



Coding techniques in Sonic GX

An **Amstrad GX-4000/Plus** game

Presented at the
Benediction Coding Party #5 (2025)

Author: Arnaud “**norecess464**” Storq

The long road to the final release

- At **Alchimie 2019**, we officially presented Sonic GX (*the project had been in development for about a year at that point!*)
- We showed our prototype and gave a short presentation (*unfortunately, it wasn't recorded!*)
- **6 years later**, we are now presenting the **final release of Sonic GX**
- Many of the people who attended Alchimie 2019 are also here today at the Benediction Coding Party 5 (2025)
...it means a lot to me! Thank you!



Disclaimer

- This isn't a tutorial about the GX-4000
- Some technical knowledge is required
 - I'm not going to explain what is a ROM, an interrupt, a memory page, etc.
- I'm not claiming to have implemented the best possible tricks
- What worked for me on this project might not work for you on yours



Before we start...

- In this presentation, the term “GX-4000” refers to both the Amstrad GX-4000 and the Amstrad Plus machines
- Think of this as food for thought. If a few concepts pick your curiosity or inspire ideas for your own projects, that’s exactly the goal! *(You don’t have to understand everything.)*
- This document will be available for download, so if something isn’t clear right now during the presentation, you’ll be able to revisit those sections later



Topics covered

- Memory organization
- Game loop
- Interrupts
- Tiles
- Scrolling implementation
- Sprite management
- Gameplay
- Collisions
- General development tips



Topics NOT covered

- Boss-fights
- Title screen
- Special stage
- Conversion tools on PC

...I should probably write a book and become a millionaire! :)



Memory organization



Memory organization

page &0000->&3FFF

- VRAM Buffer (first 16KB of the 32KB VRAM)
 - Only write access is required
- Lower ROM activated
 - When drawing column/row
 - &2000->&3FFF
 - X0Y1 char tile Z80 code
 - X1Y0 char tile Z80 code
 - Game Sound FXs



Memory organization page &4000->&7FFF

- ASIC Page
- When ASIC page is deactivated:
 - &4000→&7FFF: Unpacked map data
 - “Big tileset” pointing to “big tiles”
(more on that later)
 - “Vertical slices” of map data pointing to an index in “big tileset” *(more on that later)*



Memory organization

page &8000->&BFFF

- &8000→A000 for temporary RAM
 - Global variables (life count etc)
 - Gameplay variables
 - Temporary buffers (DMA buffers, etc)
 - Stack
- &A000→&BFFF for code
 - Interrupt handlers
 - Self-modifying code
 - Helper routines (jump from one ROM to another, etc.)



Memory organization page &C000->&FFFF

- VRAM Buffer (second 16KB of the 32KB VRAM)
 - Only write access is required
- Upper ROM activated
 - When drawing column/row
 - &E000->&FFFF
 - X0Y0 char tile Z80 code
 - X1Y1 char tile Z80 code
 - Gameplay code
 - Audio player etc.



Game loop



Game loop description

- Game loop refers to all the code executed to render one complete frame on screen
- One frame = $1/50\text{hz}$



Game loop 1/5: frame init

- **Disable interrupts**
- Fill Audio DMA buffer (play music+game sounds)
- Prepare ASIC registers for the new starting frame (set R12/R13, SPLT, SSA R12R13, DMA2_SPLT, SSA2 R12R13)
- Prepare column/row drawing tables (Stack involved for faster reading)
- Various small inits: global frame counter, palette color cycling, set sprite positions...
- **Enable interrupts**



Game loop 2/5: high-level managers

- Manage the **scripted event list**
 - A sequence of actions executed frame by frame (*more on that later*)
- Manage the **trigger boxes**
 - Triggers a scripted event when Sonic enters a bounding box (*more later*)
- Manage the **map sprites**
 - Set the screen position for a group of sprites (example: Eggman is made of 6 sprites, 2 for width and 3 for height) (*more on that later*)



Game loop 3/5: scrolling update

- Draw column AND/OR Draw row
(more on that later)
- Calculate frame deltas for next frame



Game loop 4/5: gameplay management

- Set new “shadowed” positions for moving platforms, checkpoint rotating ball animation
- Execute game-logic:
 - Grab controller input
 - Set new Sonic world PosX/Y
 - With collision against ground, walls... but also moving platforms
 - Camera logic
 - Sonic collision against enemies and platforms



Game loop 5/5: sprite management

- Set new “shadowed” positions for all sprites
- Manage rings
 - Fixed ones OR Multiplexed (not both!)
 - Sonic collision against Multiplexed Rings
- Do sprite refresh for dynamic sprites (platform, enemies)
 - Queued, 1 sprite per frame
- Do sprite refresh for Sonic
 - End of frame, **can be interrupted for new frame!**



Interrupts



Why interrupts?

- We need interrupts to perform those specific operations at precise moments during the frame generation:
 - Background rasters (PRI)
 - ASIC Splits for the scrolling
 - Multiplexed Rings
 - End of visible screen
- I use Interrupt Mode 2 (IM 2) because IM 1 forces the interrupt vector to &0038, which is already used as VRAM. IM 2 allows to use any location in RAM.



DMA vs. PRI interrupts

- PRI only allows to set an interrupt in the visible screen
- DMA is designed to send audio registers, but it can also be used to trigger DMA0/1/2 interrupts
 - Unlike PRI, not limited to the visible screen!
- Good to know: if you have 4 (!) interrupts triggered at the same line, the order of calls will always be PRI then DMA 2 then DMA1 then DMA0



Background rasters with PRI

- The animated blue sky in the background is made of rasters triggered by the PRI interrupt
- A tip from Overflow/Logon System (thanks!): use the secondary Z80 register set like the Amstrad CPC firmware does (EXX):
 - HL' points to ASIC PEN0 COLOR
 - C' is used as current PRI line index
 - B' and DE' for table management...
- This gives enough time in the left border to poke the new raster color value



DMA interrupts configuration

- DMA2
 - Play Audio (before the screen gets visible)
 - Set Second Split (Scrolling)
- DMA1
 - Sprite multiplexing (Rings)
- DMA0
 - End of visible screen

*This is not super intensive after all...
(could be worse!)*



Are the DMA interrupts bugged?

- Yes and no
- Everything is explained in the document “Plus Vectored Interrupt Bug” available on cpcwiki
- Not bugged for me when following that rule:
“Put your code between &2000-&3FFF, &6000-&7FFF, &A000-&BFFF, &E000-&FFFF”
and you will never bother again about interrupts!



