Significant tricks & techniques in MW4 by Cadaver

(This rant is mirrored from Cadaver's site)

This rant details some of the more out of the ordinary techniques used in the finished version of MW4 that make it possible to be what it is. None of them is anything special, but they're still not usually used in C64 action games, so I thought to write something of them..

Please refer to the MW4 source code to see what I'm talking about, http://covertbitops.c64.org/games/mw4src.zip (Codebase64 mirror of file: mw4src.zip).

0. Free/anydirectional scrolling

(Also discussed in Rant #4) This is like the block scrolling in any of my games, in that all action still happens in 2 routines:

- SCROLLLOGIC, determine where to scroll the screen, update “subtract-value” to position sprites correctly, determine if shifting the screen is required
- SCROLLWORK, hard work of scrolling (screen memory/color memory shifting and drawing new data to the edges)

SCROLLLOGIC is still called at the beginning of the frame, so that the sprite subtract is correctly updated and we know what the scrolling values are going to be for the rest of frame.

SCROLLWORK is called in the end of the frame, respectively.

The difference to “usual” 8-way scrollers comes in the SCROLLLOGIC routine. Usually scrollers pick one of the 8 directions, and advance the scrolling for a certain amount of frames until a distance of 1 char has been scrolled (Turrican for example).

But here the algorithm doesn't think that much forward. What it starts with are the current finescroll values (scrollx, scrolly) and the current scrolling speed (scrollsx, scrollsy). Btw. those values have 3 bits of subpixel accuracy, so they go from 0-63 instead of 0-7. The scrolling speed is a signed number, so it goes from +32 to -32 (speeds higher than 4 pixels/frame are unsupported, as screen shifting can happen on each second frame at most).

The first step is to subtract the speed from the finescroll values, separately for X and Y axes. If the finescroll values overflow, we don't care of it yet, instead we just clamp them at the ends of the finescroll range (0 or 63). This is because we are not yet ready to perform a complete screen shifting in the space of one frame!

Now we have the finescroll values for the next frame!

The next step is to subtract the speed from finescroll yet again. This time we check overflow in either positive or negative direction, and by doing this for both axes we know where the screen needs to be

shifted. If the finescroll *does not* overflow, we can discard the values from this second subtraction, no shifting is necessary, and the process starts over on the next frame.

But if we get an overflow, this tells that the screen has to be shifted. Now the rest of the process goes like this:

- A flag is set so that next time SCROLLLOGIC is executed, it just takes into use the precalculated finescrollvalues that we got from the second subtraction, and updates the sprite-subtract, but doesn't do anything else.
- At the end of the frame, SCROLLWORK will shift the screen memory. As double buffering is used, it's always a copy from the currently visible screen to the currently hidden screen, with an offset that is determined by the shifting direction. After the shift, new data is drawn to the screen edges.
- At the end of the next frame, SCROLLWORK will shift the color memory. This requires checking that the rasterbeam ($d012) is at a suitable position (at the scorescreen, or at the vertical blank) as to not cause tearing effects on

the screen.

The file “screen.s” contains the SCROLLLOGIC & SCROLLWORK routines.

1. Realtime depacked sprites

This is also something I've written about before, but how it's used in the “new” MW4 engine is a bit different.

All main characters and enemies (about 160 frames loaded at once) still reside in the videobank memory: these are not unpacked in real time, as it would be quite impossible to achieve 50Hz frame rate while doing that.

But the weapon carried by each character/enemy, all bullets/other projectiles, items lying on the ground, and “particle” effects like small smoke clouds use packed sprites. There's room for 20 packed sprites in the video bank, which is also the max. number of actors active at once, so each actor can utilize 1 packed sprite.

Packed sprites are divided into 6 “slices”, consisting of 7 bytes, like this: (each string of four numbers represents four multicolor pixels - one byte)

```
1111 2222 3333
1111 2222 3333
1111 2222 3333
1111 2222 3333
1111 2222 3333
1111 2222 3333
1111 2222 3333

4444 5555 6666
4444 5555 6666
```

```
4444 5555 6666
4444 5555 6666
4444 5555 6666
4444 5555 6666
4444 5555 6666
```

As you see, the lowest 7 rows of a packed sprite aren't used at all, to make the depacking process quicker (the lowest 7 rows have been cleared beforehand, when the program starts - this needs to be done only once). All packed sprites are so small that they fit into the 6 slices.

