What I Learned While Writing Boot Sector Games in Pure Assembly Language

Krzysztof Krystian Jankowski

THIS IS A FIRST DRAFT
Mon, 5th of 08/2024

I'm an indie game developer. I made a lot of games in many languages raging from basic, lua, python to C, C++, and Turbo Pascal. A huge number of JavaScript nonsense. Recently I was concentrated on retrocomputers, 386 era. Making those, low level, DOS prototypes I learned a lot about VGA graphics and memory management. I did one simple text-base adventure in ARM assembly few years ago. I was prepared for the real challenge - x86 boot sector assembly. In this article I will tell you the story, show some code tricks, and comment on the source of games I made so far for this category.

Make some tea, sit comfortably, and enjoy the world of pure assembly on a bare metal x86 platform.

Introduction to the Boot sector Assembly

What is a boot sector assembly you may ask? When an IBM PC starts it runs BIOS code and then search for the boot sector code. This can be the first sector of a floppy or a disk drive. It is sized at exactly 512 bytes. No more no less. Less space needs to be zero-ed as the last data of this sector needs to include a boot sector signature. I'm also adding a "P1X" characters next to it so if anyone reads the HEX dump of this file the name will appear on the screen.

This boot sector code has only one purpose - to find a sector with the Operating System and runs it, sometimes writing familiar error message after no OS was found. This is just a code - a clever programmer can make other use for it - make a demo, or a game, replacing boot sector code. The biggest benefit of this hack is that the produced binary does not need any operating system as it runs directly on the hardware - 16-bit era, Intel CPU. BIOS provides input from a keyboard and output to the VGA. 320x200 in glorious palette of selected 256 colors (16bit RGB). Classic. It also gives direct access to the memory. That is all you need to make a demo or a game. I shoot for a game, I'm a game developer not a demo scener, at least at the moment this statement is fully true.

Getting right knowledge was not that hard as there are a lot of resources for a size coding on the internee[1]. There is also a very helpful group on Discord[2] dedicated fully to the topic of (very) small demos. Mostly 256 bytes in size.

[1] size coding.org / x86
[2] discord group

Let's make it straight - to make anything usable I must fit in a 512 bytes or around 200 processor commands. Any data also needs to fit. Fighting for that last byte, optimizing code, doing all the dirty tricks you can think of is the essence of this journey.

Bootstrapping The Application

Not all 512 bytes are free to use - there is a boilerplate of code required before our code can start. In essence it can be as short as setting up 16 bit mode, memory positions for VGA graphics, boot sector code, and the signature at the end. This will take XXX bytes and everything between is for us. Even in this boilerplate part we can use some tricks to make it smaller.

```
mov ax, 0x0
mov ds, ax    ; clear ds
mov ax, 0xA000
mov es, ax    ; set output adress to VGA screen memory
```

This can be optimized as follow:

```
xor ax, ax    ; 1 bit less
mov ds, ax
push 0xA000    ; 1 bit less
pop es        ; 1 bit less
```

We saved 3 bits!

One of the first challenges I got was to keep the game loop in consistent peace - some kind of delay synced to the CPU clock or VSYNC. In the end I got two versions: one for pure boot sector and other for DOS executable.

Boot sector/BIOS version:

```
delay:
  mov ax, [0x046c]    ; Get current clock time
  inc ax              ; Increment it by 1 cycle (42ms)
  .wait
    cmp [0x046c], ax  ; Compare with the current timer
    jl .wait          ; Loop until equal (Jump Less)
```

Version for a DOS systems:

```
delay:
  push es               ; Save es (VGA)
  push 0x0040
  pop es                ; Set es to 0x0040
  mov bx, [es:0x006c]   ; Save the current tick
  .wait
    mov ax, [es:0x006c] ; Load current tick
    sub ax, bx          ; Calculate elapsed ticks (new - old)
```

```
    jz .wait            ; Loop until equal (Jump Zero)
    pop es              ; Load es back
```

As we are on the DOS versions - those needs proper exit code and checking for ESC to halt. It can be size coded to just 5 lines:

```
esc:
  in al, 0x60     ; Read keyboard port
  dec al          ; ESC is 1, if we dec it by 1 it will equal 0
  jnz game_loop   ; Loop if anything else (Jump Not Zero)

exit:
  push 0x0003     ; Text mode
  pop ax
  int 0x10        ; Back to text mode
```

This should be the last executable line in the code. I keep my procedures and data after this point.

Graphics on the Screen

At this point I got a proper loop, can exit the application cleanly and have a memory ready to write into it. Next step was to drawn a sprite on the screen.

How to actually "draw" anything with just manipulating memory? It turns out not that hard after all, VGA screen is a one, long, continuum list of pixel colors - starting from the upper-left corner of the screen, going all the way up to the right, then it wraps to the first pixel of the next line - on the left side - and the whole process repeats, up until the last pixel in the bottom-right corner will be satisfied.

