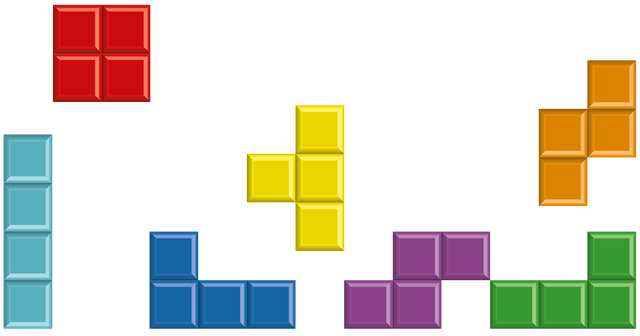


BLANC Monica-Pauline

SPILERS Lise

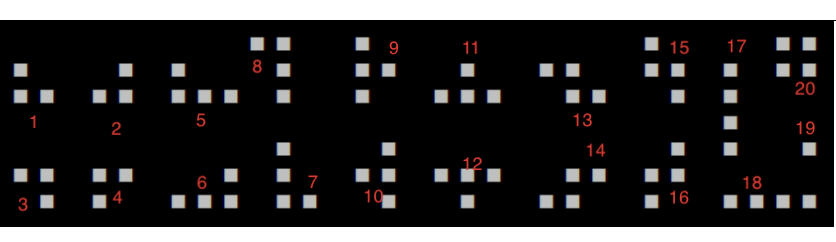
L1-INT2

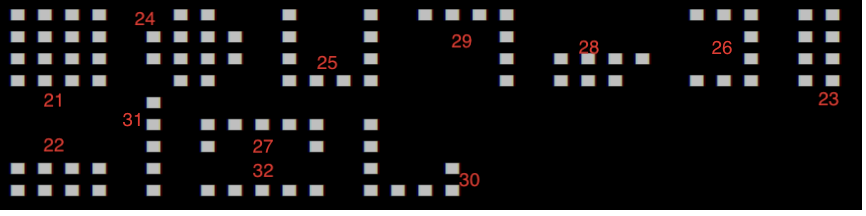
**Tetris project : report**



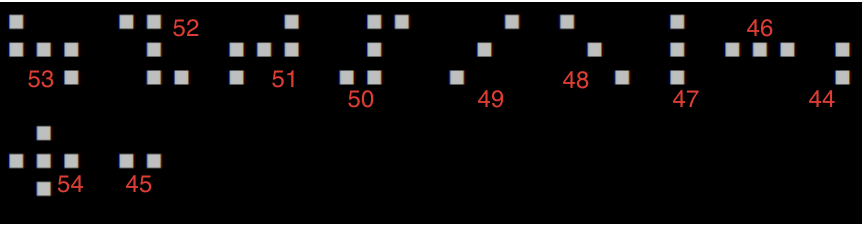
**I. How we process ?**

First of all, we decided to, for each block, put a number (for the diamond the square and the two lines of 5 points are the same block as the circle so we decided to use the same names). So this is a picture with all the number corresponding to the blocks in order to understand.









Now, as the same time that Lise started to do the first function, Pauline decided to create a Word file with all the 57 blocks in order to fill them with 1’s and 0’s to simplify our work for the functions which create each blocks.

Then after making the two first functions, the easy part was complete, we had to pass to the next level.

**II.Explanation of our functions with several with translation in algorithm in order to understand better**

1) All function for each block

Function create\_block\_1 ( )

Local variables : \*\*B: two dimensional array of integer ; i,j : integer

Begin

B = create\_2D\_dyn(5);

For i 🡨 1 to 5, do

For j 🡨 1 to 5 do

B[i][j] 🡨 0

End for

End for

For j 🡨1 to 2, do

B[5][j] 🡨 1

End for

B[4][1] 🡨 1

Return B

End

Function create\_block\_2 ( )

Local variables : \*\*B: two dimensional array of integer ; i,j : integer

Begin

B = create\_2D\_dyn(5);

For i 🡨 1 to 5, do

For j 🡨 1 to 5 do

B[i][j] 🡨 0

End for

End for

For j 🡨1 to 2, do

B[5][j] 🡨 1

End for

B[4][2] 🡨 1

Return B

End

Function create\_block\_3 ( )

Local variables : \*\*B: two dimensional array of integer ; i,j : integer

Begin

B = create\_2D\_dyn(5);

For i 🡨 1 to 5, do

For j 🡨 1 to 5 do

B[i][j] 🡨 0

End for

End for

For j 🡨1 to 2, do

B[4][j] 🡨 1

End for

B[5][2] 🡨 1

Return B

End

Function create\_block\_4 ( )

Local variables : \*\*B: two dimensional array of integer ; i,j : integer

Begin

B = create\_2D\_dyn(5);

