Developing a game for Nintendo Game Boy

Final Research Project

Final Report

Student Name (姓名): Joaquín Campos Colmenarejo (杰奎因)

Student ID (学号): LJ1706502

Introduction

The present document is the final report of my final research project, developed in Beihang University (北京航空航天大学) (Beijing, China) during the spring semester of the year 2016-17. This final report include four different reports that were sent periodically to my supervisor in Beihang and show my advancements in different parts of the project. The project consists on developing a game for Nintendo Game Boy, a really famous console from the nineties, the one I played during my early childhood and brings many good memories. Developing for Game Boy requires skills on assembler programming, as well as know the architecture of the console very deeply.

Index

Report (I)

Loading background by mapping 16x tileset

Student Name (姓名): Joaquín Campos Colmenarejo (杰奎因)
Student ID (学号): LJ1706502

Developing a game for Nintendo Game Boy

Final Research Project

Report (I): Loading background by mapping 16x tileset

Introduction

The Game Boy is a console from the late 80s and as many computers at that time, it uses an 8 bit processor, in that case it uses one called Z80. Z80 was used also by MSX, a kind of programmable computer with a successful past in Japan and USA (and actually, there is a big scene around the world and it receives many games every year).

The project in that report consists on a port from a *freeware* game developed for MSX (compatible with any version), called “Bitlogic”. I started this project with many resources, like the graphics, the design of the levels and the description of the mechanics of the game. Then, my work consists on developing the game in RGBDS (an assembler language that is closer to the original assembler from Z80 but with some extra characteristics), using all the resources I have at my disposal, that means modifying eventually the arrangement of the information using side applications, because the architecture on both platforms is quite different.

What’s in this report?

Specifically, in the present report I explain my work on the loading of the background map (BGMAP). It includes:

- Loading the tileset on the VRAM in a specific disposal.
- Developing an algorithm to map the 16x16 pixels tileset to the 8x8 pixels tileset compatible with the architecture of the platform.
- Modifying the data from the MSX version.

1. Loading the tileset

The Game Boy RAM has a portion dedicated to the graphic storage called VRAM or LCD Display RAM. This portion is located between the addresses 0x8000 and 0x9FFF and takes a total of 8192 bytes *[1]*. This portions has the data used for drawing the graphics on the LCD (2 bytes for every line of 8 pixels, 16 bytes for every tile of 8x8 pixels) from 0x8000 to 0x97FF and a table of indexes

for drawing the background from 0x9800 to 0x9FFF (2048 bytes, 2 banks of 1024 each one) *[2]*. The picture below shows the architecture of the first part mentioned before.

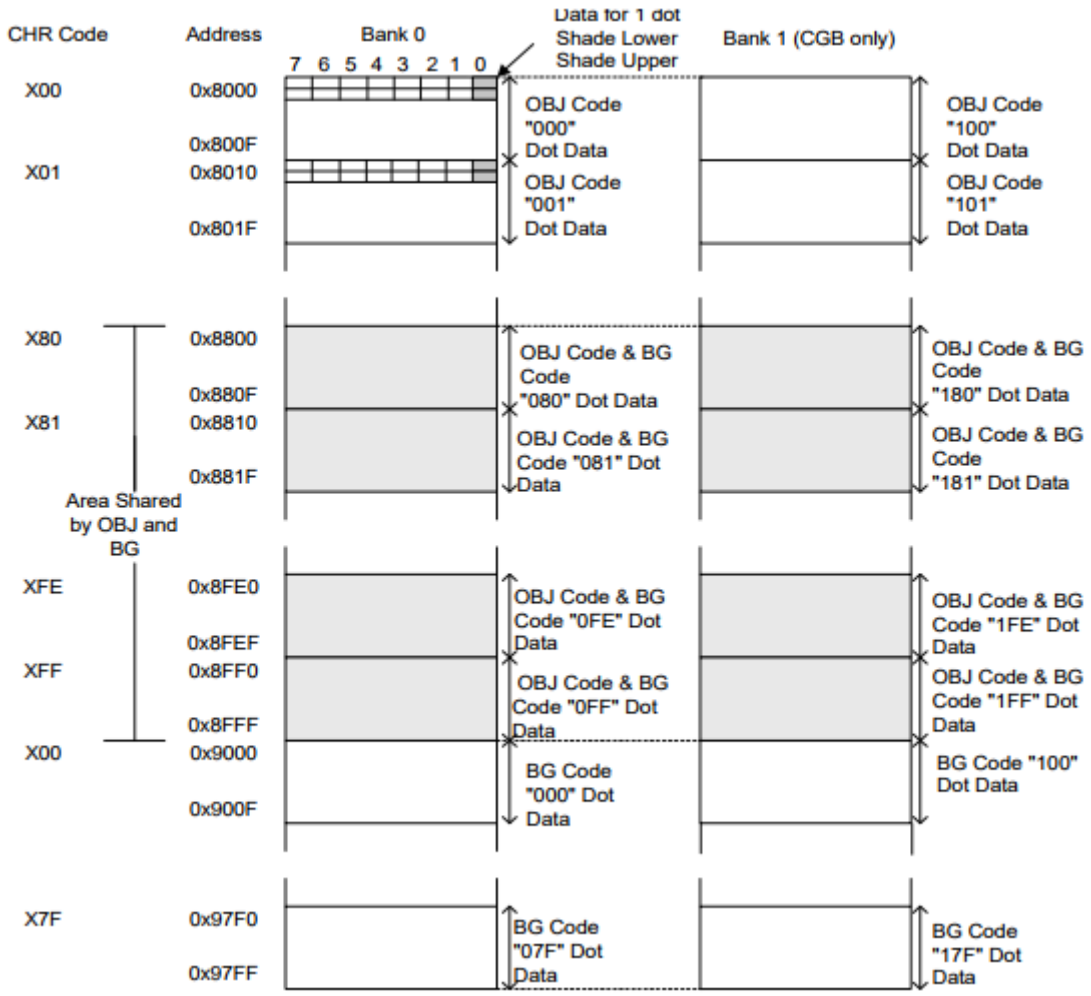

Figure 1: VRAM architecture

It is important to mention that the Game Boy recognizes two types of elements when talking about the VRAM: OBJ that is an equivalent of a sprite and BG that is the background. While the OBJ are elements with additional information and are interactive, the BG elements are just indexes to the tilesets. In the picture above we can see that there is an area that is shared by the OBJ and the BG. A flag in the LCD control register (LCDC, 0xFF40) is used to choose the area shared, that can be the area from 0x8000 to 0x87FF or the area from 0x8800 and 0x8FFF. In our case, the area chosen is the one between 0x8800 and 0x8FFF, as the picture shows. The reason of this architecture is the exploitation of an architecture that reduced to the minimum the quantity of data available. So it is not estrange to see in different games the continuous movement of data in the VRAM, depending on the visible background or enemies at a certain moment.

To insert the data into the VRAM is as simple as copying the data from the ROM to the RAM in that location: a simple loop does the work. But the data from MSX is not suitable for the Game Boy architecture.

