from the last node to the first one, it is therefore not maintained.

The final step is to convert the resulting path into instructions that can be interpreted by the assembler: calls to image description macros (cf Figure 4).

The result is not optimal, because state-of-the-art algorithms approximate the best path, there are bugs in my code, and the graph generation can be improved. To get around these limitations, the images are converted piece by piece: the basic image is cut into logical subsets that have to be displayed atomically, and the tool takes care of starting a piece at the end of the previous one, while filtering out any pixels already displayed. In this way, the plotter passes over the same nodes much less often, which is more pleasing to the eye and compresses better. Slicing is driven by image structure and converter defects. As

resolution methods are stochastic, the result is different from one conversion to the next; for example, when writing this article, the generated executable was higher than 4kb!

The calculation time and carbon footprint associated with generating this 4k are monstrous. On a machine with 20 processing cores at 4.7653 GHz, it takes 40 minutes to build the intro from scratch (most of the time being taken up by data management, not z80 assembling that is negligible). It is interesting to see how such 4k could have been produced (if we disregard *Shrinkler*) in native development on CPC (it seems impossible, but who knows...).

#### ON THE MUSIC SIDE BY TARGHAN/ARKOS

As always when working on size-restrained productions, I made a song that would be made of as most repetitive tracks as possible, using the trick of transposition present in *Arkos Tracker*: the track is encoded once, but used multiple times with a few semitones up or down.

I actually had a lot of ideas for the song, but only the last ones were used: I had found this Ben Daglish kind of melodic bass-lines, and with another melody added on top of it, considered the music was full enough and didn't need anything else. Plus, it matched more or less the duration of the demo itself, and would save a substantial amount of memory.

I tried to use as few effects as possible to save some memory, and as simple sounds as possible, making sure the song would still have appeal without the multilayered arpeggio/hardware sound tricks I often use.



Using *Arkos Tracker* as a composing tool was the logical way to go for me, being its creator, and mostly because I find it very user-friendly (hehe). The provided players are, of course, optimized, especially considering they adapt to the song itself, but even the AKM player is still bigger than CNGSoft's *Chip'n'Sfx* player, especially when shrinkled. Even though it is a burden to use his software (due to his not-so-friendly user interface, and limited sound scape), I would have converted (manually) my song if Krusty had told me to do so, just like I did for Toms' *Daymo of the tentacle*. Fortunately for me, he didn't and it saved me a few hours of work.

#### SUPPLEMENTARY DETAILS

Erasing the telescreen shakes up the screen, playing with the R3 of the CRTC without compensating that because of memory constraints (and skills ;)). This part is mandatory to get closer to the behavior of the real object. This is not problematic with a real-world CTM, but video capture on standard hardware doesn't work: for example, the OSCC goes to safety when the screen is completely black for a few seconds.

It has been necessary to go back and forth on the graphs several times to increase their compression ratio, and display several of them. There aren't many, but it's still more than was expected a few weeks before submission.

#### CONCLUSION

This article tried to explain how *J'AI PÉ-TÉLÉCRAN* was built. Only few details have been provided, but you have access to the sources to look at it in more detail. You are encouraged to do so, and to point out any errors or possible improvements.

Few 4k demos exist on Amstrad CPC. It's hard to compete with the latest 4k Vanity or Overlanders, but there are always interesting things to do, provided you have the right ideas. ■

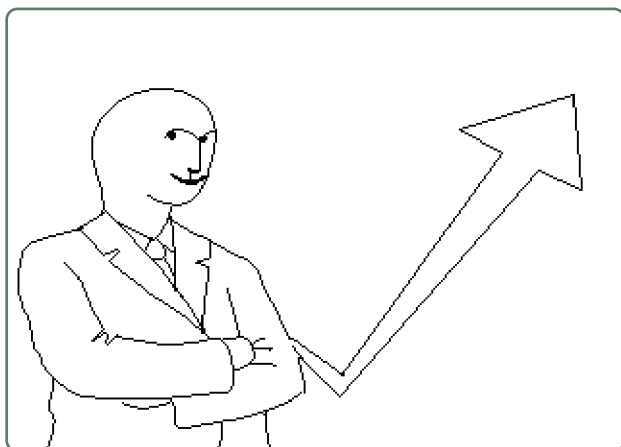
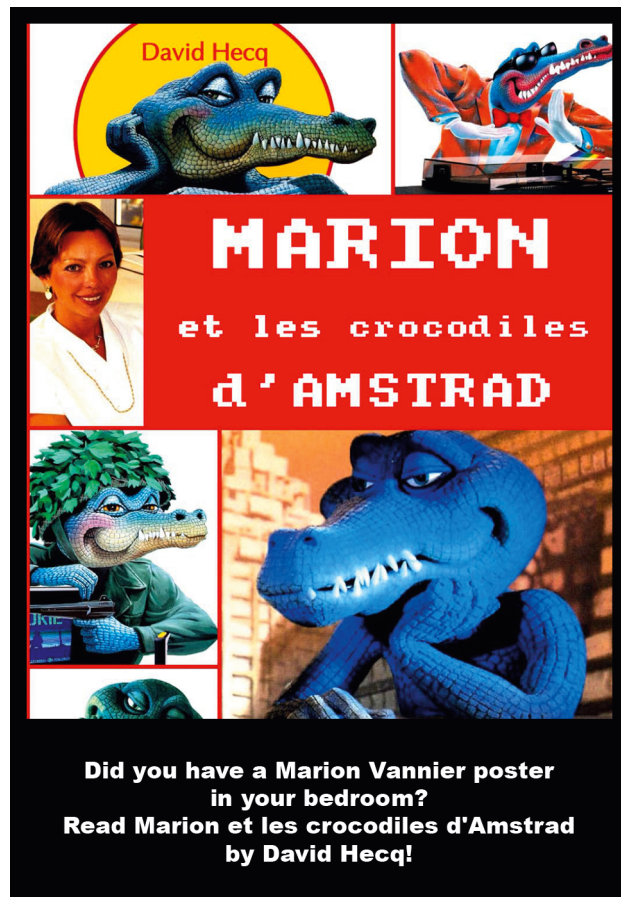


Figure 7: One of the first tries of Exocet







# A SHARP PIPE STARTS WITH YOURSELF

We'll be breaking down the general principles of a music player and looking at how *Ayane's* player manages to be so compact and fast without sacrificing any of its capabilities. If we can have a few laughs along the way, that'll be at least something.

BY MADRAM

Characters (in order of appearance):

- The Narrator, The Path of Wisdom
- Layane: the village tart
- E-dredon: polygamous, music-loving mayor
- Hicks: conscientious family man
- Roudoudou: nonchalant villager
- Targhan: big-hearted rebel
- Zik: genius bard
- Tony: the choirmaster
- Offset: evil genius on sabbatical

## COMPACT, FAST, POWERFUL. THREE CHOICES MAXIMUM!

The great Breton “pick one” means that you generally improve one aspect to the detriment of the others. However, after many walks in the woods, I've come across a player who optimises all these aspects at the same time. I have the honour and the thankless task of explaining how.

## PREMIER PRINCIPLES OF A PLAYER (PPP)

Unlike a score, which describes a piece without breaking it down, a tracker module is typically broken down into patterns, which I'll call phrases in homage to Khrisna and to denote their atomic aspect.

While some software programs use the term pattern to refer to all tracks for a specified unit of time, a phrase is here an isolated monophonic sequence that can be placed as often as desired on any track. A phrase is composed (ah!) of lines. Typically 64 lines to represent 4 bars in 4 : 4, each line representing a sixteenth note. Where a score has to multiply the use of notations to indicate the parts to be repeated, a sequencer will use a list of phrases to be played. The following terms are encountered: song list, song map or squirting if the wrong publication has been consulted.

## PLAYING A PHRASE: DELAY, AKA TICKS PER LINE

In practice, moving from one line to another routinely requires a counter initialised with a value called delay (at least in Cologne). Sometimes the term “speed” is used, which is ridiculous, because the higher the value, the slower it is. Ridiculousness does not kill (except in Japan), so let's move on.

A majority of musics uses a delay of 6, so that you move on to the next line every 6 ticks (a term paying homage to the ticking of the clock, representing a discreet but concrete time and the ineluctable march towards your physical death). 1 tick = 1 frame = ~20 ms.



For a playback at 50.08 Hz (typical on CPC), still considering that a quarter note takes up 4 lines:

- Delay 4 → Tempo 188 BPM  
(60 seconds × 50.08 Hz / (4 lines × 4 ticks))
- Delay 5 → Tempo 150 BPM  
(60 seconds × 50.08 Hz / (4 lines × 5 ticks))
- Delay 6 → Tempo 125 BPM
- Delay 7 → Tempo 107 BPM
- Delay 8 → Tempo 94 BPM

The notion of delay prohibits shorter or non-multiple durations. Unless, of course, you change the delay along the way. If the value is shared by all channels, imagine how tedious this becomes. Some professional trackers offer other commands to get around this limitation, but in my opinion this remains far from ideal.

#### IDEA 0: ENCODING THE DURATION

*Ayane* takes a completely different approach: it encodes the duration of each note. So much for triplets, heterometric syncopation and all that other nonsense. As soon as the note lasts more than one “line”, you save memory and execution time, because the counter is only reset when absolutely necessary. Even though silence after E-dredon is still E-dredon, there's no need to manage empty lines.

In terms of memory footprint, if all the lines in your phrase are filled, specifying the delay for each one seems like a waste of resources, enough to make Hicks sweat, but not Roudoudou. This mess is made up for in the lighter phrases. In any case, such filling generally compensates for a lack of power on the part of the player, and there are alternatives (see my free epub for a limited time).

In terms of performance, a global delay may be more economical, but it prevents certain musical offerings, such as playing the same phrase on two channels offset by one tick, creating a reverberation effect in front of your stunned ears. I choose power. It doesn't matter much in terms of CO2 emissions.

#### IDEA 1: MAKE A DATE WITH THE NOTE

I mentioned a counter to manage the progress of the phrase. Since each phrase is independent, that's as many counters as you'd need to update at each iteration. Forget it!

Instead, *Ayane* uses an elegant rendez-vous system. There's just one global counter (which increments to give the tick index), and for each channel, the time of the incoming rendez-vous is compared with this chronometer. When the appointment arrives, after treatment and a good shower, you add the duration to get the time of the next appointment.

The same principle is used for sequencing. Immediate benefits:

- phrase durations are independent of each other and from one channel to another. So you can repeat a short motif on one channel, while playing another of slightly different length on another (polyrhythm, for your chiptune covers of King Giz-zard and the Lizard Wizard).
- free placement (i.e. the reverb example above, and also the possibility of interrupting a phrase at will, in a very natural way). What's more, unless otherwise specified, a phrase will loop back on itself. You don't have to type in your ostinato as often as it appears. Bringing this 3000 year old innovation to CPC earned me a letter of congratulations from the heirs of Maurice Ravel.

So there are two types of appointments: notes and phrases. Actually, three, but you're too young. See you in seven paragraphs.