Tiles



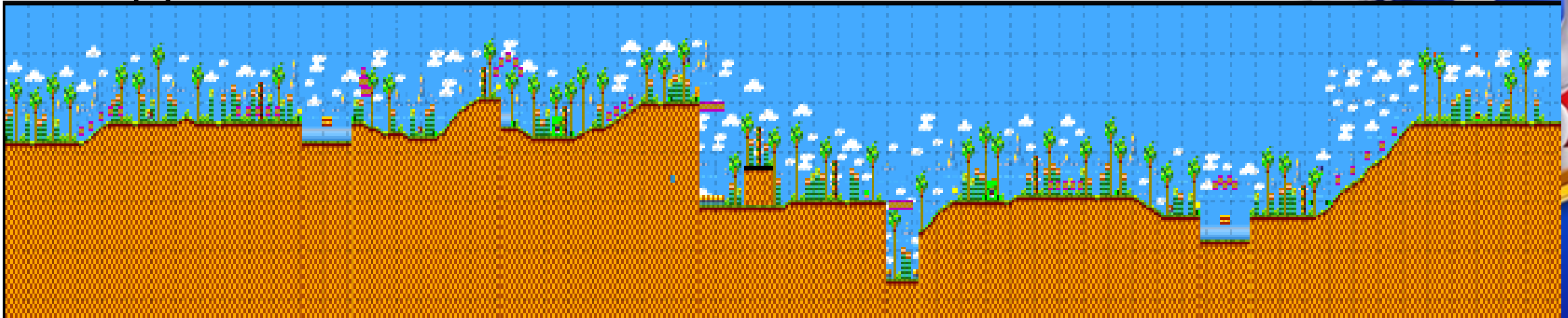
Map orientation vs. memory

- The engine allow horizontal maps, vertical maps, or squared maps
- The limits ARE NOT the width/height of the map
- The real constraint is the memory footprint in ROM/RAM



Let's focus on Green Hill Zone ACT 1

- The map is made of 636x64 tiles
- Storing a 8-bit tile index for 1 tile would mean a map in memory would be $636 \times 64 = 40704$ bytes! (*Hello Alchimie 2019 demo!*)
- That's clearly unacceptable, another approach is needed



Big tileset

- So, we said the GHZ map is made of 636x64 tiles
- Instead, we are going to group 4 tiles together (a square: 2 horizontally, 2 vertically = 4 indices pointing to “big tiles”)
- New map size is $(636 \times 64) / 4 = 10176$ bytes (plus a separate buffer for grouped tiles)
- It can now be fully unpacked into RAM!
- The only limitation is to maintain the group index < 256 , which is a real challenge with level design



Big tiles and char tiles (1/3)

- A map is made of “big tiles” (the one edited directly in Tiled)
- A “big tile” is always 8px width and 16px height (MODE 0 on the Amstrad is double-sized horizontally)
- Working with such large tiles saves a significant amount of RAM!



Big tiles and char tiles (2/3)

- The scrolling engine does not deal directly with “big tiles”
- A “big tile” is actually made of... 4 “char tiles”
- A “char tile” is always 4px width and 8px height (MODE 0 on the Amstrad is double-sized horizontally)
- This time, it’s compatible with hardware scrolling!



Big tiles and char tiles (3/3)

- “Big tiles” (2x2 “char tiles”) can be traversed horizontally (to display a row of characters during vertical scrolling) or vertically (to display a column of characters during horizontal scrolling)
- Each “char tiles” is stored in a separate ROM (4 ROMs involved)
- And thanks to the GX-4000’s powerful architecture ;-), we can use Lower and Upper ROM mapping!



Char tiles implementation (1/2)

- A “char tile” is not made of data
 - Data-only storage would mean “**READ** tile pixels from the source, **WRITE** to the VRAM destination”
- Instead, we only **WRITE** to the VRAM by using generated Z80 code

```
; below code is 45 bytes long
Tileset0_Tile0_X0Y0:
Tileset0_Tile0_X0Y0_0:
    pop hl
    ; HL = Screen Address
Tileset0_Tile0_X0Y0_1:
    ld (hl), d
Tileset0_Tile0_X0Y0_2:
    inc l
Tileset0_Tile0_X0Y0_3:
    ld (hl), d
Tileset0_Tile0_X0Y0_4:
    set 3, h
Tileset0_Tile0_X0Y0_5:
    ld (hl), d
Tileset0_Tile0_X0Y0_6:
    dec l
Tileset0_Tile0_X0Y0_7:
    ld (hl), d
Tileset0_Tile0_X0Y0_8:
```



Char tiles implementation (2/2)

- The 4 “char tiles” composing a “big tile” all share the same unique index in separate ROM
- But each tile in memory has a variable size, since it contains embedded code
- The solution I used was to rely on a jump table at offset &0000 of all the ROMs! (JP instruction always take 3 bytes, so it’s a shortcut to get aligned indices)

```
jp Tileset0_Tile0_X0Y0  
jp Tileset0_Tile1_X0Y0  
jp Tileset0_Tile2_X0Y0  
jp Tileset0_Tile3_X0Y0  
jp Tileset0_Tile4_X0Y0  
jp Tileset0_Tile5_X0Y0  
jp Tileset0_Tile6_X0Y0  
jp Tileset0_Tile7_X0Y0  
jp Tileset0_Tile8_X0Y0  
jp Tileset0_Tile9_X0Y0  
jp Tileset0_Tile10_X0Y0  
jp Tileset0_Tile11_X0Y0  
jp Tileset0_Tile12_X0Y0  
jp Tileset0_Tile13_X0Y0
```



Scrolling implementation



Scrolling overview

- We will first cover the vertical scrolling
- Then few words about the horizontal scrolling
- And finally present the draw row/column routine



Scrolling engine capabilities

- Fullscreen
- Fully 50hz
- Can scroll max. 4px width AND/OR 8px height per frame in MODE 0 (same as 1 “CRTC” char and same as 1 “char tile”)
- Take approx. 40-50% of the CPU time per frame when scrolling in both directions
 - It’s the main bottleneck, so it’s worth optimizing this part as much as possible



Vertical scrolling (1/6)

- Let's focus on vertical scrolling first
- More precisely, let's focus on character-based vertical scrolling
- Displayed screen height is 256px
- We use 2 VRAM buffers
- Each VRAM buffer is 16 chars height
- Which gives 32 chars height
- 1 char is 8px height, so $32 * 8 = 256\text{px}$



Vertical scrolling (2/6)

- Let's have a table with 32 entries, each ones having VRAM start addresses
 - `dw &0000 + 0*(R1*2)`
 - ...
 - `dw &0000 + 15*(R1*2)`
 - `dw &C000 + 0*(R1*2)`
 - ...
 - `dw &C000 + 15*(R1*2)`



Vertical scrolling (3/6)

- When no scrolling is applied, the screen is divided into a single split at its midpoint: the top half uses VRAM Buffer 0, and the bottom half uses VRAM Buffer 1
- When scrolling up the screen by 1 char (8px), the first character row is skipped. We display 15 (not 16!) char rows from VRAM Buffer 0, followed by 16 chars of VRAM Buffer 1, and finally the skipped top row of VRAM Buffer 0 is rendered at the bottom of the screen



Vertical scrolling (4/6)

- By reasoning by chars, it means we have 32 different use-cases for splitting the screen vertically
- Another table gets used here, to know where to apply screen splits + VRAM Buffer pointers
- Our use-case here with a scrolling of 1 char:

```
; SplitScreen1
db (15 * 8) - 1 ; SPLT Page1
db (32 * 8) - 1 ; SPLT Page2
dw #0000 + (2 * R1) ; R12R13 Page0
dw #3000 + (0 * R1) ; R12R13 Page1
dw #0000 + (0 * R1) ; R12R13 Page2
```



