Name: Carlos Carrillo-Calderon
09/28/2015
OSU ID: 932698326

Design description:

As requested, I started writing this document before I started coding on 09/28/2015, so that's why you will see the sentences in future tense.

First design draft:

1) I will create the 40x20 matrix by using a 2D array. "0" will represent the empty cells, and "1" will represent the alive cells. I will fill up the matrix with 0's in order to avoid junk data into the array that could interfere in the programming process.

2) Once the matrix is full of zeros, I will insert the desired pattern (a fixed oscillator, a glider, and a glider cannon) into the array. I will do that by locating the cells as if they were coordinates on a plane. For instance, the glider pattern can be drawn like this:

    - array [0][1] = 1;
    - array [1][2] = 1;
    - array [2][0] = 1;
    - array [2][1] = 1;
    - array [2][2] = 1;

    Since the user will choose the pattern, I will ask them to select the pattern before the whole program executes.

3) Once the pattern has been inserted into the matrix, I will print it out so the user can visualize the first generation/pattern.

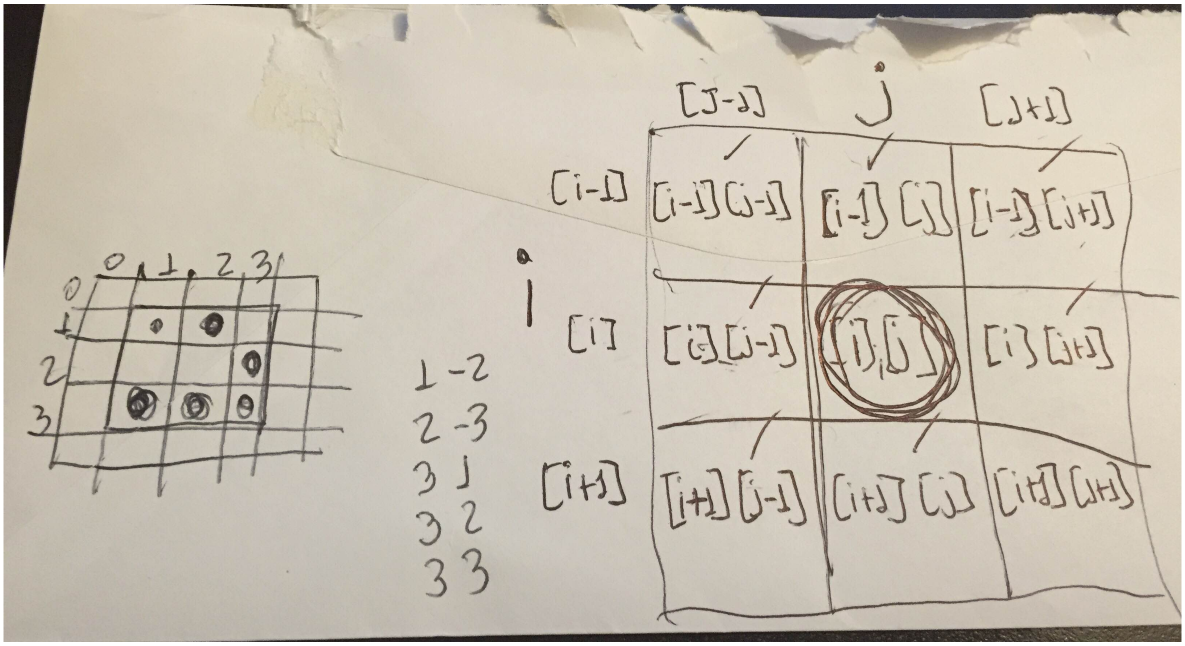
4) After visualizing it, I will copy the whole 2D array into a new array, so the changes I do over it won't affect the original matrix. I will do this because someone in Stack Exchange recommend this procedure:

    "First, let's try speeding everything up by taking advantage of the nature of Conway's Game of Life and the way C++ handles pointers. When you evaluate Conway's Game of Life, it's a two-part process. First you copy all of the entries into a new array, then you do your computation referring to that new array, and update the original one. This process takes O(n2), and involves a major memory copy; for small sizes that shouldn't be a problem, at larger sizes, the cost incurred by the memory copy will become non trivial, so let's reformulate the design a little…"
    (http://codereview.stackexchange.com/questions/47167/conways-game-of-life-in-c)