For the second type, my brain opted for a 16-bit counter, considering that a 64-lines phrase with virtual delay 6 takes  $64 \times 6 = 384$  ticks, which is apparently more than 256. This is not necessarily a happy choice, as *Ayane*'s sequencer encourages decomposition into more nuclear phrases.

What's more, with 8 bits there would be no limitations, because waits beyond 256 can be encoded through null intermediate events that have taken place, just as you can count down 12 minutes with a clepsydra that runs out in 6 (the average duration of ejaculation for pigs).

At some point you have to make a choice, otherwise you'll end up like Grimmy, drinking rum and having your toes pinched.

It's always possible to revisit this choice when compiling the module, to scrape together a bit more in case of extreme necessity. Not this weekend.

#### STATE OF CONSCIOUSNESS

The routine needs to remember where it stands from one call to the next. Track pointer, phrase pointer and current note, volume, instrument and effect. Pointer to instrument if a table is used, otherwise a bed will do.

We call the set of these variables “state”. This is partly because we like to show off, and partly because naming a concept makes it easier to understand, manipulate, communicate and juggle with, and more so if you are in good shape.



To update one of these variables, the quickest and most direct way (5 microseconds at sea level) is something like:

```
ld (phrase_pointer),h1
```

Yet there's a problem with this solid, familiar code. Stop reading and take a guess. [Answer:] Since three channels require three pointers, with static addresses as in the actually sheepish example above, the code has to be duplicated. While this is still a possibility to consider for ultimate speed, we prefer a less greedy solution. Therefore, most players use indexed addressing:

```
; IX points successively to the structures
; representing the state of each channel.
ld (ix+phrase_pointer),l
ld (ix+phrase_pointer+1),h
```

That's 10 microseconds and 6 bytes. To save space and execution time, you can fit the data associated with the pointer into an aligned 256-bytes section (Vaseline not supplied). From then on, no need to update the most significant bytes. That's great.

### IDEA 3: JUST IN TIME TO PLEASE YOU

I've recruited the good old stack pointer to manage the state. I don't like being interrupted anyway.

An update boils down to this:

```
push h1 ; update
pop h1 ; go to next field
```

Not only does it take only 7 microseconds and 2 bytes, but the code no longer depends on the field, making it highly compactable.

### IDEA 3: EFFECT = ROUTINE ADDRESS

I haven't told you the best part yet: one RET is all you need to manage an effect or the current instrument. Most effects only need one 16-bit parameter. For example:

- table pointer
- 8-bit portamento (current offset and increment to be applied)

So, to manage N simultaneous effects, our stack will have the following structure:

```
N ** [
  WORD fx ; routine address
  WORD param ; parameter we just discussed
]
WORD instrument ; instr. management routine
WORD param
```

Two implications immediately come to mind (if you are careful with your diet and sleep):

1/ Since you have to consume the stack, where there's no effect, you must still jump into a routine that does nothing but consume the parameter. The absence of effect has a price:

```
fx_null
pop h1 ; skip param
ret ; go to next fx or instrument
```

This is no more expensive than the cost of a test and a conditional jump. Ayane allows 4 simultaneous effects per phrase. If you use only 2 at a time, the compilation will take this into account and optimise the whole thing.

2/ Any additional parameter generate an overhead. Imagine two 16-bit parameters. It won't fit. So our 16-bit field will just be a pointer to the real parameters. This is expensive:

- when the effect is initialised, because we have to dynamically allocate these two words to a fresh location (so as not to interfere with the same effect triggered on another track).
- when the effect is executed, because of the indirection; you can no longer just use a PUSH to update the state. However, there's nothing to stop you trying a putsch to replace the state. Wink wink, nudge nudge, follow my lead. "Everyone has a moral responsibility to disobey unjust laws."

It's all about compromise. As it happens, this is better than reserving the maximum number of words, or allowing a dynamic number of parameters in the stack itself, which would drastically complicate management when you need to change or cut an effect. Don't take my word (ah!) for it. It depends on the music. We know the song: (word + music) = song. And I'm not even warmed up yet.

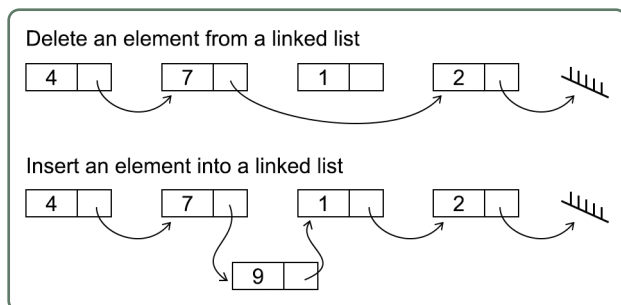
### MODULAR INSTRUMENT: A WRONG GOOD IDEA?

The jury is divided.

### CHAINS THAT SET YOU FREE

A sequence of data arranged in a row is called a table (in other languages, an array). Let's say our table describes the evolution of an instrument. How do you insert an element in the middle? You have to shift all the following elements and make sure you correct the looping. A data structure makes linking and deleting easier: linked lists. Each element is associated with a pointer to the next element. The elements no longer need to be contiguous, and an insert only requires 2 pointers to be corrected.





This insertion problem doesn't seem to concern the player<sup>1</sup>, only the tracker. However, the linked list also solves the looping problem! The last link points to the loopback element. What's more, if the data fits into a section of &100, the most significant byte remains constant, and a single instruction is enough to iterate through the instrument indefinitely, without any tests or counters.

```
ld l,(hl) ; next element or loop!
```

Fun fact: the firmware uses a linked list to browse the memory zones of arbitrary sizes reserved by each ROM. This isn't so much for ease of insertion, but to find where each one starts. An index table would have been less memory efficient, as 16 words would have had to be automatically reserved, even without any external ROM.

Is placing the link before or after the data a matter of taste? No, it isn't! There are several advantages to starting with the link:

- some routines become generic. For example, positioning at the end of the list. As long as the links are prefixed, you don't need to know the size of the attached data.
- shit, I forgot the other advantage.

Here's what such a list looks like:

```
item0 byte item1 and &ff : word 0 ; link to item 1 and first value
item1 byte item2 and &ff : word 4
item2 byte item3 and &ff : word 0
item3 byte item0 and &ff : word 7 ; loop to item 0
```

There's nothing to stop it being reused as a list of 8-bit values, because there's no need to go through all the data. It's rare for data to claim such versatility, but when it does, I'm delighted. Its minor drawbacks are that it takes up a lot of space and the links are poorly compressed. So we prefer to encode the relative distance:

```
item0 byte item1-$ : word 0 ; link : +3
item1 byte item2-$ : word 4 ; link : +3
item2 byte item3-$ : word 0 ; link : +3
item3 byte item0-$ and &ff : word 7 ; link : -9
```

## PROGRESS THROUGH THE TABLE WITHOUT POINTERS OR LINKS

There is a way to progress in a raw table without having to update the state. All you have to do is reuse the global counter, modulo the length of the table  $N$  (especially for powers of 2, since a simple  $\text{AND } N-1$  is all you need to do the modulo). There is less control over the start, since the first index calculated in this way when the table is triggered is not necessarily 0. Clever users will find this to their advantage.

Bonus. Consider what happens if, for example,  $N = 6$ , which is not a power of 2? An  $\text{AND } 5$  applied to the counter gives the following sequence 0 1 0 1 4 5 4 5 and then loops. Inputs 2 and 3 are never scanned through. This is a cheap way of generating a new sequence in the context of sizecoding.

## IDEA 5: ONLY PAY FOR WHAT YOU USE

Remember that sequencing is not done through a table, but through events. The triggering of transposition (basic modulation where all notes are shifted by the same amount) is based on the same principle. Again, power at low price! As long as there's no transposition, there's nothing to encode (which puts me in the running for the Ravel medal).

This is the third type of appointment mentioned above. You forgot it, I didn't. Not only is it economical, because you don't have to repeat the transposition value as long as it doesn't change, but it's also more versatile: you can transpose in the middle of a phrase.

## IN BULK

### STAGGERING TASKS

If you want to keep a minimal constant execution time, you have to optimise for the worst-case scenario. Vibrato is one of the most time-consuming effects (at least in Brazil). The very act of triggering the

effect adds to the bill, since the variables have to be initialised (speed counter, offset). Ayane will stop after this initialisation and concretely use

the vibrato from the next tick. Apart from Zik, no one will notice the trick. You can think of the concept of "delayed gratification", but that's not really the point.

## DELTA

A well-known trick along the Leyre is to update only those AY registers whose value has changed. Unless all the registers change at the same time, which only happens in Tony's most psychedelic works after an excessive Swedish exile, the extra cost of the tests immediately pays for itself.

<sup>1</sup> Except if evolving instrument for generative music.



The routine is no longer stable in machine time! To fix that, all it has to do is counting the number of registers updated and compensate for the maximum number of changes (depends on the music).

#### STAGGERING TASKS, THE RETURN

If you want a minimal constant execution time, you have to optimise the global worst-case scenario! We rarely have 3 simultaneous vibrato effects, it is the accumulations that create an unfortunate peak.

This typically happens when a phrase is changed: management of the change itself, possible transposition, new note, new instrument and new effect on the 3 channels, which also leads to numerous changes in the registers. To avoid this potentially disastrous accumulation, we generate the registers one frame before sending them to the PSG. Sound double buffering!

Hard-compensating the conditional branches of your routine for the worst case, as indicated in 64 NOPs #1, corresponds to a pessimistic vision where all worst cases would occur simultaneously, which hardly ever happens in practice. My preference is for a routine that returns the time it takes. The more time-consuming branches add the extra execution time to a counter. This is a generalisation of the PSG conditional reprogramming trick. This opens up another way of optimising a demo that requires a constant time player. There's no need to compensate against the global maximum time. If an effect is greedy, we look for the maximum time taken by the player during that effect, and then compensate in regard to that local maximum. However, if an effect is very demanding, the most efficient approach is to buffer the registers for as many frames as necessary.

#### REGISTERS ASSIGNED TO SPECIFIC ROLES

Rather than resorting to cumbersome memory accesses, Ayane's player manipulates throughout:

- A = current note
- DE = current pitch (initialised at 0)

By way of illustration, and to summarise the various ideas involved, the arpeggio routine simply consists of:

```
pop hl      ; table pointer
ld l,(hl)   ; next element
push hl     ; update for next iteration
inc hl      ; skip link
add (hl)
pop hl      ; skip pointer
ret         ; next effect
```

I know you like technical details, so I'll stop here.

#### CONCLUSION

Generic principles to follow:

- As simple as possible, but not simpler (generally attributed to Albert Einstein, who liked to borrow ideas without necessarily crediting the source. Source: Einstein and Poincaré - Jean-Paul Auffray. In a lecture in 1933, Albert did express a similar idea, but in a more complicated way!).
- Store in memory only when absolutely necessary (and therefore reserve as many registers as possible for the main variables). See dot record for a completely different application of this principle.

I'll tell you the truth, it was only by rewriting the code in its entirety several times that I came up with a simple version. And simple isn't easy. A lucky combination of decisions unlocked the chain of optimisations.