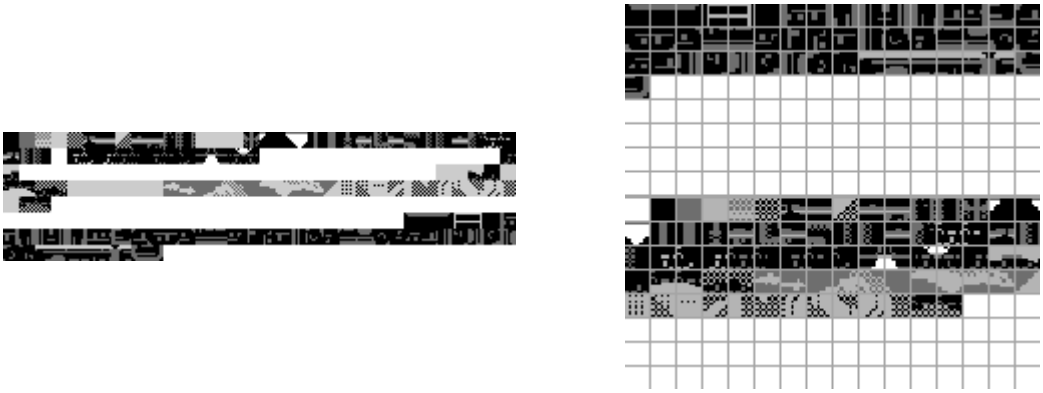

Figure 2: MSX's VRAM disposal (left picture) versus GB's VRAM disposal (right picture)

The MSX data has many redundancy due to the transformation from a full coloured palette to a 4 colours palette of Game Boy. Also, the position of the tiles does not fit in Game Boy structure. Then, for making it suitable, I change manually every position of the tiles until I have what the second picture below shows. This change have a consequence I will explain later: the tileset indexes changed, also the mapping of the background. The data is finally adapted to the Game Boy and is successfully inserted into the VRAM.

2. Setting the indexes in the BGMAP

In order to draw the background in the Game Boy's screen, indexes pointing to the data from 0x8800 and 0x97FF have to be written in 0x9800 to 0x9BFF. I don't use the second bank of the BGMAP, that is from 0x9C00 to 0x9FFF, because it is not required by the game. This indexes have values from 0 to 255.

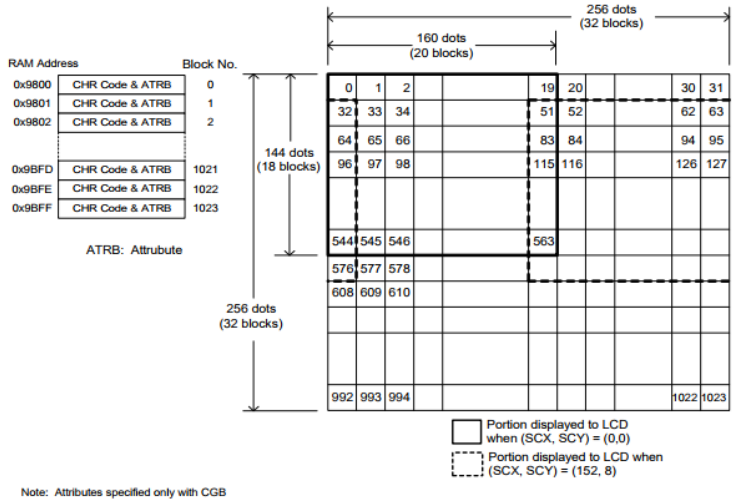

Figure 3: BGMAP indexes table

The resolution of the tileset of the game is actually 16x16 pixels, but the architecture of Game Boy works with a resolution of 8x8 pixels. So, how does it work? I use two mapping tables to get the expected result on the screen. The very first table has indexes to the 16x16 pixels tileset and has a size of 9x16; the second table has the real VRAM indexes and has a size of 20x32. Then, we can difference at least two phases in the subroutine that gets the VRAM index for every block on the LCD:

1. **Getting the 16x tileset table index.** The subroutine starts working with the addresses of the BGMAP. This addresses are from 0x9800 to 0x9BFF, but actually the game only need to paint until the address 0x99FF. With the two hexadecimal digits of the centre, the subroutine gets the row on the 16x tileset table. For example, 0x9**80**0 gives in binary a 10**0000**00, and from that binary we get the four central digits, getting the row 0 (first row); another example could be 0x9**94**0, that results in 10**0101**00, so row 5 (sixth row). To get the column, it uses the last two hexadecimal digits and applies a shift to the right. The shift is because two consecutive blocks points to the same 16x16 tile. In that case, 0x98**00** and 0x98**01**, point to the column 0 (first column), while 0x98**08** and 0x98**09**, point to the column 4 (fifth column).
2. **Getting the 8x tileset table index.** The subroutine uses the value got in the previous phase to get the row and the column: the higher bits of the value gives the information of the row, while the lower bits gives the information of the column; it uses also the address of the BGMAP to get the higher or the lower row and the left or right column. I will explain it using an example:

   16x index: 0x28
   BGMAP Address: 0x9833

   From the 16x index we get:
   - Row: 0x**2**8 -> 00100000 (32) -> 10000000 (128) -> fifth or sixth row
   - Column: 0x2**8** -> 1000 (8) -> 00010000 (16) -> seventeenth or eighteenth column.

   From the BGMAP Address we get:
   - High or low row: 0x98**3**3 -> 00**1**1 (1) -> 00100000 (32) -> low row, then sixth row.
   - Left or right column: 0x983**3** -> 001**1** (1) -> right column, so eighteenth column.

   The result is: 128 + 32 + 16 + 1 = 177 bytes from the origin address of the 8x tileset table.

But this will not give the correct index at this moment because, unfortunately, I had to change the position of the tiles to adapt them to the Game Boy architecture. The next step is modifying the 8x tileset table.