    Obviously, according to the information I have collected, the easiest way to do this will be by using a nested for loop in this way:

```
for(int i = 0; i < 20; i++)
 {
    for(int j = 0; j < 40; j++)
          array1[i][j] = array2[i][j];
 }
```

5) Once it has been copied, I would apply the rules of the game to every single cell. According to its neighbors, the cell will be deleted or remain the same. The simplest algorithm I have in mind is to check the state of all the 8 neighbors that surround the cell. How? I could put a "1" into the cells that are "alive" in order to identify these alive elements. This is the informal prototype I drew in order to try to design the algorithm:



Later, I saw the same idea in code.runnable.com, but they have used a counter to determine how many neighbors were alive. I felt happy since there was actually an implementation of my vague idea. That would solve a big chunk of the problem. This is the algorithm I found:

```
int count = 0;
count = array[j-1][i] +
array[j-1][i-1] +
array[j][i-1] +
array[j+1][i-1] +
array[j+1][i] +
array[j+1][i+1] +
array[j][i+1] +
array[j-1][i+1];
```

(Source: http://code.runnable.com/UwQvQY99xW5AAAAQ/john-conway-s-game-of-life-for-c%2B%2B-nested-for-loops-and-2-dimensional-arrays)

6) Going back to my design process, I thought that once I had a way to know the current state of each neighbor, I should just ask if the cell/element to be evaluated was dead or alive. Sot I will need to ask to every single cell in the array how many neighbors were alive. As logical, the best way to do this is by using a for loop. Since it is a 2d array, I will need a nested for loop as shown in Rosseta Code, Stack Exchange, Cplusplus and other web forums. The nested for loop can be as simple as:

```
for (int i = 1; i < 31; i++)
  {
     for (int j = 1; j < 51; j++)
       {
```

7) Thus, if the cell is alive, I will use this nested if conditional to determine if it will die or survive:

```
If (array[i][j] == 1)
 {
     if(neighbors < 2 || neighbors > 3)
         array[i][j] = 0;
 }
```

8) Or if the cell is dead, I will also use a nested if conditional to determine if it will die or survive:

```
If (array[i][j] == 0)
 {
    if(count == 3)
       temp[i][j] = 1;
 }
```

But then, I understood that I was doing wrong calculations since I wasn't accounting for all the conditions/rules of the game. And again, the algorithm I found in code.runnable.com showed me a best way to apply the rules without using a nested conditional. All I had to do was transferring the main array into a temporary array and apply the rules on only to the temporary array without affecting the original array. Thus, I modified this part of my code as follows:

```
//The cell dies.
 If(count < 2 || count > 3)
           tempArray[i][j] = 0;

 //The cell stays the same.
 if(count == 2)
            tempArray[i][j] = ruleArray[i][j];

 //The cell either stays alive or a new cell is created
 if(count == 3)
           tempArray[i][j] = 1;
```

(Source: http://code.runnable.com/UwQvQY99xW5AAAAQ/john-conway-s-game-of-life-for-c%2B%2B-nested-for-loops-and-2-dimensional-arrays)

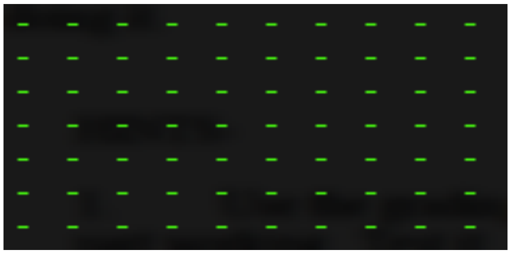
9) Coming back to my original idea, I though that all I had to do after doing the operations in the temporary array was to copy it back to the main array, print it out, and repeat as many times as I wanted. I though: “I will copy the temporary array back to the main array and use another for loop to display the actual 40x20 grid. I will display a “X” or something to represent the 1’s and a period or something to represent the zeros. Something like this:

```
for(int i = 1; i < 21; i++)
  {
      for(int j = 1; j < 41; j++)
       {
         if(gridArray[i][j] == 1)
            cout << 'X' << " ";
        if(gridArray[i][j] == 0)
            cout << '-' << " ";
        }
   }
```