For each slice, one bit determines whether it's empty (0) or whether it contains 7 bytes of data (1). Furthermore, packed sprites are stored only facing right. If they need to be displayed flipped, the flipping is also performed realtime in the program,

One more thing, before I present the actual depack routines from “sprite.s”. The depacked sprites are not doublebuffered, so they need to be depacked when the raster beam is at the scorescreen, or at the vertical blank. Since the color-RAM scrolling “competes” of this time too, I thought I was going to run into trouble, but experience indicates that during a single frame, not so many packed sprite frames change (naturally, we don't want to waste time depacking the same sprite again), and therefore not so many have to be depacked, so there's enough time even on NTSC machines.

The unflipped sprite depacking is quite straightforward. This code depacks one slice. The destination address uses X as index; the highbyte of the address has to be modified into the code. For the source address, variables temp3-temp4 + Y register are used for zeropage indirect addressing.

```
dspr_slice:     lsr temp2                       ;Check slice type
(empty/data)
                bcc dspr_emptyslice
dspr_fullslice: lda (temp3),y
dspr_fullsta1:  sta $c000,x
                iny
                lda (temp3),y
dspr_fullsta2:  sta $c000+3,x
                iny
                lda (temp3),y
dspr_fullsta3:  sta $c000+6,x
                iny
                lda (temp3),y
dspr_fullsta4:  sta $c000+9,x
                iny
                lda (temp3),y
dspr_fullsta5:  sta $c000+12,x
                iny
                lda (temp3),y
dspr_fullsta6:  sta $c000+15,x
                iny
                lda (temp3),y
dspr_fullsta7:  sta $c000+18,x
                iny
                rts
```

```
dspr_emptyslice:lda #$00
dspr_emptysta1: sta $c000,x
dspr_emptysta2: sta $c000+3,x
dspr_emptysta3: sta $c000+6,x
dspr_emptysta4: sta $c000+9,x
dspr_emptysta5: sta $c000+12,x
dspr_emptysta6: sta $c000+15,x
dspr_emptysta7: sta $c000+18,x
                rts
```

When also flipping the sprite, things become harder. I use a 256-byte lookup- table to flip the bit-pairs in each byte, but the problem is that there's no free index register to access that, so we have to modify the code to access the fliptable instead (a bit slow). Fortunately depacking an empty slice is still easy & fast.

```
dsprl_slice:    lsr temp2                       ;Check slice type
(empty/data)
                bcc dsprl_emptyslice
dsprl_fullslice:lda (temp3),y
                sta dsprl_fulllda1+1
dsprl_fulllda1: lda fliptable
dsprl_fullsta1: sta $c000,x
                iny
                lda (temp3),y
                sta dsprl_fulllda2+1
dsprl_fulllda2: lda fliptable
dsprl_fullsta2: sta $c000+3,x
                iny
                lda (temp3),y
                sta dsprl_fulllda3+1
dsprl_fulllda3: lda fliptable
dsprl_fullsta3: sta $c000+6,x
                iny
                lda (temp3),y
                sta dsprl_fulllda4+1
dsprl_fulllda4: lda fliptable
dsprl_fullsta4: sta $c000+9,x
                iny
                lda (temp3),y
                sta dsprl_fulllda5+1
dsprl_fulllda5: lda fliptable
dsprl_fullsta5: sta $c000+12,x
                iny
                lda (temp3),y
                sta dsprl_fulllda6+1
dsprl_fulllda6: lda fliptable
dsprl_fullsta6: sta $c000+15,x
                iny
                lda (temp3),y
                sta dsprl_fulllda7+1
dsprl_fulllda7: lda fliptable
```

```
dsprl_fullsta7: sta $c000+18,x
                iny
                rts

dsprl_emptyslice:lda #$00
dsprl_emptysta1: sta $c000,x
dsprl_emptysta2: sta $c000+3,x
dsprl_emptysta3: sta $c000+6,x
dsprl_emptysta4: sta $c000+9,x
dsprl_emptysta5: sta $c000+12,x
dsprl_emptysta6: sta $c000+15,x
dsprl_emptysta7: sta $c000+18,x
                rts
```

MW4 uses a total of 114 packed spriteframes, and when it's taken into account that many of them are also shown flipped, the number rises to about 170. So, the game would clearly have been impossible to implement without packed sprites!

