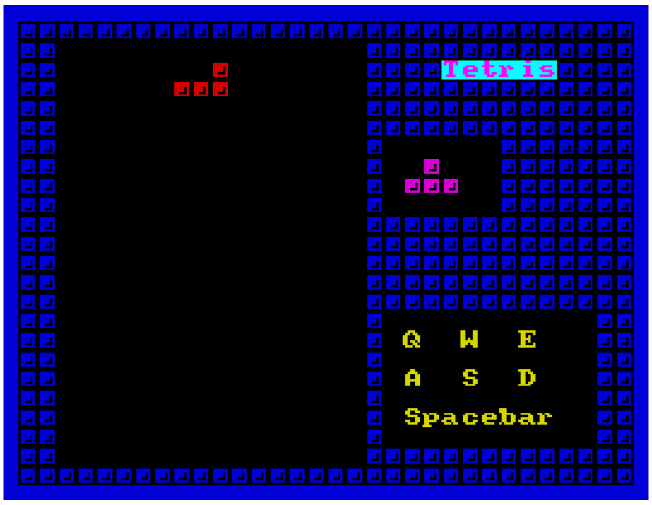


TETRIS

Universidad Francisco de Vitoria

Rodrigo Sáez Escobar
Claudia Martínez Urango
Cristina Fernández Gomáriz
3º A + BI

Content

1. Installation

The first step to play this Tetris game is to have a Z80 compiler.

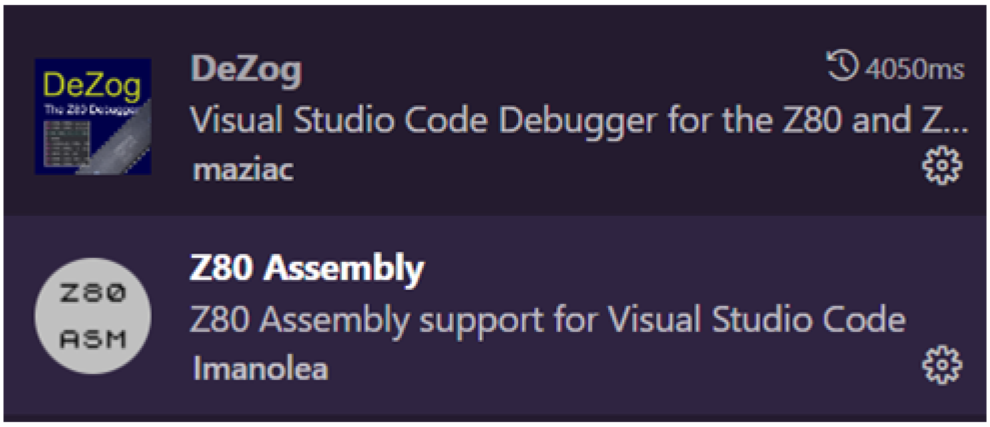
1.1. VSCode and Project

First, we need VSCODE to run the code, where you can check the last version here
https://code.visualstudio.com/download

Obviously, the project itself which is public in GitHub here
https://github.com/saezro/Tetris-Z80

1.2. Extensions

To be able to compile and run the code is necessary to have these two extensions.



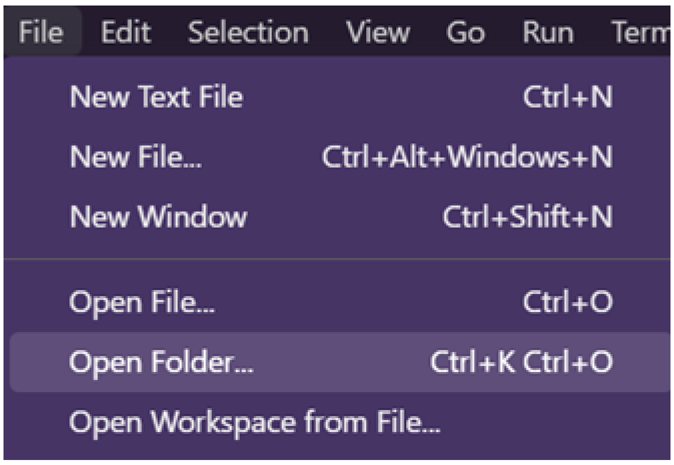
DeZog 4050ms
Visual Studio Code Debugger for the Z80 and Z...
maziac

Z80 Assembly
Z80 Assembly support for Visual Studio Code
Imanolea

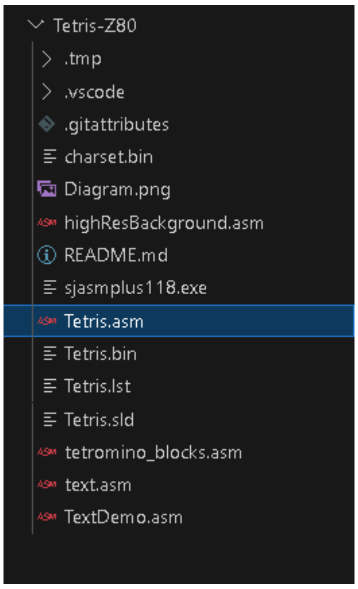
1.3. Build

The folder of the project has already been built but to save any change follow these steps.

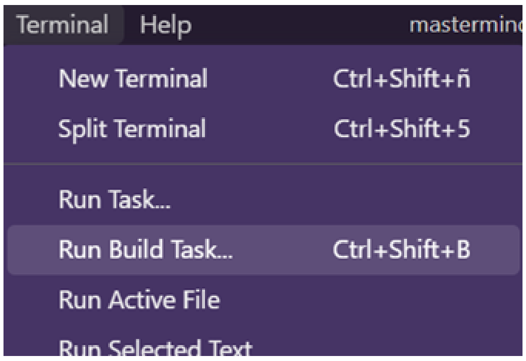
To build the project first we have to the add the folder to VSCode like this:



Then select the project folder called “Tetris” that you downloaded and that will open the project and should look like this:

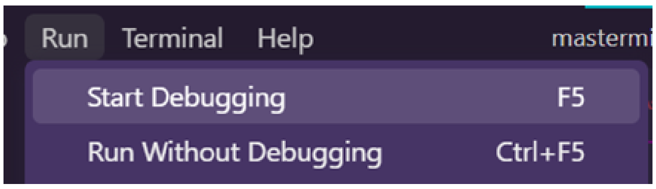


To build it we must open the archive “.asm” and select “Terminal => Run Build Task”


Terminal
Help
mastermin
New Terminal Ctrl+Shift+ñ
Split Terminal Ctrl+Shift+5
Run Task...
Run Build Task... Ctrl+Shift+B
Run Active File
Run Selected Text

1.4. Run

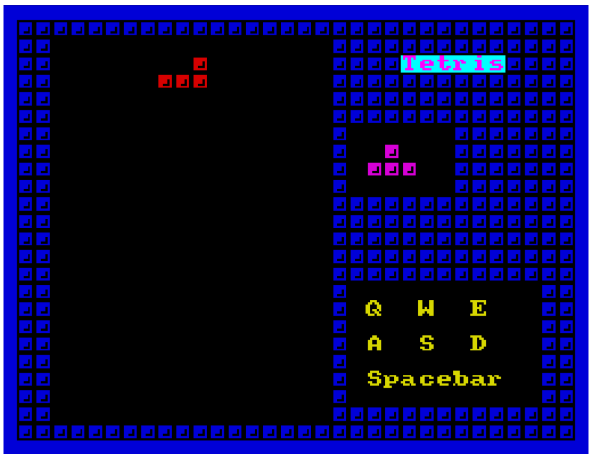
To run the game after building it select “Run => Start Debugging”



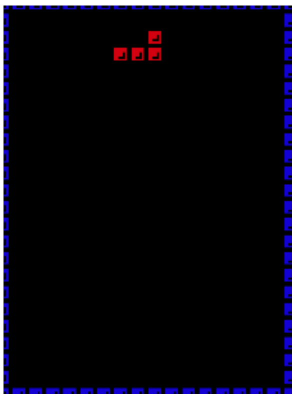
2. How to play Tetris

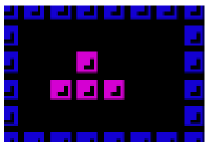
2.1. Gameboard

This is the gameboard, it has 3 important parts that need to be explained.

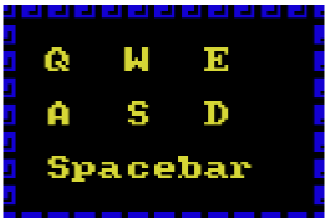


First of all, this big rectangle is where you will be able to move and place the figures.





Then this little rectangle shows the next figure to make sure you choose the best strategy.



Finally this part is just a reminder of all the possible moves you can do.

