A coordinate system for programming hexagonal boards

Alain Brobecker, september 2018 - abrobecker@yahoo.com - http://abrobecker.free.fr/

1. Arrays and square boards

Computer memory is linear, so it's easy to make a one dimensionnal board, you simply use an array which shall be a basic structure of your programming language (if it's not and you're not programming in assembly, please change programming language before resuming your reading). Here's an example in C:

```
int Board1D[7];   //creates an array of 7 integers
Board1D[6]=42;    //indexed from 0 to 6
```

0	1	2	3	4	5	6
?	?	?	?	?	?	42

Some programming languages allow to declare two dimensionnal arrays (or three dimensionnal, or more), and this can ease up programming. But it's a fraud, the computer memory still is linear and the compiler does the simple maths for you. You can see below a simple program using a bidimensionnal array, the result is in the middle, and an equivalent program using a one dimensionnal array in a bidimensionnal fashion is on the right:

```
#define Width 6
#define Height 5
int Board2D[Height][Width];
int x,y;
x=2;
for(y=0;y<Height;y++) {
   Board2D[y][x]=42;
}
```

	0	1	2	3	4	5
0	?	?	42	?	?	?
1	?	?	42	?	?	?
2	?	?	42	?	?	?
3	?	?	42	?	?	?
4	?	?	42	?	?	?

```
#define Width 6
#define Height 5
int Board2D[Width*Height];
int x,y;
x=2;
for(y=0;y<Height;y++) {
   Board2D[y*Width+x]=42;
}
```

You must understand that the compiler converts the first program into the second, and in both cases, the memory really looks like this (please note we used `Board2D[y][x]` to have exactly the same result, but you can usually use `Board2D[x][y]` if you don't care about memory organisation):

0	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
?	?	42	?	?	?	?	?	42	?	?	?	?	?	42	?	?	?	?	?	42	?	?	?	?	?	42	?	?	?

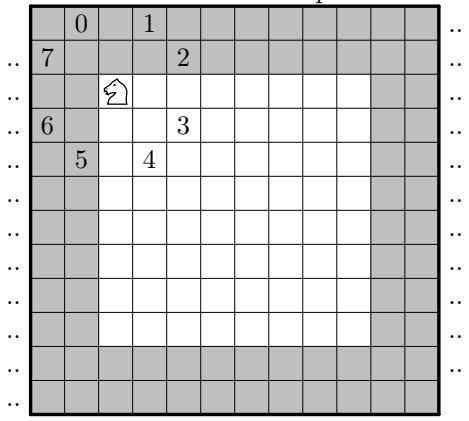
So, why would you want to make the job when the compiler does it? One reason is control, some people like me prefer to be as close to the hardware as possible. I thought another reason would be speed, but you can always do everything as fast in bidimensionnal arrays as in one dimensionnal arrays since they are the same, for example by using pointers or extra functions (memory copying, etc...).

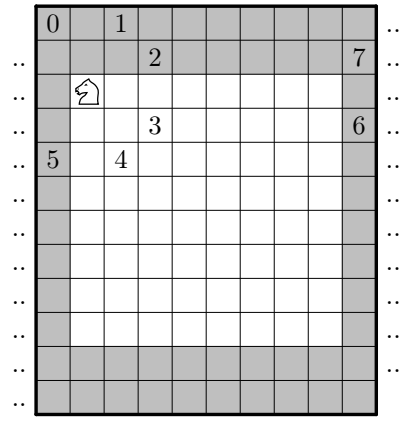
Anyway i give two simple optimisations when using one dimensionnal arrays, first is that you can reset (or copy) the whole board with only one variable: `for(c=0;c<Width*Height;c++) { Board2D[c]=0; }`

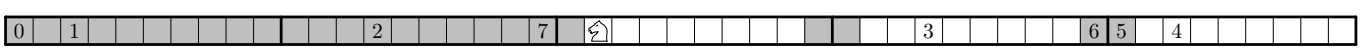
Or you can optimise the code above by setting the position to `x` and increasing it by `Width`, instead of incrementing `y`, multiplying by `Width` and adding `x`: `for(pos=x;pos<Width*Height;pos+=Width) { Board2D[pos]=42; }`

Here's an important hint often used in game programming (chess in the example below). Sometimes you have to make tests to know if a piece goes outside the board, but testing if `x`$\in [0;7]$ and if `y`$\in [0;7]$ is really slow. The hint is to create a greater board, surround the real board with a wall of forbidden squares (in grey). Since the knight can jump over two squares, the wall must be two squares large (left picture). Then instead of testing `x` and `y` values, you can just load the cell at (`x`;`y`) and check if it's a forbidden square.

