Assignment 4: Game of Life

Aniket Pratap

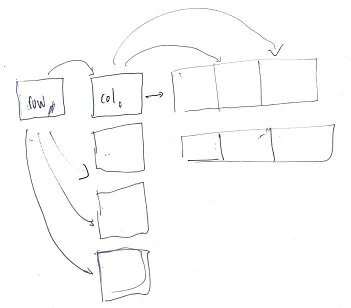
February 2022

1 Introduction

The goal for this assignment is to simulate Conway's game of life. I have to create my own abstract data types in universe.c and implement them in life.c using the header file universe.h. A grid is displayed and each spot on the grid is a cell–which is either dead or alive. In this case, a dead cell is represented by a "." and a live cell is represented by a "o". The user inputs a file and how many generations he/she wants. The program should then animate the generation process unless the user inputs -s–which silences the animation process.

2 Constructors and Destructors

The first function to code was uvCreate(). This function would allocate space to the universe struct first, and then allocates space to the actual grid. Allocating memory to the struct is simple, because you only need to allocate memory to how many bites the struct has. Allocating the grid is a little harder. In short this is what it looks like:



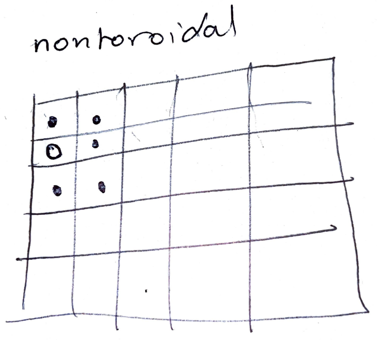
The row is first allocated memory and then the columns and things inside the columns. The allocation to the columns would be the size of uint32's because each "spot" in the array is of size uint32. When columns are being allocated to the rows, the must have a size of a pointer because the row pointer is pointing to the column pointer which is pointing to an array. After the memory is allocated, the variables are set. This can done using the -> symbol to access the variables in the struct and setting them to user input. uvDelete is the same concept except the reversing allocating the memory in uvCreate. The reason it's the reverse is because if we de-allocate the rows first, the bridge connecting us to the columns would be lost and we would not have a way to access the data in the columns. By de-allocating the data inside the columns first, we can ignore the bridge because we are starting at the inner element– Eugene explained this like a water bucket. If you want to get rid of the bucket, you need to empty it out first. uvDelete can be coded using examples from Omar's section–except he did it regarding 1d arrays. I can also set each de-allocation iteration to NULL so that I can avoid accessing an already free element. uvRows and uvCols are simple as they return the return the row and col stored in the struct. uvLiveCell and uvDeadCell are similar in that they set a grid value to either true or false depending on the user file input.

3 uvPopulate

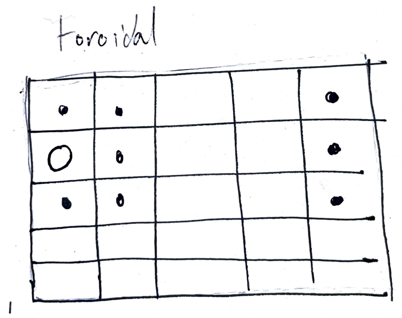
The goal of this function is to scan the numbers after the first line and set them to true if in range. I could scan in the in file, but I would also have to scan in the pattern as well. I notice that the pattern was ("number" "space" "number") so I can use the SCNu32 to get uint32's and apply a single space a newline in order to achieve the pattern. What if the data is out of range? I can just return false, since uvPopulate returns a boolean.

4 uvCensus

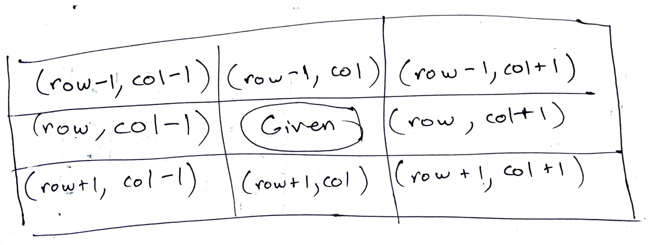
The next function, or I should say functions, counts how many neighbors each cell has. This would also implement the toroidal aspect of the struct. each cell has neighbors–all the cells surrounding it, but what if a cell was near the edge of the grid like:



Then it would have no neighbors off the edge of the grid because...they don't exist. A toroidal universe, however, is a donut shape–meaning the rows and columns wrap around to look something like:



In this case, the neighbors do exist but wrap around to the other side. To start this function, I can first check if the user wants a toroidal or nontoroidal graph. If the user selects toroidal, I can make for loops for the top row, middle and bottom with the formula being:



(row-1, col-1)	(row-1, col)	(row-1, col+1)
(row, col-1)	Given	(row, col+1)
(row+1, col-1)	(row+1, col)	(row+1, col+1)

I made for loops because I noticed that the column in the top row increased by 1, the column increased by 2 in the middle, and column increased by 1 at the bottom. 3 for loop made the most sense compared to 9 if statements. What if the grid is 2 by 2? I have implemented a check where if the neighbors go out of bounds, then don't check. I can implement the same concept with toroidal except rather than increment by 1, I have to do the modulus being $i + n - 1$ mod n for getting the previous element and $i + 1$ mod n for getting the next element where n is the size and i is the index. The reason this works is because n mod n is 0. So if you want to get the last element, you need to add the current index, the modulus will now be i, and subtract by 1 to get the previous element–(i-1). The same concept applies to getting the next element. I can create a previous step, and next step function, similar to what Eugene did in his section, and implement these functions when I want to check the next column.

5 uvPrint

This function is simple because I print an "o" if the value on the grid is true and a "." if the value is false. This is only used when printing the final generation.

6 life.c

The most difficult part about this file was learning how to take in user input and using ncurses. To make the default input stdin, I can create an input file and set it equal to stdin. The same goes for the output file except I set it equal to stdout. I can then take in inputs using switch cases and getopt–similar to previous assignments. I can then use fscanf to scan the file the user inputted–stdin as default–and I can store the first line of the file into the address of rows, and cols. I can then input these rows and cols to create, or initialize, a universe. I can then use uvPopulate to set the values of the grid as true or false when reading the rest of the file. I also create a universe B and a temp universe to swap universe pointers at the end of each generation. This is done so that the A universe doesn't get overridden when it is run. The B universe represents the next generation while A represents the previous generation. If the user inputs the -s, or silence option, then I don't run ncurses and only print the final generation. This uses 3 nested for loops: 1 for the generation, row, and columns. inside the innermost for loop, I do my if statements regarding the criteria for a dead or live cell. If the user doesn't silence ncurses, then I have to use it. Before I do however, I need to add it to add -lncurses to my Makefile so that it can be used. The process of ncurses is the same except I have to initialize the screen and set the cursor equal to false. Once that is done, I not only set something to true or false, but I use the mvpritntw() function to print the corresponding symbol for a live or dead cell. One problem that I previously had with my thought process was continuing if the cell survives. However, Miles pointed out that we don't need to do that, but rather, create a live cell on B. This can allow me to add points I could have been previously missing since I'm continuing on an all false grid and not marking that cell as survived. Finally, I can print out the final grid regardless if the user inputted a -s or not. And then I close and delete the opened files and the created universes respectively.