Vertical scrolling (5/6)

R12/R13 programming

- To sum up, we have up to 3 screens:
 - **TOP PART:** initialized before the first visible line through Set CRTC R12/R13
 - **MIDDLE PART:** also initialized before the first visible line through SPLT ASIC Register (&6801) and SSA R12/R13 ASIC Registers (&6802/&6803)
 - **BOTTOM PART:** initialized AFTER the MIDDLE's Split thanks to a dedicated interrupt, once again through SPLT+SSA R12/R13



Vertical scrolling (6/6) conclusion

- I lied ;-)

Internally, I'm not using 32 rows, but 34 rows. The 2 extra-rows are never displayed on screen. Since they remain hidden, no drawing artifacts are visible while drawing the row for the next frame

- We presented char-based vertical scrolling
- Pixel-based vertical scrolling is just a matter of using SSCR register (a value between 0 and 7 stored in bit 4→6) additionally to char scrolling



Horizontal scrolling (1/2)

- Let's iterate on top of our vertical scrolling
- It's only a matter of moving the screen left-right
- This is done like a classic hardware scrolling, let's use an offset of $\&000 \rightarrow \&3FF$ chars with CRTC R12-R13 registers
 - VRAM addresses have an offset $\&000 \rightarrow \&7FF$
- And then SSCR again (bit 0 \rightarrow 3) to fine-tune pixel-scrolling



Horizontal scrolling (2/2)

- Tip when iterating through VRAM...
 - Let's say you want to poke &C000
 - LD HL, &C000
 - Then progress by 4 pixels (1 char), you want to do
 - INC L:INC HL
 - Then do not forget to apply the following to “roll” in your VRAM buffer for your hardware scrolling
 - RES 3, H



Row/column drawing routine

- For performance reasons, all tiles are converted into optimized Z80 code
- At the start of each frame, two tables are generated: one listing VRAM addresses and one listing the corresponding chars to draw (one for row and another one for column)
- The code for tile rendering is organized in ROM in a convenient way (4 separate ROMs)
- When drawing a row or column of chars, the Lower and Upper ROMs are switched to the appropriate ones (the tile code)



Sprite management



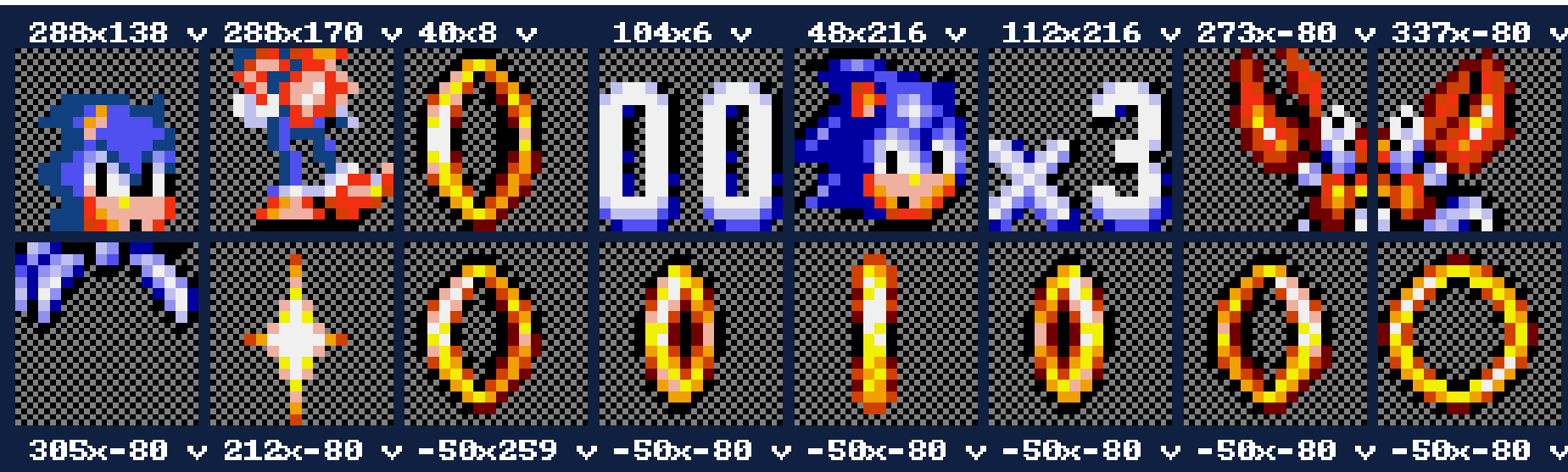
Sprite movement vs. content update

- **All sprites move on screen at 50hz**
- However, since updating them is costly for the CPU, **the content of the sprites are slowly refreshed** at 25 Hz, 12.5 Hz, or even less in some cases
- It's a perfectly acceptable compromise: the game still feels smooth to the player!



Sprite usage

- 0+1: Sonic; 2+3 Hud Rings; 4+5 Hud Lives
- 6+7+8: Enemy/Platform/Missile
- 9: Ring “collected ring animation”
- 10+11+12+13+14+15: Ring Rotation Steps



Sonic+Hud sprites

- **0+1: Sonic**

- Updated slowly: half a sprite per frame
 - I know it may sound strange, but tests showed that Sonic's animation still look acceptable!

- **2+3 Hud Rings; 4+5 Hud Lives**

- Updated even less frequently
 - Only when a frame has some “free” time available