Another hint: by merging the right+left walls you can have an even smaller array, `x` might be negative but it's normally handled correctly (right picture). The last picture shows the real look of this last array in the linear memory, you can see there are two forbidden squares between two lines:

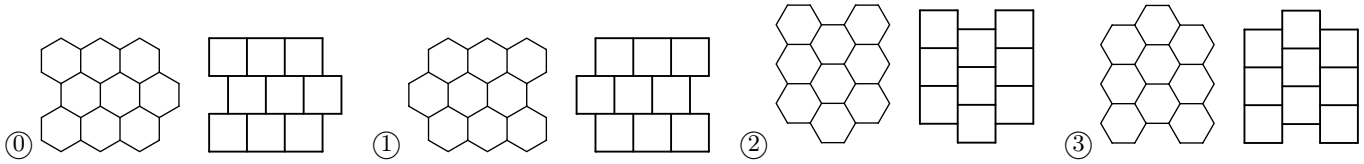






2. Hexagonal boards

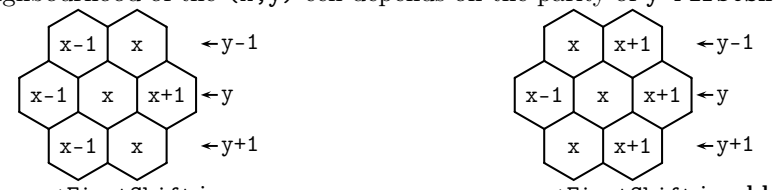
First we shall deal with the organisation of our hexagonal board in memory. Here the basic idea is that an hexagonal board is only a bidimensionnal board with odd/even lines shifted, as the following pictures show:



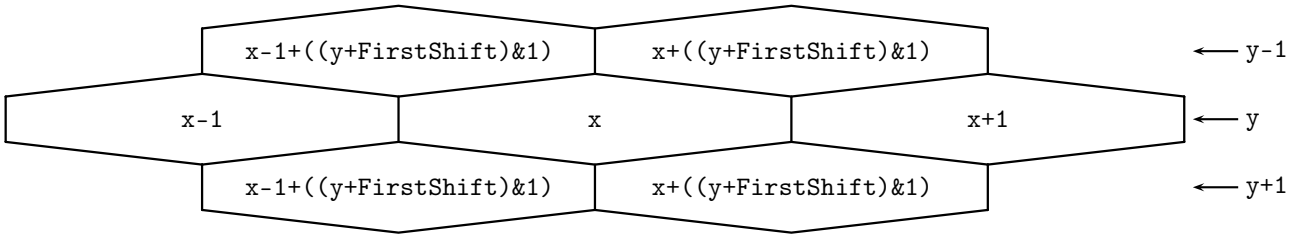
As you can see the shift of the first line is an important characteristic of our board. Thus i will put it in a `FirstShift` variable, say it is 0 in situations ⓪ and say it is 1 in situation ①.

So you have to create a bidimensionnal array, determine what kind of `FirstShift` you want, and you'll also probably need to say that some hexagonals cells are forbidden (or invisible) to have the hexagonal board you want. For example the classical 7 hexagons board can be obtained in (at least) two ways with a 3×3 array:

Then the neighbourhood of the `(x;y)` cell depends on the parity of `y+FirstShift`:

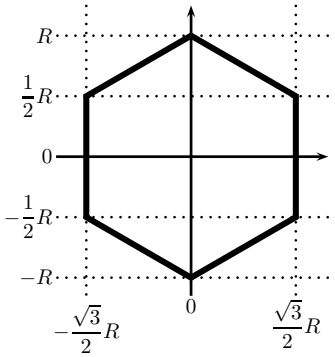


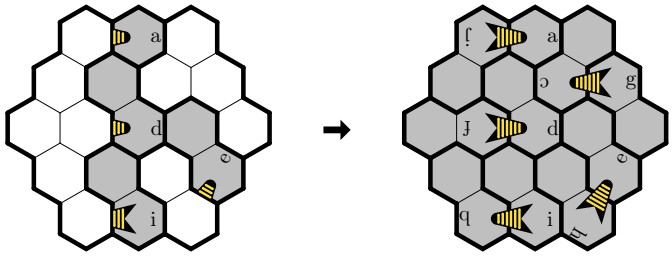
We can merge the two situations by accessing the cells of horizontal coordinates `x-1+((y+FirstShift)&1)` and `x+((y+FirstShift)&)1` on lines `y-1` and `y+1`, where `&1` is the bitwise AND operator (using the modulus operation %2 won't handle negative numbers the way we want). This is summarised in the stretched diagram below:



We also need to know how to draw the hexagons on screen. Considering we want an hexagon of radius R, with the origin on its center, some basic trigonometry gives us the coordinates seen on the right diagram.

Of course you'll have to modify the coordinates according to the `(x;y)` position of the cell in your board, and also take `(y+FirstShift)&1` in account to know if you must shift the hexagon or not (ie add $\frac{\sqrt{3}}{2}R$ to horizontal value).





If your aim is to have the hexagonal board drawn in a pdf file (i needed that to create my puzzle game Bee Logic, see picture on the left), you can alternatively use TEX and my **ffen2tex** preprocessor which can be found at http://abrobecker.free.fr/progs.htm.

Last here are two adresses if you want another, more complete, view of hexagonal boards:
https://redblobgames.com/grids/hexagons/ - https://catlikecoding.com/unity/tutorials/hex-map/