For i 🡨 1 to 5, do

For j 🡨 1 to 5 do

B[i][j] 🡨 0

End for

End for

For j 🡨1 to 2, do

B[4][j] 🡨 1

End for

B[5][1] 🡨 1

Return B

End

Int \*\*create\_block\_5

Local variables: \*\*B: two dimensional array of integer ; i,j : integer

Begin

B = create\_2D\_dyn(5);

For i 🡨 1 to 5, do

For j 🡨 1 to 5 do

B[i][j] 🡨 0

End for

End for

B[4][1] = 1

For j 🡨1 to 3, do

B[5][j] 🡨 1

End for

Return B

End

(…) 🡪 same thing until create\_block\_54()

Explanation for the function Create\_block\_n°

We won’t explain all the functions create block because we have approximately 54 functions so it will be long but we will explain the first one. We are going to explain the function : how we proceed to build the shape of the block, how we fill it… (it’s about the same for the others).

So, we add \*\* in front of ***create\_block*** because it’s a two dimensional dynamic array (5X5 because it’s the maximum height and width for all the blocks). We didn’t put parameter because it will receive anything. It will just create the shape of the block. We know that we have to fill it with 1 and 0 ( after when we display it → 0 : empty / 1: little square). So, we declare a two dynamic array called B and we create it with our function ***create\_2D\_dyn***.

Then we do two “for” in order to check all the array and fill it with 0 in order to after fill just the part we want in 1.

We made sure that all our blocks will be nearly of the corner at the bottom on the left. (in yellow)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 |
| 3 | 1 | 0 | 0 | 0 | 0 |
| 4 | 1 | 1 | 0 | 0 | 0 |

The second “for” just deals with the first column (column 0) in order to fill it with 1’s. Then, because it’s just one point that left we decided to put it manually : *B[3][0]* (third line and first column).Then we return B in order to receive it for other functions.

) Function to fill the 2D array of the board, and function to display the board

Function fill\_2D\_board ( A : two dimensional array (empty) ; size, choiceshape : integers)

Copied parameters : size, choiceshape

Changed parameter : A

Local variable : i,j : integers

Begin

For i 🡨 1 to size, do

For j 🡨 1 to size, do

A[i][j] 🡨 1

End for

End for

For j 🡨 3 to size-1, do

A[1][j] 🡨 5

A[2][j] 🡨 2

A[size][j] 🡨 2

End for

For i 🡨3 to size-1, do

A[i][1] 🡨 4

A[i][2] 🡨 3

A[i][size] 🡨 3

End for

If (choiceshape = 1)

For i 🡨 size-1 to (size/2) -1, do

For j 🡨 3 to (size/2) + 1, do

If (i+j ≥ size) then,

A[i][j] 🡨 0

End if

End for

End for

For i 🡨 size-1 to (size/2), do

For j 🡨 size-1 to (size/2), do

If (i ≥ j) then,

A[i][j] 🡨 0

End if

End for

End for

End if

If ( choiceshape = 2)

For i 🡨 3 to (size/2) +1, do

For j 🡨 3 to (size/2) +1

If ( i+ j ≥ (size/2)+2 then

A[i][j] 🡨 0

End if

End for

End for

For i 🡨 3 to (size/2) +1, do

For j 🡨 size-1 to (size/2) -1, do

If ( j-i ≤ (size/2)-2) then

A[i][j] 🡨 0

End if

End for

End for

For i 🡨 size-1 to (size/2) -1, do

For j 🡨 3 to (size/2) +1

If ( i- j≤ (size/2)-2) then

A[i][j] 🡨 0

End if

End for

End for

For i 🡨 size-1 to (size/2) -1, do

For j 🡨 size-1 to (size/2) -1

If ( I + j ≤ size +(size/2)-2) then

A[i][j] 🡨 0

End if

End for

End for

End if

If ( choiceshape =3) then,

For i 🡨 3 to size-1, do

For j 🡨 3 to size-1, do

A[i][j] = 0

End for

End for

For i 🡨 3 to (size/3)-1

For j 🡨 3 to (size/3) -1, do

If ( i+j ≤ size-(size/2)-(size/4) +1 )

A[i][j] = 1

End if

End for

End for

For i 🡨 3 to (size/3)-1, do

For j 🡨 size-1 to 2\*(size/3) +1, do

If ( j-i ≥ (size/2)+(size/4) -1 )

A[i][j] = 1

End if

End for

End for

For i 🡨 size-1 to 2\*(size/3)+1, do

For j 🡨 3 to (size/3) -1, do

If ( i-j ≥ (size/2)+(size/4) -1 )

A[i][j] = 1

End if

End for

End for

For i 🡨 size-1 to 2\*(size/3)+1, do

For j 🡨 size-1 to 2\*(size/3) +1, do

If ( i+j ≥ size +(size/2)+(size/4) -1 )

A[i][j] = 1

End if

End for

End for

End if

End

Explanation for the function fill\_2D\_board

This function is void function which has in parameter a two-dimensional dynamic array, the size and the choice of the shape the user wants. We decided to explain just for the first shape (triangle) because it the same process with for loop, if ...etc.