#### TO GO FURTHER

##### STATE OF THE ART SPEED

It's hard to do better than *FAP*. But possible. But useful?

##### STATE OF THE ART IN MEMORY USAGE

Hard to beat *Chip'n'Sfx*. Mathematical generation of periods? Dubious, the low octave only takes 12 words, the others are derived by simply dividing by 2 (and rounding to the nearest integer).

##### STATE OF THE ART IN COMPRESSIBILITY

*Ayane's* player + data set potentially takes up more space than its *Chip'n'Sfx* counterpart, but compresses much better!

For the 4k/1k I contributed to, an ad hoc player exploits the following constraints

- no volume table, only the hard envelope.
- less than 16 different phrases (whose index is encoded in 1 quartet).
- sequencing and transposition in a classical isolated table (to increase local redundancy).
- indirection: the phrase is composed of a 4-bit code per line, the code indicating the note and the instrument. These codes are redefined for each phrase by decreasing frequency of use. ■





# EVERYBODY LOVES TUNE

I am not the most prolific music creator on the CPC, but many people have asked me for advice. Many? Well, several... Let's say two people, which is a lot for the CPC community! So, here I am. Hopefully, you will take away some hints for your own creations.

By ZIK/FUTURS'

Of course, there is no universal and immutable recipe to create music. Do not be disappointed, there are many different situations and each musician has his own methods. What is fixed, however, are the capabilities of the CPC sound generator (the stock CPC, without expansion interface). Let's start with a quick reminder, you will find the main characteristics in the insert.

These characteristics naturally place us in the musical style of chiptune. Sample-based music can broaden the range of possibilities, but I will not cover that here, you may refer to the previous article. In this article I am going to assume the use of a music tracker, because that is the majority case these days. But most remarks also apply to other situations.

## ONE DIRECTION

When you are working towards a specific project, you will often have a target duration for your music and a deadline. Good! But I think it is a good practice to also inquire early about the technical constraints to avoid unforeseen late reworks. The constraints can be quite restrictive for a demo, but are usually more relaxed for a game.

Typical constraints may be the maximum memory footprint in balance with CPU time. The music player commonly used for games decodes patterns and instruments on the fly. It allows the replay routine and some data like instruments to be reused for several tunes. In contrast, a demo usually has a strong focus on CPU time optimization. In this situation, the music player of choice is based on sending raw data to the sound generator. Its main task is to decompress this data. Pattern and instrument data have already been pre-processed.

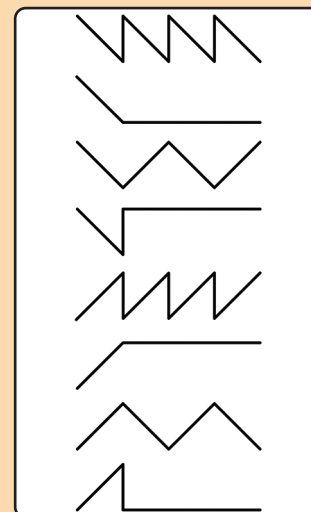
Musical variations and flourishes, subtleties added to instruments, glides and portamento, the use of many different instruments... All this will degrade the compression ratio because it results in more register value changes. So, beware and check regularly by doing intermediate exports, so that you can make adjustments as you go along. And also, know in advance which player flavor will be used, they do not all have the same capabilities.

The replay frequency will be 50 Hz in the vast majority of cases, so much that it is implicit and not even mentioned. This makes the resolution rather poor in terms of tempo choice. It can be quite annoying.



### MAIN CHARACTERISTICS AND LIMITATIONS OF THE AY3-8912 ON CPC (AUDIO PART)

- 3 square wave tone generators with programmable volume and period (fixed 50% duty cycle)
- 1 single noise generator (with selectable frequency, mix on/off per channel, channel volume applies)
- 1 single volume envelope generator
  - programmable period
  - programmable envelope shape with single-cycle or continuous shapes (see picture)
  - can generate a tone with triangle or sawtooth shape by selecting continuous & a short period
  - possible to combine with the square wave generator output to create more complex sounds
  - on/off per channel, overrides the channel volume
  - volume envelope covers the full range from 0 to 15 (no range setting)
- on CPC: 100% stereo separation Left/Right + Mid channels



Some musical styles do not sound as good if played too slow or too fast. This constraint on tempo also means you have to make compromises if you want to combine binary and ternary rhythms. You can sometimes get around this by using a trick like changing the tempo along the pattern, even at each pattern step like when you want to apply tempo groove.

#### WHERE TO START?

Okay, now that the technical framework has been established, it is time to start thinking about the music itself. For my part, I prefer to have a defined scope within which to work. It feels like a more inspiring setting to me. Knowing the overall duration, if it should loop, if a particular style or mood is expected a priori are important. Should it be ambient music or something more rhythmic and paced? Should it have multiple sections? Questions like these. Then it is up to you to decide whether you go for a thematic, melodic style, or ambient, or a rhythmic one... Of course, it can be all of the above!

Write down your ideas as they come, to avoid losing them. For this you should use a means that is immediate and natural for you, like a dictaphone, writing on a musical staff or directly in the tracker editor if you prefer. For this last option, I like trackers that have a default built-in instrument at launch, just to do this.

Likewise, I think that doing some draft patterns is a great way to see if your idea works, what the final rendering might be and whether it needs to be developed further. You can start with an idea of melody, then harmonize it. You can also start with a bass

line, a rhythm, a chord progression... There are no rules. But personally, I need to have a fairly precise idea before I sit down in front of the music editor on my computer to lay down the first notes.

#### INSTRUMENTS FOR SUCCESS

On the question of instruments, my advice would be to create most of them from scratch with each new project. This is a nice way to match what you have in mind and find new inspiring things. This is also a good method to avoid losing your idea while browsing through dozens of instruments to find the one that fits your purpose. Instead, I prefer to create a rough version of the instrument at first, and refine it later when the pattern starts to fill.

The AY sound chip does not offer a wide variety of timbres. For very low tones, such as a trendy bass voice, our square wave tones are not very powerful, and can sound crushed and hollow. You can work on their attack if you want to make them sharper, double them on several channels, or use another waveform, such as sawtooth.

If you struggle to obtain the sound you are looking for, but know a song having a similar one, there is a helper tool in *Arkos Tracker* that is called YM analyzer. It is quite convenient to capture one part of a YM file to an instrument. You may start from this base and shape it to your goal. Remember, too, that music-editing software often comes with sample tunes. Reading through them is a good way to learn how to do it yourself. Another good exercise, which I enjoy occasionally, is to reproduce by ear as closely as you can an excerpt of a CPC music you like. Very instructive!



## PATTERN FACTORY

Having only three audio channels can be frustrating, because, let's say, you cannot easily add the counterpoint or the pads you wanted besides your melody. Probably, you would always like to have one more channel than the platform you are composing on offers! But for sure, having only three requires a few contortions. With noise only, you can give the illusion of a fourth channel, but this is not without constraints, since the volume of the channel applies to the noise. The tracker editing software I know of is not very practical for this, it is a little tedious. Of course, it is also possible to do without percussion at all, or very little, and leave more room for melodic lines. Why not?

To make more space and ultimately free up a channel, the method I often use is to write flat on several channels side by side, then combine them. It is easier than writing merged and interleaved voices at once. For example, write bass and drums on two separate channels, then combine them all on a single channel. Another example is to write chord progressions flat on three channels and then transform them into arpeggios on a single channel. In an editor like *Arkos Tracker*, you can add one PSG to the configuration to have supplemental tracks for your drafts. This trick is not perfect, but it might come in handy every now and then.

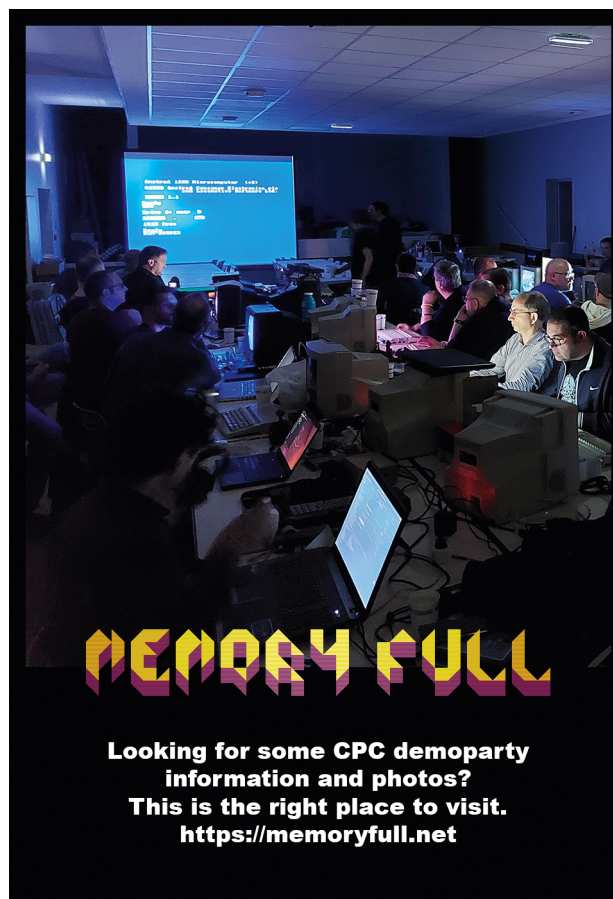
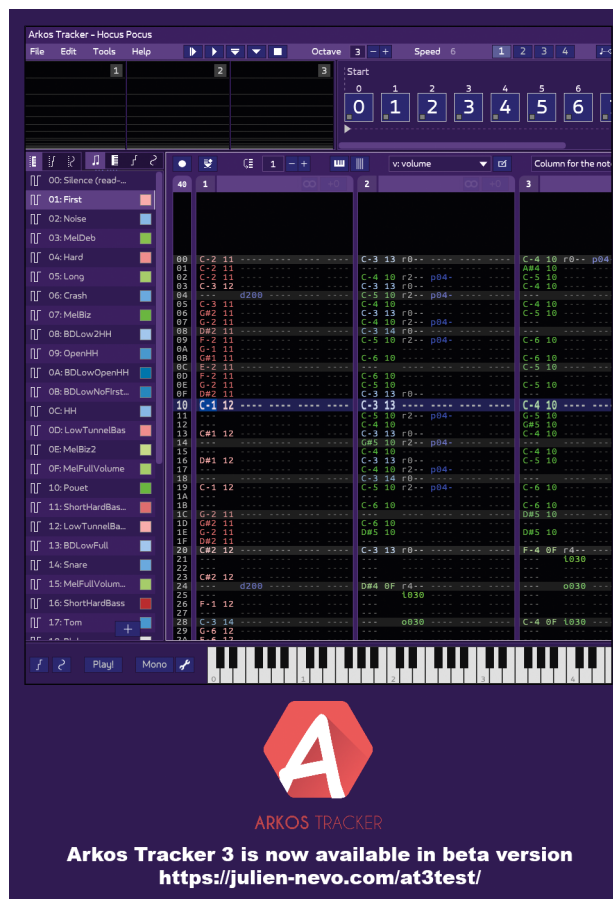
## WRAPPING UP

Throughout your work, I think it is quite beneficial to manage collisions on noise and volume envelope (see insert), at least to some extent to limit disappointments at export. The external player might not handle collisions and produce the exact same result you hear in the tracker editor. You have been warned!