2. Flipping of sprites while loading them

Near the end of MW4 development, I began to run out of diskspace. It would be nice to have at least 3 save game slots, but they'd take 48 blocks off the disk. To somewhat help this,“flip duplicates” of sprites aren't saved on disk, instead there's just emptiness at their place (Exomizer packs that quite well) and a “command code” to instruct the sprite loading routine to rebuild these flip duplicates at load time.

This routine is also in “sprite.s”. The trouble with it is that sprites can also reside under the I/O area, so it needs to be switched off/interrupts be disabled. Of course, this can't happen for a prolonged time, or the raster interrupts would freak out.

The “fliptable” is also used by this routine, and the process goes like this for one sprite: temp2 is the row counter, alo-ahi are the zeropage source pointer, and tempadrlo-tempadrhi are the destination pointer. The routine first takes the rightmost source byte of a sprite row, flips the bitpairs, and puts it to the leftmost byte of destination sprite, then flips the middle byte, and then, at last, leftmost source byte is flipped and put into the rightmost destination byte. This process is repeated for all 21 rows of the sprite.

```
                lda #21                 ;Row counter
                sta temp2
                clc
loadspr_rowloop:ldy #2
                sei
                dec $01
                lda (alo),y
                ldy #0
                tax
                lda fliptable,x
                sta (tempadrlo),y
```

```
                iny
                lda (alo),y
                tax
                lda fliptable,x
                sta (tempadrlo),y
                dey
                lda (alo),y
                ldy #2
                tax
                lda fliptable,x
                sta (tempadrlo),y
                inc $01
                cli
                lda tempadrlo   ;Sprites never cross page boundaries
                adc #$03
                sta tempadrlo
                lda alo
                adc #$03
                sta alo
                dec temp2
                bne loadspr_rowloop
```

3. Scripting system

The scripting system as implemented in the “new” MW4 engine is truly raw. No interpreters, no virtual machines, no paged bytecode, just loading of ASM code in 2KB chunks and executing it. All code corresponding to this is in the file “script.s”.

Scripting performs various activities of the game such as the title screen, parts of the game menu system, starting a new game, conversations and such. There are 2 ways to call a script:

- “One-shot”: the X register is loaded with scriptfile number, A is loaded with entrypoint number, and the routine EXECSCRIPT is called. If the scriptfile number is not what is currently loaded, EXECSCRIPT loads the new scriptfile first. In the beginning of each 2KB script chunk, there is a jump table for the entrypoints, and so it knows where to jump.
- “Continuous” or “latent” execution. The game mainloop (in “main.s”) will call a certain script routine, using the EXECSCRIPT call, each frame until told to stop or execute a different script routine.

A script routine has no “state information”, so it must use the game's variables (defined in “var.s”) to know where it's going. For example, in the beginning, when Ian gets hit by the alien craft, the appearances/actions of the other characters are timed by a simple delay counter.

To ease the pain of scripting, I defined some macros, these are in the file “scriptm.s”. Most of them pertain to conversations.

An important concept of the scripting system is to have access to all subroutines and variables of the game main program. Therefore, all its symbols are dumped in the makefile for use by the script

routines. A script routine can naturally crash the game if it wants to, so care had to be exercised when writing them, just like writing the main engine code.

Care must also be taken of what routines are called in the script routine: if it JSRs off to a subroutine that also calls EXECSCRIPT, a different script file might be in memory upon return and a crash would be inevitable. Therefore, whenever this is suspected, the script routine uses a JMP instead, or sets a latent script to be executed on the next frame.

Because each script routine is identified by a 16bit number (highbyte = file number, lowbyte = entrypoint), assigning script routines to the objects (switches, computers and such) in the levels was easy using the level editor (AOMEDIT2). The bytecode-based scripting system of the preview wouldn't have allowed that as easily.