Assignment 4 Design Document

Nicholas Chu

January 30, 2022

1 Introduction

The purpose of this assignment is to replicate John Conway's The Game of Life. The game takes place on a matrix with each element of that matrix representing a cell. Cells can have two states, alive or dead, and is updated during the passing of a generation based off certain conditions. These conditions are:

- Live cells need 2 or 3 neighbors to survive a generation or they become dead.
- Dead cells with exactly three live neighbors becomes a live cell in the next generation.
- All other cells who do not fit the two conditions above will be set to dead.

For this it is first necessary to create an abstract data structure in order to represent the universe that the game will be on.

2 The Universe

The universe, the grid that the game will take place on, will be implemented as an abstract data structure. The structure representing the universe should have four variables and should look like this:

```
struct Universe{
    uint32_t rows;  //Number of rows of universe
    uint32_t cols;  //Number of columns of universe
    bool **grid;    //2D array of booleans
    bool toroidal;   //If the grid is torroidal or not
}
```

The universe data type should not be manipulated directly, meaning instead the program will need to use functions to construct, deconstruct, access, and manipulate this data type indirectly.

2.1 uv_create

This function is the constructor for the data-type and will create the universe to the size specified by parameters for rows and columns. First it will need to initialize an instance of the Universe structure and allocate the appropriate memory for it by using malloc() with the size of the Universe struct and would look like:

```
//Gives a pointer to memory in the heap to store u
Universe *u = (*Universe)malloc(sizeof(Universe))
```

Then we need to save the parameters into the initialized Universe struct so they can be used and accessed later:

```
u->row = row;
u->col = col;
u->toroidal = toroidal;
```

After saving these values, the function then needs to create the grid that will represent the universe by using the dimensions specified by the rows and columns. The created grid will be saved to **grid and should be a bool. The grid will be represented by a multi-dimensional array, each array within the array representing a row and each element in those arrays representing an element in the column of that row. You first need to allocate the proper memory for the array that will represent the grid using calloc() and using rows to indicate how much arrays, in this case rows, are going to be in the grid. Then inside a for loop iterating over the the arrays in the created multi-dimensional array, using calloc() and columns for size, allocate the size of the row arrays. This loop would look like:

```
for every row:
    grid[r] = (bool *) calloc(cols, sizeof(bool))
```

After the function is done setting up the instance of the universe structure it will then return it.

2.2 uv_delete

This function is a destructor to delete all the memory that had been allocated for creates Universes in order to prevent memory leakage. To do this you need to free each everything that was allocated memory during uv_create. You do this by calling free() with:

- u− >grid
- u

It is important that the function uses free(u) last, as you need to free everything within u before you can free u itself.

2.3 uv_rows

This is an accessor function and all it should do take a Universe structure, u, and return its rows, u− >rows.

2.4 uv_cols

This is an accessor function and all it should do take a Universe structure, u, and return its columns, u− >cols.

2.5 uv_live_cell

This function is a manipulator and meant to set a cell at the position based off the parameters, row r and column c, to a live state. To do this it should set the value at u− >grid[r][c] to true.

2.6 uv_dead_cell

This function is also a manipulator and meant to set a cell at row r and column c to a dead state. To do this it should set the value at u− >grid[r][c] to false.

2.7 uv_populate

This function is supposed to take an input file and use the data in that file to set the states of all the cells in a universe. In order to see the contents of the file, the function will first need open the data file using: File *file = fopen(infile, ”r”). The data file should contain lines with two spaced integers per line. An example of the contents of the input file the function would accept would resemble:

```
20 21
5 10
15 20
```

The first lines get the rows and columns for the universe. Using the example above the rows would be 20 and the columns 21. The function will use the following command to save the numbers of the first line into variables called rows and cols.

```
fscanf(infile, "%’ SCNu32 %" SCNu32 "\n" rows, cols)
```

It should save rows and cols into the universe structure u. The lines following are supposed to represent the rows and columns that should be populated with live cells. The function should have a loop to read all the remaining lines and and take the rows and columns for each line using scanf, then use these as the parameters and call uv_live_cell to set the cell at the specified position to a live one. It will also need to check if the rows and cols that it scanned are viable ones. This would be structured like:

```
while there are lines still in the file:
    fscanf(infile, "%’ SCNu32 %" SCNu32 "\n" rows, cols)

    if specified cell does not exists within the universe:
        return false

    uv_live_cells(u, rows, cols)
```

After finishing with this, it is important to close the opened infile when finished working with it, this would not happen inside the actual function but in the main program.