It may seem counter-intuitive, but I try not to listen too much to the music I am writing. This is to avoid getting too used to it, with the risk of not having enough hindsight to rework the linking, change a musical line, delete elements... A good way to deal with this, too, if you have the time, is to put your work aside for a few days before going back to it and listening again with fresh ears.

When played on a big sound system, the AY high-pitched notes (even more so with some wild arpeggio) and also light noises can sound aggressive. Maybe think about this when adjusting mix volumes. Full stereo separation may also be a drawback, depending on the room.

If you would like to get started creating music on our good old computer, I hope the few tips and thoughts discussed in this article will help you find your way around. Now get to your favorite tracker! ■







# A SIMPLE AND FAST HASH ALGORITHM IN Z80 ASSEMBLER

Developing a spreadsheet application like *SymCalc* is incredibly fun because you constantly encounter problems you never thought of before. One such problem, for example, is determining the set of all cells that need to be automatically recalculated when the content of another cell changes. Another challenge is figuring out whether a cell is even occupied and where its data is located.

BY PRODATRON/SYMBIOSIS

## THE PROBLEM

At first glance, this might seem trivial, but it's crucial for the performance of the entire application. There are constant situations where you need to retrieve a cell's data - whether it's while moving the cursor around, copying cells, or during recalculations after a cell update, which could require searching through hundreds of cells. In operations affecting large portions of the entire table, it could even involve thousands of cells.

In *SymCalc*, a table is 64 columns by 252 rows, theoretically providing 16,128 possible cells. Of course, not all of them will be occupied at the same time - that would be far too many. The basic information alone (12 bytes per cell) would already require 200 KB. For performance reasons, all related cell data should reside in the same 64K bank. Currently, the maximum is therefore limited to 2,048 cells.

Now, let's say we have a table with 500 occupied cells. Each cell record stores its column and row. If we now move the cursor through the table, *SymCalc* needs to figure out each time what is under the cursor. The cells may have been created in completely random order, so in the worst case, all 500 records would need to be searched.

Using the following stupid algorithm:

```
; input -> e=column, d=row

    ld hl,cell_record_memory
    ld bc,cell_record_length-1
    ld ix,(cell_count)
    ; ... convert IX to IXL,IXH counter

loop  ld a,(hl)   ; 2
      inc hl      ; 2
      cp e        ; 1 5
      jr nz,next  ; 2/3
      ld a,(hl)   ; 2/0
      cp d        ; 1/0
      jr z,found  ; 2/0 7/3
next  add hl,bc   ; 3
      dec ixl     ; 2
      jr nz,loop  ; 3 8 -> 5+(7+3)/2+8=18

      dec ixh
      jr nz,loop
      ; ... not found

found ; ... found
```



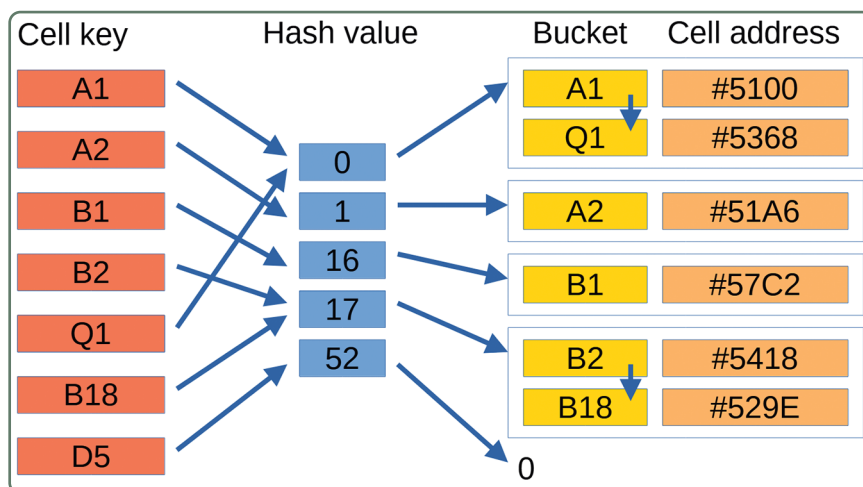
Without surrounding overhead, this takes 18 NOPs per record. With 500 cells, on average, 250 would need to be searched to find the correct one. If the cell is empty, however, all 500 records would need to be searched. Assuming an average of 300, each search would take  $18 \times 300 = 5,400$  NOPs, or 84 raster lines. This means we could perform a little less than 200 searches per second. For cursor movement, where only one cell is searched, this is sufficient. But recalculating a table, which may require hundreds of searches due to chained cell references, could take several seconds. Similarly, inserting whole columns or rows, where thousands of cells need to be checked for existence, could become very annoying without further optimizations.

Instead of the brute-force algorithm mentioned above, one could search the cells using an array of 16,128 entries. Each key (composed of column and row) would point to an entry containing either 0 or the address of the cell. This would make the search extremely fast, but the array would need to be almost 32 KB in size. While *SymCalc* is already memory-hungry, this might be overkill.

#### HASH-BASED SEARCH

I chose a hash algorithm because it seemed to be the best solution for the given circumstances. In a hash algorithm, the *long* key that uniquely identifies an element - in this case, the column and row - is first transformed into a *short* hash value. This hash value points to a much smaller array, the so-called hash table, which requires significantly less memory.

Since a hash value is shorter than the original key, it will inevitably happen that multiple keys will lead to the same hash value. This is called a "collision" and means they all point to the same place in the hash table. Therefore, the hash table can't directly reference the element but must instead point to a so-called bucket containing the pointers of all elements with that hash value.



Transforming a key into a hash value to find a bucket entry

16 entries per hash value

A bucket shouldn't get too large, meaning there shouldn't be too many collisions; otherwise, time is lost searching through the bucket entries. Hash values should therefore be generated in such a way that a well-balanced hash table is created, where each bucket contains a similar number of elements. We're in luck: a two-dimensional table that tends to be filled evenly from top to bottom and left to right is ideal for generating a hash value. We simply use the lower 4 bits of the column and the lower 4 bits of the row and combine them into an 8-bit hash value.

As a result, within a grid of  $16 \times 16$  filled cells, there will be no collisions, as each of the 256 cells will have its own hash value. A grid of  $16 \times 48$  filled cells would also be perfectly balanced, with three entries per hash value for a total of 768 cells. A less optimal case would be a grid of  $1 \times 252$  cells, where up to 16 entries could share the same hash value. The worst case occurs when only cells in rows and columns 1, 17, 33, and so on are occupied, creating up to 64 entries for a single hash value - the maximum for a  $64 \times 252$  table. Such a scenario doesn't make sense, so in practice, we can assume a well-balanced hash table.

Generating the hash value from the lower 4 bits of the column and row and finding the corresponding hash table entry is relatively easy in Z80 assembly (see source next page).

Now, the entry either contains 0 for the bucket size, indicating no cells with this hash value, and the search ends immediately. Or it contains the size of the bucket and the address of the first entry.



```

; input  -> l=column, h=row
; output -> hl=bucket size (0=no entries)
;         hl+256/512=address of first
;         bucket entry (low/high byte)

```

```

ld a,l          ; 1
rrca            ; 1
rrca            ; 1
rrca            ; 1
rrca            ; 1
and #f0         ; 2
ld l,a          ; 1
ld a,h          ; 1
and #0f         ; 2
or l            ; 1
ld l,a          ; 1  l=hash value
ld h,hash_tab/256 ; 2  15

```

Each bucket entry contains the element's key, its address, and the address of the next bucket entry, for a total of 6 bytes. Since entries can be added or removed, we must store them as a linked list rather than simply placing them sequentially.

The only task left is to compare the key in the bucket entry with the key of the sought element. If they match, we have found the element's address; if not, we proceed to the next bucket entry. But we already know it must be one of them.

The actual search loop is compact and fast:

```

; input  -> hl=bucket entry
;         c=column, b=row
; output -> de=element address

ld de,3

loop ld a,(hl) ; 2      check column
inc hl ; 2
cp c ; 1 5
jr nz,next ; 2/3/2
ld a,(hl) ; 2/0/2 check row
cp b ; 1/0/1
jr z,found ; 2/0/3 7/3/8
next add hl,de ; 3
ld a,(hl) ; 2
inc hl ; 2
ld h,(hl) ; 2
ld l,a ; 1
jr loop ; 3 13 -> 5+(7+3)/2+13
; -> 23 (not found)

found inc hl ; 2
ld e,(hl) ; 2
inc hl ; 2
ld d,(hl) ; 2 8 -> 5+8+8
; -> 21 (found)

```

Let's again assume we have 500 occupied cells. If the hash values are ideally distributed, we would have only two elements per bucket ( $500 / 256$ ). Being pessimistic, let's assume three entries, and we find the element at the second entry on average. The actual search would take  $23 + 21 = 44$  NOPs. Including hash value calculation and some overhead, we would take just over a raster line to find a cell among 500 - an 80-fold speedup, wow!

This approach comes at a price, though, since we must store as many bucket entries as there are possible cells. With 2,048 potential cells and 6 bytes per bucket entry, that's a hefty 12 KB. While it's a lot, it's still much less than the 32 KB mentioned earlier, and the search remains lightning-fast compared to the brute-force method. In *SymbOS*, switching between functions in different 64K RAM banks happens quite quickly, provided not too many parameters need to be passed, which is the case here. Therefore, the entire hash algorithm, including the 12K bucket list, can be placed in another bank, costing just one more raster line in overhead.

#### MANAGING BUCKET ENTRIES

As mentioned earlier, bucket entries are stored as linked lists, allowing elements to be inserted or removed without moving large amounts of data. When an element (and thus a bucket entry) is removed, it is simply unlinked from the list, meaning the previous entry no longer points to it but to the next one instead. It is marked as "free" by setting the element's address to 0.

When adding a new element, we first need to find the first free bucket entry, i.e., one with an element address of 0. To avoid searching the entire bucket list each time, we remember the first known free entry. In an empty list, this is the first entry. For adding a new entry, we start looking for a free entry from that point onward until reaching the end of the used memory. The next known free entry is then set just beyond the newly added one. Conversely, if an entry is removed and it is located before the first known free entry, the next known free entry now points to the removed one. This massively shortens the search for a free entry.

What do you think of the algorithm presented here? Does it make sense at all? Can you think of any optimizations, or would you implement it completely differently? Could it be made even faster without using more memory, or equally fast with less memory? Perhaps a search tree would be more suitable, and if so, why? This would be very interesting to know! ■

The complete hash library can be found at:

<https://github.com/Prodatron/symapp-symcalc/blob/main/App-SymCalc-Extend.asm> ("HASH ROUTINES")

It includes functions for initializing the hash table, adding new entries, removing existing entries, modifying entries, and of course, searching for entries.