Testing Plan:

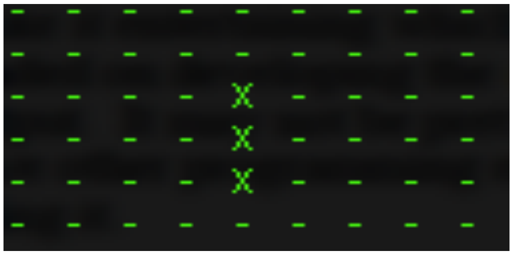
Obviously when I tested my first design, after debugging, it did not work at all. It doesn’t even print the array.

1) So my first goal was to print the grid itself. After changing and trying new methods I could finally print the grid as I wanted



```
- - - - - - - - - -
- - - - - - - - - -
- - - - - - - - - -
- - - - - - - - - -
- - - - - - - - - -
- - - - - - - - - -
- - - - - - - - - -
```

2) Then, my second goal was to print all the patterns. After some time of working on it, I could finally print the fixed oscillator pattern:

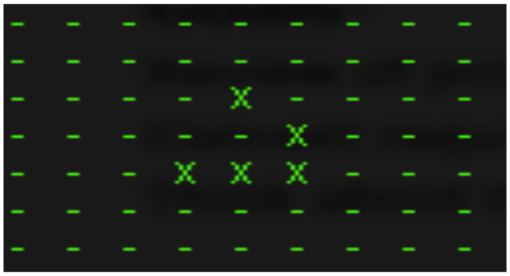


```
- - - - - - - - -
- - - - - - - - -
- - - - X - - - -
- - - - X - - - -
- - - - X - - - -
- - - - - - - - -
```

The code I implemented to created this shape was also based on the array coordinates:

```
patternArray[xCoord1][yCoord1] = 1;
patternArray[xCoord1][yCoord1+1] = 1;
patternArray[xCoord1][yCoord1+2] = 1;
```

3) Then I could print the simple Glide:

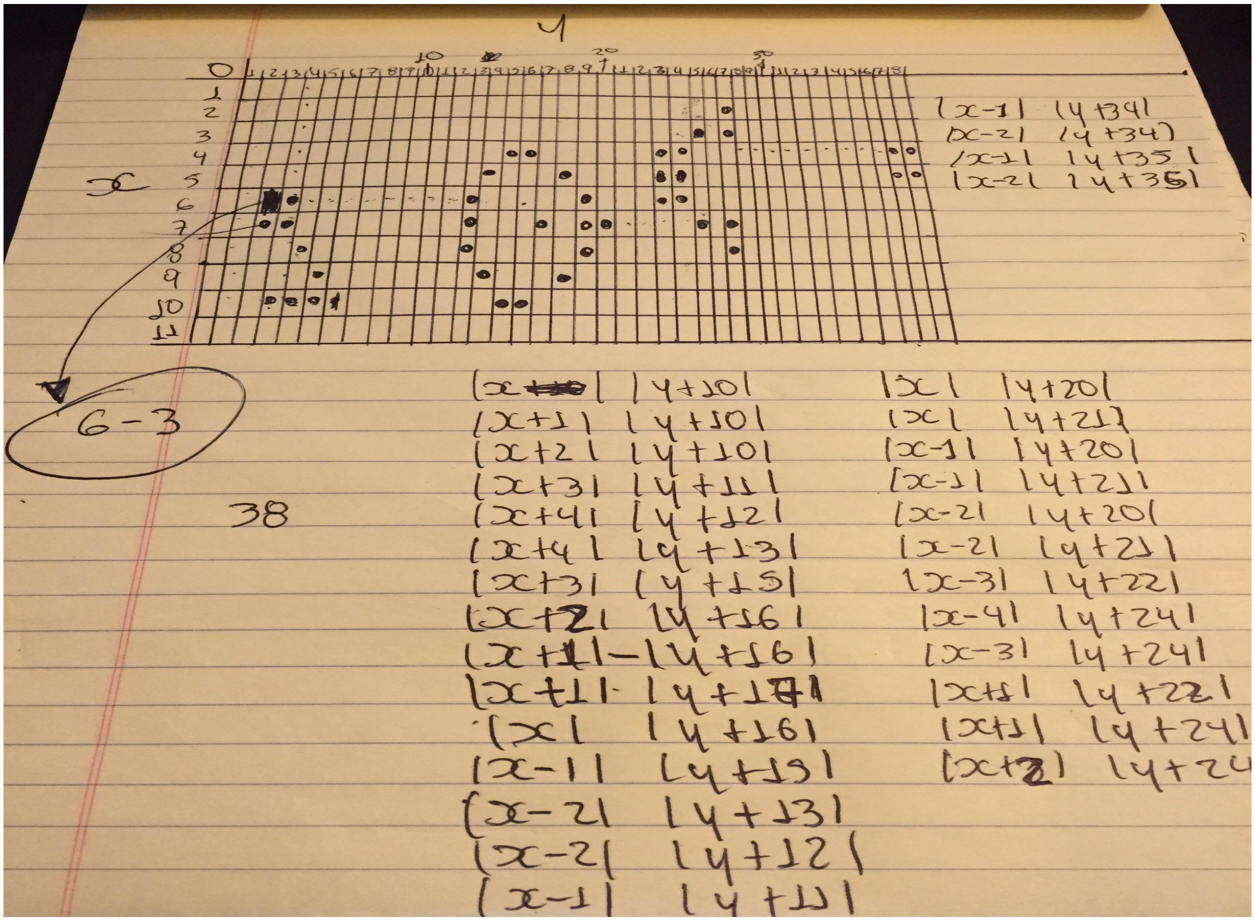


```
-  -  -  -  -  -  -  -  -
-  -  -  -  -  -  -  -  -
-  -  -  -  X  -  -  -  -
-  -  -  -  -  X  -  -  -
-  -  -  X  X  X  -  -  -
-  -  -  -  -  -  -  -  -
-  -  -  -  -  -  -  -  -
```