With the comments we can see that at the beginning with a double “for” that goes from 0 to the size: we fill the board with 1 that corresponds to spaces (see in the part *How we proceed*).

Then we create the border with letters (5 correspond to the horizontal letters and 4 to the vertical ones)  and little lines (vertical is 2 and horizontal is 3).

* *for (j=2;j<size-1;j++)* :
* A[0][j] = 5 : all the first line will be filled with letters
* A[1][j] = 2 : we fill the second line with little horizontal lines
* A[size-1][j]: we fill the last line with little horizontal lines too
* *for (i=0;i<size-1;i++)* :
* A[i][0] = 5 : all the first column will be filled with letters
* A[i][1] = 2 : we fill the second column with little vertical lines
* A[i][size-1]: we fill the last line with little vertical lines too

Then we fill with points to create the shape of what the user wants :

* For the triangle and the diamond we fill with points the shape directly.
* For the circle we first fill all with points and take of the corner.
* For the triangle, we do two double for in order to divide the triangle in two.
* For the diamond and the circle we do four double for in order to do the four triangle (diamond) and the four corners (circle).

As we said before we decided to explain just the triangle. In order to find the right number to subtract, add or divide we do a sketch on paper.

First part: left part:

The first “for” is for the left part : from size-2 (because we start at the bottom and we don’t forget the border !) to size/2-1 (the middle of the line) and the other for from 2 (border again but for the column) to size/2+1 (middle of the column). First, we will try to fill the diagonal :

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 10 |  |  | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U |  |
| 11 |  |  | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | || |
| 12 | K | || |  |  |  |  |  |  |  |  |  |  | . |  |  |  |  |  |  |  |  |  |  | || |
| 13 | L | || |  |  |  |  |  |  |  |  |  | . | . | . |  |  |  |  |  |  |  |  |  | || |
| 14 | M | || |  |  |  |  |  |  |  |  | . | . | . | . | . |  |  |  |  |  |  |  |  | || |
| 15 | N | || |  |  |  |  |  |  |  | . | . | . | . | . | . | . |  |  |  |  |  |  |  | || |
| 16 | O | || |  |  |  |  |  |  | . | . | . | . | . | . | . | . | . |  |  |  |  |  |  | || |
| 17 | P | || |  |  |  |  |  | . | . | . | . | . | . | . | . | . | . | . |  |  |  |  |  | || |
| 18 | Q | || |  |  |  |  | . | . | . | . | . | . | . | . | . | . | . | . | . |  |  |  |  | || |
| 19 | R | || |  |  |  | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . |  |  |  | || |
| 20 | S | || |  |  | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . |  |  | || |
| 21 | T | || |  | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . |  | || |
| 22 | U | || | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | || |
| 23 |  |  | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ |  |

In this example, we have size = 24, and we remark that the diagonal is defined by i+j=size (U+A = 22+2=24, R+D= 19+5=24…), and then, for all others point, if we add I and j, we will find a number above 24. It’s why we have :

if (i+j >=size)

A[i][j] =0

Second part: right part:

We proceed as the same way : first find the diagonal, and then, find the formula to fill the good part of the diagonal. Here, we ant to fill the part under the diagonal.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 10 |  |  | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U |  |
| 11 |  |  | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | || |
| 12 | K | || |  |  |  |  |  |  |  |  |  |  | . |  |  |  |  |  |  |  |  |  |  | || |
| 13 | L | || |  |  |  |  |  |  |  |  |  | . | . | . |  |  |  |  |  |  |  |  |  | || |
| 14 | M | || |  |  |  |  |  |  |  |  | . | . | . | . | . |  |  |  |  |  |  |  |  | || |
| 15 | N | || |  |  |  |  |  |  |  | . | . | . | . | . | . | . |  |  |  |  |  |  |  | || |
| 16 | O | || |  |  |  |  |  |  | . | . | . | . | . | . | . | . | . |  |  |  |  |  |  | || |
| 17 | P | || |  |  |  |  |  | . | . | . | . | . | . | . | . | . | . | . |  |  |  |  |  | || |
| 18 | Q | || |  |  |  |  | . | . | . | . | . | . | . | . | . | . | . | . | . |  |  |  |  | || |
| 19 | R | || |  |  |  | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . |  |  |  | || |
| 20 | S | || |  |  | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . |  |  | || |
| 21 | T | || |  | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . |  | || |
| 22 | U | || | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | || |
| 23 |  |  | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ |  |

We still have size = 24, and we remark that the diagonal is defined by i=j, and then, for all others point, we remark that we have i>j . It’s why we have:

if (i >=j)