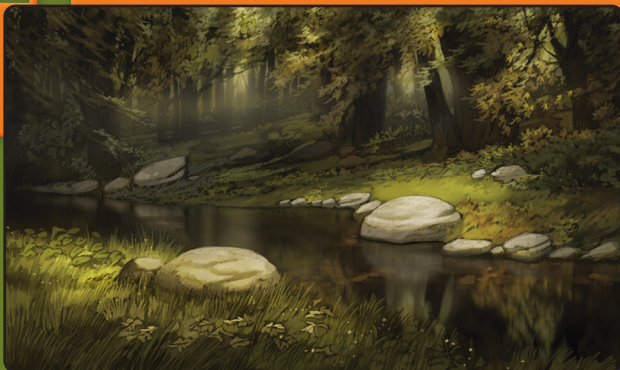




# I SEE YOUR TRUE COLORS SHINING THROUGH

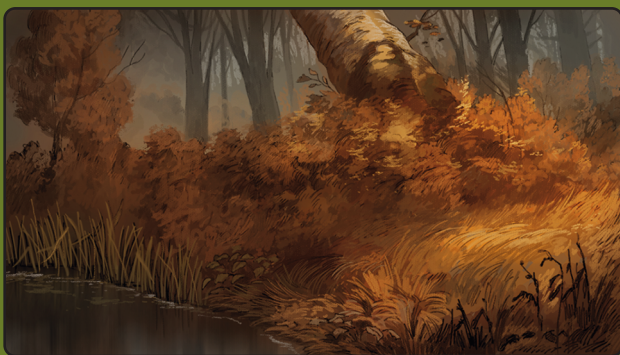
In the previous articles in this series, we explored the fundamental concepts of visual weight and contrast and their critical roles in creating compelling visuals on the Amstrad CPC. These topics have helped us understand how to manipulate shapes, brightness, and saturation to create effective image composition with the CPC's limited graphical capabilities. Now, it's time to dive into one of the most important aspects of visual design—color theory—and examine how it relates to the limited 27-color palette of the Amstrad CPC.

BY HWIKAA/PRALINE



## ORANGE CRUSH

I'm a child of Halloween. Because of (or thanks to?) that, my home is full of twisted pumpkins, black cats (one of which is alive), mysterious potions, antique grimoires and spell books, candles, candies, tricks and treats. I'm the happiest person in the world as soon as the first red dead leaf falls to the ground, and all of a sudden it's all spooky music and autumnal movies and shows until Christmas.



Among them, one of my all-time favorite Halloween-esque animated TV shows is *Over the Garden Wall*, created by showrunner Patrick McHale. For those of you who haven't watched it yet (which, of course, is unthinkable), it's a 10-part 2D animated series in which brothers Wirt and Greg have gotten lost in the woods and try to find their way home and escape the Beast, who rules this mysterious land called "The Unknown".



A huge part of the show's success is due to Art Director Nick Cross, who created a whimsical world set in an everlasting 19th-century American fall, surrounded by mist, eccentric characters, and nonsensical, sometimes spooky situations. Here are a few examples of the fantastic work he and his team delivered (you can find more on his Tumblr, [@ncrossanimation](#).)

*"All right, Hwikaa, that's fantastic. Now, will you tell us where exactly you're going with this?"*

Of course, my lovely audience. Let's dive into it.



## FIELDS OF GOLD



This is one of the backgrounds Cross painted for the opening of the show, and it's a callback to the second episode, *Hard Times at the Huskin' Bee*.

If we were to study this painting, we'd likely pick out some desaturated oranges, grays and yellows, as well as a muted light blue for the sky, with more saturated highlights for the road and the pumpkins.

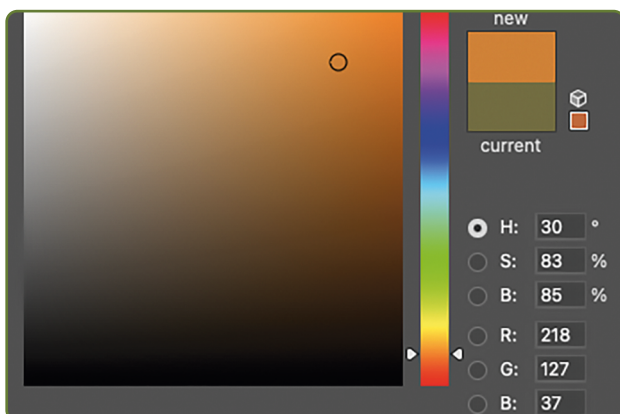
Let's give it a try.



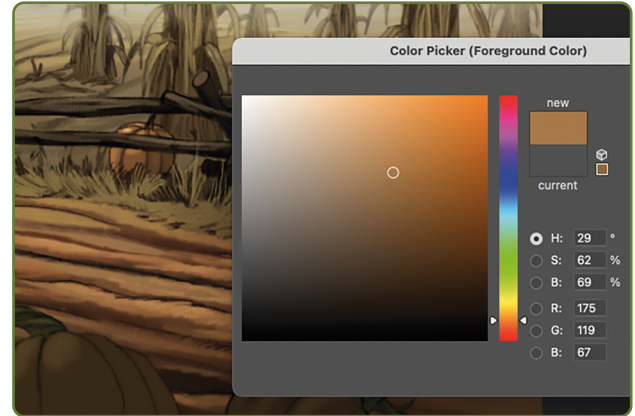
Ouch. That didn't turn out really well. Obviously, my colors are wrong—too bright, too saturated, and not working together. It looks like a patchwork, with each color competing for attention.

All right then. I'm not gonna fiddle around forever. Let's cheat and color-pick directly from the original painting using the eyedropper tool.

This is the kind of orange I assumed the road was:

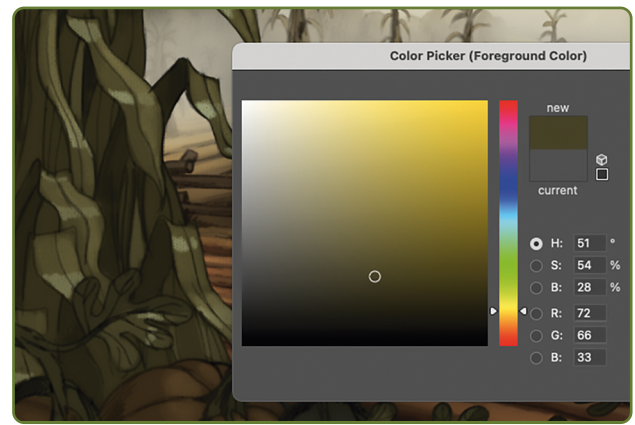


But this is the actual color used in the painting:



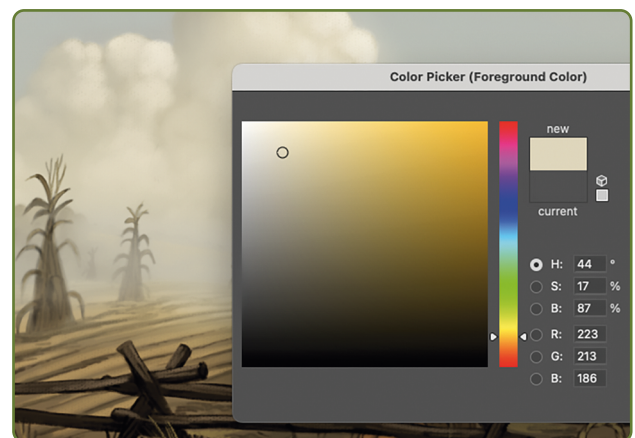
Although I guessed the **hue** fairly well (an orange around 30°), I saw it as about 20% brighter and more saturated than it actually is.

OK. Let's keep exploring. What kind of green are the corn leaves in the foreground?



Now that's weird. Like, really weird. They're not green at all. They're... yellow. A dark, desaturated yellow.

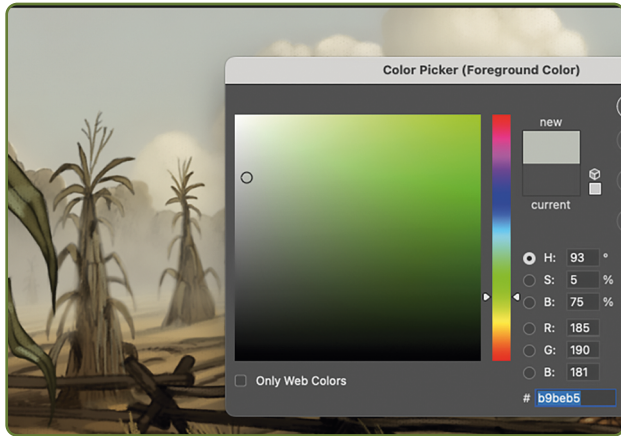
Wait, what? What about the corn field? And the clouds?





Yellow too. All the colors I'm picking from the painting seem to be oranges and yellows, in the hue range of 20° to 45°.

Hold on. What about the sky, then? Are you going to tell me that this light blue sky isn't light blue at all? Hahahahaha! Haha... Ha...



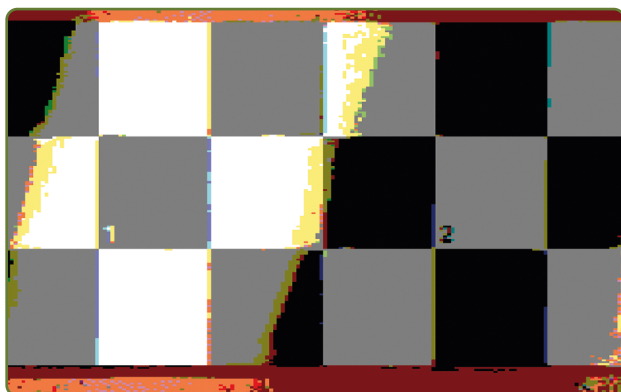
Holy macaroni. It's... green.

Remember that strawberry picture I showed at the end of the last article? (If not, now's the time to grab your copy of *64NOPs #2!*) This is the exact same principle I'm demonstrating here, and it's called **color relativity**.

#### GREY STREET

Color relativity is the idea that colors don't exist in isolation but are **perceived differently depending on their context**. The same hue can appear lighter, darker, warmer, cooler, or even more saturated, depending on the neighboring colors.