The code I implemented to created this shape was also based on a coordinates approach:

```
patternArray[xCoord2][yCoord2] = 1;
patternArray[xCoord2+1][yCoord2+1] = 1;
patternArray[xCoord2+2][yCoord2-1] = 1;
patternArray[xCoord2+2][yCoord2] = 1;
patternArray[xCoord2+2][yCoord2+1] = 1;
```

4) It took me some time and some planning but I did it. Here is the sketch I made to create the coordinate arrangement for this shape:

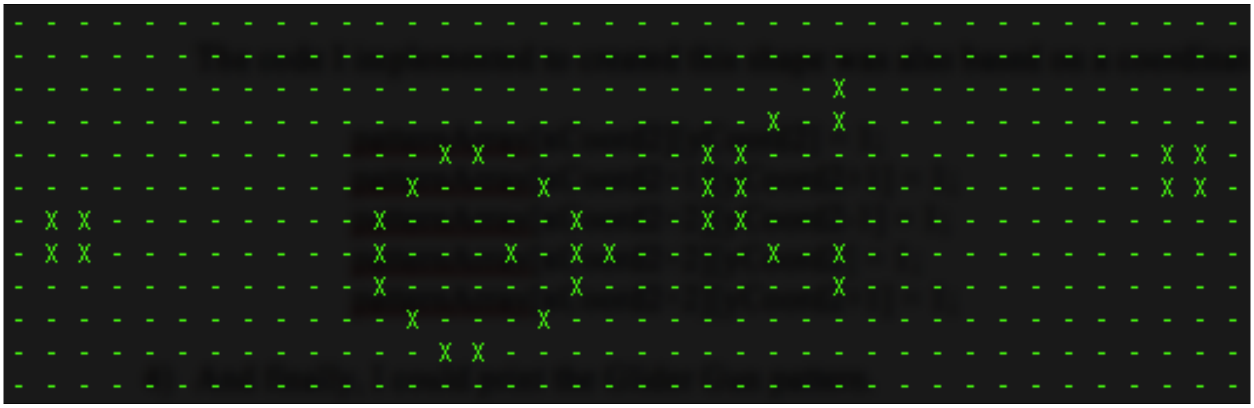


[x-1]	[y+34]
[x-2]	[y+34]
[x-1]	[y+35]
[x-2]	[y+35]