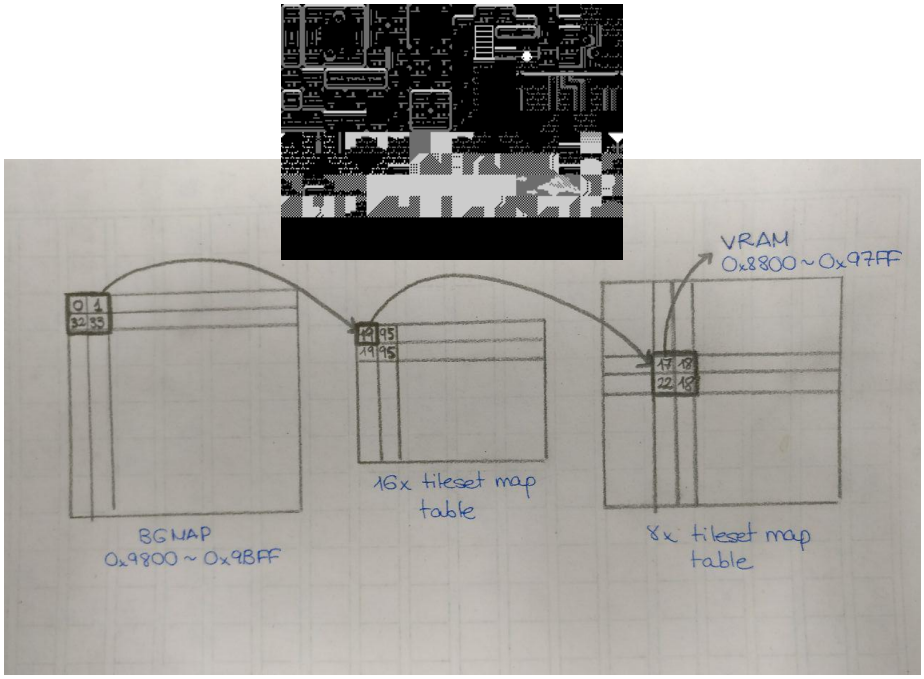

Figure 4: graphical scheme of the subroutine and the 16x tileset configuration

3. Adapt mapping to the new tileset disposal

As I moved all the tiles, the previous indexes in the 8x index table do not fit into the new disposal. To find the new indexes is as easy as getting the new position and compare it with the previous position. With that values I could manage to code a small script that changes all the values. For that purpose I chose Javascript as scripting language and *hardcoded* the differences to change the values, as the next piece of code shows:

```
for (let j in indexes) {
     let index = parseInt(indexes[j].trim());

     if (index >= 0 && index <= 1) index += 1;
     else if (index == 2) index = 4;
     else if (index == 3) index = 3;
     else if (index >= 4 && index <= 11) index += 1;
     else if (index >= 12 && index <= 14) index = 3;
     else if (index >= 15 && index <= 34) index -= 2;
     else if (index == 35) index = 0;
     else if (index >= 36 && index <= 47) index -= 3;
     else if (index >= 48 && index <= 62) index = 0;
```

```
        else if (index == 63 || index == 64) index -= 18;
        else if (index >= 65 && index <= 90) index = 0;
        else if (index == 91 || index == 92) index = 3;
        else if (index == 93 || index == 94) index -= 46;
        else if (index == 95) index = 3;
        else if (index >= 96 && index <= 99) index -= 47;
        else if (index >= 100 && index <= 105) index = 3;
        else if (index >= 106 && index <= 127) index -= 53;
        else if (index == 128) index = 3;
        else if (index == 129 || index == 130)  index -= 54;
        else if (index >= 185) index -= 57;
        else index = 0;

        indexes[j] = '$' + ('00' + index.toString(16)).slice(-2);
    }
```

After that processing, the code generates a file prepared to be inserted into the game's ROM.

References

1. Nintendo of America Inc., (1999), Character Code Mapping In Display Functions, *Game Boy Programming Manual (version 1.0)* (p. 48), (n.p.).
2. Nintendo of America Inc., (1999), BG Display In Display Functions, *Game Boy Programming Manual (version 1.0)* (p. 48), (n.p.).

Report (II)

Moving the player and scrolling the window

Student Name (姓名): Joaquín Campos Colmenarejo (杰奎因)
Student ID (学号): LJ1706502

Developing a game for Nintendo Game Boy

Final Research Project

Report (II): Moving the player and scrolling the window

Introduction

The Game Boy is a console from the late 80s and as many computers at that time, it uses an 8 bit processor, in that case it uses one called Z80. Z80 was used also by MSX, a kind of programmable computer with a successful past in Japan and USA (and actually, there is a big scene around the world and it receives many games every year).

The project in that report consists on a port from a *freeware* game developed for MSX (compatible with any version), called “Bitlogic”. I started this project with many resources, like the graphics, the design of the levels and the description of the mechanics of the game. Then, my work consists on developing the game in RGBDS (an assembler language that is closer to the original assembler from Z80 but with some extra characteristics), using all the resources I have at my disposal, that means modifying eventually the arrangement of the information using side applications, because the architecture on both platforms is quite different.

What’s in this report?

Specifically, in the present report I explain my work on the movements of the player. It includes:

- DMA transfer routine, a hardware-specific operation to transfer data from the WRAM to the OAM. Showing the player on the LCD.
- Reading the physical direction pad and buttons of the Game Boy.
- Modifying the position of the player depending on the values read from the direction pad and detecting the collisions with the background.
- Animating the player.

1. OAM Register and DMA Transfer Routine

The OBJ elements, usually called “sprites”, are the interactive elements in the game and are located between the positions 0xFE00 and 0xFE9F, a total of 160 bytes. Game Boy has a limitation on its

LCD, it is able to paint only 40 tiles simultaneous; so, the 160 bytes mentioned before respond to that 40 tiles, each one occupy 4 bytes in the OAM. [1]

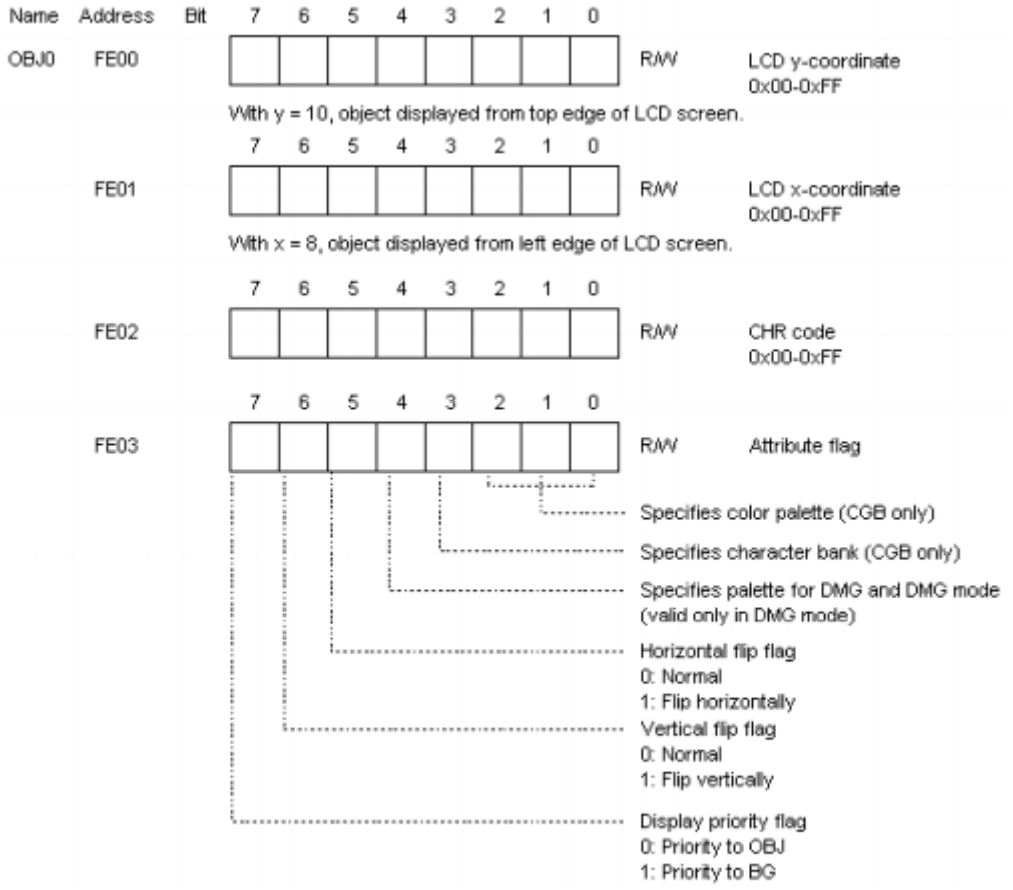

Figure 1: OAM Register for OBJ0 (4 bytes)

As can be seen in the figure above, every object has 4 bytes of information on the OAM register. The first byte of each object has the information of its Y coordinate in pixels; the second, the X coordinate; the third is the character code, the index of the tile from VRAM; the last one is a byte called “attribute flag” and is used typically to flip horizontally the sprite (that way same tiles can be used for different directions) *[2]*. Filling this bytes with the proper information makes the Game Boy print the tiles on the screen every refresh or VBlank (Vertical Blanking, irrelevant for the topic of this report, so I will not go into detail).