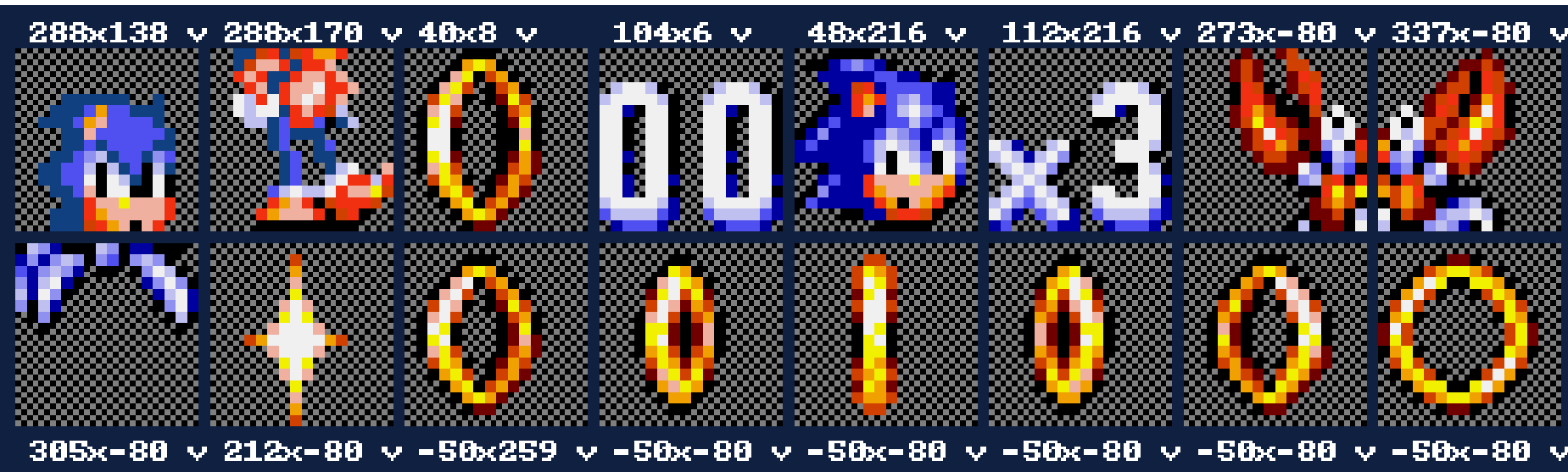
Enemy/platform/missile sprites

- **6+7+8: Enemy/Platform/Missile**
- **Content is streamed dynamically** based on Sonic's position in the world
- A small “**update queue**” is put in place
 - only one sprite per frame gets actually updated (*more on that later*)
- **Always executed at the very end of the frame.** Since the timing isn't constant, if the sprite isn't fully copied before the next frame begins, another attempt to draw it will be made on the following frame.



Ring sprites

- **10→15: Ring Rotation Steps**
- Which means up to 6 sprites can be shown on the same screen (without multiplexing)
- No sprite copy done for animation; only sprite positions are swapped



Multiplexed ring sprites

- Sonic GX can display an array of max. 5x8 rings on screen
- A DMA interrupt is used to update the new sprite position for the current row of rings
- Think in term of columns: one bitfield per column, where 0=ring hidden and 1=ring displayed
- The main challenges were clipping rings at the top of the screen, and implementing a fast collision detection routine (*more on that later*)



PIX2 sprite copy 1/3

- A 16x16 ASIC sprite takes 256 bytes in memory
- A byte being a value from 0→15 (a palette index per pixel)
- By default, it means bits 4→7 are unused
- Copying directly sprites from ROM to ASIC RAM is a waste of space
- “PIX2” gets used in Sonic GX: 1 byte in ROM (all bits used) contains 2 bytes in ASIC RAM
- A copy remains fast thanks to an optimized Z80 routine



PIX2 sprite copy 2/3

- But wait, there is more!
- During the copy process, **nothing forces us to copy data linearly from ROM to ASIC RAM**
- It's almost "free" to mirror the sprite on Y axis during the copy
- This avoids to store 2 versions of Sonic in ROM (left and right) – only right is needed
- Not used in Sonic GX: could be extended to support all rotation routines or X mirroring etc.



PIX2 sprite copy 3/3

- Just for fun, I ran some tests comparing PIX2 with the ZX0 data packer
- PIX2 is faster to process than ZX0 unpacking
- And to my surprise, on average, it also achieves better data compression than ZX0!
- But this is only true for a single sprite. Once you want to unpack a group of sprites, ZX0 becomes the clear winner for data compression



Gameplay



World vs. projected coordinates

- Always use World PosX/PosY coordinates for everything (Sonic, enemies, moving platforms, etc)
- Use “projected” positions only for the final 2D rendering on screen
- This approach makes it much easier to manage a camera that follows Sonic, collision detection, and more



Trigger boxes

- A simple invisible box placed in the world (authored in Tiled)
- When Sonic enters it, a **scripted event** is triggered
- Examples
 - Checkpoints
 - End-of-level rotating board (Eggman→Sonic)
 - The box that explodes to release the animals



Scripted events 1/4

- A scripted event is a data structure containing a sequence of actions
- Almost a scripting language! With one instruction processed per frame
- An instruction is a command identifier + a command parameter



```
; *** COMMANDS ***  
EVENTLIST_STOP = 0 ; do nothing next frame, till another address gets into Vars.wEventListAddress  
EVENTLIST_CALL_ROUTINE = 1 ; used to call a routine for this frame  
EVENTLIST_SET_SPRCOPYLIST_ADDRESS = 2 ; used to set a sprite copy list  
EVENTLIST_SET_EVENTLIST_ADDRESS = 3 ; used for loops  
EVENTLIST_SET_MAPSPRITES = 4 ; used to display map sprites  
EVENTLIST_DISABLE_MAPSPRITES = 5 ; used to disable map sprites  
EVENTLIST_WAITFRAMES = 6 ; used to wait few seconds
```


Scripted events 2/4

- Features
 - Can call a dedicated ASM routine
 - Can loop the sequence at any instruction
 - Can copy sprites from ROM to ASIC RAM
 - Can enable sprites on screen at a given position
 - Can wait for a few frames
- More than enough to animate anything in a video game!



Scripted events 3/4

- Example 1: prepare Eggman in ASIC RAM

```
;
EndGameSequence_EventList_ShowEggman:
    db EVENTLIST_DISABLE_MAPSPRITES
    db EVENTLIST_SET_SPRCOPYLIST_ADDRESS:db ENDGAMESEQUENCE_ROM:dw SprCopyList_EggMan

    db EVENTLIST_CALL_ROUTINE: dw OnEvent_CopySmoke0
    db EVENTLIST_CALL_ROUTINE: dw OnEvent_CopySmoke1
    db EVENTLIST_CALL_ROUTINE: dw OnEvent_CopySmoke2

EndGameSequence_EventList_ShowEggman_InfiniteLoop:
    db EVENTLIST_CALL_ROUTINE: dw OnEvent_ShowDrEggManAnim
    db EVENTLIST_STOP
```

Scripted events 4/4

- Example 2: the explode anim on animals box

```
;
EventList_ShowUnbrokenLapinBox_ButtonPressed:
; Show Unbroken LapinBox MapSprites With Button Pressed
db EVENTLIST_CALL_ROUTINE:dw OnEvent_SetLevelMapSprites_UnbrokenWithButtonPressed

; Wait few frames...
db EVENTLIST_WAITFRAMES:db 10

; Show Unbroken LapinBox MapSprites
db EVENTLIST_CALL_ROUTINE:dw OnEvent_SetLevelMapSprites_Unbroken

; Explode Steps
db EVENTLIST_WAITFRAMES:db EXPLODESTEP_FRAMEDELAY
db EVENTLIST_CALL_ROUTINE:dw OnEvent_SetLevelMapSprites_ExplodeStep0
db EVENTLIST_WAITFRAMES:db EXPLODESTEP_FRAMEDELAY
db EVENTLIST_CALL_ROUTINE:dw OnEvent_SetLevelMapSprites_ExplodeStep1
....
db EVENTLIST_CALL_ROUTINE:dw OnEvent_SetLevelMapSprites_ExplodeStep11
db EVENTLIST_WAITFRAMES:db EXPLODESTEP_FRAMEDELAY
db EVENTLIST_CALL_ROUTINE:dw OnEvent_SetLevelMapSprites_ExplodeStep12

; Return back to default (no explode)
db EVENTLIST_WAITFRAMES:db EXPLODESTEP_FRAMEDELAY
db EVENTLIST_CALL_ROUTINE:dw OnEvent_SetLevelMapSprites_Unbroken

; Copy Broken Base at SPR0-1
db EVENTLIST_SET_SPRCOPYLIST_ADDRESS:db DELIVERANCE_SPRITES_ROM:dw SprCopyList_LapinBox_BrokenBase

; Show Broken Base at SPR0-1
db EVENTLIST_CALL_ROUTINE:dw OnEvent_SetLevelMapSprites_BrokenBase

; Copy Unbroken LapinBox sprites
db EVENTLIST_CALL_ROUTINE:dw OnEvent_SetSprCopyList_JumpingAnimals

; Start jumping animals routine
db EVENTLIST_CALL_ROUTINE:dw OnEvent_BeginJumpingAnimals

db EVENTLIST_STOP
```