[x]	[y+10]
[x+1]	[y+10]
[x+2]	[y+10]
[x+3]	[y+11]
[x+4]	[y+12]
[x+4]	[y+13]
[x+3]	[y+15]
[x+2]	[y+16]
[x+1] -	[y+16]
[x+1]	[y+17]
[x]	[y+16]
[x-1]	[y+15]
[x-2]	[y+13]
[x-2]	[y+12]
[x-1]	[y+11]

[x]	[y+20]
[x]	[y+21]
[x-1]	[y+20]
[x-1]	[y+21]
[x-2]	[y+20]
[x-2]	[y+21]
[x-3]	[y+22]
[x-4]	[y+24]
[x-3]	[y+24]
[x+1]	[y+22]
[x+1]	[y+24]
[x+2]	[y+24

5) And finally, I could print the Glider Gun pattern:



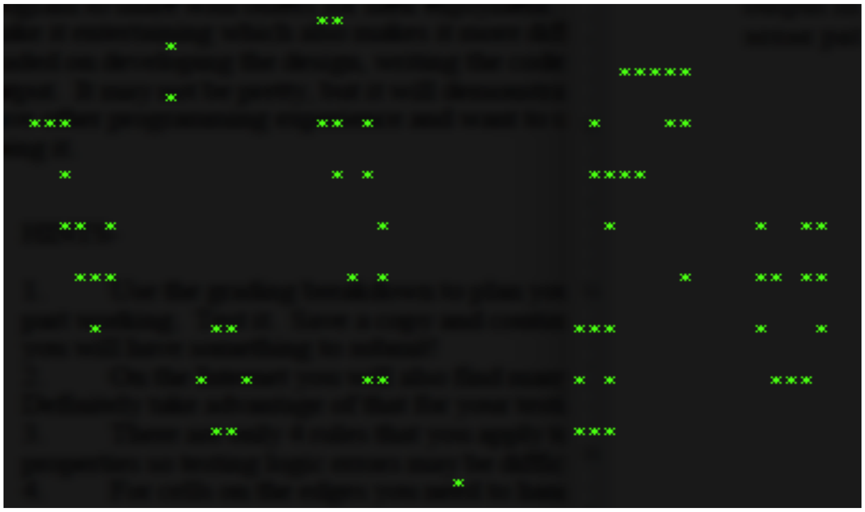
```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - - - - - - X - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - - - - X - X - - - - - - - - - - - -
- - - - - - - - - - - - - X X - - - - - - X X - - - - - - - - - - - - X X -
- - - - - - - - - - - - X - - - X - - - - X X - - - - - - - - - - - - X X -
- X X - - - - - - - - X - - - - - X - - - X X - - - - - - - - - - - - - - -
- X X - - - - - - - - X - - - X - X X - - - - X - X - - - - - - - - - - - -
- - - - - - - - - - - X - - - - - X - - - - - - - X - - - - - - - - - - - -
- - - - - - - - - - - - X - - - X - - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - X X - - - - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

And this is the code I implemented to created this shape (also based on coordinates):

```
//left square
patternArray[xCoord3][yCoord3] = 1;
patternArray[xCoord3][yCoord3+1] = 1;
patternArray[xCoord3+1][yCoord3] = 1;
patternArray[xCoord3+1][yCoord3+1] = 1;

// left cannon
patternArray[xCoord3][yCoord3+10] = 1;
patternArray[xCoord3+1][yCoord3+10] = 1;
patternArray[xCoord3+2][yCoord3+10] = 1;
patternArray[xCoord3+3][yCoord3+11] = 1;
patternArray[xCoord3+4][yCoord3+12] = 1;
patternArray[xCoord3+4][yCoord3+13] = 1;
patternArray[xCoord3+3][yCoord3+15] = 1;
patternArray[xCoord3+2][yCoord3+16] = 1;
patternArray[xCoord3+1][yCoord3+14] = 1;
patternArray[xCoord3+1][yCoord3+16] = 1;
patternArray[xCoord3+1][yCoord3+17] = 1;
patternArray[xCoord3][yCoord3+16] = 1;
patternArray[xCoord3-1][yCoord3+15] = 1;
patternArray[xCoord3-2][yCoord3+13] = 1;
patternArray[xCoord3-2][yCoord3+12] = 1;
patternArray[xCoord3-1][yCoord3+11] = 1;