2.8 uv_census

This function checks the surrounding 8 neighbors of the cell specified by the rows and columns and returns the number of live neighbors. The diagram below has (r,c) representing the cell that is having its neighbors checked.

```
(r-1,c-1)   (r-1,c)   (r-1,c+1)
 (r,c-1)     (r,c)     (r,c+1)
(r+1,c-1)   (r+1,c)   (r+1,c+1)
```

To get the live cells it would check 8 indexes of the cells surrounding (r,c), see if they are true or false, and have it increment a count for live neighbors to be returned at the end of the function. There should be an if statement before checking to index to see if the index exists. If the index does not exist and the structure of the universe has torroidal set to true, then depending on if r-1 or c-1 does not exist then it would use (i+n-1)%n to get the appropriate previous index, where n would be either r or c and n would be the either the total of rows or cols of the universe. This is to simulate the index wrapping around to the start and simulate the torroidal nature of the universe. Similarly if r+1 or c+1 do not actually exist, it would get the appropriate next index by using (i+1)%n instead. Then using these new indexes you would find the state of all eight of the neighbors and the number of them that are live. This would look like:

```
next rows = r + 1
prev rows = r - 1
next cols = c + 1
prev cols = c - 1
live neighbors = 0

if universe is toroidal:
    if next rows does not fit in grid:
        next rows = (r+1) % total rows
    if prev rows does not fit grid:
        prev rows = (r+u.rows-1) % total rows
    if next cols does not fit in grid:
        next cols = (c+1) % total cols
    if if prev cols does not fit in grid:
        prev cols = (c+u.cols-1) % total cols

    \\Counts live neighbors
    if cell at (r-1,c-1) is live:
        live neighbors += 1
    if cell at (r-1,c) is live:
        live neighbors += 1
    if cell at (r-1,c+1) is live:
        live neighbors += 1;
    //and you would continue for the rest of the eight neighbors
}
```

If u is not torroidal you need to take a different approach. You would not adjust the previous or next rows and cols, and instead you would only call uv_get_cell when the next and previous indexes actually exist. Take for example for going over the neighbors in the top row above the cell (r,c):

```
//Finding neighbors on r+1
if universe is not toroidal:
    if prev rows fits:
            if prev cols fits:
                if uv_get_cell(r-1,c-1) is true:
                    live neighbors += 1


        if uv_get_cell(r-1,c) is true:
            live neighbors += 1

        if next cols fits:
            if uv_get_cell(r,next_cols) is true
                live neighbors += 1
```

2.9 uv_print

This function will print a representation of the grid onto an output file. To do this there will be a for loop that iterates through each cell of the grid, checking if the cells are alive or dead and prints 'o' if the cell is live and '.' is dead. This would resemble:

```
for every row r:
    for every col c:
        if uv_get_cell(r,c) is true:
            print O to outfile
        else:
            print . to outfile
    print an next line once row has been filled
}
```

3 Life

The file life.c will be the main program and uses the functions and struct defined universe.c to simulate the Game of Life. The command line options for this program are:

- -t: Specifies if the universe will be torroidal. This will be used for uv_create and creating the universe struct where it will set bool **torroidal to true. Otherwise this will be false.
- -s: Makes it so that ncurses will not display anything.
- -n: Sets the number of generations that the universe will go over throughout the course of the game. By default it will be set to 100.

- -i input: Sets the input file that will be read and used in uv_populate in order to set the live cells. The default input would be stdin.
- -o output: Sets the output file which will have the final universe printed into it by using uv_print. The default output would be stdout.

These command line options should be stored in a set for better usage similar to Assignment 3. First the program will create 2 universe structures, one will be universe A and the other will be universe B. If -t is specified then the parameter for torodial in uv_create will be true, if not it will be left false. You can think of universe B as a kind of temp universe. A is the reference and B will hold the results of the next generation, then A will be set to B. This is because we can not instantly just update all the cells in the universe at once, instead we need to loop over each element within the universe and individually update each cell with each passing generation. If we just use only one universe the cells will not be updated properly, because when a cell is updated, it would compromise the following cells from being updated properly. Using uv_populate, populate one of these universes, specifically A, which will be used as the recent universe. Then it will need to loop in order to simulate generations passing. The steps for looping through each generation are:

- If -s is not specified as an option, the program will need to display the game and each generation passing, and will need to use the ncurses.h library. This will create a display in the terminal which will be used for displaying the game. This part would look very similar to uv_print in which it loops over every cell in the grid for universe A checking if its true or false and printing 'o' or '.' respectively, using mvprintw() instead of fprintf(). The screen then needs to be refreshed and then slept for 50000 microseconds using usleep(50000).
- To update the universe with the passing of each generation, you need to use a loop in which uv_census is called for each cell in A. If the cell is a live cell then the corresponding cell in universe B is set to true if uv_census returns 3 or 2 or set to false if it returns anything else. If the cell is a dead cell then the corresponding cell in B will be set to a live one when uv_census returns exactly three, or will be set to dead.

```
neighbors = uv_census(A, r, c)
if uv_get_cell(A, r, c) is true and number of live neighbors are 3 or 2:
    uv_live_cell(B, r, c)
else:
    uv_dead_cell(B, r, c)

if uv_get_cell(A, r, c) is false and number of live neighbors is 3:
    uv_live_cell(B, r, c);
else:
    uv_dead_cell(B, r, c);
```

- Swap universe A with B, *A = *B, as A needs to be changed to represent the most recent generation.

After this just use endwin() to stop the display and output the results of universe A using uv_print to the output file specified by the command line. Then use uv_delete the free the memory that had been allocated for universes A and B.