Map sprites

- It represents a group of sprites displayed on screen as a single entity
 - Checkpoint sprite, Eggman, etc.
- Only one X/Y position needs to be set
 - No need to manage the positions of all individual ASIC sprites
 - Handle sprite clipping automatically
- Possibility to define a pivot point
- Very convenient



C implementation 1/2

- I found it very useful to rely on C for the game logic implementation, especially with all those *if* statements and gameplay variables (readability)

```
// Direction change ?
if (bIsHeadingToRight != YES)
{
    // if player was in ball mode but is changing direction, quit ball mode
    if (bIsBall == YES)
    {
        bIsBall = NO;
        bIsLongJump = NO;
    }

    // Does not allow direction change when Sonic is running (only walking is accepted)
    if ((bIsJumping == YES) && (bIsWalkingOrRunning == RUN))
    {
        FUNCTION_QUIT;
    }
    else if (bSlidingFrameCount != 0)
    {
        FUNCTION_QUIT;
    }

    // Does not allow direction change when Sonic falls from the top of a wall
    // and is willing to return back to the opposite direction
    wColliderEdgePosY = GetGroundElevation(wSonicWorldPosX, COLLIDER_RIGHT_OFFSET + COLLIDER_RIGHT_EDGE_POSX);
    if (wColliderEdgePosY < wSonicWorldPosY - 24)
    {
        FUNCTION_QUIT;
    }
}
```


C implementation 2/2

- Z80 code generated by SDCC 4.5.0 is great
 - Approx. 10 raster lines, I'm fine with that
- SDCC converts C→ASM→Z80 code, but I only use the C→ASM step and wrote my own translator to the RASM assembler, since its syntax is different
- I don't use anything from the C standard library -- just plain C syntax!
- For performance reasons, I avoid passing parameters on the stack and rely entirely on global variables. That's the most obvious optimization with C for Z80!



Collisions



Sonic vs. enemies

- Sonic vs. enemies use simple collision boxes:
 - We first update the new world positions for both Sonic and the enemies
 - Then, iterate through all close enemies
 - Is Sonic's bounding box colliding with an enemy's bounding box?
 - If Sonic was jumping or rolling ball → enemy dies
 - Sonic dies otherwise
 - No need to be complicated



Sonic vs. rings

- Sonic's collisions with Rings use a similar box-based collision detection routine as the one used for enemies
- Easy!



Sonic vs. multiplexed rings (1/2)



Sonic vs. multiplexed rings (2/2)

- We need to handle an array of up to 5x8 rings
- My first (naive) approach was to check collisions between Sonic and every single ring individually → too much time consuming
- So I had to come up with a different solution
- Since the multiplexed sprites form a 2D grid (the visibility bitfield), I calculate Sonic's position within that grid and test the collision against a single box → just one ring gets tested!



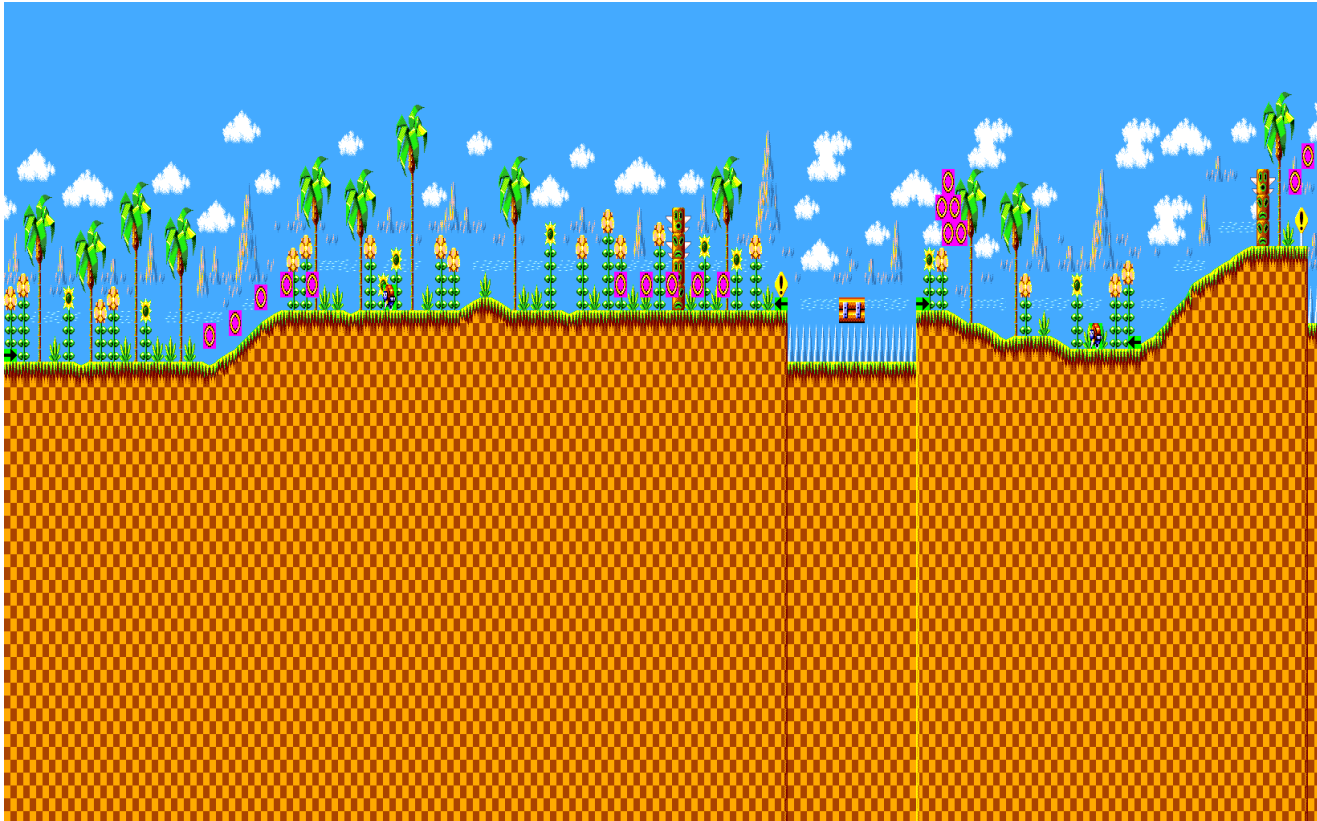
Collisions (Sonic vs. ground) 1/5

- Sonic vs. terrain is handled separately: for each X position in the map, an elevation Y defines the ground level
- When Sonic jumps, his falling position is checked against the elevation map
- Walls are easy to detect (a big elevation difference between X and X+1)
- It's actually a bit more complex (support for "tunnels" with multiple elevation levels, vertical levels are problematic, but the overall approach remains the same)



Collisions (Sonic vs. ground) 2/5

- The map in Tiled...