// right cannon
patternArray[xCoord3][yCoord3+20] = 1;
patternArray[xCoord3][yCoord3+21] = 1;
patternArray[xCoord3-1][yCoord3+20] = 1;
patternArray[xCoord3-1][yCoord3+21] = 1;
patternArray[xCoord3-2][yCoord3+20] = 1;
patternArray[xCoord3-2][yCoord3+21] = 1;
```

```
patternArray[xCoord3-3][yCoord3+22] = 1;
patternArray[xCoord3-4][yCoord3+24] = 1;
patternArray[xCoord3-3][yCoord3+24] = 1;
patternArray[xCoord3+1][yCoord3+22] = 1;
patternArray[xCoord3+1][yCoord3+24] = 1;
patternArray[xCoord3+2][yCoord3+24] = 1;

//right cannon
patternArray[xCoord3-1][yCoord3+34] = 1;
patternArray[xCoord3-2][yCoord3+34] = 1;
patternArray[xCoord3-1][yCoord3+35] = 1;
patternArray[xCoord3-2][yCoord3+35] = 1;
```

But obviously, visualizing the patterns wasn't enough. So I decided to run the program that I was taken as a base in order to see its output and better understand how it was implemented, but surprisingly, it just produced a non-sense pattern that nothing had to do with any known pattern of the Conway's Game of Life itself. It just displayed blinking junk:



```
                     **
           *
                                         *****
           *
  ***                **  *              *    **