2.2. Gameplay

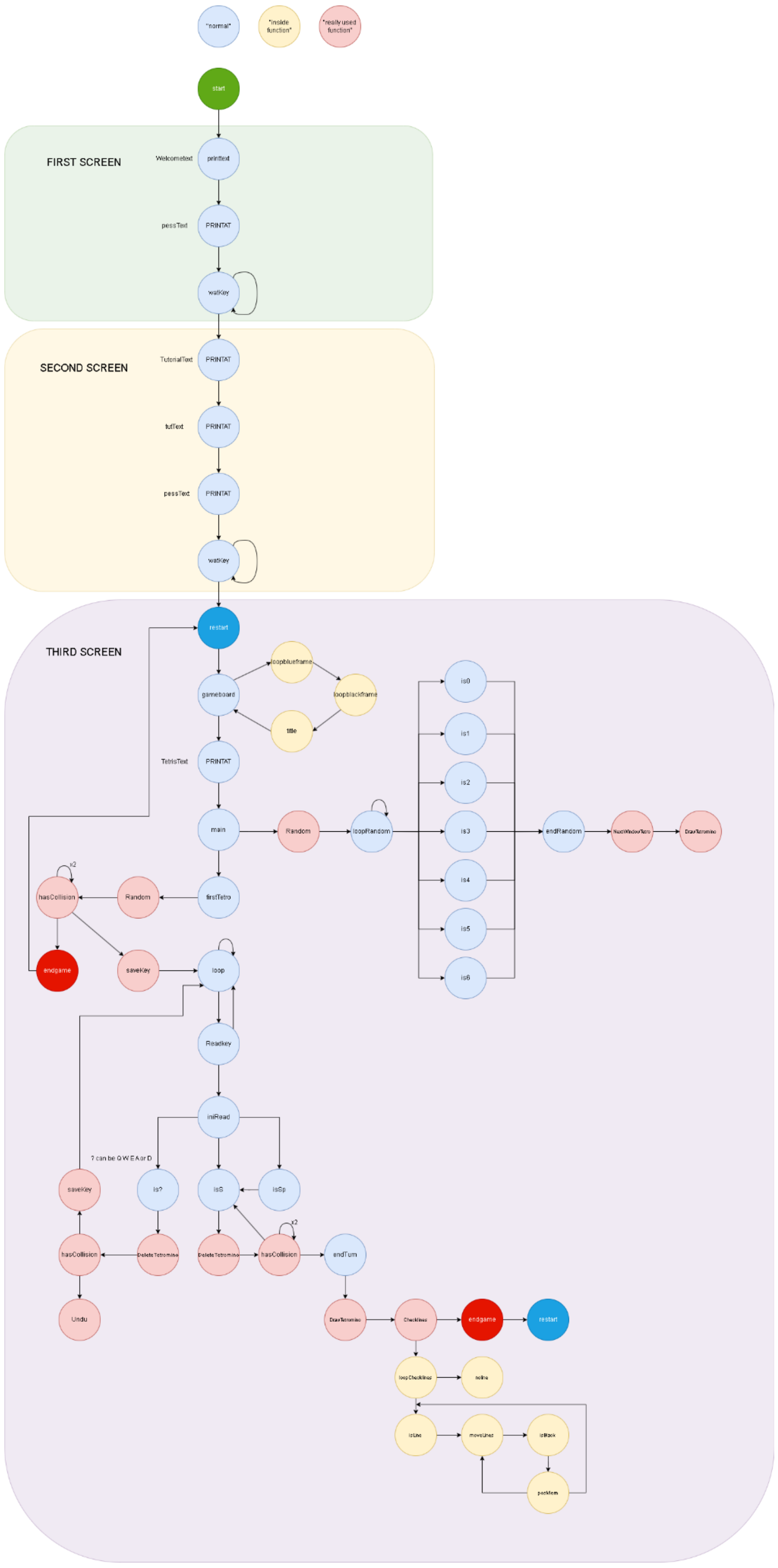
The concept of this game is to place the tetrominoes in the best way to fit the most amount of them, if you place a full line with color this line will be deleted and all the above will go down 1 line.

The controls are really easy, but after every move (except rotating and going down)you will go down 1 position ALWAYS so be careful. With “A” you move 1 position to the left, with “D” you move to the right, with “W” you go up, with “S” is down and with “Spacebar” you drop the figure to the lowest point it can in the same X.

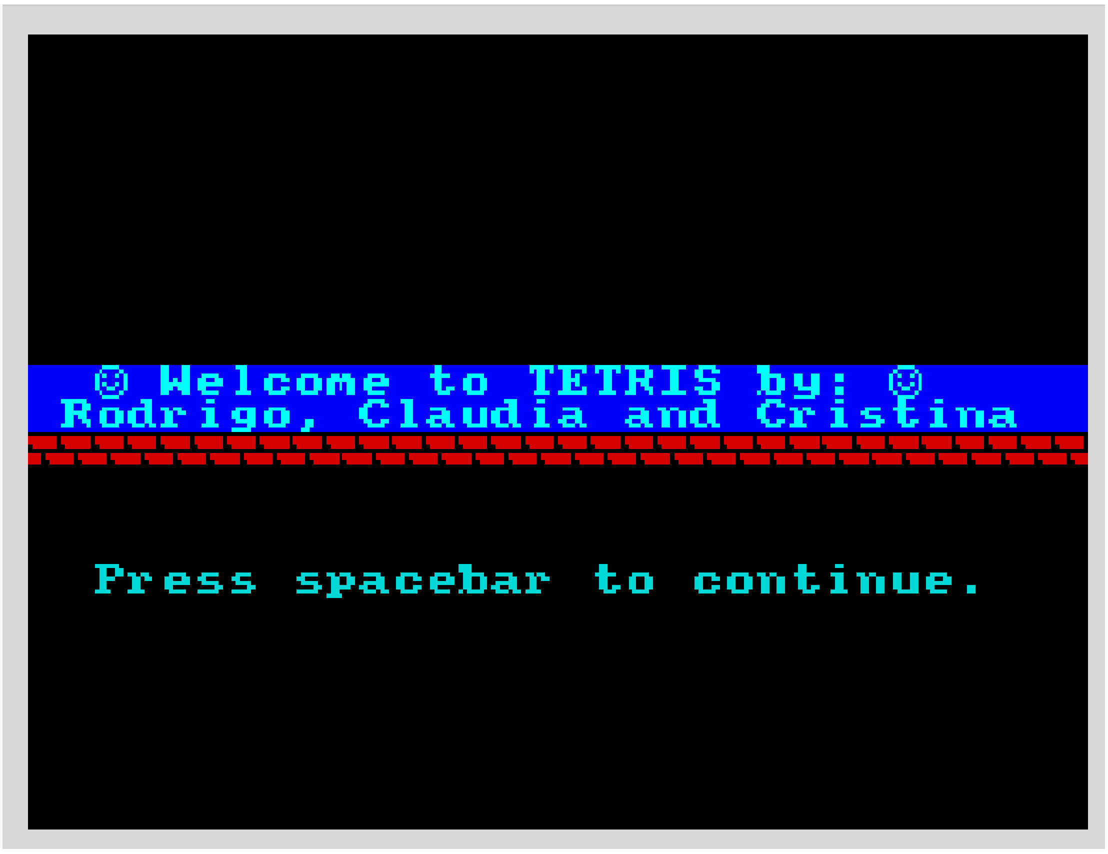
Finally the rotate controls are just “Q” and “E” which turn the figure left and right respectively without going down.

3. Program Flow

This diagram can be seen in the “Tetrisdiagram.png”



4. First screen


☺ Welcome to TETRIS by: ☺
Rodrigo, Claudia and Cristina
Press spacebar to continue.

For this, we use the "TextDemo.asm". The labels used are there. Inside "TextDemo.asm" is an include to "text.asm".

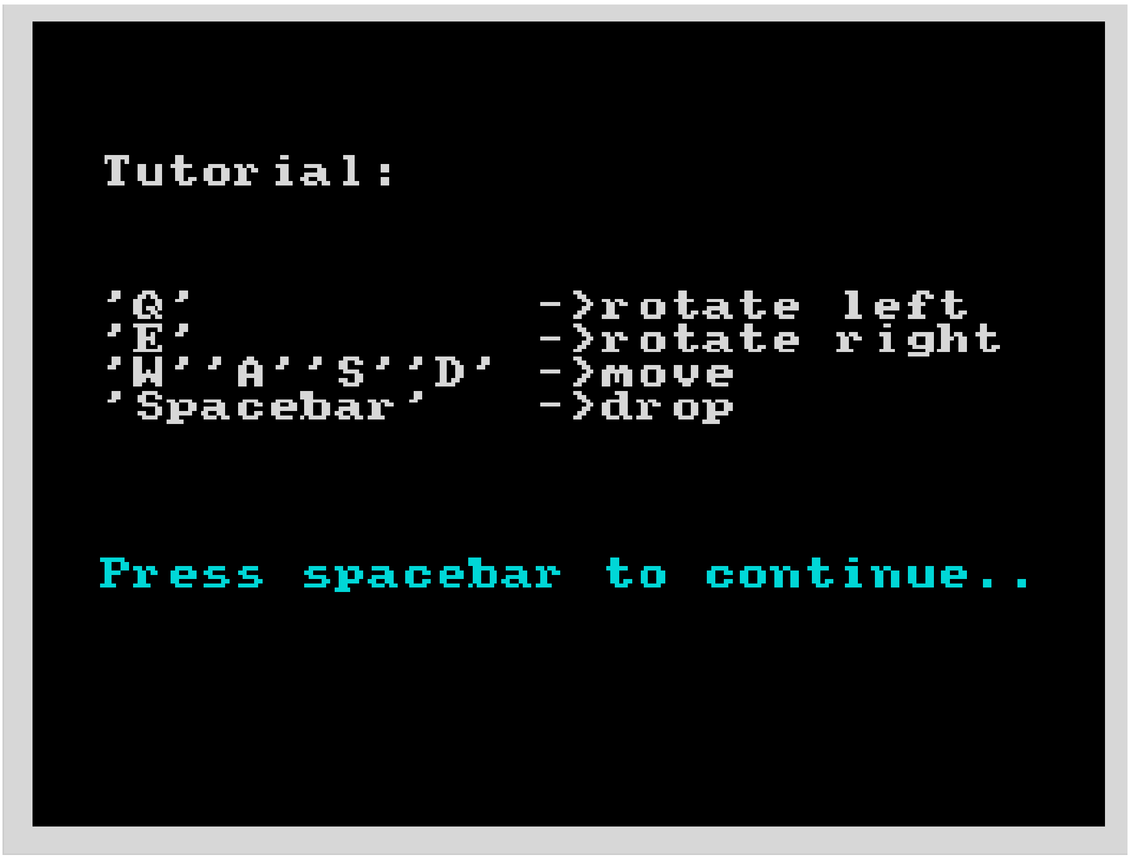
```
include 'TextDemo.asm'
```

```
include "text.asm"
```

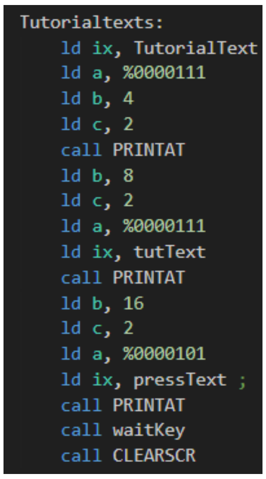
Start: the text we want to introduce is loaded on ix, and the color on a. The text loaded on ix is printed on the screen. "Print text" is the text and the brick bar. "Press Text" is the text below where the space is needed to continue in the label waitKey. Finally, the screen is cleared.

```
start:
    ld ix, Welcometext
    ld a, %01001101
    call printtext
    ld b, 16
    ld c, 2
    ld a, %0000101
    ld ix, pressText
    call PRINTAT
    call waitKey
    call CLEARSCR
```

5. Second screen



```
Tutorial:

'Q'            ->rotate left
'E'            ->rotate right
'W''A''S''D'   ->move
'Spacebar'     ->drop


Press spacebar to continue..
```



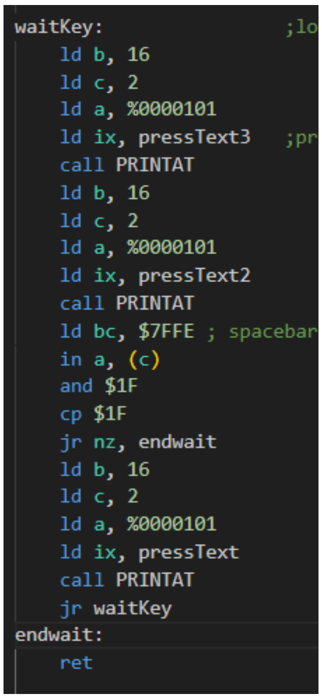
```
Tutorialtexts:
    ld ix, TutorialText
    ld a, %0000111
    ld b, 4
    ld c, 2
    call PRINTAT
    ld b, 8
    ld c, 2
    ld a, %0000111
    ld ix, tutText
    call PRINTAT
    ld b, 16
    ld c, 2
    ld a, %0000101
    ld ix, pressText ;
    call PRINTAT
    call waitKey
    call CLEARSCR
```

Tutorialtexts: this function follows the same dynamic as “Start”.

