Assignment 4 Design Document

Program Requirements

Our program must run the game of life simulation, given a starting universe.

The Game of Life is a routine where a potentially infinite, 2-D universe begins with a grid of cells, which are either alive or dead. Over the course of multiple generations (iterations), cells come to life or die in accordance with the following rules.

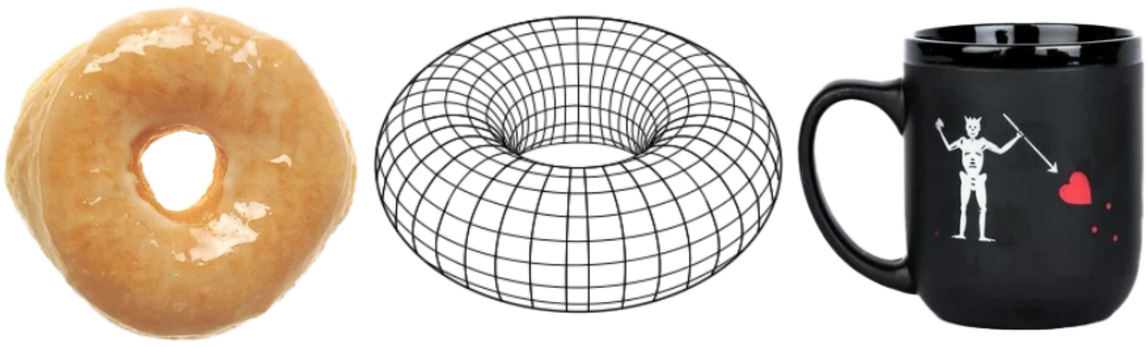
1. Any live cell with two or three live neighbors survives.
2. Any dead cell with exactly three live neighbors becomes a live cell.
3. All other cells die or remain dead.

The cells to be live for the first generation are determined by the input file, which specifies the number of rows and columns the 2-D universe must contain, as well as the initial placement of live cells in that universe.

The user has the following command-line options:

- Specify whether or not the universe is toroidal.
- Select whether to display or hide the state of the universe with each generation.
- Cap the total number of generations (default: 100).
- Provide the input file to create the starting grid.
- Specify which output file to print the final state of the universe to (default: stdout).

To further clarify the concept of a toroidal universe, the following are examples of spaces that are topologically equivalent to a torus:



The commentary describing the pseudocode in all ensuing sections are in blue, while the pseudocode itself is in black.

life.c Pseudocode

Include universe.h as well as the following header files:

- stdlib
- inttypes
- unistd

- ncurses

1. Play the game per the given rules for the desired number of generations. In each generation, live cells with 2 or 3 neighbors survive, dead cells with 3 neighbors come alive, while all others die. We start with two identical universes, A and B. We count the number of neighbors of each cell in universe A, and update universe B per the rules of the game based on that information. If the silent mode is off we display the evolution of the universe in each generation. At the end of each generation, uA and uB are swapped.

Input parameters:

- num_generations: uint32_t: Number of generations
- uA: Universe *: Universe A
- uB: Universe *: Universe B
- silent: bool: Determines if evolution is to be displayed or not

Returns: void

```
void play_game() {
    Universe *tmp;
    // Take a census of each cell in universe A, as below.
    For each row in universe A…
        For each col in each row…
            neighbor_count = uv_census();
            // Cells with 3 neighbors and live cells with two neighbors live.
            If neighbor_count is equal to 3:
                Mark the cell in universe B as live.
            Else, if neighbor_count is equal to 2, and the cell is live:
                Mark the cell in universe B as live.
            Else:
                Mark the cell in universe B as dead.
    Set or clear the corresponding cell in universe B, based on the 3 rules above.
    // Swap the universes as follows:
    tmp = uA;
    uA = uB;
    uB = tmp;
}
```

2. If the silent mode is off, display the evolution by showing live cells of each generation for 50000 ms.

Input Parameters:

- uA: Universe *: Universe to be displayed

Returns: void

```
void show_evolution() {
        Clear the ncurses screen.
        Display universe A.
        Refresh the screen.
        Sleep for 50,000 microseconds.
}
```