A[i][j] =0

5) Function block and display\_block

Function block ( block\_tab[34][5][5])

Changed parameter : block\_tab

Local\_variable : \*\*Ar: two dimensional array dynamic ; index, j, k :integer

Begin

for index🡨1 to 32, do

Ar🡨fill\_location\_tab(index+1)

for j🡨1 to 5, do

for k🡨 1, to 5, do

block\_tab[index][j][k]🡨Ar[j][k]

end for

end for

end for

for index🡨1 to index<46, do

Ar🡨fill\_location\_tab(index+1)

if (index<=20 OR index => 29)

for j🡨1 to 5, do

for k🡨1 to 5, do

block\_tab[index][j][k]🡨Ar[j][k]

end for

end for

end if

end for

for index🡨1 to 57, do

Ar=fill\_location\_tab(index+1)

if (index <= 20 OR index => 45)

for j🡨1 to 5, do

for k🡨1 to 5, do

block\_tab[index][j][k]🡨Ar[j][k]

end for

end for

end if

end for

end

Explanation for the function block

It’s a void function because it returns anything. It has three-dimensional static array called blocks\_tab in parameter (*blocks\_tab [57][5][5]* : index for the 57 blocks / dynamic array for one block).

We will explain just for the circle and the diamond because we add a thing in the diamond so it’s important to explain it, but for the triangle it’s the same as the diamond.

* For the circle :

In the first “for” we go to 32 because it corresponds to the last block of the circle. We can see it in our part “How we proceed” with a picture of the blocks with their corresponding number. Then, we say that the double pointer Ar (that corresponds to the array of a block) takes the function **fill\_location\_tab (index+1)** : that means for each i (we say index+1 because **fill\_location\_tab** start at 1), *Ar* will takes the corresponding blocks. Then, we do a double “for” in order to implement in the case of dynamic array for one block ([5][5]) the corresponding block. So, *block\_tab* will fill in the first case the index (between 0 and 31) and then the corresponding block.

Example :

* *index = 2*
* *Ar =* **fill\_location\_tab(3)** : that corresponds to **create\_block\_3()**, the third block.

Thus, for *j=0 to 5* and for *k=0 to 5*, *block\_tab [3][j][k] = Ar[j][k].*

* For the diamond and the triangle:

As we say before, it’s actually the same with a little difference: we know that it exists blocks which are in common with all shapes. When we put the number we saw that it was for 1 to 20.

We do as the circle, a first for to 46 (46 =  the last blocks that correspond to the diamond) and then the double pointer *Ar* that takes the **function fill\_location\_tab**. Now we can see the little difference: we add an “if”. We add it because of what we said before, we know that all the shapes contain 20 blocks in common. So we say that *if index<20* or *index>29* (29 and not 32 because the circle and the diamond has 3 blocks in common) that means if it corresponds to a common block or to a diamond block we can continue.

Function display\_block1(choiceshape: integer ; C[57][5][5] three dimensional array)

Copied parameter : choiceshape, C

Local variable :

triangle\_array([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,46,47,48,49,50,51,52,53,54,55,56,57])

diamond\_array([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46])

circle\_array([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32]) : arrays of integer

i,j,k : integers

Begin

if (choiceshape=1)

for i🡨1 to 31, do

for j🡨1 to 5, do

for k🡨1 to 5, do

if (C[triangle\_array[i]][j][k]=1)

Write (“■ “)

else

write (“ ")

end if

write (" ")

end for

write (triangle\_array[i], “ “)

end for

end for

end if

if (choiceshape=2)

for i🡨1 to 37, do

for j🡨 1 to 5, do

for k🡨1 to 5, do

if (C[diamond\_array[i]][j][k]=1)

Write (“■ “)

else

write (" ")

end if

write (" ")

end for

write (" ")

end for

write (diamond\_array[i], “ “)

end for

end if

if (choiceshape==3)

for i🡨 1 to 32, do

for j🡨1 to 5, do

for k🡨 1 to 5, do

if(C[circle\_array[i]][j][k]=1)

Write (“■ “)

else

write (“ ")

end if

write (" ")

end for

write (" ")

end for

printf (circle\_array[i], “ “)

end for

end if

end

6) Functions to select a block and choose a block

Function select\_block (choicepolicy, choiceshape : integer ; \*\*randomblock[37]