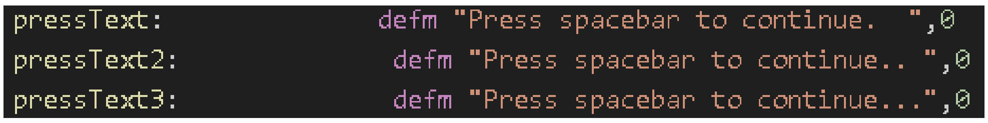
- TutorialText: the text “Tutorial:”.
- TutText: the text including the explanation of the different keys.
- PressText: The text that introduces that the spacebar is needed to continue.

WaitKey: same dynamics as “TutorialText” and “Start” but with the difference that it compares if the spacebar is pressed to continue.

To imitate the impression that the dots in “Press spacebar to continue…” are changing. We print 3 times the same print but with different numbers of dots to give the illusion.

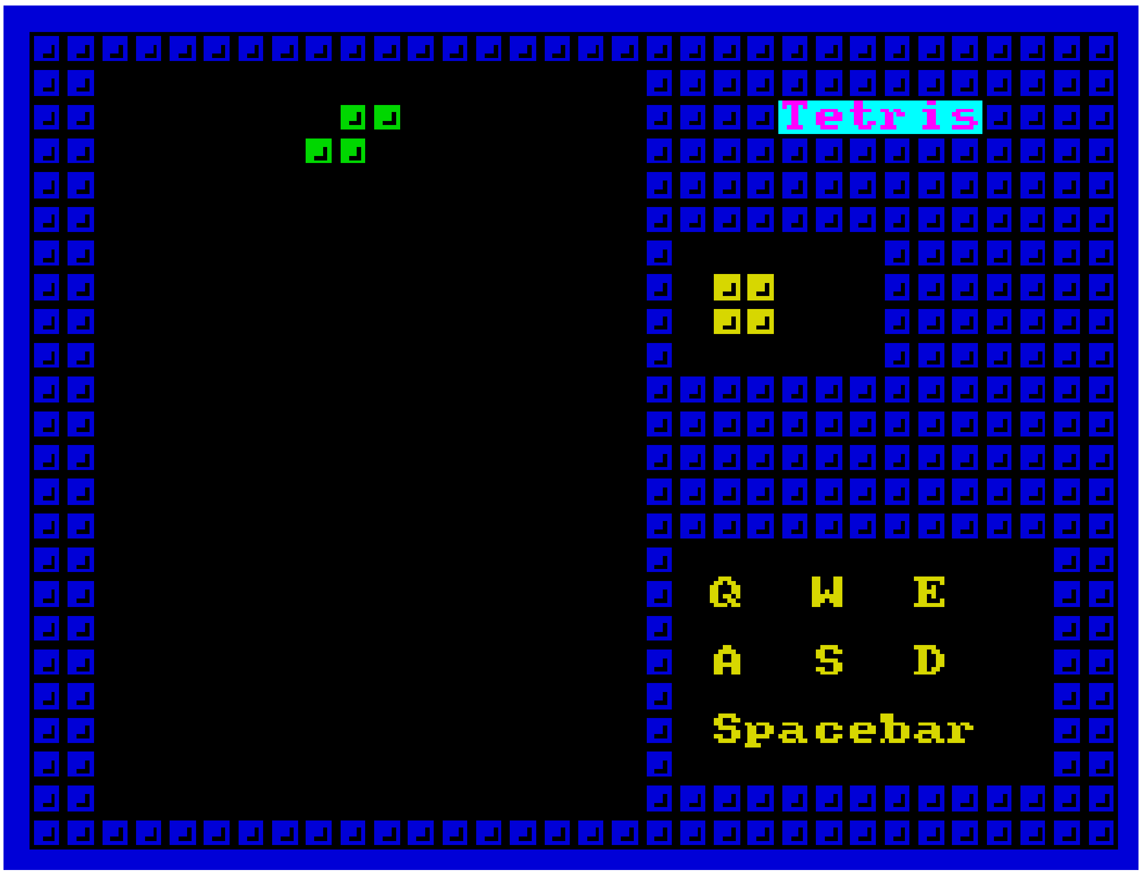


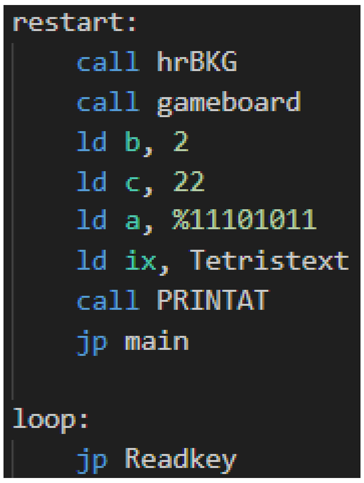
```
waitKey:                ;lo
    ld b, 16
    ld c, 2
    ld a, %0000101
    ld ix, pressText3    ;pr
    call PRINTAT
    ld b, 16
    ld c, 2
    ld a, %0000101
    ld ix, pressText2
    call PRINTAT
    ld bc, $7FFE ; spacebar
    in a, (c)
    and $1F
    cp $1F
    jr nz, endwait
    ld b, 16
    ld c, 2
    ld a, %0000101
    ld ix, pressText
    call PRINTAT
    jr waitKey
endwait:
    ret
```



```
pressText:              defm "Press spacebar to continue.  ",0
pressText2:              defm "Press spacebar to continue.. ",0
pressText3:              defm "Press spacebar to continue...",0
```

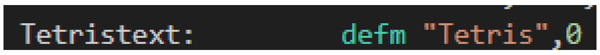
6. Third Screen





```
restart:
    call hrBKG
    call gameboard
    ld b, 2
    ld c, 22
    ld a, %11101011
    ld ix, Tetristext
    call PRINTAT
    jp main

loop:
    jp Readkey
```

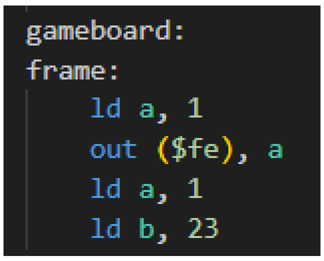
Restart: the function "hrBKG" is called. <u>This function is given by the teacher</u>, to see it in a more beautiful way. Then the print of the gameboard is done. The color is `%11101011 pink-blue` alternated between them. Then it jumps to main.



```
Tetristext:          defm "Tetris",0
```

Gameboard: it includes the functions:

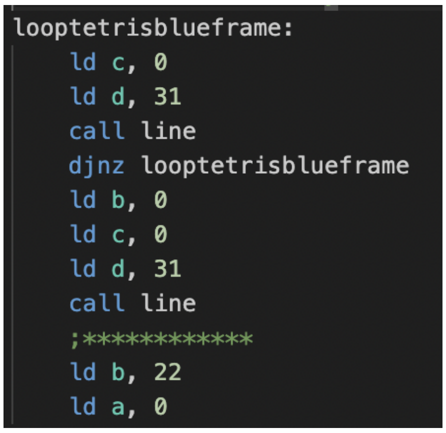
- Frame.
- Looptetrisblueframe.
- Looptetrisblackframe.
- Title.



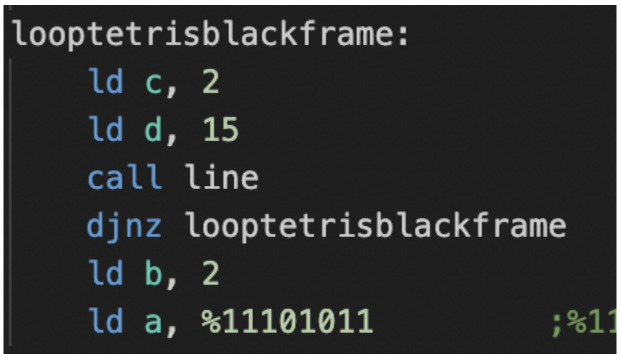
```
gameboard:
frame:
    ld a, 1
    out ($fe), a
    ld a, 1
    ld b, 23
```

Frame: we use this function to print the game frame on the screen. We load in "a" the color of the frame and the out ($fe), the instruction sends the value in register a to the output of that port.

Looptetrisblueframe: controls a loop that sets lines, and calls the **line** subroutine, which sets the X and Y coordinates and calls **DOTYXC** to draw a pixel at the calculated position. The memory address to draw is calculated in **DOTYXC** based on the coordinates and set with the provided color (in this case is blue).The finality of this function is to print all screen blue due to the given parameters (b,c,d). To print the last line we call the function "line" (because the value of b). The last two lines in function "frame" belongs to "looptetrisblueframe".

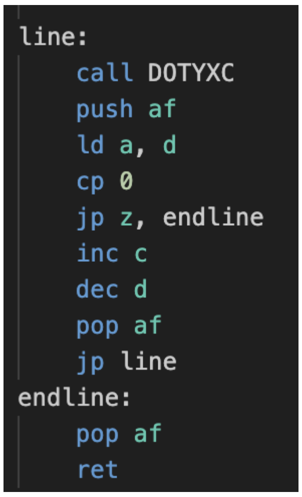


```
looptetrisblueframe:
    ld c, 0
    ld d, 31
    call line
    djnz looptetrisblueframe
    ld b, 0
    ld c, 0
    ld d, 31
    call line
    ;************
    ld b, 22
    ld a, 0
```



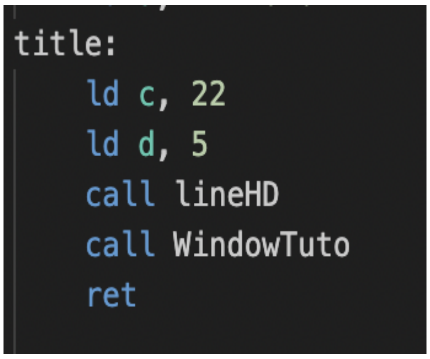
```
looptetrisblackframe:
    ld c, 2
    ld d, 15
    call line
    djnz looptetrisblackframe
    ld b, 2
    ld a, %11101011              ;%11
```

Looptetrisblackframe: the functionality is similar to **looptetrisblueframe**, but this time the position of the coordinates and the color will be different. In this case the lines will be black. With this we make the **game zone** where the game will take place. The color on a %11101011 pink-blue and b are for the function "title".