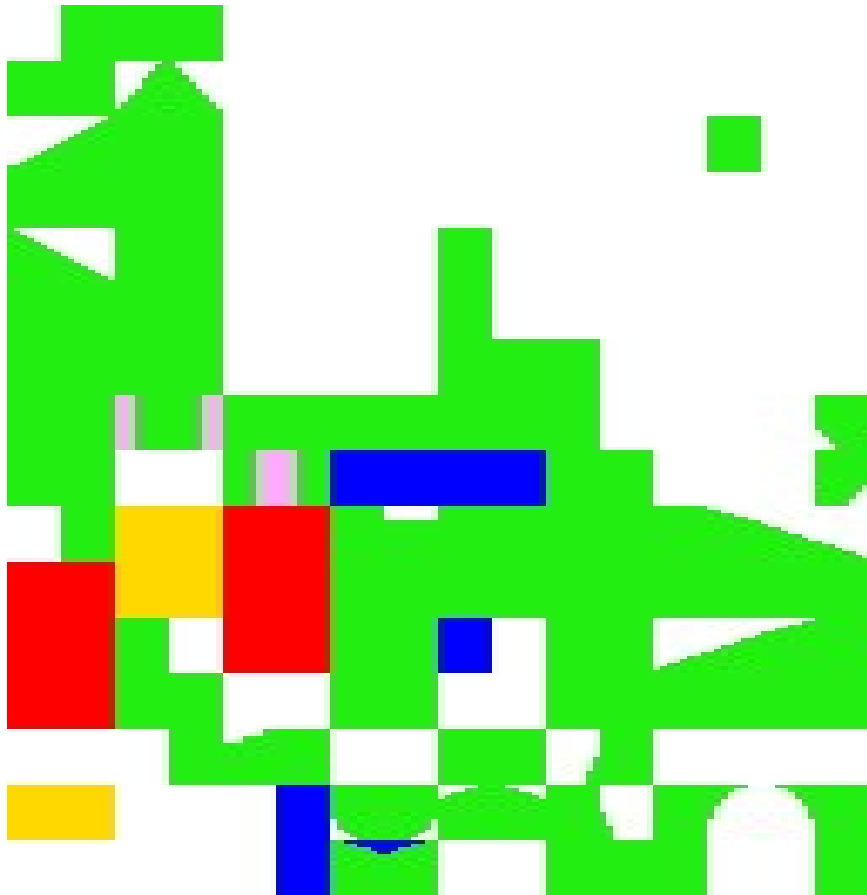
Collisions (Sonic vs. ground) 3/5

- The corresponding tileset...



Collisions (Sonic vs. ground) 4/5

- The corresponding collision mask tileset...



WHITE: air
GREEN: ground
BLUE: water
RED: die (spikes...)
ORANGE: jumper



Collisions (Sonic vs. ground) 5/5

- This is corresponding generated collision map
- For each X position in the map, **read** the corresponding Y elevation and check its **collision status**: Green for ground, Red for die, Yellow for jumpers and so on.



General development tips



Forget the VBL

- On old Amstrad CPCs, we used to wait for the VBL
- With ASIC programming, you can instead trigger an interrupt on the last visible line to start a new frame
- This gives you some extra-time to perform more initialization routines before the first visible line of the next frame



Sprite update vs. raster beam

- This may surprise you, but if you update the content of a sprite while it's being rendered at the current raster beam position...
... then some white artifacts gets displayed on screen!
- To my knowledge, no emulators simulates that behavior yet
- To prevent this, try updating your sprite either before or after the raster beam's location



Context for hidden sprites

- A sprite rendered at position PosX=-80, -79 or -78 will be hidden from the display in the same way
- So instead of -80, -79 or -78, you can define constants like `HIDDEN_FROM_<reason>`
- Those values remain visible when debugging in an emulator, so you can know exactly how the sprite got hidden
- Could be also applied to PosY, or PosY>256, etc.



Controller schemes

- Testing for key or button presses is CPU-intensive on the Amstrad
- Advice: don't check multiple inputs (example: Button 1, Control key, and Z key) for the same action
- Instead, let the player press a key in the game menu to define their control scheme
- If they press the Control key, use that input method for the rest of the game and ignore the others



Determinist timing

- Avoid using separate time counters -- they quickly become time-consuming to maintain
- Instead, rely all your time-based operations on a single 16bit frame counter
 - You can access it as a 8bit value too
 - You can even AND it for a smaller range
- When calculating the position of an enemy, a moving platform, or similar elements, always make the value depend on that time counter
- This ensures the same in-game experience every time a level starts or is restarted



Use "shadowed" sprites

- Never update your sprite positions while in the middle of your frame
- Instead, store the new positions in dedicated variables, and apply them at the start of next frame (when the visible screen is not yet displayed)
- Much cleaner, no visual glitches



Configuration files

- Avoid hard-coded values directly in your implementation
- Instead, Sonic GX relies on 3 dedicated configuration files:
 - **GlobalConfig.asm**
 - **GameConfig.asm**
 - **BonusConfig.asm**
- These files only contain constants, allowing the entire game to be tweaked from there



Not everything needs to be optimized

- Optimize only where you spend the most of your time
- Refer to the good old “80/20” optimization rule:
“Usually only 20% of executed code requires to be ultra-optimized”
(and that’s what I did in Sonic GX)
- Drawing/copy operations are usually slow, that’s where your attention should go first!



Your V1 sucks!

- Iterate, and iterate again
- The first version can often be optimized, and optimized again
- That's perfectly fine to hit the DELETE key on existing code, even if it took you 3 weeks to write or takes 3000 lines of code
- Be result-oriented



Good design sucks! (1/2)

- So you built well-structured modules, all nicely separated with independent routines... reusable code... all variables nicely packed together...
 - “I’m writing object-oriented code in Z80 assembly!”
 - “It’s totally data-driven!”
 - “I wrote unit-tests!”
- Well, I agree... *BUT IT’S A TRAP.* ☺