The Game Boy offers a special transfer performed by the hardware that makes the updating of the sprites much faster. This transfer moves 160 bytes from the specified starting address between 0x8000 and 0xDFFF (the starting address has to be multiple of 0x100) to the OAM. The implementation and use of this operation is quite strange at first sight, though. To make this

operation starts, it is as simple to set the start address in the DMA register (0xFF46) *[3]*. But actually this operation does not work if the instructions of the address setup are not located in the Internal RAM (0xFF80 to 0xFFFE) because the DMA operation can only access there during its execution, so the developer by itself has to write these instructions in the Internal RAM during the execution of the program. Below is shown the code used in the program, the "DMARoutineCopy" subroutine is called just at the beginning of the execution of the game:

```
; DMA Transfer Routine ($C000 as start address)
DMARoutine:
     db $3E, $C0, $E0, $46, $3E
     db $28, $3D, $20, $FD, $C9

;--------------------------------------------------
; DMARoutineCopy - copies the DMA Transfer Routine
; into HRAM (DMA can only access to HRAM during operation)
;--------------------------------------------------
DMARoutineCopy::
     ld c, $80
     ld b, 10
     ld hl, DMARoutine
.dma_copy_loop
     ld a, [hli]
     ld [c], a
     inc c
     dec b
     jr nz, .dma_copy_loop
     ret
```

The "DMARoutine" is the routine in machine code, extracted from the original Game Boy Programming Manual published by Nintendo (published only for granted developers), and it just set the DMA register value to 0xC000 and makes a loop of 160 cycles, making the program to wait until the DMA transfer finished. As the DMA transfer is copying the OAM data from 0xC000 (the address setted in the DMA register), I have to reserve 160 bytes starting from that address and set there the local copies of the sprites. From now on, the program modifies the local copies of the sprites and every refresh of the screen the data of the local copies is moved to the OAM registers.

2. Reading the DPAD and buttons

The Game Boy has some built-in physical controllers for the interaction of the player with the game. Four buttons as directional pad, called also DPAD, two action buttons "A" and "B" and two

extra buttons “Select” and “Start” that are used typically for choosing options in the menus in the game or starting the “pause” screen.

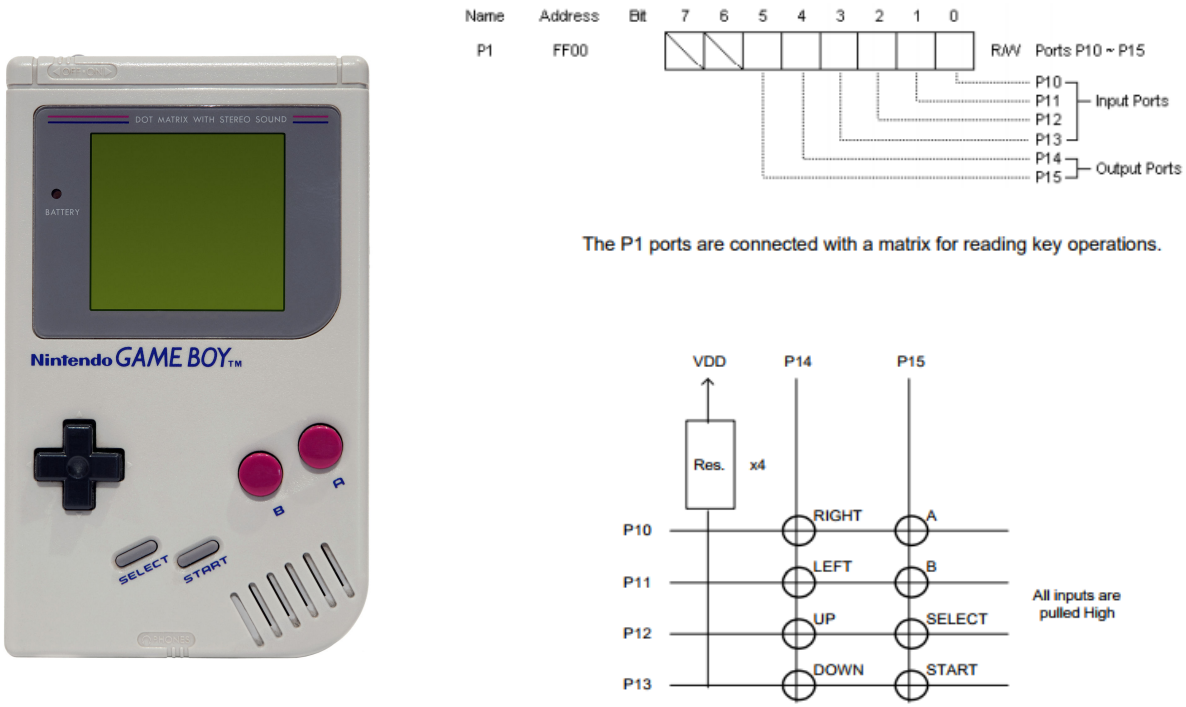

Figure 2: Game Boy on the right, showing the physical interface; on the left, P1 register and matrix disposal

The hardware reads the values from the bottons pressed and set them up in the P1 register (0xFF00) *[4]*. The developer can obtain the value from that register . The way to get that value is pretty standard; the developer just has to take care about the bouncing effect of the physical interface, as the reliability of the mechanical elements lecture is low (after the player stopped pushing one button, the console can get a positive contact during few cycles). So, basically, the developer has to read many times the value on P1 to ensure the input is correct. After reading the input, it is saved in a temporary variable setted up in the WRAM, that way it can be read by another subroutines. The piece of code that makes that reading is the one I show here:

```
ReadJoypad::
    ; get the d-pad buttons
    ld a, $20     ; bit 5 = $20
    ld de, rP1
    ld [de], a    ; select P14 by setting it low
    ld a, [de]
    ld a, [de]    ; wait a few cycles
    cpl          ; complement A
    and $0F      ; get only first 4 bits
```

```
swap a       ; swap it
ld b,a       ; store A in B

; get A / B / SELECT / START buttons
ld a, $10
ld [de], a    ; select P15 by setting it low
ld a, [de]
ld a, [de]
ld a, [de]
ld a, [de]
ld a, [de]
ld a, [de]    ; Wait a few MORE cycles
cpl          ; complement (invert)
and $0F      ; get first 4 bits
or b         ; put A and B together
ld b,a       ; store A in B

; calculate the buttons that went down since last joypad read
ld a, [joypad_held] ; read old joy data from ram
xor b        ; toggle w/current button bit
and b        ; get current button bit back
ld [joypad_down], a ; store just-went-down button bits

ld a, b       ; put original value in A
ld [joypad_held], a ; store the held down button bits


ld a, $30     ; deselect P14 and P15
ld [de], a    ; RESET Joypad


ret          ; done
```

3. Moving the player

Moving the player consists on changing the values of the two first bytes (the coordinates) in every OBJ used to build the player sprite. Actually, the sprite of the player has a resolution of 16x32 pixels (width x height), so as the tiles are 8x8 pixels, it needs 8 OBJ, 32 bytes. In that change, few variables are involved:

- Collision with the walls (blocking tiles)
- Gravity
- Value read from the “ReadJoypad” subroutine
- Presence of ladders or steps

I am going to explain superficially every variable and how they affect to the movement of the player, I think going deeply on it is not interesting in a research point of view. Everything explained below is inside a big subroutine called “MovePlayer”, so the reader has to consider that one movement makes the other movements not check (if the user pressed left, checking right has no sense).