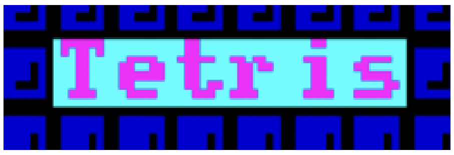
```
line:
    call DOTYXC
    push af
    ld a, d
    cp 0
    jp z, endline
    inc c
    dec d
    pop af
    jp line
endline:
    pop af
    ret
```

Line: it is responsible for drawing a horizontal line on the screen moving from left to right. It uses registers c and d to control the position and length of the line. The memory address on the screen is calculated by calling **DOTYXC**. The line ends when the length of the line reaches zero.

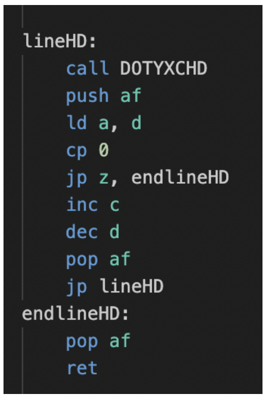


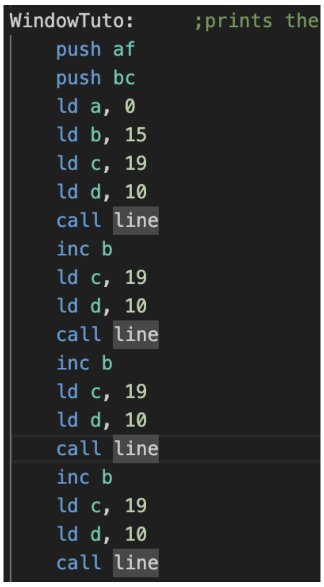
```
title:
    ld c, 22
    ld d, 5
    call lineHD
    call WindowTuto
    ret
```

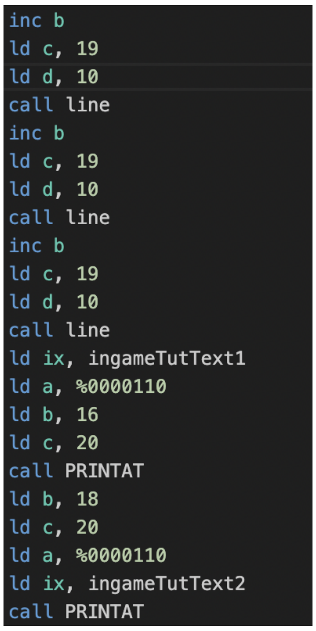
Title: is responsible for drawing a horizontal line on the screen with the call to the **lineHD** function and then displays a window with tutorial information and explanatory text related to the game keys by calling this time the **WindowTuto** function. This code is used at the start of the game to display a title and provide information to the player. Registers c and d are used to indicate the position coordinates.

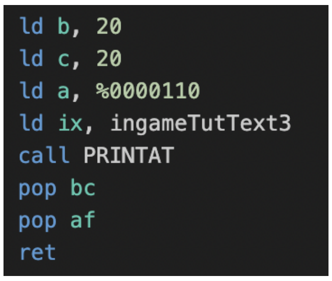


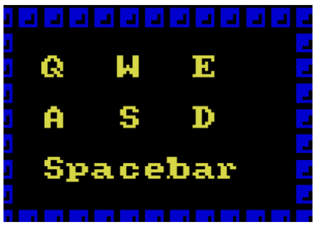
LineHD: It does the same thing than the "line" function, but the only difference is that this call DOTYXCHD that doesn't modificate the value of the color introduced as a parameter.


```
lineHD:
    call DOTYXCHD
    push af
    ld a, d
    cp 0
    jp z, endlineHD
    inc c
    dec d
    pop af
    jp lineHD
endlineHD:
    pop af
    ret
```


```
WindowTuto:     ;prints the
    push af
    push bc
    ld a, 0
    ld b, 15
    ld c, 19
    ld d, 10
    call line
    inc b
    ld c, 19
    ld d, 10
    call line
    inc b
    ld c, 19
    ld d, 10
    call line
    inc b
    ld c, 19
    ld d, 10
    call line
```

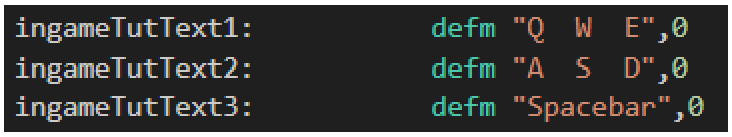

```
inc b
ld c, 19
ld d, 10
call line
inc b
ld c, 19
ld d, 10
call line
inc b
ld c, 19
ld d, 10
call line
ld ix, ingameTutText1
ld a, %0000110
ld b, 16
ld c, 20
call PRINTAT
ld b, 18
ld c, 20
ld a, %0000110
ld ix, ingameTutText2
call PRINTAT
```

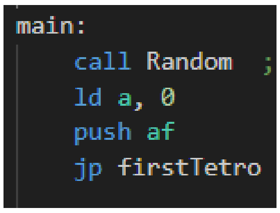

```
ld b, 20
ld c, 20
ld a, %0000110
ld ix, ingameTutText3
call PRINTAT
pop bc
pop af
ret
```



WindowTuto: draws a window, and then prints specific text within that window. With ld ix, ingameTutText1, ld ix, ingameTutText2 and ld ix, ingameTutText3 loads into the ix register the memory address of the text to be printed, in this case the letters of the keys to be used in the game and the word "Spacebar". It also calls the PRINTAT function to print the text in the specified position and color.

- Register A: color value for the lines to be drawn. If a = 0 is black. If a = %0000110 is yellow.
- Register B: specify the vertical position (Y) of the lines being drawn.
- Register C: Used to specify the horizontal position (X) of the lines being drawn.
- Register D: length of the line.

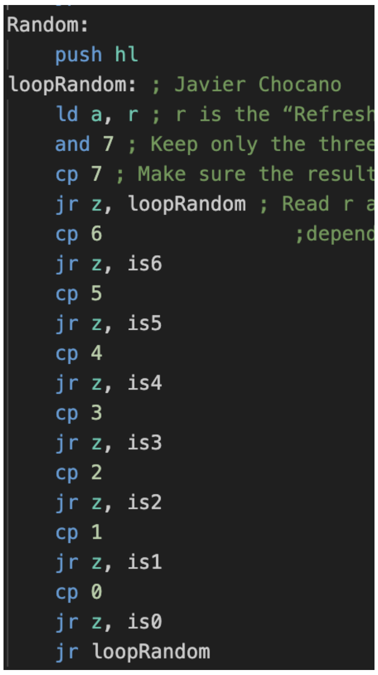

```
ingameTutText1:         defm "Q  W  E",0
ingameTutText2:         defm "A  S  D",0
ingameTutText3:         defm "Spacebar",0
```



```
main:
    call Random  ;
    ld a, 0
    push af
    jp firstTetro
```

main: the function “Random” is called so the first random tetromino is printed on screen (on the window that shows the next tetromino) and the value of “a” is saved. Then the first tetromino on the black playable background.

6.1. Random



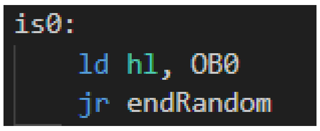
```
Random:
    push hl
loopRandom: ; Javier Chocano
    ld a, r ; r is the “Refresh
    and 7 ; Keep only the three
    cp 7 ; Make sure the result
    jr z, loopRandom ; Read r a
    cp 6                ;depend
    jr z, is6
    cp 5
    jr z, is5
    cp 4
    jr z, is4
    cp 3
    jr z, is3
    cp 2
    jr z, is2
    cp 1
    jr z, is1
    cp 0
    jr z, is0
    jr loopRandom
```

Random: It stores hl on the stack.

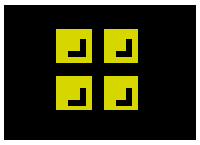
LoopRandom: The first 4 lines of this function are provided by the teacher on Canvas (AOC_Tetris_Hackathon II.pdf). Then the value given randomly is compared to represent one figure in the function (is6, is5,is4…).

We are going to explain one example of this kind of function like for example is0.

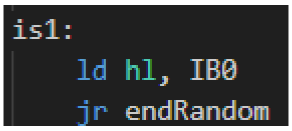
Is0: the figure is loaded in OB0. Each function keeps one figure.