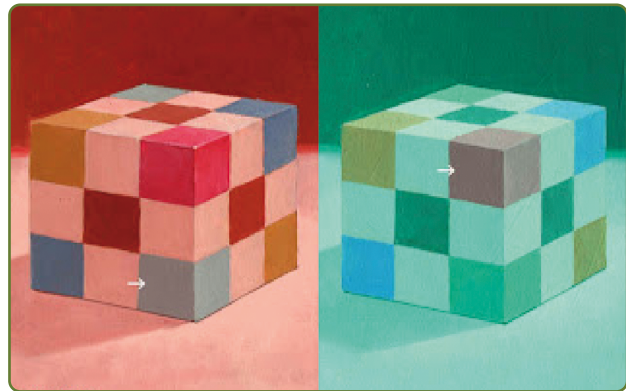
For instance, a mid-gray color may appear much lighter when placed next to a dark color, and darker when placed near a bright color. This is what this famous optical illusion illustrates:



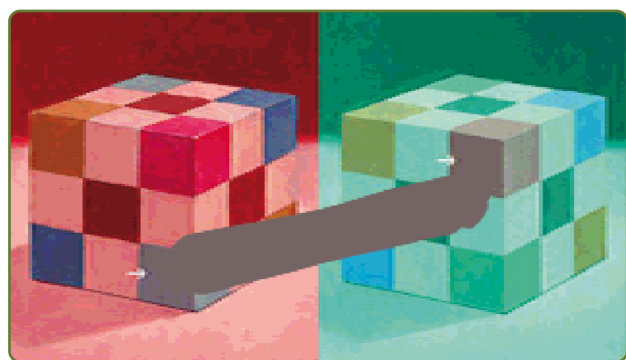
In the image above—a dirty CPC conversion from James Gurney's book *Color and Light: A Guide for the Realist Painter*—you can see that squares 1 and

2, although they look like two different values, are actually the exact same old CPC #13 mid-gray. What makes it look darker on the right side of the image are the surrounding pixels.

In this other example, also from James Gurney's book, you can see how the same gray (indicated by the white arrow in both cases) can be interpreted as green in a red context or red in a green context:



They are the exact same color, though. Don't believe me?



This phenomenon is known as **simultaneous contrast**: when your eye sees a color, it simultaneously perceives its complementary, even if it's not actually present. This is why gray surrounded by red looks cyan, for instance.

You can take advantage of this effect by playing with the **hue, saturation, and brightness** of your gray to shift it toward a cooler or warmer, lighter or darker color.

All right. Got it. With this in mind, let's try redoing our study. If I squint at the painting, I can roughly see four large blocks in the image: the main orange ground, a more greenish area for the cornfield with a darker variant in the foreground, and a bluish area for the sky.



I'll start by using the actual orange that we found to block in the ground. To find the green of the field, I'll apply what we've learned: I'll desaturate my basic orange to get closer to a gray and shift the hue toward a cooler color, which lands me in the yellow range. Shifting even further towards the greens, bringing down both saturation and brightness, I get the darker green for the foreground. Finally, to find the right color for the sky, I'll reduce saturation to 5%, increase brightness to 75%, and move closer to the greens.

And this is what we get:



Now that's way better! Let's refine that study a little more:



Not bad. So, as crazy as it seems, we get better results by picking out colors from a limited palette of yellow variants than when we think in **absolute** colors like "green", "cyan", or "orange".

All right, this is all very nice in theory, but how do we apply this to the 27 colors of the Amstrad CPC?

#### COMPUTER BLUE

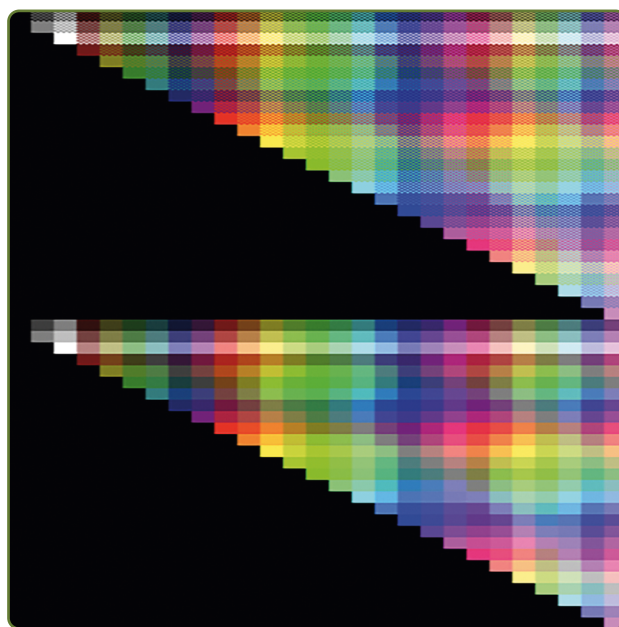
I won't be covering the Plus range this time, as its 12-bit color palette makes it easier to subtly shift hues.

The case of the CPC, however, is much more interesting. With only 27 predetermined and highly saturated colors, artists need to be particularly thoughtful about how they approach color choices. These limitations can be frustrating at first, but with a good un-

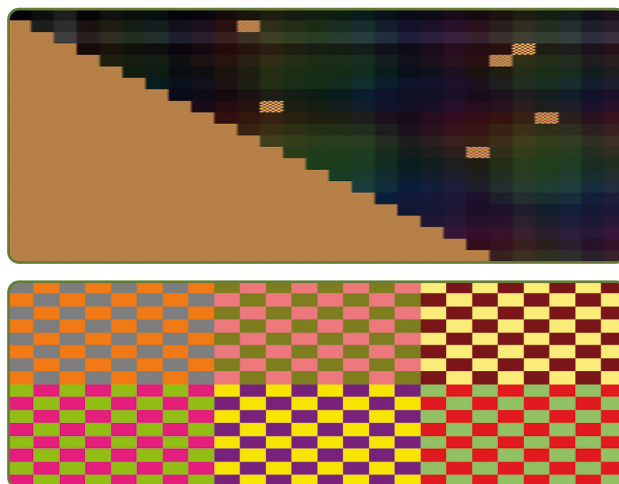
derstanding of color theory and some creative thinking, they can also be a source of inspiration.

So, if I wanted to reproduce the same kind of autumnal colors on my CPC, how should I proceed? While the CPC's palette might initially feel restrictive, the key lies in leveraging color relativity and the very convenient fact that, on our CTM monitors, pixels tend to **bleed** into one another. We can take advantage of this effect.

Below is a screenshot featuring all the different combinations of our beloved 27 colors. The bottom half shows pure colors, while the top half shows their counterparts using **optical mixing** via **dithering**. From there, it's pretty easy to pick out the color we want and see what color mixing allows us to simulate on screen.



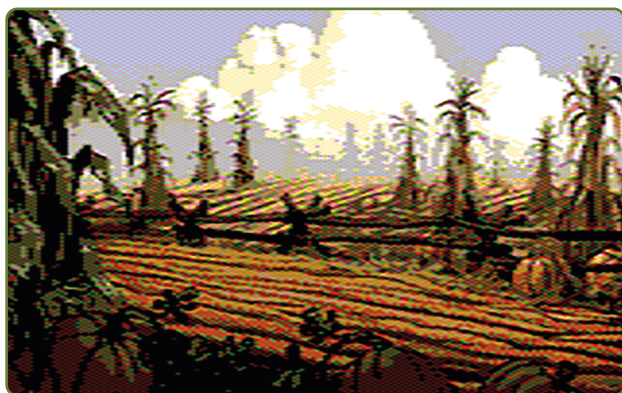
For instance, if I wanted to simulate the color #BF7F3F (R:191, G:127, B:63), which doesn't exist in the Amstrad CPC's palette, I'd have six different options, highlighted in the screenshot below:





In the close-up, you can better see the colors being mixed. Given the lower value contrast between their components—and since we want to keep our palette as limited as possible—options 1 and 2 (either mixing orange #15 with gray #13, or pink #16 with dark yellow #12) blend more smoothly and provide the best results. The choices you make will depend, of course... on the surrounding colors! As we saw earlier, value contrast, simultaneous contrast, and warm-cool relationships will help you build an effective color palette and achieve unexpected results.

Applying all the aforementioned concepts, I quickly created the image below, which demonstrates the subtlety of shades achievable even within the CPC's highly saturated palette.

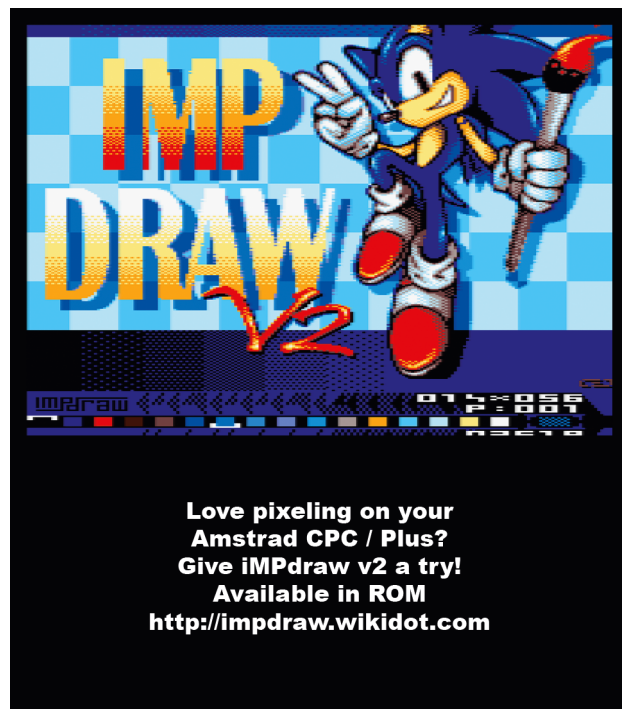


Of course, there's always room for improvement, but it's fascinating to see how we can achieve such nuance in color with these limitations.

Now that you have these potentially new concepts in mind, try experimenting with them. You'll get better results on real hardware, but any pixel art tool will do the job just fine.

In the meantime, I'll be working on a brand-new article about another powerful tool for creating impactful images: lines of force, or **motion**.

Until next time! ■







# IN BED WITH FDC

When the first CPC emulators were developed in the late 90s, FDC emulation was very closely tied to the DSK format. This made it easy to pre-mash the physical format into a logical format that included the expected return codes. FDCs then emulate very little and simply take from the DSK what they are supposed to return. Even today, the DSK limits the quality of emulation because the FDC/DSK pair was designed from the outset to make existing stuff work, not to reproduce how generic hardware works. While this isn't a particular problem for the gamer, it's a different story for the programmer who wants to create his own trackload, and the years that have passed don't contradict the state of the art. Productions that run on emulators, at best on Gotek and not on a real drive, are on the increase.

BY ROUDOUDOU/PRALINE

That's why I'm researching the FDC, to understand how the chip works normally and also how it works when you step out of line, because it's very easy to do something without meaning to! I've been playing with the FDC for some time now, and I've also had many problems, problems that some people have had at one time or another when trying to run their programs on a real floppy drive, or even just a real CPC. Reality is often capricious, especially for those programming on an emulator or simply using a Gotek. As I said in the introduction, FDC management is often superficial, and we suffer from the ubiquity of the DSK format, which is incapable of correctly representing physical tracks. So we often have this quadruple problem of the poor FDC emulation, the drives, the physical media and poor programming, mainly caused by poor documentation.

As I have often complained in the *À la découverte du FDC* series of articles (editor's note: on the 64 NOPs blog), the FDC documentation does not give detail enough, does not warn of the side-effects we are about to talk about, and it's a bit the same with the documentation for physical drives, whose timings do not always match what they are supposed to describe. We will see later that some crucial be-

haviours are however mentioned in these documents, sometimes with errors, sometimes lost in the mass, in very small print at the bottom of the page.