As the Game Boy has a very poor computation performance, it is non-viable to define a complex collision system involving collision boxes or collision with non-rectangular shapes. For our game, I defined a very simple way to check if there is a collision or not: finding the tile index from the BGMAP’s block involved in the collision. If the tile is greater than 0x80 (it is actually a constant I defined in the code, called “BLOCKING_TILES_START_NUMBER”), then the player collided with a wall or with the ground. The tiles of the stairs are also blocking when the player is over them, but not when the player crosses them, so I defined another two constants “LADDER_TILE_NUMBER_00” and “LADDER_TILE_NUMBER_01” to avoid stopping the player when it crosses a stair while walking.

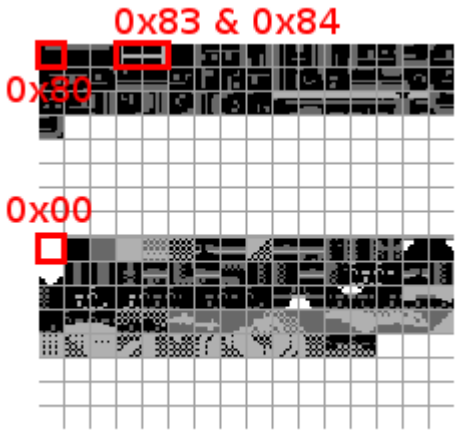

Figure 3: VRAM showing the values used to detect collision

To find the index, it is as easy as transform the incremented coordinates into a byte value, the displacement value, increment the BGMAP start address by this value and get the index of the VRAM in that block.

The gravity is actually a very simple process that is checked every time the program access to the subroutine, except in the case the player is using a ladder, but I will talk about it later. That gravity affects the player just if it has not a blocking tile under its feet, and consists on an increment of the first bytes of every object (coordinate Y). This increment value is defined by a constant that I called

“PLAYER_FALLING_SPEED”. As the increasing could make the player’s Y coordinate not to fit correctly the ground (for example, the player feet is only 3 pixels above the ground and the constant has a value of 5, the new position will be 2 pixels below the ground), before moving definitely the player, we check if the player is currently over a blocking tile and correct its position.

The up and down movements only affect the player in case of an existing ladder close to it. In case of going up, it uses a subroutine called “CheckLadder”. This subroutine checks if the player is over a ladder tile (0x83 or 0x84), taking as a reference point the center of the sprite (8, 16). If it is over a ladder, a variable “onLadder” is set, and the movement is started. When the player reached the top of the ladder and the “CheckLadder” gives a negative result (that is, the centre of the player is not over a ladder tile anymore), then a decrement of 8 pixels is made over the player and the variable “onLadder” is cleared. The reason of that increment is just for the appearance and the feeling of the player.

In case of going down, a subroutine called “CheckBelowLadder” is used. This subroutine checks if the player has a ladder below its feet, taking as a reference point the centre of the feet of the player (8, 32). If the subroutine gives a positive result, the variable “onLadder” is set and the movement is started. Also, 8 pixels are incremented. When the player reached the bottom of the ladder, the same subroutine used in the gravity process to check if the tile below the player is blocking is called, and if the result of this subroutine (called “CheckVCollision”) is positive, then the variable “onLadder” is cleared.

The movement over the ladder is different than the other movements. In this case, I increment or decrement the Y coordinate of the player 4 pixels every time, making the player’s feet to coincide with the steps on the ladder’s sprite. For that purpose I use a counter that I increment by 1 every time the “MovePlayer” subroutine is called and the up or down buttons are pressed. Every time the counter reaches 4, then the counter is reset and the player is move up or down according to the button pressed.

Also, the player sprite position is corrected horizontally in case it is not centred over the ladder. To do that, the subroutines “CheckLadder” and “CheckBelowLadder” gives information about the tile that they found (0 in case of 0x83, 1 in case of 0x84). In case of receiving a 0, the X coordinate of the OBJ has to be incremented; in case of a 1, has to be decremented. The value to be incremented

or decremented by, is calculated by the difference between the central point of the ladder horizontally (in fact, the value in X of the second tile) and the centre of the player.

In case of going left or right, a subroutine called “CheckHCollision” is called, and it checks if the tile next to the player is blocking or not. The reference point is the most left or most right pixel of the player’s sprite at a centre height, (1, 16) or (16, 16), respectively. If the result of the subroutine is positive, the player is stopped; if the result is negative, the player’s X coordinate is incremented or decremented by a constant value called “PLAYER_SPEED”.

But that subroutine only stops the player if the blocking tile has a height of 16 pixels, what happen if the tile is a “step”, that is, 8 pixels height? Then another subroutine called “CheckStep” is called and when the central horizontal point of the user reached the most left or most right pixel of the blocking tile, then the Y coordinate of the player is decreased by 8 pixels.

4. Scrolling

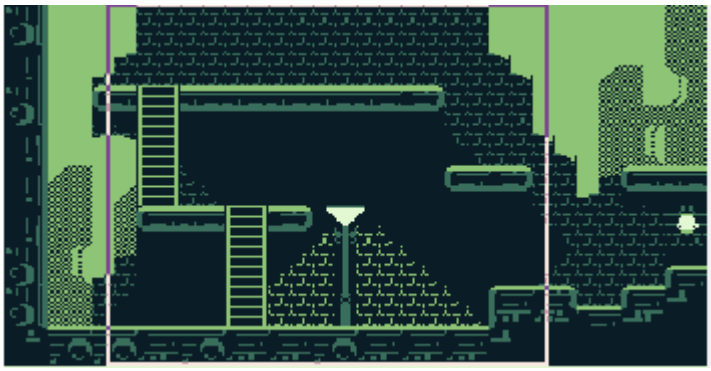

Figure 4: Scrolling view over the BGMAP

As the resolution of the LCD of the Game Boy is smaller than the resolution from the MSX game (could be any screen, that game was implemented to fit in a 16:9 resolution), I am forced to use the scrolling capability of the Game Boy. Actually, the screen of the Game Boy has a resolution of 160x144 pixels but the BGMAP size is 256x256 pixels; the scroll system let the developer move a window of visible objects around the BGMAP, so the developer has not to change the BGMAP every time the player is moving. The scroll in this game is only horizontal because the screen height fits in the Game Boy LCD height.

Therefore, the scroll affects to any right or left movement, even to the ladder horizontal correction (when the player is not centred). In this case, I used a system based on a window centred on the screen with half width of the LCD of the Game Boy. The player can move freely while it is inside

the window, but when it reaches the horizontal limits of the window, then the scrolling over the BGMAP is done.

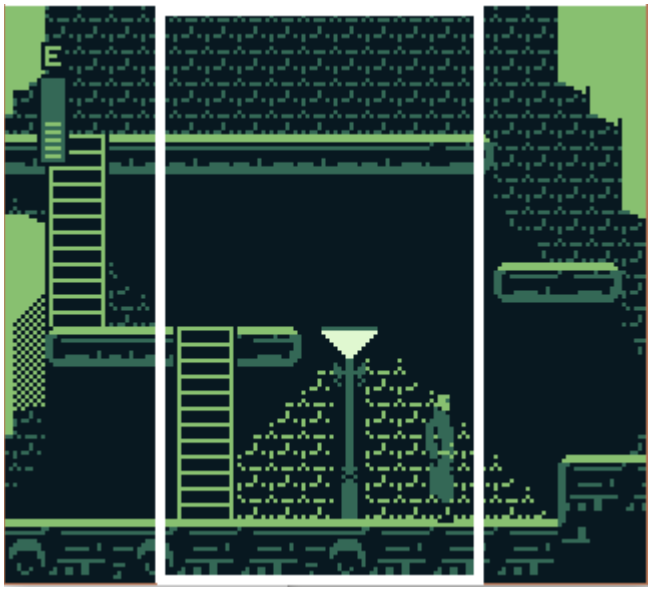

Figure 5: Scrolling window bounds

To move the scrolling window over the BGMAP the SCX register (0xFF43) *[5]* value has to be modified, indicating the value in X of the left edge of the window. In the case of this game, this value is limited to the range [0, 96].