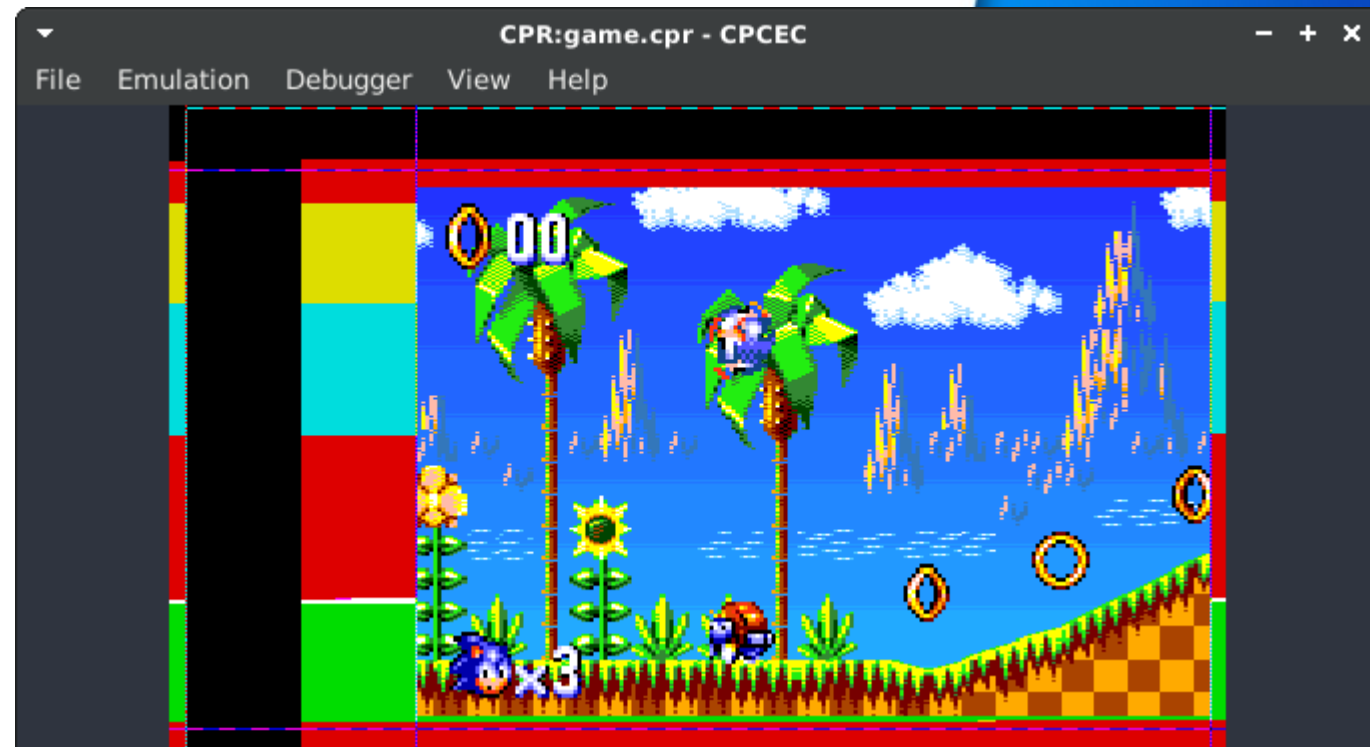
Good design sucks! (2/2)

- Z80 heavy optimizations often rely on:
 - self-modifying code
 - global variables
 - included files in unexpected places
 - pre-generated code
 - code that isn't interrupt-friendly
 - repeated code
 - wasted memory (due to alignment constraints)
- *No worries, it's Halloween-time!* 🎃



Benchmarking

- Those good old raster-lines are still useful for a quick overview
- One color per major processing routine (yellow/turquoise for column/row drawing...)



Naming conventions 1/2

- I use the **prefix “b” for byte values** and **“w” for word values**
- The variable name immediately tells me whether I’m dealing with a byte or a word
- **BAD:**

LifeCount: db 0

FrameCounter: dw 0

- **GOOD:**

bLifeCount: db 0

wFrameCounter: dw 0



Naming conventions 2/2

- I use the **“RAM_”** prefix for code/data stored in RAM
- Very practical guideline, since most code/data should ideally reside in ROM to minimize RAM usage



Data structures

- RASM provides support for Structures
- I use 2 structs in RAM:
 - **GlobalVars** (life count, current level, etc)
 - **GameVars** (Sonic World Pos X/Y, etc)
- All variables are nicely packed into the same memory area
- The Special Stage has also its own BonusVars structure. Both GameVars and BonusVars actually share the same fixed location in memory!



Memory footprint report+build for every Git commits (1/3)

- PCs are made to suffer; I don't care if we don't only store code/data files in Git
- For every commit, two files are always updated:
 - a small text file describing the cartridge's memory layout (handy for comparing different versions)
 - the current CPR cartridge file
- This practice has saved countless times when chasing regressions between Git commits!



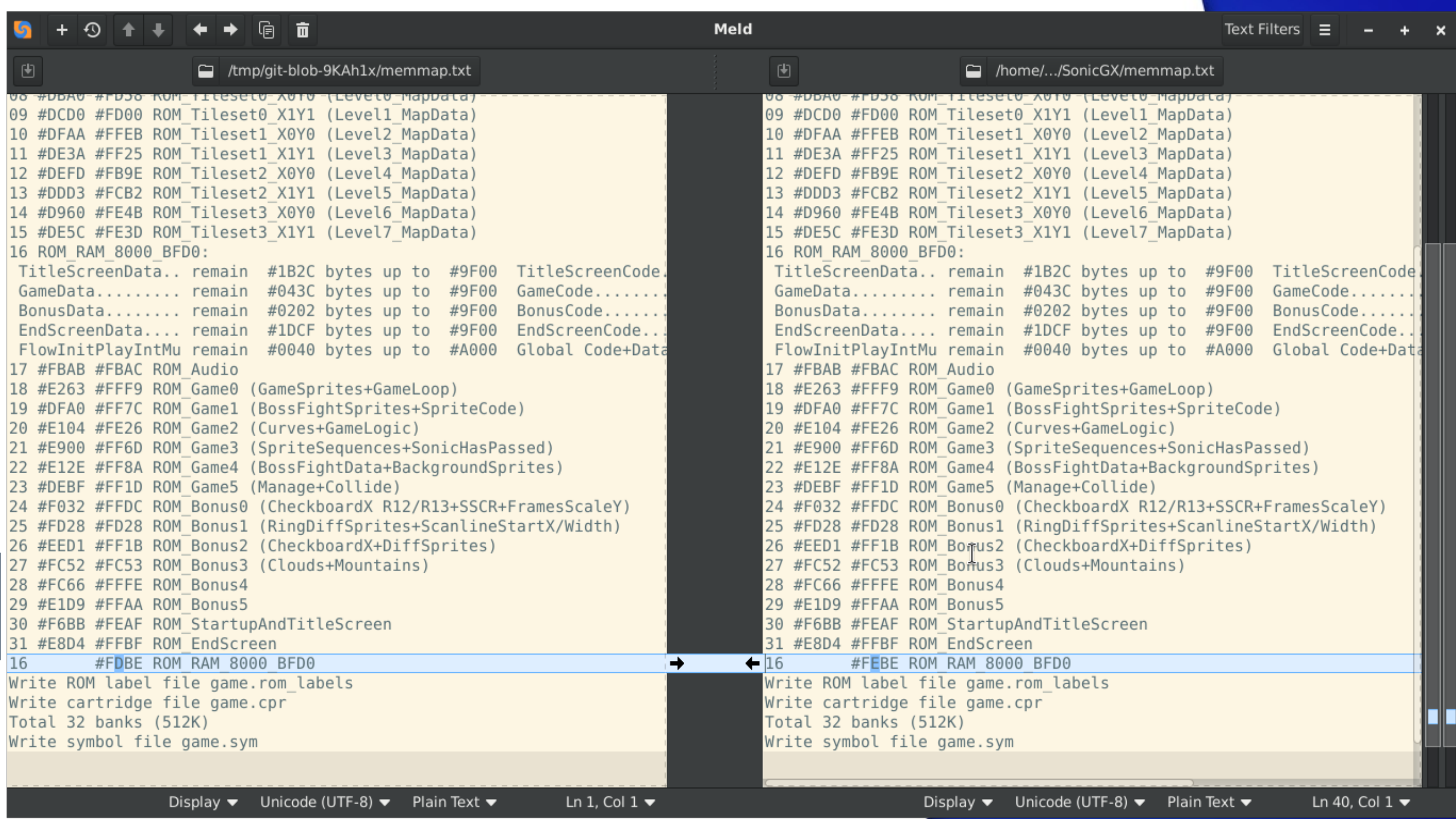
- **Result of a build**
(Terminal output)