I re-read the series of articles by Sined le Barbare that appeared in Amstrad Cent Pour Cent. Interesting reading, with a few errors of interpretation. For example, it is said that it would be stupid to write in the status<sup>1</sup> register whereas the FDC simply does not decode the address bit when a data is sent to it. On the contrary, it would be stupid not to do so, thus avoiding the gymnastics of INC C / DEC C every time you want to write after reading a status. At the same time, you save two precious bytes, invaluable for those who want to create a nice boot sector, for example! When reading data, you have no choice but to enable / disable bit 0 so that the FDC knows what you want to get back!

The series of articles led to the development of an efficient format (good-length GAP, 5K per track) with routines that went round the demo and game world, including one that I was very critical of because it is the typical example of a function that goes off the rails<sup>2</sup>. Again, making sure that the return of a variable number of parameters is correctly emulated is

1 Amstrad Cent Pour Cent n°35 page 46

2 Amstrad Cent Pour Cent n°37 page 46



no easy task, because then comes the whole issue of managing messy accesses to the FDC. As I mentioned in *À la découverte du FDC*, you're reading a second result that doesn't exist, initiating a command because reading from the port when the FDC has nothing to send... sends the last data present on the port! And off you go, repeating the routine at least twice. The *ACE-DL* emulator will have a lot of fun with this and will write "Bad FDC access routine detected" in the log, but traces of this kind of abuse can be found in Ocean games, for example.

#### KNOWING IS NOT UNDERSTANDING

As I have written before, I have produced a short series of articles on the FDC for the *64 NOPs* blog. During the writing of this series of articles, I went back several times to correct some parts, and almost each article offered an erratum to the previous one, as I came across behaviours that called into question my initial hypotheses. I'd even shamefully added an "essential pause" for real drives while it wasn't necessary. This explains my reluctance to resume writing articles for the *64 NOPs* blog. Fun fact: writing this article will also require some corrections as discoveries are made (writing started in early 2024).

It has to be said that there are many legends circulating, spread by one person or another, sometimes a few lines of great importance that are carelessly included in the documentation. Even worse, there are biased experiments, some of which I have carried out. And I've believed or deduced a lot of things that weren't true! Physical drives are slower, electronics have aged, blah blah blah. And your Z80, has it aged too<sup>3</sup>? We like to find reasons for the unexplained (without proof!), but it's harder to explain what we don't know. We reassure ourselves as best we can by distancing ourselves from reality.

But the pause found in ROMs and many other sources did not come out of nowhere. Indeed, if you read the small print in the documentation, which is designed for an FDC running at 8 MHz (the CPC's is at 4 MHz), you will learn that it would be a good idea to leave 12 microseconds (24 in our case) before reading the status again, as this is the maximum time the FDC will take to set DIO and RQM. In practice, the FDC will drop DIO and RQM immediately (I may be getting ahead of myself, but I haven't seen any examples to the contrary, quite the opposite!). On the other hand, it can be particularly slow to remove CB (CommandBusy or InProgress) which, according to the documentation, will not accept a new command. In fact, this is only partially true. In Ocean loaders, you can find an unorthodox way to get back the results by testing CB aggressively (I should point this out to emulator authors who want to get started, the difficulty of emulating FDC flaws is to keep programs that work by accident!).

Wanting to write a high performance loader, I continued to use the Targhan optimised system<sup>4</sup> that calls functions according to the current status returned by the FDC, you know, the one that does `IN L, (C) : JP (HL)?` It's compact, powerful and *FatMag 2* uses this method for the boot sector. I had in mind to reuse this system to stress the FDC a bit, and what I got really made me go crazy! Indeed, I noticed that if I got back the last result too quickly, the FDC seemed to go off the rails and send me back a lot more, a multiple of 16, following a more or less logical sequence... If I gradually added the famous delay, it could get out of its infinite retrieval of results. Even better! I had noticed that it crashed even more with sectors that weren't size 2.



*Fatmag 2 (Praline)*

Wrong conclusions: the FDC shows optimisations for sectors of size 2, if you get back too fast it gives different results, etc. All this seemed very complicated for a small FDC, but that was what I was observing, what I knew, without understanding it. I even asked various members of the Impact Discord to see if they all had the same timing problem (answer: yes). So, in the unknown, I wrote a fragment of emulation of the misunderstood phenomenon, without much satisfaction. Then came the information that unlocked everything. RunBowCPC told me that he was observing some strange statuses on function returns, as he was working in the same way, using `IN L, (C)` to get back his results. The simple solution? Add pauses to your code like everyone else? Out of the question! I have to emulate this bug, understand it and come up with something correct, so that if someone codes like a pig, it won't work on the emulator...

#### FDC DESIGN FLAW FINALLY FOUND!

So I restart my favourite CPC to see how the status evolves after the last result, and I launch a series of `INI : INC B`, which is the quickest way to get a lot of results in a row. And... what a surprise! After getting back the last result, the FDC status takes its time. First I get #10 (instruction still in progress),

<sup>3</sup> It seems that the Z80's internal resistance can change over time and cause bus conflicts with the CRTC, but that's another story.

<sup>4</sup> As far as I know, rare production using this principle.

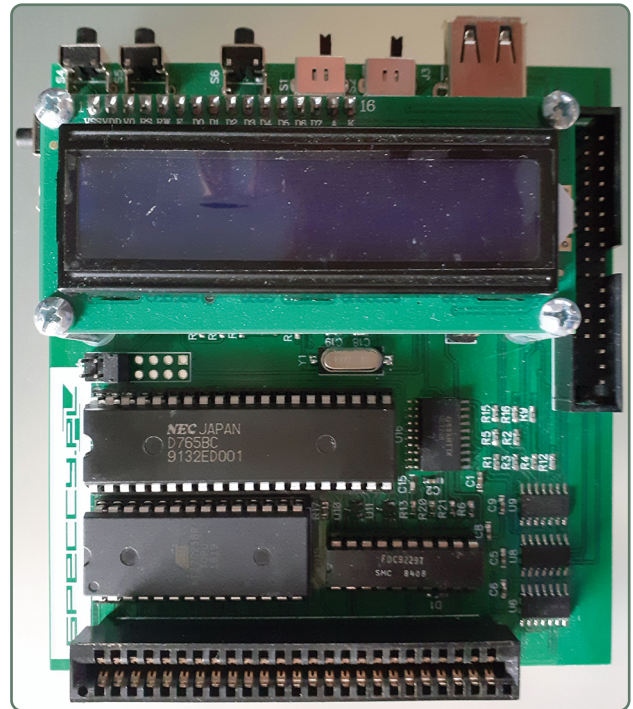


then a series of #90 #90 #90 #90 #90 #90 (hello, I'm available AND still in progress) and finally #80 (available and ready to do something else). This was something we wanted to test later: at what point does the FDC take a new command into account if you ask it too early? Here, in relation to our routines, status #90 is when the command parameters are sent. Our routines were not good, because if we read the status too early, we thought we had parameters to send! So we sent a new command using what was in memory, using the results we'd just read! With a size 2 sector, we asked for a diagnostic read. This new command to read 9 values and the 7 results made 16 values! We had found the reason for what we thought was a bug in getting back the results, and a good explanation!

### CHALLENGING THE EXISTING KNOWLEDGE

OK, so the FDC didn't go into panic mode and send back lots of results, it just restarted a command. By the way, I went back to the calibration and the Seek to "see" when the bit of the drive in question was set. I quickly saw that the bit was set at the end of the command, but when? I found a similar sequence to before, with the drive bit set: #10 #90 #90 #90 #91 #91 #91 #91 #91 and finally #81 (command completed but head movement in progress). The drive bit was set after 18 NOPs during which the FDC said it was available for any new command (although CommandBusy #10 says it can't be). This part was one of the hardest to get right because (for example) the *Chase H.Q.* loader (Ocean) uses this flag (CommandBusy #10) to get back its results! One more NOP and the game won't load. One NOP less... Fortunately, I still had my naive test program to stress test this particular point.

There were still the borderline cases to analyse. As much as I wanted to postpone this test, the analysis of *LazerLoad* forced me to come to a quick conclusion. This fast-loading routine is very clumsy and uses several unconventional ways to calibrate and move its read head. For example, instead of reading interrupts after calibration, it uses SenseDrive, which has the peculiarity of setting a bit when the drive is in track 0. Completely dirty, the loader has the luxury of sending a new command only 19 NOPs after the end of the previous result delivery. So I restarted *Organs* and took my turn at stuffing the FDC, first at 19 NOPs, then reducing it to 6 NOPs later. And then it's another surprise, and at the same time so logical, so simple... To cut a long story short, as soon as we relaunch a command, if the available status was raised, for example #10 #90 #90, then it loses its "available" status and returns to #10. The status bit will continue to be set at the same time. Yes, because it doesn't leave the function any faster. This is followed by a series of #11 #11 #11 #11 #11



DDI-5 whose FDC has been replaced by a 765B (model not used on CPC / Plus)

#11 #11 #11 then #01 then #11 again. The #01 indicates that it has completed its command and relaunched the other (all the bits are cleared except those of the drives whose heads are being moved at the start of the commands). We had confirmation that it was finishing what it had to do, but it told us that it had taken the request into account for later.

### WHAT OTHER TOPICS DID I MISS OR GET WRONG?

I'd noticed for a long time that if I launched several GetVersion on the FDC, every 2000 NOPs or so it would pause for just over 100 NOPs before responding to me. I literally had over 100 NOPs with a status of #00. To simulate this, I kept it simple with a counter, a modulo and an addition of 100 NOPs. That seemed to do the trick. Then, later (I talked about this in the third article of *64 NOPs*), I noticed that if a command hadn't been executed for a long time, its preparation time fluctuated from about 1800 to 3800 NOPs, a difference of about 2000 NOPs, still this magic value.

After re-reading the documentation, I decided to take an interest in the pooling of the drives, which - funnily enough - the total time given for pooling the 4 drives is about 1000 NOPs. Once again, the UMC documentation provides a crucial detail by writing not 1 ms but 1.024 ms ^\_^ As the documentation gives times for an 8 MHz FDC (sometimes the time is given for a 4 MHz FDC), we found our magic value of 2048 NOPs. OK, while we're at it, let's simulate a pooling of the drives according to the documentation, with an approximate distribution as described at

5 Simulated pooling solves the problem of many delays, but implies complete management including integration of the Seek.



1/5, 1/5, 1/5 and 2/5 of these 2048 NOPs. I mention this to RunbowCPC who immediately tells me: no, the pooling is much faster than that (using another piece of documentation), it actually lasts 30 NOPs per drive and the signals from drive 3 remain until the end of the 2048 NOPs, but the pooling should last 30 NOPs like the others. I didn't think of this immediately, but the interesting thing is that the phenomenon observed during the GetVersion stress test is exactly the same if we assume that during pooling we delay the execution of the command for 120 NOPs. All you have to do is really simulate pooling<sup>5</sup>, block the start of a new command for the time it takes to complete it, and all our statuses will be as they are in real life.