To "draw" a pixel - set given address as color - we just need to move the color value to the correct memory position. VGA graphics in 0x13 mode has 200 lines, each 320 pixels long - this gives us 320x200 pixels long memory line. First pixel will be at position 0 and the middle one at 320*100+160 - this means 100th line (320*100) and 160 row (320/2), and last one as simple as 320*200.

Colors - we got default palette of 256 colors at start. It is easy to make custom palette, wasting space, keeping it or not is one of the hard decisions one must decide to choose in sub-512 bytes code space. Given the limit, the base 256 colors are fine, including gradients and three levels of brightness. First sixteen colors are for compatibility purposes - before VGA we got only those colors.

The Sprite

Ahh, a sprite. What it is and what it can be. A building stone for every game imaginable, at least in the 2D world, the only one we know at the times of 386 processors. Even Wolfenstain has sprites. We needed to wait for the first Quake to change that.

In my case I needed as small and optimal sprites imaginable - to save precious space, for data itself and code that displays it on the screen. This means 1-bit, 8 pixels wide sprites. You can store the whole line of a sprite in one byte. 8x8 will take 8 bytes. Each 1 in the data represents pixel light up and the each 0 represents transparency - this sprite is skipped.

This is how a simple smile sprite is encoded, first in the binary format:

```
sprite_smile:
  db 00000000b   ; 0x00 in hexadecimal
  db 01100110b   ;
  db 01100110b   ;
  db 00000000b   ; 0x00
  db 10000001b   ;
  db 01000010b   ;
  db 00111100b   ;
  db 00000000b   ; 0x00
```

You can clearly see the image just looking at the code. This is good for prototyping but for the final code it should be converted to the hexadecimal values:

```
sprite_smile:
  db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
```

Drawing Sprite

Once we got the data set and loaded, it is time to draw the sprite on the screen. This is a two step process: 1) set sprite settings, 2) manipulate memory accordingly by teh draw_sprite procedure. Let's start with the color and sprite data address:

```
mov bl, 0xF             ; 15 is white color
mov si, sprite_smile    ; Set source pointer to the sprite data
mov di, 320x100+156     ; Set position to the center of the screen
call draw_sprite        ; Call the procedure
```

Procedure expect color in bl and sprite data (8 lines) in si - source address, and a position set in di - destination adress. After the call we should see a nice smile at the center of the screen in white color.

Here is the most size shrieked version of the drawing procedure I got so far. It reads one byte - a line - of a sprite data, shift left it 8 times for each pixel, compare the removed value and set color in that place or skips it, all of this repeated 8 times for each line. Let's jump to the code:

```
draw_sprite:
  mov dx, 8             ; dx will hold the number of lines
  .draw_row:
    mov al, [si]        ; Read the first byte of a sprite to al
    mov cx, 8           ; Set cx counter to 8 pixels (wide)
    .draw_pixel:
      shl al, 1         ; Shift left the data by 1, this sets
                        ; the carry flag representing value
                        ; removed by shifting - left most pixel
      jnc .skip_pixel   ; if 0 then skip this pixel
                        ; using Jump Not Carry
      mov [es:di], bl   ; Set destination memory to color (bl)
    .skip_pixel:
      inc di            ; Increase destination adress - next pixel
    loop .draw_pixel    ; Loop back until cx more than zero
    inc si              ; Increase source address - next line
    add di, 312         ; Move destination address to next line
                        ; 320 for a line minus 8 pixels sprite
                        ; width, hence 312
    dec dx              ; Decrease dx after each line
    jnz .draw_row       ; Jump if dx > 0 using Jump Not Zero
  ret                   ; Return
```

Double Buffering for Smooth Animations and lack of flickering

To avoid flickering on the screen that is a typical problem when we update the memory of the VGA at the wrong time - when it is actually drawing something on the screen - one can implement Double Buffering. This technique dedicates part of the available memory to a screen buffer - in the game loop this memory is updated as needed, and in the end pushed to the main VGA memory in a few consequent, and very fast, copying from buffer. No more flickering. The downside is that it shrinks our space for the game play code or data even more. Any game needs this to be playable so it's a good compromise.

At start of the program set es to the new buffer position - 0x1000 in this case. Overall code will look like this:

```
push 0x1000
pop es

game_loop:

  ...manipulate memory by game code...

  VGA_blit:
    push es
    push ds           ; Save es, ds

    push 0xA000
    pop es            ; Set target as VGA
    push 0x1000
    pop ds            ; Set source as Double Buffer
    mov cx, 0x7D00    ; We will be moving words ( 4 bits )
                      ; So it needs to be repeated 1/4 of
                      ; the length of buffer - 0x7D00
    xor si, si        ; Clear source pointer
    xor di, di        ; Clear destination pointer
    rep movsw         ; Repeat (cx-times) move word

    pop ds
    pop es             ; Restore ds, es

jmp game_loop
```

And that is it. The engine is ready for the actual game logic code and some nice sprites.