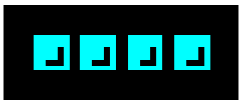
```
is0:
    ld hl, OB0
    jr endRandom
```



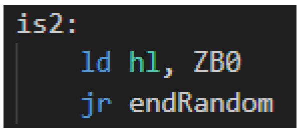
is1:



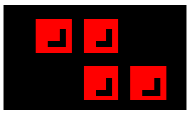
```
is1:
    ld hl, IB0
    jr endRandom
```



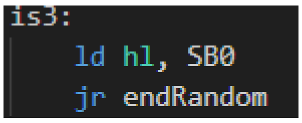
is2:



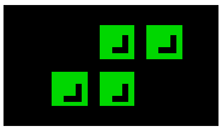
```
is2:
    ld hl, ZB0
    jr endRandom
```



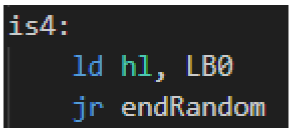
is3:



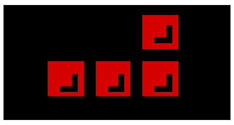
```
is3:
    ld hl, SB0
    jr endRandom
```



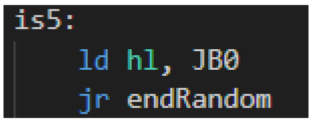
is4:



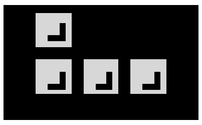
```
is4:
    ld hl, LB0
    jr endRandom
```



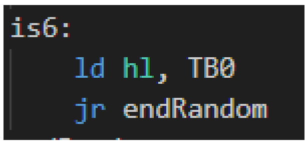
is5:



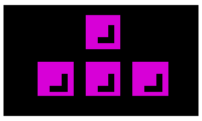
```
is5:
    ld hl, JB0
    jr endRandom
```



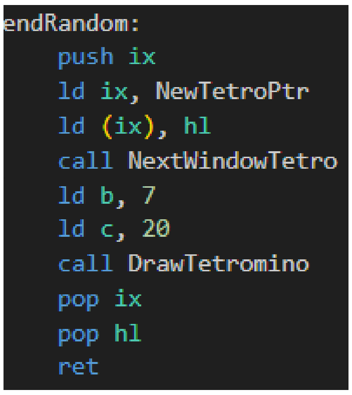
is6:



```
is6:
    ld hl, TB0
    jr endRandom
```

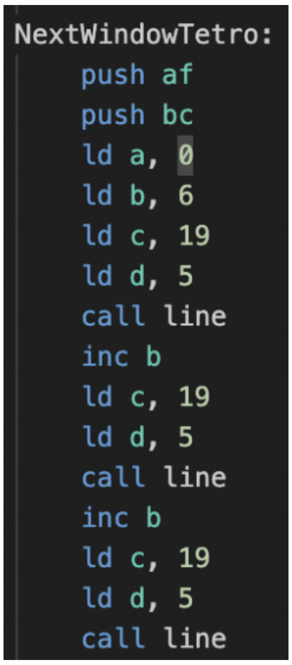


EndRandom: the value of the new tetromino is loaded in ix, the window screen is printed again into black with the function “NextWindowTetro” and the random tetromino is drawn on that window.

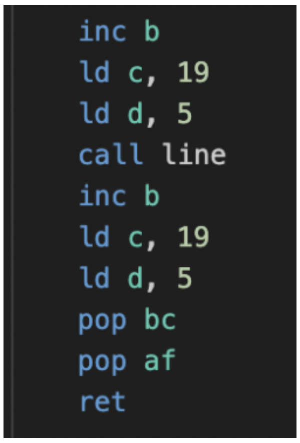


```
endRandom:
    push ix
    ld ix, NewTetroPtr
    ld (ix), hl
    call NextWindowTetro
    ld b, 7
    ld c, 20
    call DrawTetromino
    pop ix
    pop hl
    ret
```

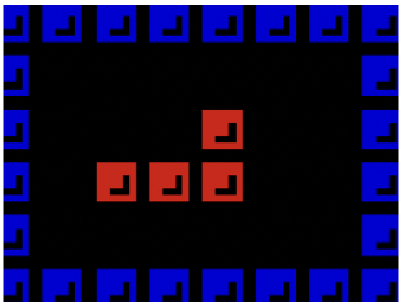
NextWindowTetro: it is responsible for printing on the screen the visual representation of the window of the next tetromino that will appear in the game. It would be the same as the picture but without the figure.



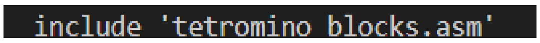
```
NextWindowTetro:
    push af
    push bc
    ld a, 0
    ld b, 6
    ld c, 19
    ld d, 5
    call line
    inc b
    ld c, 19
    ld d, 5
    call line
    inc b
    ld c, 19
    ld d, 5
    call line
```



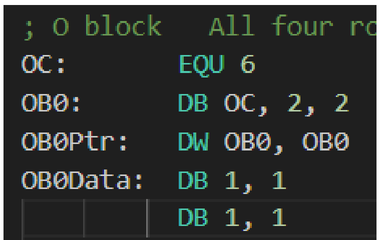
```
    inc b
    ld c, 19
    ld d, 5
    call line
    inc b
    ld c, 19
    ld d, 5
    pop bc
    pop af
    ret
```



After doing the include of the next file.asm we can work with the tetrominoes.



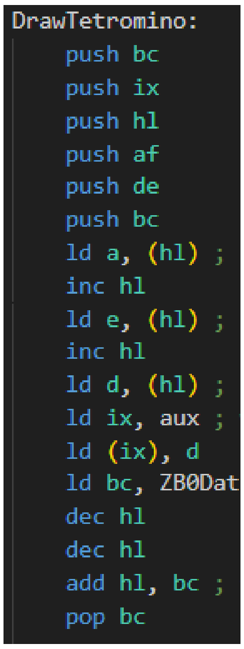
```
    include 'tetromino blocks.asm'
```



```
; O block    All four ro
OC:         EQU 6
OB0:        DB OC, 2, 2
OB0Ptr:     DW OB0, OB0
OB0Data:    DB 1, 1
            DB 1, 1
```

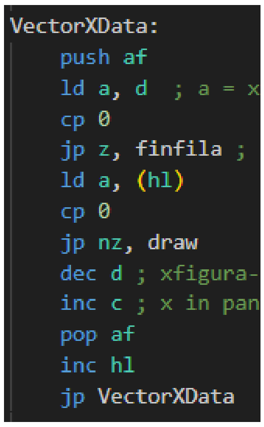
DrawTetromino: the intention is that the following variables contain:

- A = color. In this example, it would be 6.
- E = yfigure. In this example, it would be 2.
- D = xfigure. In this example, it would be 2.
- Hl = the first position of the tetromino. In this example, it would be the first position of "0bOData". The first 1 of the array.



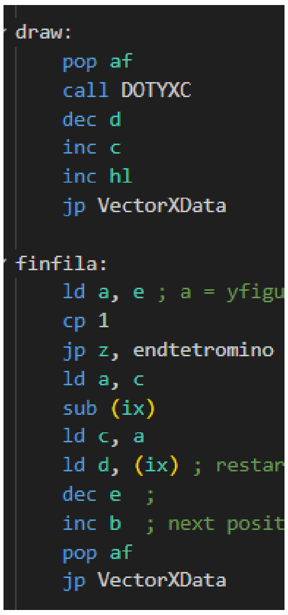
```
DrawTetromino:
    push bc
    push ix
    push hl
    push af
    push de
    push bc
    ld a, (hl) ;
    inc hl
    ld e, (hl) ;
    inc hl
    ld d, (hl) ;
    ld ix, aux ;
    ld (ix), d
    ld bc, ZB0Dat
    dec hl
    dec hl
    add hl, bc ;
    pop bc
```

VectorXData: it compares if the xfigure ends to jump to "finfila". If not, when "TetrominoData" has a 1, it jumps to "draw", decreasing xfigure and increasing x on screen.



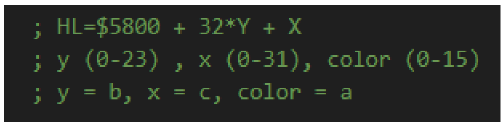
```
VectorXData:
    push af
    ld a, d   ; a = x
    cp 0
    jp z, finfila ;
    ld a, (hl)
    cp 0
    jp nz, draw
    dec d ; xfigura-
    inc c ; x in pan
    pop af
    inc hl
    jp VectorXData
```

Draw: it paints the dot in the position of hl calculated in "DOTYXC". Then it goes to the next position (hl) of the "TetrominoData" and the next position on the screen.



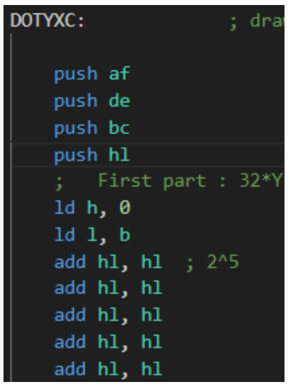
```
draw:
    pop af
    call DOTYXC
    dec d
    inc c
    inc hl
    jp VectorXData

finfila:
    ld a, e ; a = yfigu
    cp 1
    jp z, endtetromino
    ld a, c
    sub (ix)
    ld c, a
    ld d, (ix) ; restar
    dec e  ;
    inc b  ; next posit
    pop af
    jp VectorXData
```

DOTYXC. It realizes the next operation in hl, so it calculates the position:

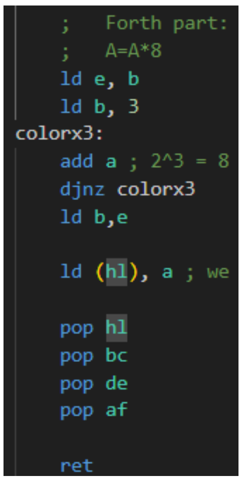


```
; HL=$5800 + 32*Y + X
; y (0-23) , x (0-31), color (0-15)
; y = b, x = c, color = a
```

In this function, there is colorx3.



```
DOTYXC:               ; dra

    push af
    push de
    push bc
    push hl
    ;  First part : 32*Y
    ld h, 0
    ld l, b
    add hl, hl  ; 2^5
    add hl, hl
    add hl, hl
    add hl, hl
    add hl, hl
```

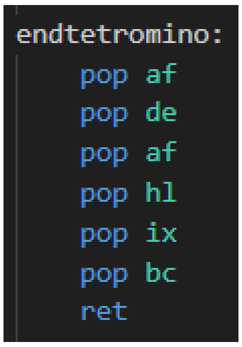


```
    ;  Forth part:
    ;  A=A*8
    ld e, b
    ld b, 3
colorx3:
    add a ; 2^3 = 8
    djnz colorx3
    ld b,e

    ld (hl), a ; we

    pop hl
    pop bc
    pop de
    pop af

    ret
```

Colorx3: Moves the color to the correct attribute positions. It realizes 2^3. The color is stored in hl.

Finfila: when the row ends, it compares if it is the last row, to see if the tetromino ends. Then, the value of xfigure is restored in d, the yfigure is decreased and the position on screen (x) is increased.

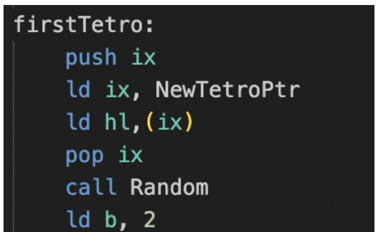
EndTetromino: the tetromino ends.



```
endtetromino:
    pop af
    pop de
    pop af
    pop hl
    pop ix
    pop bc
    ret
```

DeleteTetromino: uses some of the functions as "DrawTetromino" with different names but it draws the tetromino in black.

- VectorXData2.
- Draw2.
- DOTYXC (same function as DrawTetromino).
- Finfila2.
- EndTetromino2.



```
firstTetro:
    push ix
    ld ix, NewTetroPtr
    ld hl,(ix)
    pop ix
    call Random
    ld b, 2
```

firstTetro: this function is responsible for initializing the first figure (first tetromino) of the game.

