Game of Life Readme

Detail and Design

I kept the original game.c file for my program, which takes in a width and a height to create a game board fill with random values. The main function calls an assembly update function, which is used to change the values based on the rules of Conway's Game of Life.

Update takes in an int** (the 2d array board that needs to be updated), as well as the width and height to use for boundary checking.

It then creates space for an int**, a 2d board array I will refer to as newboard, and allocates space for that board with the same amount of space used to create the original int** board. Then, it loops through the indexes created and fills the newboard with zeros. This creates a 2d array of the same size of the original board filled with zeros to be used to store the new values that board will have after being updated.

Update then creates an int I will refer to as count, and goes into a double loop that iterates through the rows and then the columns. It compares each index of that particular row and column with all 8 indexes surrounding it, checking to make sure it is not reaching out of bounds for those cases. For each of those surrounding indexes, count is incremented each time there is a 1 in that index.

After the count variable has stored all of the 1's surrounding an index, it then checks to see if that index contains a 1. If it does, it checks if count is less than 2, in which case that index's value in the newboard becomes a 0 (dies by under-population). It then checks to see if count is 2 or 3, in which case that index's value in the newboard becomes a 1 (lives on to next generation). Finally, it checks to see if the count is greater than 3, in which case that index's value in the newboard becomes a 0 (dies by overcrowding).

If the index value was not a 1 to begin with, Update checks to ensure that the value at that index is indeed 0, and also checks to see if count is equal to 3. If both these cases are true, it sets the value at that index of the newboard to 1 (alive by reproduction).

Update then changes count to 0 for the next iteration of the loop.

After the loops have ended and the newboard has been filled with the correct values, Update then loops through all of the rows and columns again and replaces the value at that index of the original board to the updated value in the newboard.

Design/Implementation Challenges

Basically all of the problems I encountered with the project were from assembly syntax issues, as it is very time consuming to understand how to program assembly to do simple tasks. It was difficult to iterate through the indexes of a board in Assembly.

Space and Time Performance Analysis

I kept the original game.c file for my program, which simplifies the run time of main (if c is the number of columns and r is the number of rows) to O(c*r). This is due to the fact that main iterates through the 2d array several times in order to allocate memory, fill with random values, etc. Main also calls printboard, which also has a run time of O(c*r), as it iterates through the length and width of the board to print the values.

Update itself has a run time of O(c*r).

Update first loops through the newboard to fill it with zeros, which takes (if c is columns and r is rows) O(c*r). Update then loops through every index of the old board, which has the same amount of rows and columns, so this takes O(c*r). After the newboard is filled in depending on if-statement checks on the indexes of the old board, Update iterates through the indexes of board one last time to replace board's values with newboard's updated values, which again takes O(c*r) run time.

The total run time, since it only takes an O(1) for comparisons and values appointed to registers, is a simplified O(c*r) run time.

The total space performance of the program is O(r*c).
The entire program mallocs enough space for 2 2d arrays, size of c*r (columns times rows).
As an int is 4 bytes, and there are 2 arrays, this results in 8(r*c) space complexity, which simplifies to a big O of O(r*c).