Game play Mechanics I Did So Far

I made six games, four of them playable and actuary finished and published on my site, pouet, and itchio for any financial support. In each of them I tried something new, mostly to have a new thing to crack and learn something of it. The once that did not finish as playable game used as a jumping place for a next project with some new trick.

Land Me
Basic Engine
First Dive into the Bootsector Waters

I wanted to start very simple - just enough to learn the basic. One button gameplay. Player mission is to safely land a roket on the ship, that is drifting rapidly over the ocean. Payer only changes direction, left or right, to steer the rocet. To make the game interesting I've added obstacles, stationary floating platfroms, one touch and level restarts. There are four levels and a helth system.

First trick was to store the levels data efficiently. Keeping a list of bloks with postion and size is not very effcient as you need word-szized (2 bytes) for those. Instead I dividd the screen into 32px wide blocks. In memory I keep two 1 byte values: block position and block width.
Another cool trick I added here is that the last blocks of last level are made of "P1X" signature I'm always keep in the end of the file.

Collistion detection is always tricky - there are many strategies to choose from. For my first attempt I came up with a pixel color cheking, platforms are colored in "dangerous" color, ending mission if bottom part of the rocket ever be in the same position. Same check also looks at the "good" color, that is the ship, sucessfuly ending current level and loads next one.

Game is challanging, having each level harder with unique soution, someimes few solutions.

Fly Escape

Animated Sprites

I wanted enemies, action, more drama, easy to learn, hard to master type of a game. In Fly Escape player steers a fly that must go to the fower to pass level. There are spiders, a lot of them, more after each level. One touch and the life meter goes one point down, all to the end of the game.

I introduced animatd sprites (two frames) for a fly, flower and spiders, colorful, unique background for each level, and there are infinte number of levels, all driven by a random number generator. Still a one-button gameplay: changing fly angle it is flying in a clock-wise direction after each press. There is an entitie system and fly and enemies has eight direction movement. I even squished a life metter over the fly. A lot for a 512 bytes!

Bit of a Treasure

Isometric Grahics

So far I made flat, 2D, graphics in a classic frnt view. The logical next step was to fake the third dimenssion by incorporating isometric tiles. On a first glace this seams simple, everything is stil flat and there is no perspective involved as it uses otographic camera. This type of rendering game world is not new to me so I knew how to handle it. In essence sprites needs to be drawn from the back to front, top-right to bottom-left in isometric world, and with a right shift of position for each tile. This way the oclussion works as expected and we get the isometric illusion.

It will ony works with a correctly drawn sprtes. The main trick is to drawn each diagonal line as two pixels wide, one pixel height. In this game I was also using 1-bit sprites that adds a little bit of a challange to make them readable. To fix this problem I made two sprites for each: one with a body/shilluet and another with different color for highights/details that was drawn on to of the first. End product came out suprisingly good.

Moth Hunter
Muse

Next Game

Two-bit Color Graphics

I was thinking of a way to upgrade my sprite rendering with adding more colors. With 2 bits per pixel I can have 4 colors and still quite small data, making each line of a 8 wide sprite a word length. First color (two zeros) represents transparent, skip pixel, pairs: (0,1), (1,0), and (1,1) are shades of a given color for a sprite. It took me a moment to figure out how to draw this on the screen, while doing this I also manage to add mirroring feature. Now I only need to keep half the date for a sprite, mirroring the other part, having sixteen pixel wide sprites for free!:

With the mirroring in mind I started sketching some pixel art that takes this concept and made full advantage of it. The space ship is made of just two sprites: 8x12 and 8x8, mirrored, placed on each other. Each sprite have own palette: a start index, next colors are selected using +2, +4, +6 positions. It's a big step up over my previos games. Game Boy sprites are composed in simillar way.

Another advancment I made is in the steereng the ship dpartment.I've added velocity, increased when cursor button is pressed, giving illusion o anaog movent - shorter press moves the ship less than longer press. Now steering the ship fees more like arcade games. The code is not optimzed yet:

Entities, the core of many arcade like games. Each enemy, powerup, bullet is an entity. Some of them have complex logic like enemies, other just moves in one directoion, bullets, other hovers, power-ups. It adds a bit more code at start but gives endless posibilities, esential as game grows in complexity.

Main idea is simple: ddicated part of memory is filled with data - entity type id and its properites, position, velocity. Algorithm scans the memory and depending on sprite type and velocity modifies the positions. Collision checking is also performed here for enemy type, if it touches player bullet.

Collisions.