- Then DIFF in next slide...

```

00 #DC65 #FD78 ROM_Tiles0 X0Y1 (BigIndices0/1+BigTiles0)
01 #DFE0 #FD0B ROM_Tiles0 X1Y0 (Tiles0 Tile0opcodeOffsets+BigTiles1)
02 #DD5C #FB0B ROM_Tiles1 X0Y1 (BigIndices2/3)
03 #DCA0 #FFC8 ROM_Tiles1 X1Y0 (Tiles1 Tile0opcodeOffsets+BigTiles1/3+BONUS CODE)
04 #DECB #FF00 ROM_Tiles2 X0Y1 (BigIndices4/5)
05 #DD71 #FAC1 ROM_Tiles2 X1Y0 (Tiles2 Tile0opcodeOffsets+BigTiles4/5)
06 #DDD0 #FCF3 ROM_Tiles3 X0Y1 (BigIndices6/7+BONUS CODE/DATA)
07 #DD52 #FE54 ROM_Tiles3 X1Y0 (Tiles3 Tile0opcodeOffsets+BigTiles6/7)
08 #DBA0 #FD58 ROM_Tiles0 X0Y0 (Level0_MapData)
09 #DCD0 #FD00 ROM_Tiles0 X1Y1 (Level1_MapData)
10 #DFAA #FFEB ROM_Tiles1 X0Y0 (Level2_MapData)
11 #DE3A #FF25 ROM_Tiles1 X1Y1 (Level3_MapData)
12 #DEFD #FB9E ROM_Tiles2 X0Y0 (Level4_MapData)
13 #DDD3 #FCB2 ROM_Tiles2 X1Y1 (Level5_MapData)
14 #D960 #FE4B ROM_Tiles3 X0Y0 (Level6_MapData)
15 #DE5C #FE3D ROM_Tiles3 X1Y1 (Level7_MapData)
16 ROM_RAM_8000_BFD0:
   TitleScreenData.. remain #1B2C bytes up to #9F00 TitleScreenCode. remain #0421 bytes up to #B400
   GameData..... remain #043C bytes up to #9F00 GameCode..... remain #0002 bytes up to #B400
   BonusData..... remain #0202 bytes up to #9F00 BonusCode..... remain #0700 bytes up to #B400
   EndScreenData... remain #1DCF bytes up to #9F00 EndScreenCode... remain #12EA bytes up to #B400
   FlowInitPlayIntMu remain #0040 bytes up to #A000 Global Code+Data. remain #0046 bytes up to #BFD0
17 #FBAB #FBAC ROM_Audio
18 #E263 #FFF9 ROM_Game0 (GameSprites+GameLoop)
19 #DFA0 #FF7C ROM_Game1 (BossFightSprites+SpriteCode)
20 #E104 #FE26 ROM_Game2 (Curves+GameLogic)
21 #E900 #FF6D ROM_Game3 (SpriteSequences+SonicHasPassed)
22 #E12E #FF8A ROM_Game4 (BossFightData+BackgroundSprites)
23 #DEBF #FF1D ROM_Game5 (Manage+Collide)
24 #F032 #FFDC ROM_Bonus0 (CheckboardX R12/R13+SSCR+FramesScaleY)
25 #FD28 #FD28 ROM_Bonus1 (RingDiffSprites+ScanlineStartX/Width)
26 #EED1 #FF1B ROM_Bonus2 (CheckboardX+DiffSprites)
27 #FC52 #FC53 ROM_Bonus3 (Clouds+Mountains)
28 #FC66 #FFFE ROM_Bonus4
29 #E1D9 #FFAA ROM_Bonus5
30 #F6BB #FEAF ROM_StartupAndTitleScreen
31 #E8D4 #FFBF ROM_EndScreen
16 #FDBE ROM_RAM_8000_BFD0
Write ROM label file game.rom_labels
Write cartridge file game.cpr
Total 32 banks (512K)
Write symbol file game.sym
norecess464@x220:~/Development/Bitbucket/SonicGX$

```

Closing words



Great things take time (1/2)

- IMO our most memorable works from Condense on the Amstrad:
 - **Pheelone** demo: **9 months**
 - **Phortem** demo: **1.5 years**
 - **PhX** demo: **5 years**
 - **Sonic GX**: **7 years**



Great things take time (2/2)

- It's perfectly fine to spend **years** developing something
 - The Amstrad CPC is 40+ years after all
 - Nobody is waiting on you
- It's perfectly fine to progress slowly
 - It's just a hobby, so don't "force it"
 - What matters is to make coding a weekly routine
- The more it's polished, the better the reception will be!



Reference docs

- Many online tutorials are incomplete
 - But they may help as a starting point
- **Arnold V Revised spec** doc on cpcwiki is very good (https://www.cpcwiki.eu/index.php/Arnold_V_specs)
- **Plus Vectored Interrupt Bug** on cpcwiki is very good to understand correct usage of DMA interrupts (https://www.cpcwiki.eu/index.php/Plus_Vectored_Interrupt_Bug)
- **Z80 Timing Cheat Sheet** (https://www.cpcwiki.eu/imgs/b/b4/Z80_CPC_Timings_cheat_sheet.20230709.pdf)



Thanks

- **Gerald** for the C4CPC
- **TotO**, **Rabs** and **Nemo Kantio** for their extensive testing and spot-on feedback
- **Targhan** for the amazing, perfectly converted audio
- **CeD** for the INCREDIBLE graphics conversion
- **Slype** for the instruction manual
- **Overflow** and **Longshot** for their technical discussions when the project started
- Everyone from **CPCWiki**, **CPCRULEZ** and the **demoscene community**
- And **YOU!** *Damn, it's 2025 and we are still poking bytes with a Z80! =)*



Q&A
(GAME OVER :)
(we just went through 100 slides!)