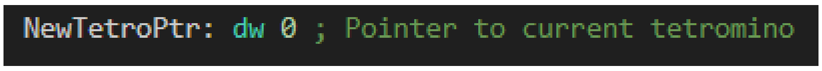
First, it initializes “NewTetroPtr” with a memory address that points to the new tetromino that will be random thanks to the call to the **Random** function, which generates it randomly. It does this by loading ix the value of NewTetroPtr and then that value stored at the memory address is loaded into the hl register.

Then, it checks if there is any collision at the start position of the tetromino by calling the **hasCollision** function and if there is one, it jumps to the **endgame** label.

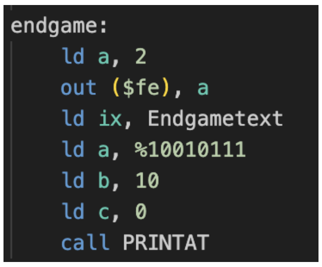
The position (y-1) is then incremented by y and checked again for a collision. If there is a collision, it jumps to the **endgame2** label.

After checking for collisions a call is made to the **savePos** function and it saves the current position of the tetromino and then jumps to the **saveKey** function.

In case there is a collision, **DrawTetromino** is called to draw the figure and the game ends.



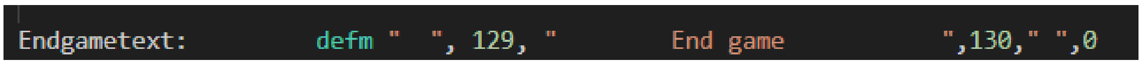
```
NewTetroPtr: dw 0 ; Pointer to current tetromino
```



```
endgame:
    ld a, 2
    out ($fe), a
    ld ix, Endgametext
    ld a, %10010111
    ld b, 10
    ld c, 0
    call PRINTAT
```

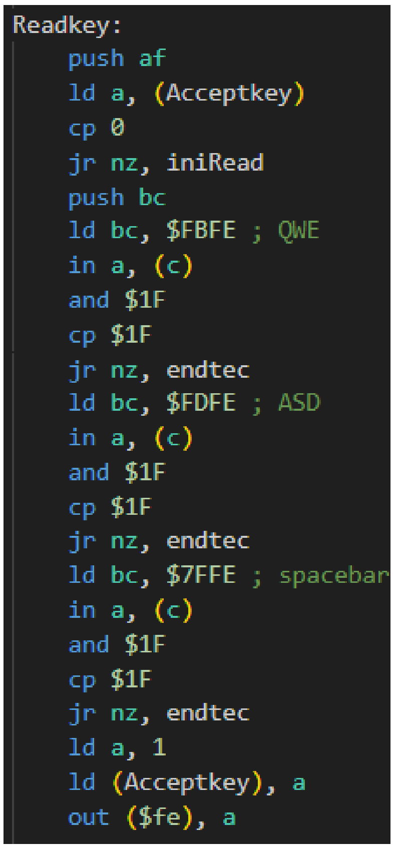
endgame: this function indicates the end of the game and loads the memory address of the text that appears on the screen. It loads in ix “Endgametext” so then, after call PRINTAT , the message appears on the screen at the end of the game in the indicated position. To do this, you must perform the “include ‘TextDemo.asm’ “.

%10010111 red-white.



```
Endgametext:            defm "   ", 129, "          End game              ",130," ",0
```

6.2. Keyboard read



```
Readkey:
    push af
    ld a, (Acceptkey)
    cp 0
    jr nz, iniRead
    push bc
    ld bc, $FBFE ; QWE
    in a, (c)
    and $1F
    cp $1F
    jr nz, endtec
    ld bc, $FDFE ; ASD
    in a, (c)
    and $1F
    cp $1F
    jr nz, endtec
    ld bc, $7FFE ; spacebar
    in a, (c)
    and $1F
    cp $1F
    jr nz, endtec
    ld a, 1
    ld (Acceptkey), a
    out ($fe), a
```

Readkey: this function reads the QWE, ASD and spacebar keys on the keyboard and if either of these keys is pressed, sets Acceptkey to 1.

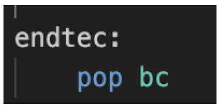
With ld a, (Acceptkey) the value of the Acceptkey variable is loaded into register a and with cp 0 it compares the value in register a with 0. If the value in register a is not zero, it jumps to the **iniRead** label.

If Acceptkey = 1, then you can use the keys.

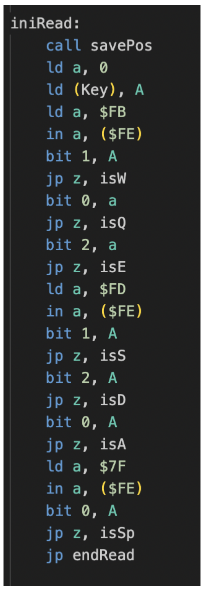
With ld bc, $FBFE the value $FBFE is loaded into registers b and c, which represents the QWE key on the keyboard.

In a, (c) is used to read the state of the QWE key and places it in register a. By doing an and $1F we will keep only the 5 least significant bits and then compare with $1F, which checks if those keys are pressed. If they are not, it will make a jump to the "endtec" label.

For ASD and spacebar keys, the process is repeated . Then the border is reestablished with blue again.



```
endtec:
    pop bc
```

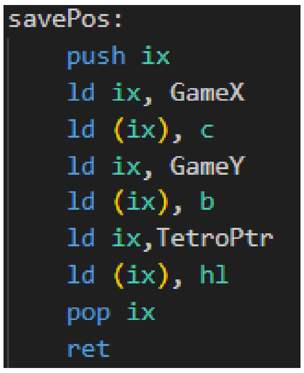


```
iniRead:
    call savePos
    ld a, 0
    ld (Key), A
    ld a, $FB
    in a, ($FE)
    bit 1, A
    jp z, isW
    bit 0, a
    jp z, isQ
    bit 2, a
    jp z, isE
    ld a, $FD
    in a, ($FE)
    bit 1, A
    jp z, isS
    bit 2, A
    jp z, isD
    bit 0, A
    jp z, isA
    ld a, $7F
    in a, ($FE)
    bit 0, A
    jp z, isSp
    jp endRead
```

iniRead: this function detects which key has been pressed and jumps to the corresponding subroutine (isW, isQ, isE, isS, isD, isA e isSp) to handle the action that corresponds to that key. To do this, it makes a call to the **savePos** function, which saves the current positions of the tetromino.

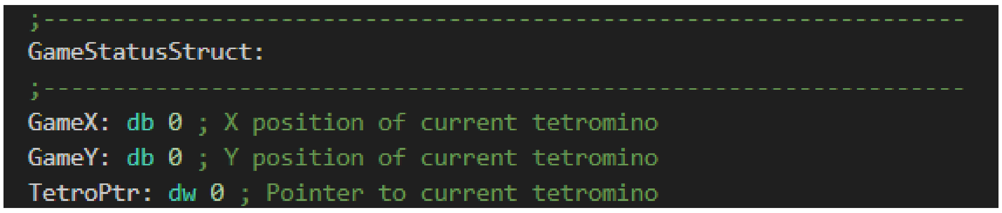
To check which key has been pressed, for example, in the case of W key, it uses bit 1, a to check if bit 1 is the one that is active and if it is, it jumps to the isW label. This is repeated for the other keys.

If none of the keys are used in the game, it jumps to endRead, which restores the contents of the flag register and jumps to the loop label.



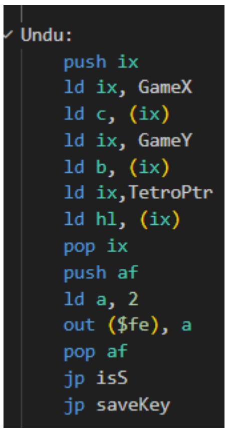
```
savePos:
    push ix
    ld ix, GameX
    ld (ix), c
    ld ix, GameY
    ld (ix), b
    ld ix,TetroPtr
    ld (ix), hl
    pop ix
    ret
```

SavePos: saves the actual values of the tetromino into GameX, GameY and TetroPtr. To do this we use the register ix.

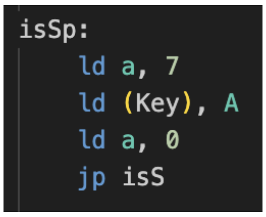


```
;-------------------------------------------------------------
GameStatusStruct:
;-------------------------------------------------------------
GameX: db 0 ; X position of current tetromino
GameY: db 0 ; Y position of current tetromino
TetroPtr: dw 0 ; Pointer to current tetromino
```

Undu: The values of GameX, GameY and TetroPtr are restored and the frame is now of the color red because there was a collision.



```
Undu:
    push ix
    ld ix, GameX
    ld c, (ix)
    ld ix, GameY
    ld b, (ix)
    ld ix,TetroPtr
    ld hl, (ix)
    pop ix
    push af
    ld a, 2
    out ($fe), a
    pop af
    jp isS
    jp saveKey
```



```
isSp:
    ld a, 7
    ld (Key), A
    ld a, 0
    jp isS
```

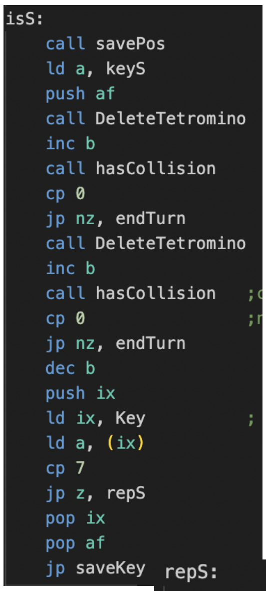
The **isSp** label has the functionality to address the Spacebar key. When the Spacebar key is pressed, the code in isSp sets the value of Key to 7 and then jumps to isS. The Spacebar key is handled similarly to the S (isS) key, but with the distinction that a specific value is set to Key before jumping to the subroutine.

isS: handles the downward movement of the tetromino, checks for collisions (so the figure doesn't go off the board), and performs specific actions depending on the key pressed.

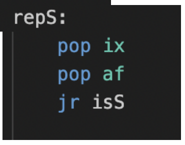
Calls the **savePos** subroutine, which saves the current position of the tetromino. It calls the **DeleteTetromino** subroutine, which deletes the current representation of the tetromino figure on the board. It also makes a call to the **hasCollision** function, which checks if there are any collisions with other pieces or boundaries on the board. Compare the result with zero and if it is not equal to zero there will be a collision and the tetromino will not be able to move, then does it again to make sure there's nothing below and next turn wouldn't be able to move, any further down, jumping to the **endTurn** label, which marks the end of the turn.