Copied parameters: choicepolicy, choiceshape

Changed parameter : randomblock

Local variable:

triangle\_array([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,46,47,48,49,50,51,52,53,54,55,56,57])

diamond\_array([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46])

circle\_array([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32]) : arrays of integers

random, j, i: integers

Begin

srand((unsigned)time(NULL))

if (choicepolicy=1 AND choiceshape=1)

for i🡨1 to 31, do

triangle\_array[i] 🡨 randomblocks[i]

end for

end if

if (choicepolicy=1 AND choiceshape=2)

for i🡨1 to 37, do

diamond\_array[i]🡨randomblocks[i]

end for

end if

if (choicepolicy=1 AND choiceshape=3)

for i🡨1 to 32, do

circle\_array[i]🡨randomblocks[i]

end for

end if

if (choicepolicy=2 AND choiceshape=1)

for i🡨1 to 3, do

if (i=1)

random 🡨 rand()%31

end if

if (i=2)

do

random = rand()%31

while(random=randomblocks[1])

end if

if (i=3)

do

random = rand()%31

while(random=randomblocks[1 ] OR random=randomblocks[2])

end if

randomblocks[i]🡨triangle\_array[random]

end for

end

if (choicepolicy=2 AND choiceshape=2)

for i🡨1 to 3, do

if (i=1)

random = rand()%37

End if

if (i=2)

do

random = rand()%37

while(random=randomblocks[1])

end if

if (i=3)

do

random = rand()%37

while(random=randomblocks[1] OR random=randomblocks[2])

end if

randomblocks[i]🡨diamond\_array[random]

end for

end if

if (choicepolicy=2 AND choiceshape=3)

for i🡨1 to 3, to

if (i=1)

random = rand()%32

end if

if (i=2)

do

random = rand()%32

while(random=randomblocks[1])

end if

if (i=3)

do

random = rand()%32

while(random=randomblocks[1] || random=randomblocks[2])

end if

randomblocks[i]🡨circle\_array[random]

end for

end if

Explanation for the function Select\_block

The first part of this function just fill the array randomblocks according to the shape. (randomblocks has a size of 37 because the array of the diamond contains 37 and it’s the bigger size). We decided to explain more the second part (second choice : 3 random blocks) of the function **select\_block** because we think that it has to be explain.

Also, we explain just for the first shape (here *choiceshape==1* : the triangle) because it’s approximately the same process just some number that changes.

So first we do a “for” that start at 0 until it’s lower than 3 in order to have 3 random blocks. For each i we decided to put in the variable *random* a random value between 0 and 31. (31 because it’s the size of the array *triangle\_array*, for the circle and the diamond changes this value in order to respect the size of the array).

Then, we do a “do...while” in order to not put the same block, that’s why we have while(*random==randomblocks[0]* = it means that we continue the loop if the random when i=0 it’s the same when i=1. We do the same for i=2, but we look for i=0 and i=1. Then in the “for”  the array *randomblocks*[in function of the i] will stock the correct block in function of the index random in the *triangle\_array*. (same for the other shape)

Example :

* i = 0 ; random = 20
* i = 1 ; random = 2
* i = 2 ; random = 10

*randomblock[0] = triangle\_array[20]*

*randomblock[1] = triangle\_array[2]*

*randomblock[2] = triangle\_array[10]*

The first position of *randomblocks* will take the 20th position of the *triangle\_array* that’s correspond to the block n°20.

Explanation for the function Choosecoordinate

This function is an int because it returns x which is the number of the line or column because in our main we use this function twice. At the beginning we declare two array of letter (the alphabet in upper and lower) and a x equals to 0. Then we enter in a “do...while”. The “do...while” ends when okay equals to 0 that means when the letter the user has entered is in one of the two arrays.

The first condition when we enter in the “do...while” is an “if” (*if x==0*) we did it because as we said before we use this function twice for lines and columns so when we asked for the line x =0 but if we want the column we have to change the x (that’s why the column is after the condition “else”. Thus, at the beginning x=0, it asks for the line and stored in the variable x2 and checks with getchar if it’s a number or a letter (if it’s not a letter it asks again) and then it enters in a “for” from  0 to c-3 : it means that it looks at the column (or line it’s the same size) minus 3 because we have to be careful because of the border. We check if the letter the users enters is in one of the two arrays and if it is okay takes 1 and x takes i + 2 (because A=2 because of the border) and returns x. If it’s not, okay and x stays at 0 so we ask again to enter a line.

Example :

* *When it works*
* x=0, and okay=0
* the game board has a size of 22
* We assume that the user enters F.

For i=0 to 19, so to A to S we look if F is in the list of upper or lower letter. It is so :

* F it’s at position i=6
* okay = 1
* x = 8
* *When it doesn’t work*
* x=0, and okay=0
* The game board has a size of 22
* We assume that the user enters U .

For i=0 to 19, so to A to S we look if U is in the list of upper or lower letter. It is not because U is at position 21 but we stops at position 19. So okay and x stays at 0 so we ask again.

7) Function to check if the coordinate given by the user are good.