Similarly, in line with the FDC's default settings, I quickly found that the duration beyond which the optimisation no longer works is quite simply the read head withdrawal time, with the variability inherent in pooling continuing unnoticed most of the time from 2048 NOPs onwards. It may not seem like much to use 2048 instead of the 2000 or so observed, but 2% on durations in milliseconds adds up to a lot of NOPs!

#### SEARCHING FOR THE BUG IN SIZE 0 SECTORS

According to the documentation, the FDC cannot read / write / format size 0 sectors in MFM. However, if you try, you have no problem formatting size 0 sectors and reading them almost correctly (if you haven't written any data to them). Funnily enough, one of the first Speedlock protections has a size 0 sector at the end of the track (this is the only minor difficulty with an automated copy), but doesn't bother to read its contents, even though it contains data and can easily be read for at least 80 bytes. What do you mean 80? Those sectors are 128 bytes long! Yes, what doesn't seem to work is the amount of data to be read (a trainee bug?). Only 80 bytes are read, whether in ReadData or ReadDiagnostic. See what I mean? 80 bytes are read for sectors whose size is #80 in hexadecimal! Coincidence? Probably, because if the FDC only returns 80 bytes, the 128 bytes are read and the CRC of a freshly formatted sector is correct! Well, that's not too serious - if you can write to it, you can always read back 80 bytes, or even more with ReadDiagnostic! But can you write to it?

#### GOTEK TO THE RESCUE IN WRITING!

Until now, my zero size write tests resulted in the loss of the DAM. The data just disappeared! I put an empty HFE on my Gotek to make it easier to analyse my MFM stream. The big advantage of the Gotek is that all you have to do is get back the file from a PC to re-read any stream. I write a few size 0 sectors to a track full of sectors. On this occasion, the 9<sup>th</sup> and last parameter of the read / write commands finally comes into its own and allows you to ask for only N bytes to be written into the sector, the rest being filled with zeros until you reach 128 bytes of data. The CRC written in the 128<sup>th</sup> position is always calculated correctly. How do I know it if these sectors are unreadable? Well, it's because analysing the stream reveals the surprise of seeing new DAM marks for the data! On any other type of sector, the DAM is #FB for a classic sector and #F8 for a ControlMark sector (the famous disco CM). Here we have #FD for normal and #FC for ControlMark. The writing almost does what it's told! Is this a big loss? I don't think so, but you can imagine copy protection for a program that can copy itself, hehe.



IBM 3742

I looked at the documentation for other controllers to see if these markers were used on other systems, but the #FC marker seems to be reserved for the index, and #FD is marked "spare" on the Western Digital controller. Other DAMs exist, #F9 and #FA as "user" DAMs, but we are not concerned. However, as the documentation says, there is still the case of single density use (FM coding). Would it be possible to read and write FM? According to the *Discolgy* 5.1

6 CP/M uses 128-byte sectors, which are logical but not physical.



manual, only the 464 can write in single density. We're obviously talking about the external drive, which alone has all the necessary architecture. When it comes to writing data, the 6128's FDC sends its impulses directly to the drive, whereas with a DDI-1 the impulses pass through the data separator. It's a shame that the "driveless" computer is the only one capable of writing in all modes! On some CPC models, the MFM pin of the FDC is not connected at all. Just like DMA (yes, we could have had DMA access for floppy disks...), the decision was made to omit this functionality. After all, what machine could have written a 3-inch single-density floppy? None. AMSDOS was CP/M<sup>6</sup> compatible, not MS/DOS!



IBM 3741 Data Station

Through experimentation, an FM formatting command will run correctly on a 6128 (timings according to FM, OK return codes, etc.), but it is impossible to read anything with the 464, it has written nonsense. The same program on a 464 + DDI-1 will confirm that it can read and write in single density, but the 6128 will still not be able to read it. So we won't be able to use this ancient format, which was used in the early 70s when 8-inch floppy drives appeared on the 3740 "Data Station" of the IBM-370 mainframe. Back then, double density didn't exist and we were just about out of punched cards, so it was a godsend to have a portable, direct-access format. Some old platforms that used the first DOS (Thomson / IBM / Atari-400) would use size 0 sectors, but on either 8" or 5 1/4".

The Nec765 FDC was released a few years later, in 1978, with the arrival of the double-density drive from Tandon (the manufacturer who had been challenging Shugart) and boasted compatibility with the old IBM systems, their 8-inch floppies and their tiny sectors, as well as future compatibility with MFM support and gigantic sectors. The Atari 400 with its first single-density drive (model 810 caddy / 1980) will have its

DOS quickly updated in 1981 to stop using these sectors and optimise formatting. In other words, NEC had been right to neglect size 0 in MFM (128 bytes of data to which we must add 62 bytes of header and a minimum of GAP, we lose half the capacity!). In the same year, Atari abandoned single density by releasing the new 815DD model, which only supported double density. The desire to have a MAXIMUM of space was well known, all the more so as the frequency of use of the FDC in the CPC meant that disks were already being written in single density, the cheapest disks were already being used! So it's easy to see why, a few years later, low-density management was not a topic on the 664 and 6128. As for Amstrad, we can only regret the lack of DMA, which would have made our platform a worldwide 8-bit killer.

#### HIDDEN FDC FEATURE OR SICK BUG?

The Nec documentation says that the FDC interrupts must be acknowledged of any head movement before doing anything else, otherwise no reading or writing is possible. The Seek description also says that you can't start a new command during Seek (is this true?). I remember trying - during a head movement! - to run a formatting command. The movement was briefly interrupted, and then the head went to the upper reaches of the Gotek, levelling off at track 99... I left this strange phenomenon aside, although I thought it would be fun to understand and simulate it one day. Later, when I started a test campaign to continue on ACE's FDC, I tried to launch a GetID after moving the head, and everything went fine. No errors. OK, I'll test the formatting later to see if the head is still dancing the salsa, but the impossibilities mentioned in the documentation seemed very odd to me...

Let's change the topic for a moment and talk about money: for cost reasons, not all FDC pins are physically connected, which disables some functions. This is the case for the MFM pin, DMA and also US1, which normally provides access to the 3<sup>rd</sup> and 4<sup>th</sup> drives. The FDC is actually capable of managing 4 drives. It certainly has 4 counters for the current track, probably 4 counters for its shifts, and so on. In any case, the status register provides 4 bits for reading for the drives. In order to check this, I'm going to have a bit of fun. I calibrate drive 0 and 2 (since pin US1 is not connected, drive 2 is drive 0, but with different counters!). Their counters are at zero. I move drive 2 to track 2, and I move drive 0 to track 8 (as the physical drive has received 10 pulses, it is on track 10). And I launch a GetID without acknowledging the interrupts, as I've just tested that it's not necessary (why bother?). I get the information from my current sector (track 10) and then I hear the read head move briefly. What? I launch GetID again, it tells me it's moved to track 12 and I hear the head move again. Now we're back to the heart of the matter!

7 Fun fact: the Hynix GM82C765B documentation states 0,1,3,2 pooling, but analysis of the signals confirms that the Nec actually does 0,1,2,3. This inversion is not due to the pooling order.



What a brilliant trick! On a disc with the correct IDs and an FDC that hasn't been fooled, this would mean that if you did a GetID after moving the head on drive A, it would automatically set itself to the correct track in the event of a shift. Well, that's a bit far-fetched, but why not! Then I remembered that the GetID command in single density returns its results in a rather peculiar way. Indeed, if you calibrate and ask for a lousy GetID, the track information will be at most #4D (remember? #4D as 77 maximum calibration pulses), because a GetID that doesn't return any results keeps the previous C, H, R and N values. And if the interrupts are not acknowledged, the drive goes to track 77. I later understand that this counter is a residue of the calibration pulses. If I start from track 16, calibrate and do a lousy GetID, my counter is no longer #4D but #3D (because it took 16 pulses to get to track 0) and the drive goes to track 61... For formatting, the nuance is slightly different, and understanding this bug told me that the C result of the command has nothing to do with the track info used for formatting. Actually, when the command writes the sectors, it doesn't store the ID-Track in C. It simply takes the C value from the previous command, usually a correctly executed Seek. So there is no overloading of the value, as I might have thought at the beginning of my research.

not C, H, R and N, but the last two are reversed! To move our drives A, B, C and D, it uses C, H, N and R<sup>7</sup> respectively! Now that we've understood the sick mode, what's the point?

#### A MAGIC LOADER? PUT IT INTO PRACTICE!

This applies to all instructions that return 7 parameters as a result. This means that all the SenseDrive, Seek and Calibration functions can be used without interference (take care to allow a few delays so as not to interrupt a movement in progress). As with the *Dark Sceptre* trackload, the SenseDrive can falsely acknowledge calibration without dropping the Seek bit.

By having fun putting in shifted IDs from the very first track (risky for the catalogue), or by pretending to write, the wrong track info is enough to trigger the head to move to the next track. A failed entry to position yourself where you want! More maliciously, you could initiate ONE movement on track 1 via drive 2 (which is actually drive 0) and everything would follow as if by magic, with the FDC trying to catch up after each new reading. You could also (and this seems to me to be the perfect solution) organise your sectors so that the last sector read from the track is the indication of the next track! That way there is no command in the air, no Seek, diabolical!



Atari 810 floppy disk drive

I'm still investigating the overall workings of this "bug". Is it a coincidence that the drive uses the track result to move his head? It seems to be, as drive B doesn't move on any of my commands. So I try a GetID on the second side of my drive B and, lo and behold, it moves to track 1! I then test whether the sector result moves drive C (i.e. drive A) and whether the sector size moves drive D (i.e. drive B). It does move, but I don't get the expected result! In fact, on a second test run, I realise that the order is

#### THE LIMITS OF THE BUG?

The C, H, N and R registers contain the number of pulses remaining from a completed calibration. Any calibration from track 0 will write 77 to the relevant drive's registers. Any calibration from another track will write 77 - the current track. That's the basics. However, if the FDC command changes the values of C, H, N and R, these values will be erased. The Format command does not change the current C. This is where we will have the most surprises! On the other hand, the read/write commands or the GetID rewrite C, H, N and R, so everything remains easily predictable, especially for the read/write command, where you can tell it to go wherever you want. Be careful, however, when using a GetID on an unformatted track or in

FM reading. If it finds nothing, C, H, N and R will be preserved by the residual values of the previous command!

#### ENCOURAGE YOU TO FIND THINGS

By sharing these anecdotes and discoveries, I hope to have motivated you to play with the FDC, or at least to have stimulated your creative side as an algorithm explorer. If there are any secrets left to discover, you'll find out in the next episode... ■





## **64 NOPs**

**Editorial staff:** Hicks, toms

**Contributors to this issue:** Eliot, Hwikaa, Krusty,  
Madram, Prodatron, rexbeng, Rhino, roudoudou, Zik

**Cover:** Beb

**Layout:** toms, Hwikaa

Thanks to the authors of the photos published in the articles!

First print run · 120 copies

**To contact the editorial team:**

**Email :** [admin@memoryfull.net](mailto:admin@memoryfull.net)

**Websites :** [64nops.wordpress.com](http://64nops.wordpress.com) · [www.memoryfull.net](http://www.memoryfull.net)









# SIXTY FOUR HOPE