It also compares the value of the key (the Spacebar) with 7 and if it has been pressed, it jumps to the **repS** label again until it collides with something making the drop move (if the spacebar is pressed, the tetromino will go down until it detects a collision).

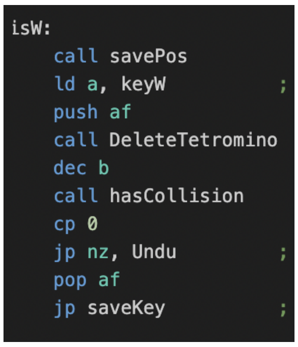
The last thing this function does is a jump to **saveKey** after having retrieved the values from the stack.



```
isS:
    call savePos
    ld a, keyS
    push af
    call DeleteTetromino
    inc b
    call hasCollision
    cp 0
    jp nz, endTurn
    call DeleteTetromino
    inc b
    call hasCollision   ;
    cp 0                ;
    jp nz, endTurn
    dec b
    push ix
    ld ix, Key          ;
    ld a, (ix)
    cp 7
    jp z, repS
    pop ix
    pop af
    jp saveKey
```

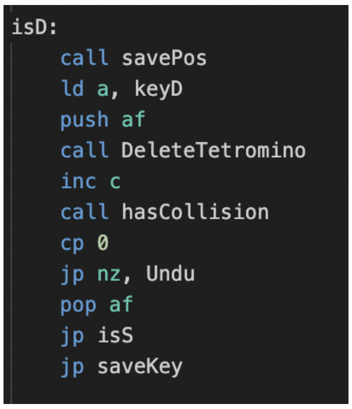


```
repS:
    pop ix
    pop af
    jr isS
```



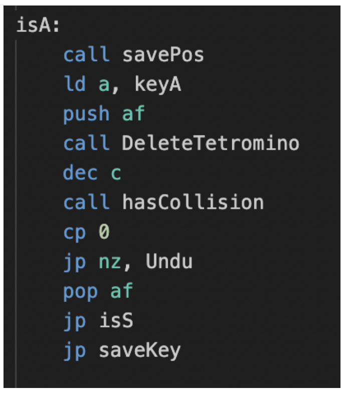
```
isW:
    call savePos
    ld a, keyW             ;
    push af
    call DeleteTetromino
    dec b
    call hasCollision
    cp 0
    jp nz, Undu            ;
    pop af
    jp saveKey             ;
```

isW: This function handles the upward movement of the tetromino. Before making the move the tetromino is deleted with the function “DeleteTetromino”, it makes sure that there are no collisions by calling the **hasCollision** function with the tetromino itself and with other elements on the board. If there is a collision, the movement is reversed by calling the **Undu** function. Also, save the key pressed with the call to **saveKey.**



```
isD:
    call savePos
    ld a, keyD
    push af
    call DeleteTetromino
    inc c
    call hasCollision
    cp 0
    jp nz, Undu
    pop af
    jp isS
    jp saveKey
```

isD: this function handles the rightward movement of the tetromino. Before making the move, it makes sure that there are no collisions with the tetromino itself and with other elements on the board by calling the **hasCollision** function. If there is a collision, the movement is reversed by calling **Undu**. Additionally, it saves the key pressed with the call to **saveKey**. If there is no collision, it jumps to **isS,** which handles downward movement. **DeleteTetromino** deletes the current tetromino to ensure there are no collisions with itself.



```
isA:
    call savePos
    ld a, keyA
    push af
    call DeleteTetromino
    dec c
    call hasCollision
    cp 0
    jp nz, Undu
    pop af
    jp isS
    jp saveKey
```

isA: This function handles the leftward movement of the tetromino. This function does the same as isD but moving into the left.

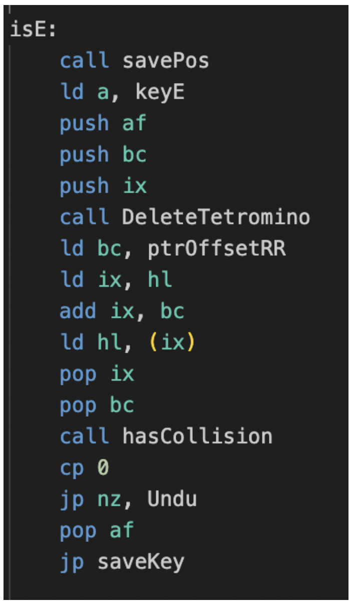
isE: handles the rotation of the tetromino to the right, ensuring there are no collisions and restoring the original position in the event of a collision.

The value associated with the E key is loaded into register a.

The current tetromino is deleted by calling **DeleteTetromino**.

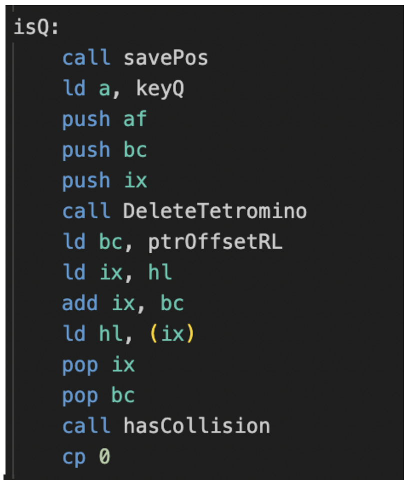
The offset is then added to the current tetromino position stored in the ix register and the new tetromino shape pointer is loaded into the hl register.

It is then checked if the rotation produces a collision with other pieces by calling the **hasCollision** function. If there is a collision, the rotation is undone by jumping to the **Undu** label. And to save the pressed key, jump to the **saveKey** label.

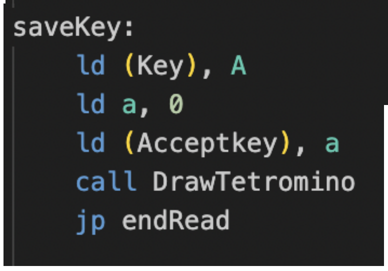


```
isE:
    call savePos
    ld a, keyE
    push af
    push bc
    push ix
    call DeleteTetromino
    ld bc, ptrOffsetRR
    ld ix, hl
    add ix, bc
    ld hl, (ix)
    pop ix
    pop bc
    call hasCollision
    cp 0
    jp nz, Undu
    pop af
    jp saveKey
```

isQ: has the responsibility of rotating the tetromino to the left. This does the same as isE, but it rotates the tetromino to the left instead of to the right.



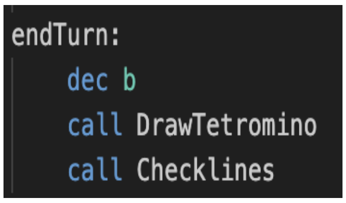
```
isQ:
    call savePos
    ld a, keyQ
    push af
    push bc
    push ix
    call DeleteTetromino
    ld bc, ptrOffsetRL
    ld ix, hl
    add ix, bc
    ld hl, (ix)
    pop ix
    pop bc
    call hasCollision
    cp 0
```



```
saveKey:
    ld (Key), A
    ld a, 0
    ld (Acceptkey), a
    call DrawTetromino
    jp endRead
```

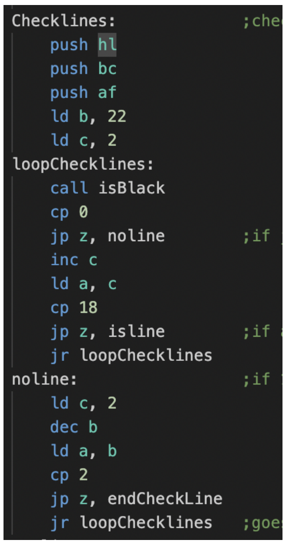
saveKey: it is responsible for saving the information associated with the pressed key, equals Acceptkey to 0, draws the tetromino in its new position and ends key reading.

endTurn: the turn is finished when the key “s” is pressed, the tetromino collides and can’t move anymore. So the tetromino is printed again. Then the function “Checklines” is called.



```
endTurn:
    dec b
    call DrawTetromino
    call Checklines
```

6.3. Checklines



```
Checklines:                 ;che
    push hl
    push bc
    push af
    ld b, 22
    ld c, 2
loopChecklines:
    call isBlack
    cp 0
    jp z, noline            ;if
    inc c
    ld a, c
    cp 18
    jp z, isline            ;if
    jr loopChecklines
noline:                     ;if
    ld c, 2
    dec b
    ld a, b
    cp 2
    jp z, endCheckLine
    jr loopChecklines       ;goe
```

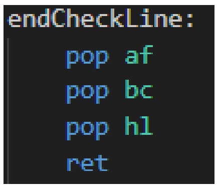
Checklines: is the function that deletes a line when it is done, after that moves all the lines from above 1 down.

loopChecklines: is the loop to check if there's 1 pixel black if that goes 1 up to the next lines, this is down till the line 2 on the top. If there's no black pixels in the line it deletes the line and starts moving all the lines pixel by pixel down.

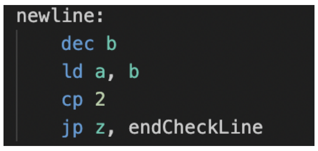
noline: restart the x position to the start of the line and go 1 line up to start again.

newline: it checks if all of the lines are done and with the function “endCheckLine”, ends.

EndCkeckLine: the pops are realized.