Function check\_validity (\*\*A, \*\*Block: two dimensional array; size, posR, posC: integer)

Copied parameters : A, Block, size, posR, posC

Local variable : i,j,hb,wb

Begin

For i🡨5 to 1, do

For j🡨5 to 1, do

If (block[i][j]=1), then

If ( i=0), then

hb=5

End if

If (i=1), then

hb=4

End if

If (i=2), then

hb=3

End if

If (i=3), then

hb=2

End if

If (i=4), then

hb=1

End if

If (wb < j+1)

wb= j+1

End if

End if

End for

End for

If ((posR - hb < 2) OR (posC +wb > size-1))

Return 0

End if

For (i🡨 5 to 1, do

For (j🡨1 to 5, do

If ((A[posR-4+i][j+posC]=1 AND block[i][j]=1) OR (A[posR-4+i][j+posC]=2 AND block[i][j]=1) OR (A[posR-4+i][j+posC]=3 AND block[i][j]=1) OR (A[posR-4+i][j+posC]=6 AND block[i][j]=1))

Return 0

End if

End for

End for

Return 1

End

Explanation for the function check\_validity

This function is an int function because it will return 1 if the check it corrects otherwise it will return 0. This function was very important so we stay blocked a long time but we find the solution.

Firstly, we compute the height and the width of the block we want to check in order to see if when we place the block it goes out of the gameboard (border) or the shape.

We do a double “for” that start at the bottom on the right of the array of 5x5 and then it ascends because for several block if we had to start at the bottom on the left it doesn’t work (we see it trying on paper : we draw blocks on paper and think of how we can have the height and the width). Then, we add the if in order to check in the array the case when it corresponds to 1’s. Then for each i, we implement a different number for the height, and we say that if it’s less than j+1 we implement j+1 for the width.

Example with the block  n°33:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
| 0 | 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 2 | 0 | 1 | 0 | 0 | 0 |
| 3 | 1 | 1 | 0 | 0 | 0 |
| 4 | 1 | 0 | 0 | 0 | 0 |

We start as we said at [4][4] ([line][column]):

So in the first line (starting at the bottom on the right: in yellow), we reach a 1 at [4][0], we enter in the if :

* i = 4 so hb = 1
* j+1 = 1, wb=0 so is < 1, so wb = 1.

Then in the second line, we reach a 1 at  [3][1] and  [3][0] after passing to the third line so we going to do do for the last one (it’s the last wb that counts when we pass to another line):

* i = 3 so hb= 2
* j+1 = 1, wb=1 so is < 1, so wb = 1.

Third line :

* i = 2 so hb = 3
* j+1 = 2, wb=1 so is < 2, so wb = 2.

Fourth line :

* i = 1 so hb = 4
* j+1 = 3, wb=2 so is not <1, so wb = 2.

Last line :

* at [0][3]
* i = 0 so hb = 5
* j+1 = 4, wb=2 so is < 4, so wb = 4.
* at [0][2]
* i = 0 so hb = 5
* j+1 = 3, wb=4 so is not < 4, so wb = 4.

So at the end we got the right size:

height = 5

width = 4

Now we have the height and the width we have to compare it in the gameboard.

First Case :

We do a “if” to check if it’s out of range(out of the gameboard), if yes it returns 0, otherwise it returns 1 : (posR correspond to the line where the user wants to place the block and posC the column)

* posR - hb<2 : because in our case A starts at 2 because of the border so it can’t be less than 2.
* posC+wb >c-1 : because in our case if for example the user wants to play it at U and it have a size of 21, the only block that it can place it’s those with wb=1.

In our example, posC +wb = = 21 + 1 = 22 so it can’t place it because it’s higher than c-1 (20). We put c-1 because we have the border.

Second Case :

Now that we exclude cases where it can be out of range we can do a second check now looking at the game board and the block.

We do a double “for” starting at the last line but first column and we check if :

* *A[posR-4+i][j+posC]==1 && block[i][j] ==1*: the first condition checks if in the gameboard the line where we wants to place and the columns where we wants to place corresponds to a space in the gameboard so out of the shape. The second condition is that we check if it correspond to a 1 in the array of the block
* *A[posR-4+i][j+posC]==2 && block[i][j] ==1* : the first condition checks if in the gameboard the line where we wants to place and the columns where we wants to place corresponds to an horizontal straight line so when it’s reach the border of the gameboard. The second condition we check if it corresponds to a 1 in the array of the block
* *A[posR-4+i][j+posC]==3 && block[i][j] ==1 :*  the first condition checks if in the gameboard the line where we wants to place and the columns where we wants to place corresponds to a vertical straight line so when it’s reach the border of the gameboard. The second condition we check if it correspond to a 1 in the array of the block
* *A[posR-4+i][j+posC]==6 && block[i][j] ==1 :* he first condition checks if in the gameboard the line where we wants to place and the columns where we wants to place corresponds to a little square in the gameboard so when the user wants to put its blocks on a block already placed .The second condition we check if it corresponds to a 1 in the array of the block

Then if we are not in both cases we can place our block so the check return 1.