    *                 *  *              ****

    **  *                 *              *         *  **

     ***               *  *                   *    **  **

      *       **                      ***          *   *

             *  *       **            * *           ***

              **                      ***

                              *
```

(Source output: http://code.runnable.com/UwQvQY99xW5AAAAQ/john-conway-s-game-of-life-for-c%2B%2B-nested-for-loops-and-2-dimensional-arrays)

Then I learned that the model I was following was wrong and I had to figure out how to create a new model that worked as requested by the assignment. But I know that even though this program was just a piece of junk for my purposes, it had very good bones and concepts. I knew I could create my program out of that skeleton. So I start my testing process.

Although it took a lot of time, I finally understood what I needed to create my program, so those were going to be the elements to be tested:

1) A nested for loop to fill up the pattern array with 0's in order to eliminate/avoid junk into the array.
2) A main menu to prompt the user to choose the shape and the starting location of the

pattern.

3) Insert the chosen pattern into the pattern array by using a coordinates approach.
4) A nested for loop to fill up the main array with 0's in order to eliminate/avoid junk into the array.
5) A nested for loop to transfer the pattern array into the main array.
6) A nested for loop to print the main array with the pattern in it.
7) A temporary array.
8) A nested for loop to transfer the main array into the temporary array.
9) A nested for loop to apply the rules algorithm to each element of the array.
10) A nested for loop to transfer the elements from the temp array to the main array
11) A do loop to make the whole process be repeated a finite amount of times.

After several experimental main functions, I came up with an idea/design. I tried to create this function as original as possible in order to avoid coy right issues. However I properly quoted the source of the algorithm that perform the rules of the game over the array elements (see http://code.runnable.com/UwQvQY99xW5AAAAQ/john-conway-s-game-of-life-for-c%2B%2B-nested-for-loops-and-2-dimensional-arrays).

Test results:

After the program was giving the desires output the next challenge was to test if the starting location entered by the user was accurate. Then I discovered some facts that were inherent to the nature of the program:

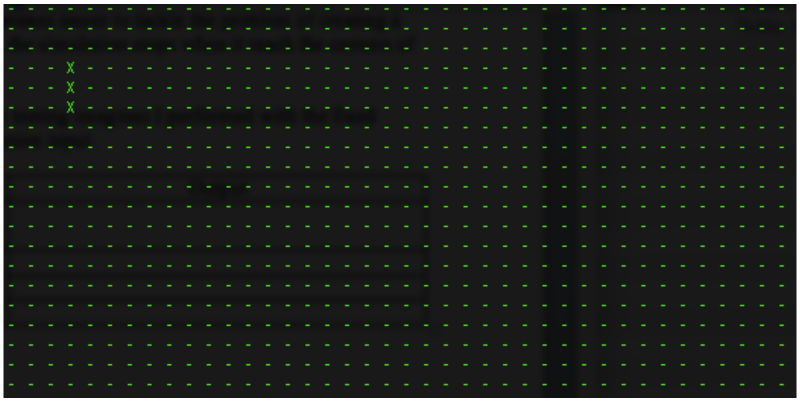
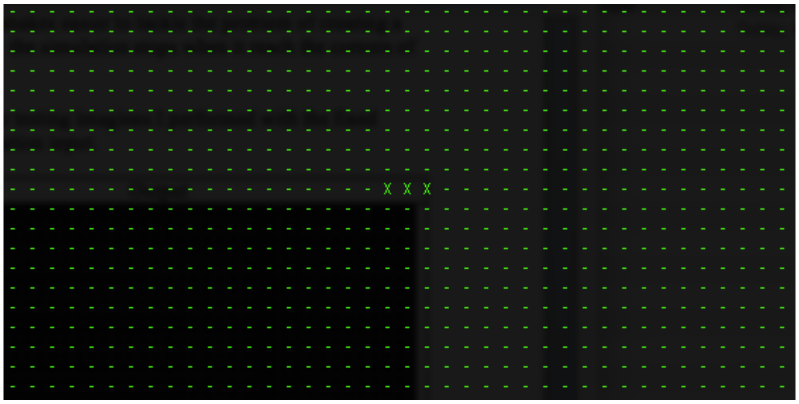
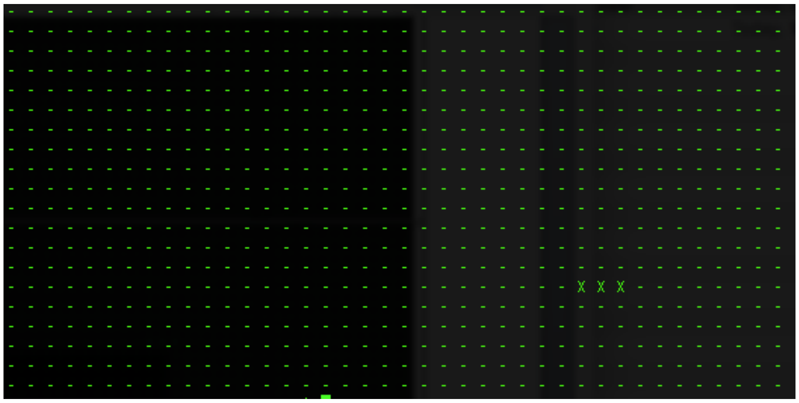
1) For the Fixed Oscillator pattern, the user would have to enter a row coordinate in the range of 2 to 19 and a column coordinate in the range of 1 to 38 in order to see the whole pattern.

2) For the simple Glider shape pattern, the user would have to enter a row coordinate in the range of 1 to 18 and a column coordinate in the range of 2 to 39 in order to see the whole pattern.

3) For the simple Glider Gun pattern, the user would have to enter a row coordinate greater than 5 and a column coordinate fewer than 5 in order to see the whole pattern.

After discovering these restrictions, I had no other option but displaying a message into the program in order to warn the user about these restrictions. Also I realized that there was nothing I could do about it since these issues were related to the ghost rows and column that were not visualized for the user, since although the program displays a 40x20 grid, the real size of the arrays used to develop the program is 52x32. It makes easier to tackle the problem of creating a frame for the actual grid visualized and to avoid the simulation stops when it reach the corners of the grid.

In order to abbreviate, I just going to show some testing imagines I performed with the fixed oscillator pattern with respect to the starting location input:

Input	Output
row coordinate = 5 column coordinate = 6	 - X - X - X -
row coordinate = 10 column coordinate = 20	 - X X X -
row coordinate = 15 column coordinate = 30	 - X X X -