When moving left or right, the “MovePlayer” subroutine has to see if the player reached the bounds of the window and if so move only the window, because the position of all of the sprites is dependant of the scrolling window (that is, the value 0 in X of the object is in the same position as the SCX value). If the SCX value is equal to 0, then the left bound disappear, letting the user to move freely on the left side of the screen; if the SCX value is equal to 96, then the right bound disappear and the user can move freely on the right side of the screen.

When correcting the horizontal position of the player in the ladders, the scrolling has to be done in case the correction reaches the bounds of the box. In that case, both scrolling window and player’s positions can be modified, or only the scrolling window’s position or only the player’s position.

5. Animating the player

Animating the player involves changing the third byte of every OBJ that builds the player’s sprite. In case that the player is moving left, it also involves activating a flag in the fourth byte to flip horizontally the tiles, and also swapping the tiles between the OBJs that share the same height (if not, the tiles are flipped and the sprite is totally split).

To make the animation less frenetic, the program uses a temp variable called “gameTick” that is incremented by 1 every Vblank. Then, the frame is changed every 8 blanks, that is every 8 refreshes. As an exception, in the case of the ladder, the frame is changed every time the player is moved.

References

1. Nintendo of America Inc., (1999), Character Composition In Display Functions, *Game Boy Programming Manual (version 1.0)* (p. 46), (n.p.).
2. Nintendo of America Inc., (1999), OAM Register In Display Functions, *Game Boy Programming Manual (version 1.0)* (p. 59), (n.p.).
3. Nintendo of America Inc., (1999), DMA Transfers in DMG In Display Functions, *Game Boy Programming Manual (version 1.0)* (pp. 60-61), (n.p.).
4. Nintendo of America Inc., (1999), Controller Data In System, *Game Boy Programming Manual (version 1.0)* (pp. 23-24), (n.p.).
5. Nintendo of America Inc., (1999), LCD Display Registers In Display Function, *Game Boy Programming Manual (version 1.0)* (p. 55), (n.p.).

Report (III)

Collision boxes (damage), shooting and life bar

Student Name (姓名): Joaquín Campos Colmenarejo (杰奎因)
Student ID (学号): LJ1706502

Developing a game for Nintendo Game Boy

Final Research Project

Report (III): Collision boxes (damage), shooting and life bar

Introduction

The Game Boy is a console from the late 80s and as many computers at that time, it uses an 8 bit processor, in that case it uses one called Z80. Z80 was used also by MSX, a kind of programmable computer with a successful past in Japan and USA (and actually, there is a big scene around the world and it receives many games every year).

The project in that report consists on a port from a *freeware* game developed for MSX (compatible with any version), called “Bitlogic”. I started this project with many resources, like the graphics, the design of the levels and the description of the mechanics of the game. Then, my work consists on developing the game in RGBDS (an assembler language that is closer to the original assembler from Z80 but with some extra characteristics), using all the resources I have at my disposal, that means modifying eventually the arrangement of the information using side applications, because the architecture on both platforms is quite different.

What’s in this report?

Specifically, in the present report I explain my work on the collision boxes system, that is damage over the player and the enemies, the shooting action and the life bar. It includes:

- Adapting the GUI of the MSX game to the Game Boy’s screen. Drawing new tiles for the life bar and set the animation.
- Shooting action. Load a new frame for the animation, create and move the bullets.
- Collision boxes system to detect the damage received by the player and the enemies.

1. Adapting the GUI

As shown in previous reports, the resolution of the MSX game has 16:9 format. Actually, the height of it fits perfectly with the Game Boy screen (144 pixels, 9 tiles of 16x16 pixels each one), but then we lose all the space for the GUI of the game. After evaluating which in-screen information is

valuable for the user, I decided to keep only the life bar, because the score, the number of lives and the number of “chips” collected are extra information that can be shown at the pause screen.

Figure 1: Screen of the original game where the GUI is visible

I decided to draw the new life bar based on the original Megaman games, where the life bar appears in a vertical position as a floating element. It has some implications over the game experience, because something can be hide unintentionally behind the life bar, but this is something that skips the purpose of this project. What it is important to evaluate is the implication that the life bar has technically: as the life bar is an interactive object (it changes if the user is damaged), it needs to be located in the OAM as a sprite. Special mention here, the OAM is little, letting the developer use only 40 sprites in the game at the same moment.

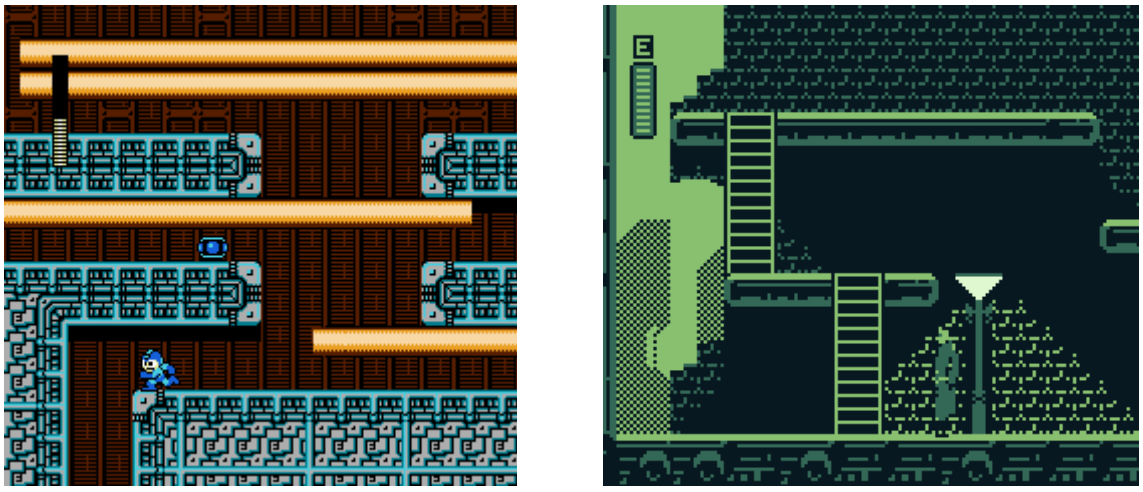

Figure 2: life bar design of Megaman series (left picture) and life bar design in the project (right picture)

2. Shooting action

Shooting involves creating a bullet from a specific position, depending on the current position of the character, moving the bullets every frame and checking the collision of the bullets with the enemies or the edge of the screen. Actually, the player only can shoot horizontally, left or right; also, if there are 2 bullets in the screen, the game don't let the user to shoot more. The last design decision was made because of the little OAM in MSX and Game Boy, so the bullets are defined as sprites also.