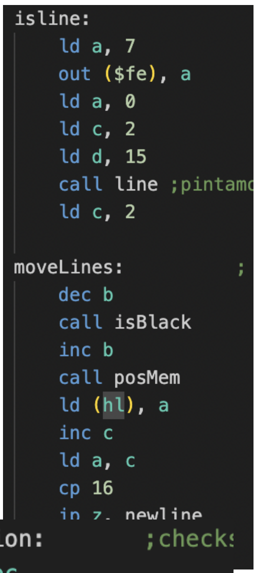


```
endCheckLine:
    pop af
    pop bc
    pop hl
    ret
```



```
newline:
    dec b
    ld a, b
    cp 2
    jp z, endCheckLine
```

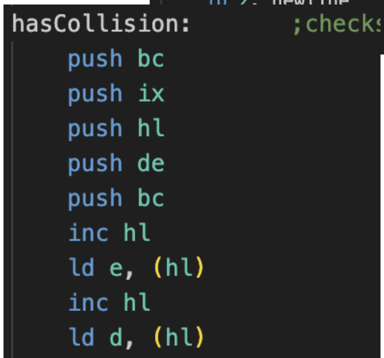
isline: if gets here means that one line is full with color, deletes the line and restarts the x position to start moving lines.



```
isline:
    ld a, 7
    out ($fe), a
    ld a, 0
    ld c, 2
    ld d, 15
    call line ;pintamo
    ld c, 2

moveLines:                ;
    dec b
    call isBlack
    inc b
    call posMem
    ld (hl), a
    inc c
    ld a, c
    cp 16
    jp z, newline
```

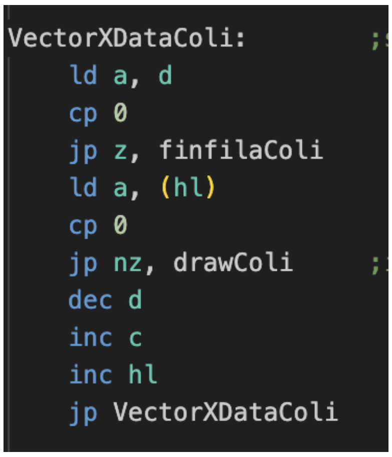
moveLines: Checks the color of the pixel, increase b (y of the position in screen) so goes down 1 pixel and then prints the color, this is done each time for each pixel of the line in every line

6.4. Collisions



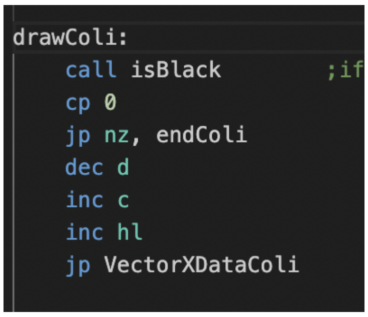
```
hasCollision:           ;check
    push bc
    push ix
    push hl
    push de
    push bc
    inc hl
    ld e, (hl)
    inc hl
    ld d, (hl)
```

hasCollision: this function checks if there is a collision between the tetromino and whatever is on the screen at the current position. This function allows the program to know, in other parts of the code, whether the tetromino figure can move around the board or not to a new position. It stores the positions



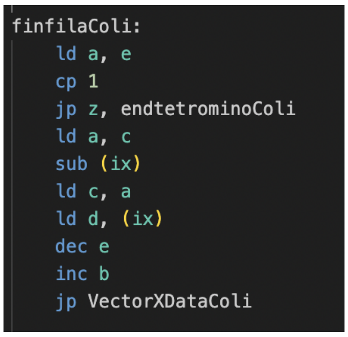
```
VectorXDataColi:                ;
    ld a, d
    cp 0
    jp z, finfilaColi
    ld a, (hl)
    cp 0
    jp nz, drawColi             ;
    dec d
    inc c
    inc hl
    jp VectorXDataColi
```

VectorXDataColi: verifies the collision of the tetromino figure with the screen horizontally.



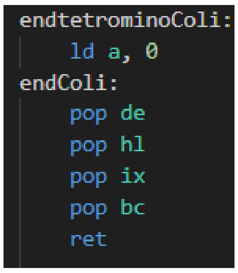
```
drawColi:
    call isBlack          ;if
    cp 0
    jp nz, endColi
    dec d
    inc c
    inc hl
    jp VectorXDataColi
```

DrawColi: this function checks if there is a collision. It makes a call to **isBlack** function to check if there is black color in the current pixel and if there is that color it means that there is a collision and jumps to **endColi**. If there is no collision, it adjusts the positions to continue.



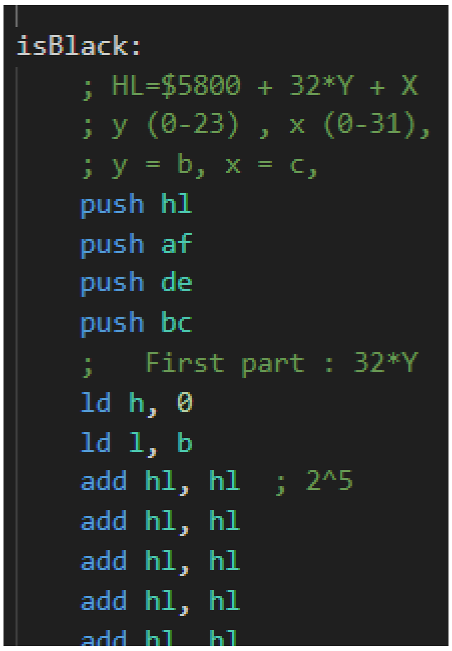
```
finfilaColi:
    ld a, e
    cp 1
    jp z, endtetrominoColi
    ld a, c
    sub (ix)
    ld c, a
    ld d, (ix)
    dec e
    inc b
    jp VectorXDataColi
```

finfilaColi: when e is not equal to 1 it adjusts the position of the tetromino and when it is equal to 1 it indicates that a row has been completed and jumps to **endtetrominoColi,** restoring the value of a.

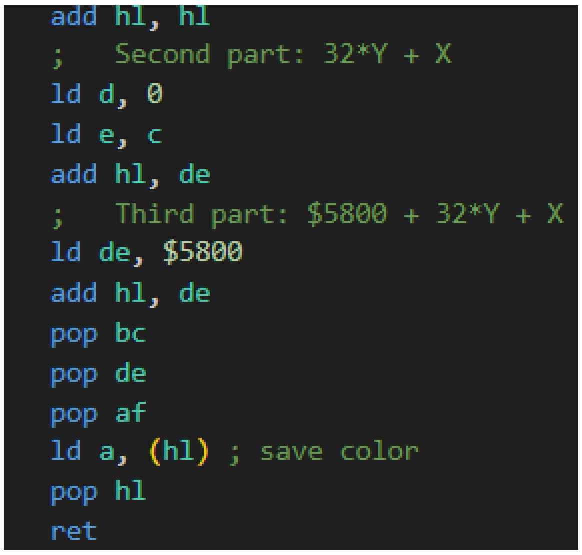


```
endtetrominoColi:
    ld a, 0
endColi:
    pop de
    pop hl
    pop ix
    pop bc
    ret
```

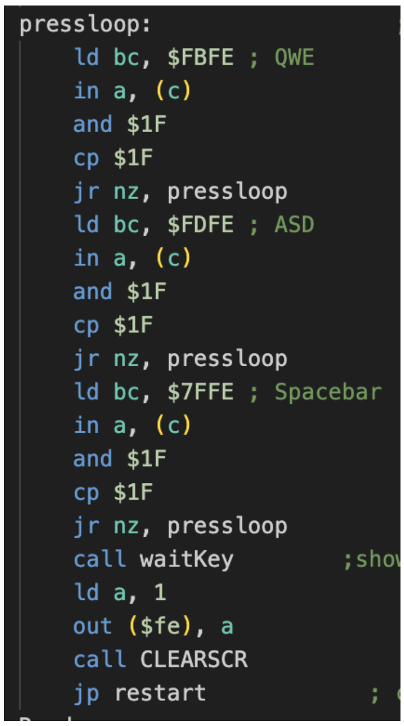
isBlack: indicates whether a pixel at the indicated position on the screen is black.



```
isBlack:
    ; HL=$5800 + 32*Y + X
    ; y (0-23) , x (0-31),
    ; y = b, x = c,
    push hl
    push af
    push de
    push bc
    ;   First part : 32*Y
    ld h, 0
    ld l, b
    add hl, hl   ; 2^5
    add hl, hl
    add hl, hl
    add hl, hl
    add hl, hl
```

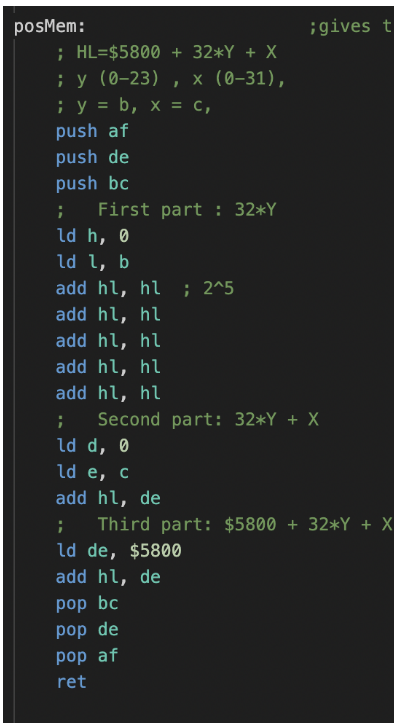


```
    add hl, hl
    ;   Second part: 32*Y + X
    ld d, 0
    ld e, c
    add hl, de
    ;   Third part: $5800 + 32*Y + X
    ld de, $5800
    add hl, de
    pop bc
    pop de
    pop af
    ld a, (hl) ; save color
    pop hl
    ret
```



```
pressloop:
    ld bc, $FBFE ; QWE
    in a, (c)
    and $1F
    cp $1F
    jr nz, pressloop
    ld bc, $FDFE ; ASD
    in a, (c)
    and $1F
    cp $1F
    jr nz, pressloop
    ld bc, $7FFE ; Spacebar
    in a, (c)
    and $1F
    cp $1F
    jr nz, pressloop
    call waitKey          ;sho
    ld a, 1
    out ($fe), a
    call CLEARSCR
    jp restart              ;
```

Pressloop: this function waits until all keys that are used in the game are free. When they are free, a message appears and it says: "press to continue" and the program waits for a key to be pressed. Calling the **CLEARSCR** function clears the screen and then jumps to the restart label. With that function the game is playable again.



```
posMem:                        ;gives t
    ; HL=$5800 + 32*Y + X
    ; y (0-23) , x (0-31),
    ; y = b, x = c,
    push af
    push de
    push bc
    ;   First part : 32*Y
    ld h, 0
    ld l, b
    add hl, hl   ; 2^5
    add hl, hl
    add hl, hl
    add hl, hl
    add hl, hl
    ;   Second part: 32*Y + X
    ld d, 0
    ld e, c
    add hl, de
    ;   Third part: $5800 + 32*Y + X
    ld de, $5800
    add hl, de
    pop bc
    pop de
    pop af
    ret
```

PosMem: it calculates the memory address of a pixel on the screen (the screen is of 32x24). It uses the b and c registers.

7. Percentage

Claudia MU:	33%

- Gameboard
- Random
- Draw and delete Tetrominos

Cristina Fernandez:	31%

- ReadKeys
- Documentation

Rodrigo Sáez:	36%

- HD Graphics
- Collisions and Undo
- Checklines