8) Function which place the block

Function place\_block( \*\*A,\*\*btp : two dimensional array of integers ; x,y : integers) : integer

Copied parameters:  x,y,btp

Changed parameter  : A

Local variable: i,j,hb,wb: integers

Begin

hb 🡨 0

wb 🡨 0

              For i 🡨5 to 1 do

        For j🡨5 to 1

                            If ( btp[i][j]  = 1) then

                If (i=0) then

                    hb 🡨 5

                End if

                If (i=1) then

                    hb 🡨 4

                End if

                If (i=2) then

                    hb 🡨 3

                End if

                If (i=3) then

                    hb 🡨 2

                End if

                If (i=4) then

                    hb 🡨 1

                End if

                If (wb<j+1) then

                    wb 🡨 j+1

                End if

            End if

        End for

    End for

For i →5 to 1 do

        For j→ 1 to 5

                            If ((A[x-4+i][y+j]  = 0 AND btp[i][j]=1) then

                          A[x-4+i][y+j]  → 6

            End if

        End for

    End for

End

14) Function to compute the score

Function score (\*\*A : two dimensional array; num, size, corl: integers) : integer

Copied parameters: A, num, size, corl

Local variable : i,j,score

Begin

score🡨0

If (corl = 1), then

For i🡨3 to size-1, do

If (A[num][i] =6), then

score🡨 score+10

End if

End for

End if

If (corl=2), then

For j🡨3 to size-1, do

If (A[i][num]=6)

score🡨 score+20

End if

End for

End if

Return score

End

Explanation for the function score

This function is pretty simple but we thought that we had to explain just one thing : the variable *corl*. In fact, this function count the elimination of the column and the lines (we add 20 for the column because it’s more difficult to eliminate a column).

The variable *corl* allows us to see the difference when we eliminate a line or a column. So, in our main, when the function **state\_line do** its job (that means that the line is filled with blocks), corl =1 so in the function score we enter in the first case. It’s the same case for the column but it’s with the function **state\_column**.

**III- The main**

The main is crucial in a lot of projects, in fact we use the main file to summarize all of our function in a specific order, so our program has some kind of continuity. First of all, we include our header file named “Tetris” and our file .c where all the functions are.

The main file starts with a configuration of the type of game that the user wants (shape, size, 3 blocks or all of them) and then with this: (**system("cls");**), the menu disappear and we can start the game.

Then, we implement our function in the right order so as to play to Tetris!

The first part consists in choosing all parameters for the game. In this part, we need 9 parameters, we use 3 declarations:

* First declaration: r, choice, choice2, choice2\_2, choice3 and choice3\_3. (we will not use r2 in this part, but it has the same utility than r)
* r: verifies that you enter a number.
* choice: allows you to choose if you want to see the rules or to play.
* choice2: allows you to choose the shape: triangle, diamond or circle.
* choice2\_2: ask you if you are sure of your choice2.
* choice3: ask you if you want to choose one block among all of them, or among three random blocks displayed by the game.
* Choice3\_3: ask you if you are sure of your precedent choice.
* Second declaration: SIZE, which will be the dimension of your game.
* Third declaration: gameboard, which is our 2-dimensional dynamic array for the game board.

Then, we ask we ask the user to do all its choice, and finally we use all its answer to display and show him its game-board.

The second part concerns the blocks: now we want to display three or all blocks to let the user choose which one he wants, and then where he wants to place it. We begin this part with a “do... while” because we will repeat all the actions until the end of the game, so this do… while contains all the parts 2 and 3. The game stops when you want to place a block at a wrong place for the third time. It’s what our while contains, with the variable “error” that compute how many time you try to place your block at a wrong place. Then we use the function block to fill the array that we declared at the beginning to have a variable block which has all the blocks.

We display all blocks, or only three blocks thanks to the functions select\_block which and display\_block, and we ask the user to choose which one he wants to place, thanks to the variable “chosen”. The user just has to enter the number corresponding to the block he wants to place, and chosen will take the value. After that, we ask him to enter the coordinate to place its block, but he will enter two letters, so we need to convert it in number to find the good place. To do so, we use the function choose coordinate. This function will take a letter given by the player, verifies that the letter is in the board, if no, ask the user to enter his letter again, and if yes, it will transform the letter to the corresponding number and return it.

