# Assignment 4 Design

Arnav Nepal

February 13, 2023

## 1 Introduction

In this assignment we will be implementing John Horton Conway's *Game of Life*. The Game of Life (or simply "Life") is a zero player game where successive stages of the game's universe are determined purely by the initial state and nothing more. Life can be played on an infinite grid, however due to the limitations of computers we will be implementing it in a finite universe. The files we will be writing are:

- universe.c
- life.c
- Makefile
- DESIGN.pdf
- README.md
- WRITEUP.pdf

## 2 Design and Psuedocode

This assignment requires us to write just 2 (albeit complex) C source files: universe.c and life.c. The first, universe.c, will contain functions for the Universe *Abstract Data Type* (ADT), while the second, life.c, will contain the main() function to run the game within the universe and other helper functions.

### 2.1 universe.c

```
#include any relevant files

//this typedef allows us to call just "Universe" instead of
// "struct Universe" to initialize the universe struct
typedef struct Universe Universe;

#the ADT reference:
typedef struct Universe {
    uint32_t rows //num of rows
    uint32_t cols //num of cols
    bool **grid
    bool torodial
}

//We use calloc here since calloc initializes each grid to 0 or false
//which means the grid starts off with all dead cells (as is required)
Universe *uv_create(uint32_t rows, uint32_t cols, bool toroidal);
    calloc memory for Universe uv: calloc(4, sizeof(Universe *));
    uv->rows = rows
    uv->cols = cols
    bool **grid = (bool **)calloc(rows, sizeof(bool *))
    for (uint32_t i = 0; i < rows; i++){
        grid[i] = (bool *) calloc(cols, sizeof(bool))
    }
    uv->toroidal = toroidal
    uv->grid = grid //sets the pointer of uv->grid to grid array
    return grid //pointer to uv struct

void uv_delete(Universe *u);
    loop( int i = 0; i < u->rows; i++):
        free(u->grid[i])
        set u->grid[i] to NULL
    free(u->grid)
    set u->grid to NULL
    free (u)
    set pointer to u to NULL

uint32_t uv_rows(Universe *u);
    return u->rows

uint32_t uv_cols(Universe *u);
    return u->cols

void uv_live_cell(Universe *u, uint32_t r, uint32_t c);
    if r < u->rows AND c < u->cols:
        u->grid[r][c] = True
    return

void uv_dead_cell(Universe *u, uint32_t r, uint32_t c);
    if r < u->rows AND c < u->cols:
        u->grid[r][c] = False
    return

//same boundary condtions as live and dead cell, so we can reuse
//the conditions
bool uv_get_cell(Universe *u, uint32_t r, uint32_t c);
    if r >= 0 AND c >= 0 AND r < u->rows AND c < u->cols:
        return u->grid[r][c]
    else:
        return false

//fscanf returns number of read inputs, so returning 0 either means end of -
//file or error. I use uv_get_cell right after uv_live_cell to check if that -
//cell was properly populated. if live_cell works, get_cell on that same cell -
//will always return true

bool uv_populate(Universe *u, FILE *infile);
    int x, y
    //fscanf returns 0 when it cannot read anything more from input
    while (fscanf(infile, "%d %d", &x, &y) != 0):
        uv_live_cell(u, x, y)
        if (uv_get_cell(u, x, y) == false):
            return false
    return true;
```

*Changes from the initial design* : I removed the checks for whether row and column are less than 0 since they are unsigned integers (and cannot be less than 0, so it's a redundant check). I also edited the uv_delete() to loop over u->rows instead of u->cols. The reason I initially used u->cols is because I mistakenly assumed it meant the number of columns. It means the *length* of the column. Thus I changed it to u->rows since it means the length of the row, which is equivalent to the number of columns.

uv_census is one of the more complicated functions, requiring us to calculate the number of adjacent neighbors of each cell. This is not too complicated, and can be represented as:

| | | |
|---|---|---|
| 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 |

Where *r* represents the row and *c* the column. However, our actual implementation is a bit more complicated because the calculation needs to take place on different planes depending on whether or not the Universe is toroidal. If the universe is toridal, the universe wraps around on itself, meaning a cell on the edge, in col(max), will have to check col(0) for neighbors. This is not *too* complicated; it just requires us to caculate the index *modulo* len(array).

```
uint32_t uv_census(Universe *u, uint32_t r, uint32_t c);
    int num_neighbors = 0; //will hold number of neighbors
    if (u->toroidal):
        for (i = -1, i <= 1; i++):
            for (j = -1, j <= 1; j++):
                if (j==0 AND i==0):
                    continue

                if (r == 0 && i == -1):
                    row_index = u->rows - 1
                else:
                    row_index = (r + i)%u->rows

                if (c == 0 && j == -1):
                    col_index = u->cols - 1
                else:
                    col_index = (c + j)%u->cols
                if(uv_get_cell(u, row_index, col_index):
                    num_neighbors++
    else:
        for (i = -1, i <= 1; i++):
            for (j = -1, j <= 1; j++):
                if (j==0 AND i==0):
                    continue
                if(uv_get_cell(u, r + i, c + j)):
                    num_neighbors++
    return num_neighbors


void uv_print(Universe *u, FILE *outfile);
    for (int r = 0; r < u->rows; r++):
        for (int c = 0; c < u->cols; c++):
            if(uv_get_cell(u, r, c):
                fputc(111, outfile) //111 is ASCII for lowercase o
            else:
                fputc(46, outfile) //46 is period char
        fputc(10, outfile) //newline char
```

*Changes from the initial design*: I updated uv_print() to print a newline at the end of of every row. On the other hand, uv_census required significant changes. To avoid cluttering my file, I chose to use nested loops to implement the census. I also ran into problem with just using “(r-1)%u->rows” to calculate the row to check, and had to create a case for the census of numbers in the 0th row and columns.

### 2.2 life.c

This file will contain the main() function. It will contain more logic than main() in other assignments due to the way this game is set up.

```
#include relevant files
#include universe.h

int main(argc, argv):
    use getopt with switch statements to parse command line options:
        open files to read and write to
    handle errors if something fails (or -h used as option)
    scan the number of rows and cols from input file
    create universe A and B with above row and col conditions
    populate universe with points from the input file

    if ncurses enabled:
        initialize ncurses

    for (loop for number of generations specified in options or default):
        if (ncurses enabled):
            print the grid to the ncurses window using mvprintw()
            for this part, follow the formatting of uv_print from universe.c
            refresh the screen after all elements printed
            usleep(specified delay)
        for loop (int r, r < uv_rows(A), r++):
            for loop (int c, c < uv_cols(A), c++):
                //folow the rules of the game of life for each cell
                if uv_get_cell(A, r, c): //checks if cell alive
                    if uv_census results are 2 or 3:
                        uv_live_cell(B, r, c)
                    else:
                        uv_dead_cell(B, r, c)
                else:
                    if uv_census results are 3:
                        uv_live_cell(B, r, c)
                    else:
                        uv_dead_cell(B, r, c)
        swap universe A and B (swap their pointers)
    if (ncurses enabled):
        end ncurses window
    open output file
    print final grid to file with uv_print(A)
    close output file
```

```
        free all allocated memory by deleting universe A & B with uv_delete()
```