For shooting I created two subroutines: "ShootBullet" and "MoveBullets". The first one is called during the execution of the "MovePlayer" subroutine, as it is called when the player presses the buttons A or B. The second one is called every game loop. For both subroutines I needed to create some variables:

- "bullets", that is a 1 byte variable that have the information of both bullets (only 2 bullets simultaneously). The high bytes correspond to the second bullet in OAM, while the low bytes correspond to the first bullet in OAM. The information I save here is the status of the bullet (alive or not) and the direction (right or left). So, for example, a value of 0x32, that in binary is 00110010, mean that both bullets are alive (second bit set of each 4 bits part) and that the first bullet is moving right (first bit clear) and the second is moving left (first bit set).
- "bulletsTmr" is a simple timer that controls the time a bullet can be shoot just after another bullet. It is set with the value in the constant "BULLET_TIMER" and updated every frame in the VBlank subroutine.
- "bulletsCnt" is a simple counter, its value determine how many bullets are alive in the screen at the same time.
- A constant called "BULLET_SPEED" to increment or decrement the position of the bullet every frame.

The subroutine "ShootBullet" works as it follows. When the player pressed the button A or B, it is detected in the subroutine "ReadJoypad" and after treated in the subroutine "MovePlayer". The design of the game does not permit the player to move the character while it is shooting, so the entering to the "ShootBullet" subroutine will avoid the game checking if another button is pressed. At this moment, the execution enter to the subroutine "ShootBullet" and what it does is pretty simple. Firstly, it checks if the variable "bulletsTmr" is equal to 0, that means the player can shoot; if it is so, the timer is set with the value "BULLET_TIMER". The next checking is the variable

“bulletsCnt”; if its value is 2, the player can not shoot. Then it is time to see which bullet is available in OAM, so what it does is getting the value on the variable “bullets” and doing and bitwise AND operation over the lower bits (the first 4 bits). If this results in a 0, then it means that the first bullet in OAM is available to be written; if this results in a 1, the available is the second one. In any case, the direction is checked, the variable “bullets” updated and the information is written in the OAM as mentioned in other reports. The variable “bulletsCnt” is increased by one.

“MoveBullets” will read the variable “bullets” and check the status of both bullets. On that point, the subroutine will calculate the new position of the bullet, that could be decreased or increased depending on the orientation of the bullet (also information got from “bullets”) by the value set in the constant “BULLET_SPEED”. After the assignation of the new position to the bullet, the subroutine will call to another subroutine “CheckBulletEnemyCollision”, that I will treat below. Of course, if the bullet collided an enemy, it will be destroyed; if not, the subroutine will check if the position of the bullet is at 8 pixels from the edges of the screen (also, taking on count the orientation). If it is, the bullet will be destroyed, that it means removing the information from the OAM, clearing the bits in the “bullets” variable and also decreasing the “bulletsCnt” variable.

Finally, the player animation is changed. As this sprite has an offset of few pixels, when shooting the subroutine has to change the position on the OAM by that few pixels. To control that the offset is not applied every time the user presses A or B buttons, a variable “isShooting” is set. Then, if the variable is set, it means that the user is holding the shooting buttons and the offset has not to be applied; if the variable is clear, the offset is applied and the variable is set.

3. Collision boxes and damage

I start explaining the collision boxes way to check collision between two objects. As the console is not able to do some complicated calculations like pixel overwriting detection or trigonometric operations to calculate collision between different shapes, I decided to implement a simple and generic collision detection: collision boxes *[1]*. This way consists on checking the collision between two bound boxes assigned to each object. Basically, the minimum and maximum points of each box in X and Y are compared and if there is at least one point in common, then the two objects collided. The below figure shows this more clearly.

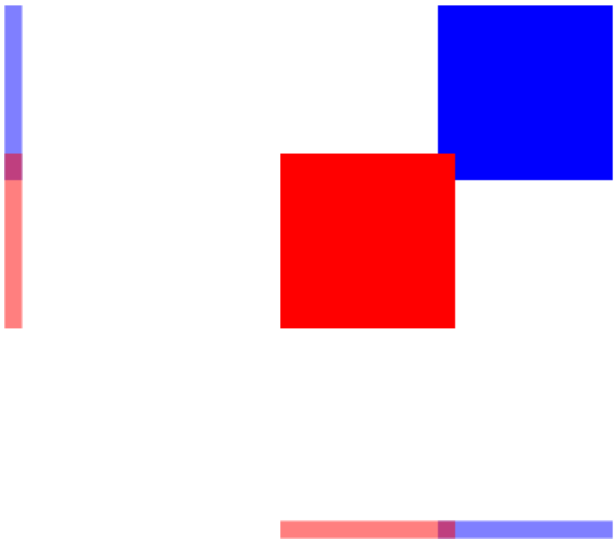

Figure 3: Collision detection

In the case of the present project, the bounding box over the player does not fit the sprite completely, it is a bounding box of 8 pixels width and 16 pixels height centred in the player. The bounding boxes for the enemies fits on their shapes completely and, in fact, the designs of them were made to be as bigger as possible to make the user feel that the game is not cheating. As can be seen in the below code, copied and pasted from the original code of the game, when getting the values “xmin” and “xmax”, adds of 3 are made to the values (in case of xmax, what the code gets is the center of the object).

```
;--------------------------------------------------
; CheckEnemyCollision - checks if the player is colliding with any enemy
;--------------------------------------------------
CheckEnemyCollision::
      ld a, [player_attrib + 24] ; get ymin player's box
      add $08
      ld b, a
      ld a, [enemies_attrib]  ; get ymax enemy's box
      cp b
      jp nc, .no_collision

      add $28     ; get ymin enemy's box
      ld b, a
      ld a, [player_attrib] ; get ymax player's box
      cp b
      jp nc, .no_collision

      ld a, [player_attrib + 5]
      add $03     ; get xmax player's box
```

```
        ld b, a
        ld a, [enemies_attrib + 1] ; get xmin enemy's box
        cp b
        jp nc, .no_collision

        ld a, [enemies_attrib + 5] ; get xmax enemy's box
        add $08
        ld b, a
        ld a, [player_attrib + 1]
        add $03 ; get xmin player's box
        cp b
        jp nc, .no_collision

        ld a, 1
        ld [getDamage], a

        call GetDamage
        jp .finish_checking_enemy_collision

.no_collision
        ld a, 0
        ld [getDamage], a

.finish_checking_enemy_collision
        ret
```

When a collision is detected, the subroutine “GetDamage” is called. This subroutine decreases the life of the user by the value from the constant “DAMAGE_VALUE” and set a timer “damageTmr” with the value from the constant “DAMAGE_TIMER”. The player will receive damage after the timer reached the 0 value, in which case it returns to be set. Also, the sprite of the player is turned off and turned on every few ticks, noticing the user that it is being damaged. Every time the life is multiple of 16 (the life bar has 10 lines and the life of the player is set to 160), the tiles of the life bar are changed. In the case there is only one line left in the life bar, the line starts to flickers.

In the case of the bullets, they have a bounding box that fits on the 8x8 pixels sprite completely. The collision detection is the same as in the code above, the difference is that the bullet is destroyed when it touches an enemy.

References

1. Collision Detection, (2014, September 9). Retrieved from http://lazyfoo.net/tutorials/SDL/27_collision_detection/index.php

Report (IV)

Dynamic loading of screens

Student Name (姓名): Joaquín Campos Colmenarejo (杰奎因)
Student ID (学号): LJ1706502

Developing a game for Nintendo Game Boy

Final Research Project

Report (IV): Dynamic loading of screens

Introduction

The Game Boy is a console from the late 80s and as many computers at that time, it uses an 8 bit processor, in that case it uses one called Z80. Z80 was used also by MSX, a kind of programmable computer with a successful past in Japan and USA (and actually, there is a big scene around the world and it receives many games every year).