Example: if you enter B, it will return the number of the position of B in the array of the gameboard, so it will return 3:

*Here we see that B is at position 3*

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Positions | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0 |  |  | A | B | C | D | E | F | G | … |
| 1 |  |  | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ |
| 2 | A | | |  |  |  |  |  |  |  |  |
| 3 | B | | |  |  |  |  |  |  |  |  |
| 4 | C | | |  |  |  |  |  |  |  |  |
| 5 | D | | |  |  |  |  |  |  |  |  |
| 6 | E | | |  |  |  |  |  |  |  |  |
| 7 | F | | |  |  |  |  |  |  |  |  |
| 8 | G | | |  |  |  |  |  |  |  |  |
| 9 | … | | |  |  |  |  |  |  |  |  |

We use two variable to collect the position : co\_x for the number of the line, and co\_y for the number of the column. Afterward, we create a new two-dimensional array blocks2 and fill it in order to obtain the block chosen by the user thanks to create\_2D\_dyn and fill\_location\_tab. Now we have our block to place so we can call the function check\_validity wich will compare the two-dimensional array of the block and the board:

* To don’t be out of range, we calcul the dimension of the block, its weight and its high. If the position + the weight is greater than the total dimension of the board, it means the you can not place your block because it will overfill the board! In the same way, if position – high is smaller then 2, it will overfill the board.
* Now we compare what is contained in each array.

Check\_validity returns 1 if the position is correct, and 0 if not, so we use a new variable called check to now what is returns, then if check is equal to 0 (if check\_validity had returned 0), we add 1 to the variable error, else, we can place the block in the board. We will repeat this action (ask the user to enter its coordinate) until check is different from 0 thanks to another do…while. Part 2 is finished, go to the next part!

Part 3: in this part, we have placed our block, so now we want to check if a line or a column is empty and if yes, eliminate it and compute the score. If a line is eliminated, we have to shift all lines above the eliminated line. Let’s do it! We begin a line with a for because we are going to check every lines and column one by one! So then, it’s what we do thanks to state\_line and state\_column. If a line is empty, we have a variable named “CorL” which take the value 1. We will use this variable in score, because in our function score, the points gained are not the same when you eliminate a line or a column, so this variable allows us to now what to do in the function score if we apply the function score before eliminate a line or a column. So, it takes the value 1 if we are in lines, and 2 for the columns. Now, we apply the function score, and then the function eliminate\_line.

We apply it before the function because the function score counts how many little square you have in the line (or column) to add 10 (or 20) for each little square which will be eliminate. After eliminating, we shift. We process in the same way with column: CorL take the value 2, then apply score and eliminate\_column. Then we compute the total score and reinitialize scoreL and scoreC to 0 to don’t have wrong values. Finally, we display again the board, and part 2 start again : display again blocks, ask you to choose one, then the position …

IV. Remarks and conclusion

1. **Remaks**

We had some difficulties to empty the two dimensional array of the board, in particular with 0 and 1 to create the shape of the gameboard (triangle, diamond or circle). It was our main difficulty with also the function Check\_validity that takes more time. We had to be sure that the check will be efficient and good because our game reply on it. So during long days of calling each other we succeed. Then our other difficulty was the function shift that we changed a lot. We also add different functions that aren’t asked because we thought that adding them will turn our project into something more efficient, secured and simple for our main or for us. Indeed, we decided to add 5 other functions :

1) Choicesize

This function is just a function to choose the size of the game board. Indeed, it was not necessarily to add it but we thought that it was right to add it in our file function in order to don’t add line sin our main. It’s secured according to the shape (the user can’t write an odd number for the triangle or for the diamond), to the interval (it can’t write a number below 21 and above 29) and the user can’t write a letter.

* 1. Fill\_2D\_board

This function was very important and not asked (we saw it in the file that Klais posts in Moodle). What was asked is just the **display\_board** so we thought that it was clearer that we do this function. This function fill the array with number (0= point, 1= space, 2= horizontal line ,3= vertical line,4 and 5 = letters) that we use in the function **display\_board** using ASCII codes (vertical and horizontal lines, letters, little point, little square…)

* 1. Choiceblock

We decided to create this function in order to secure the choice of the block. Thus, if the user is not sure of the block he chose he can go back and chooses another one. Also, we can display the chosen block.

* 1. Choosecoordinate

This other function is for the choice of the coordinate and also to simply our main. In this function the user can put a letter and then this letter will be converted in number. We thought that it was clearer to create a function to convert instead of converting in the main.

* 1. The division of the display\_blocks (display\_block1 and display\_block2)

This is not a new function: we don’t know why but when we had one function for displaying a block, the strategy for all blocks wasn’t displayed. So, we decided to divide this function for the strategy 1 (display all block) and the strategy 2 (display 3 random blocks).

1. **Conclusion**

To conclude, we would like to say that we learned a lot while working on this project: we learned more about pointers, about 2D arrays also about interesting library function such as malloc and free. Even if our project isn’t efficient enough, we managed to compute a real Tetris and that’s very cool. We are proud of what we did and proud of what you are going to see: this is the first time that we finish a project (our pythons projects were not finished) and that’s why we are happy. Indeed, we could do the Optional parts, but we wanted to send you the most perfect Tetris, so we didn’t focus on this part.