The project in that report consists on a port from a *freeware* game developed for MSX (compatible with any version), called “Bitlogic”. I started this project with many resources, like the graphics, the design of the levels and the description of the mechanics of the game. Then, my work consists on developing the game in RGBDS (an assembler language that is closer to the original assembler from Z80 but with some extra characteristics), using all the resources I have at my disposal, that means modifying eventually the arrangement of the information using side applications, because the architecture on both platforms is quite different.

What’s in this report?

Specifically, in the present report I explain my work on the dynamic load of the screens and the movement of the player between them. It includes:

- Reading the data values and load the map based on the position of the user.
- Loading new screens when player changes its position.

1. Reading the mapzone file

The mapzone file has all the information about the world (a zone in the game) and its screens. Every mapzone has a header with the information about the size of the mapzone, the initial screen in the world and the matrix of screens that configure the world. Also, for every screen, it has the 16x tile map and the enemies in that screen (type and position). A fragment of that data is shown below:

```
MAPZONE1:
      db    8                 ; Width                 (+0)
      db    4                 ; Height          (+1)
      db    0                 ; Init Room X           (+2)
      db    1                 ; Init Room Y           (+3)
      ; List pointers to rooms by row & columns      (+4)
      ; if dw 0 this room is empty
      dw    0
      dw    0
      dw    0
      dw    0
      dw    SCREEN_4_0_MAPZONE1
      dw    SCREEN_5_0_MAPZONE1
      dw    SCREEN_6_0_MAPZONE1
      dw    SCREEN_7_0_MAPZONE1
      dw    SCREEN_0_1_MAPZONE1
      dw    SCREEN_1_1_MAPZONE1
      dw    SCREEN_2_1_MAPZONE1
      dw    SCREEN_3_1_MAPZONE1
      dw    SCREEN_4_1_MAPZONE1
      dw    SCREEN_5_1_MAPZONE1
      dw    SCREEN_6_1_MAPZONE1
      dw    SCREEN_7_1_MAPZONE1
      dw    0
      dw    0
      dw    SCREEN_2_2_MAPZONE1
      dw    SCREEN_3_2_MAPZONE1
      dw    0
      dw    0
      dw    SCREEN_6_2_MAPZONE1
      dw    SCREEN_7_2_MAPZONE1
      dw    0
      dw    0
      dw    0
      dw    0
      dw    0
      dw    0
      dw    0
      dw    SCREEN_7_3_MAPZONE1

SCREEN_4_0_MAPZONE1:
db     19, 95, 95,108, 43,107, 95, 95, 95, 95, 16,102,102,102,102,139
db     19, 95,108, 14, 14, 14,107, 95, 95, 95, 16,140,141,102,102,102
db     19, 95, 14, 14, 14, 25, 42, 42, 25, 42, 32,110, 26, 25, 42, 42
db     69, 51, 25, 52, 14, 25, 94, 95, 25, 14,128,127, 14, 25, 14, 14
db     19, 93, 25,105, 31, 25, 94, 95, 95,106, 14, 14, 14, 25, 14, 14
db     19, 95, 25, 95, 95, 25,105, 95, 26, 27, 14, 14, 26, 42, 25, 14
db     19, 95, 25, 95, 26, 27, 95, 95, 95,106, 14, 14,107, 95, 25, 14
db     19, 95, 25, 95, 95, 95, 95, 95, 95, 95, 14, 14, 94, 95, 25, 14
db     69, 25, 52, 95, 95, 95, 95, 95, 95, 95, 31, 26, 42, 42, 27, 14

      db    3     ;Cantidad de enemigos en room
      db    ENE2_FLYER  , 64, 48
      db    ENE2_FLYER  ,176, 16
      db    ENE3_SPIKE  ,192, 64
```

When the program starts, the initial room values are read and the map is loaded as seen in previous reports. This load is started in the subroutine “LoadInitRoom”, that sets the variable “currentScreen” based on the value in the variable “currentMapzone”. This subroutine just finds the third and fourth bytes on the data file and sets the “currentScreen” with the format 0xYX (fourth byte value as high bits and third byte value as low bits).

All the great job is done in “LoadScreen” subroutine. This subroutine reads the matrix shown above to find the position in ROM of the 16x tile map of the current screen and send it to the subroutine “LoadBGMap”, subroutine explained in previous reports. “LoadScreen” starts getting the value of the “currentMapzone” variable and finding the position in memory of the mapzone. Later, it gets the width and the heght of the map, so it knows the size of the matrix. With the value of the “currentScreen” variable, get the row and the column of the matrix of screens. As can be seen, the instruction “dw” is used and that instruction acts like as a pointer, writing 2 bytes with the position in ROM of the tag written. So, after taking the position in ROM of the 16x tileset, it is sent to “LoadBGMap”.

2. Moving the character between screens

To move the character between screens, firstly the program needs to detect if the user reached the limit of a screen. It is as easy as check if the sprite touched the edge of the screen in the “MovePlayer” subroutine, then the value in “currentScreen” is changed. If the user went right, the value is increased by 1; if the user went left, decreased by 1; if went up, the value is decreased by 0x10; if it went down, is increased by 0x10. In case of up and down, the detection is made based on the centre of the sprite, so half body of the player went out the screen before changing the background.

Then it is important to turn off the LCD, because if not the program get glitched and many tiles are located in wrong position, making the game unplayable. In order to do that, it sets the highest bit of the register LCDC (0xFF40) *[1]* to 0, making the screen completely blank. On that time, the “LoadScreen” is called and the background map is loaded properly.

The new position of the player has to be set. In case of going left, the new position is the most right place, 8 pixels from the right edge of the screen; in case of going right, the new position is the most left place, 8 pixels from the left edge of the screen. In this two cases, the position in Y does not

change, but the scroll is modified, being 0xA0 in case of going from right to left, and being 0x00 in case of going from left to right.

In the case of up, the player is positioned in a way its sprite fits with the sprite of the ladder, that is the centre of the player being on the bottom edge of the screen. The same change is made when going down, but the player being in the upper edge of the screen. In both cases, the scroll is never changed, so the player did not move horizontally.

After that changes are made, the LCD is turned on, setting the highest bit of the LCDC register to 1.

References

1. Nintendo of America Inc., (1999), LCD Display Registers In Display Function, *Game Boy Programming Manual (version 1.0)* (p. 54), (n.p.).

Conclusions

What I learnt from this project is how to develop software in architectures with few resources and how to write code thinking more about the performance. At the beginning, coding assembler was really tough, used to coding in high-level languages I could not even implement a product between two numbers. But time by time I got the way to understand all the code and how to enjoy programming the game, it became easy to implement and I got rid of all the useless tools and mechanics that come with the current game engines. Also, programming a game like as they were programmed decades ago, shown me how to develop some generic topics, and this can be applied in every language for every architecture.

Future improvements

The game is still incomplete. A lot of things were left because they were not in the bounds of the research; another ones were left because of their complexity; another ones just because they need another professional works. A list below shows the main points to be improved/completed:

- Load dynamically the enemies from the mapzone data files.
- Implement the different enemies movements.
- Implement the “chips” and “cartridges” mechanisms. Open doors based on the quantity of “chips” that the player has.
- Implement the start screen and the game over screen. Make the player die if they get damaged.
- Implement the pause screen. Implement the map and show the GUI.
- Set music and SFX to the game.
- Load the rest of the mapzones.
- Redraw the tileset to adapt to Game Boy 4 colours screen.
- Fix minor bugs and test all the game.