3. Create a function to print the usage message.

Here is the usage function.

```
void usage(char *exec_name) {
    Print a usage message.
    return;
}
```

4. Close open files and free the memory allocated.

Input Parameters:

- infile: char *: Name of the input file
- ifp: FILE *: File pointer to the input file opened for reading
- uA: Universe *: Pointer to universe A
- uB: Universe *: Pointer to universe B

Returns: void

```
void cleanup(char *infile, FILE *ifp, Universe *uA, Universe *uB) {
    Close the input file.
    Free the allocated space.
    return;
}
```

5. Build the main function as follows.

Here is the main function.

```
int main(int argc, char **argv) {
    Declare and initialize the variables.
    // We need to parse the input options and store the sorting options in a set.
    Make calls to getopt() to get the parameters.
    Make calls to fscanf on the input file to get the number of rows and columns.
    Create two universes, A and B.
    // If there is no correct entry in the input file…
    Populate universe A with data in the remaining lines of the input file.
    Set up the ncurses screen.

    For each generation…
        play_game();

    endwin();
    Output universe A using uv_print().
}
```

universe.c Pseudocode

Include universe.h and stdlib.h.

1. rows and cols specify the dimensions of our finite universe. A value of false indicates that a cell is dead, while true means it's alive. A true value for toroidal means our universe is toroidal, but flat otherwise.

```
struct Universe {
    uint32_t rows;
    uint32_t cols;
    bool **grid;
    bool toroidal;
};
```

2. Initialize the universe by allocating memory for the universe, and all the rows and columns, and setting the toroidal boolean variable.

Input Parameters:

- rows: uint32_t: Number of rows
- cols: uint32_t: Number of columns
- toroidal: Boolean: Specifies the shape of the universe

Returns: Pointer to the universe

```
Universe *uv_create(uint32_t rows, uint32_t cols, bool toroidal) {
    Universe *u;

    Allocate memory for Universe and grid **


    For r between 0 and rows…

        // The memory size of each row will be the number of columns * size(bool).

        Allocate memory for grid[r].


    Initialize, u->rows, u->cols, and u->toroidal.

    return(u);

}
```

3. The destructor method frees all the memory allocated.

Input Parameters:

- u: Pointer to the universe

Returns: void

```
void uv_delete(Universe *u) {

    Free memory for each row.


    Free memory for grid ** and u.

    return;

}
```

4. Return the number of rows in the universe.

Input Parameters:

- u: Universe *: Universe

Returns: Number of rows

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

5. Return the number of cols in the universe.

Input Parameters:

- u: Universe *: Universe

Returns: Number of cols

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

6. Mark the cell as live.

Input Parameters:

- u: Universe *: Universe
- r: uint32_t: Row number

- c: uint32_t: Column number

Returns: void

```
void uv_live_cell(Universe *u, uint32_t r, uint32_t c) {
    Set the cell value to true.
    return;
}
```

7. Mark the cell as dead.

Input Parameters:

- u: Universe *: Universe
- r: uint32_t: Row number
- c: uint32_t: Column number

Returns: void

```
void uv_dead_cell(Universe *u, uint32_t r, uint32_t c) {
    Set the cell value to false.
    return;
}
```

8. Return the status of the cell. If the input is out of bounds, return false.

Input Parameters:

- u: Universe *: Universe

- r: uint32_t: Row number
- c: uint32_t: Column number

Returns: true if the cell is live, false otherwise

```
bool uv_get_cell(Universe *u, uint32_t r, uint32_t c) {
    If r and c are in range:
        return(u->grid[r][c]);
    return(false);
}
```

9. Populate the universe based on the information in the input file. The first row of the file that contains the number of rows and columns has already been read, and the universe has already been created. The remaining rows of the file contain row and column indices of one live cell per line.

Input Parameters:

- u: Universe *: Universe to populate
- infile: FILE *: File containing information on live cells

Returns: false if an index is out of bound, true otherwise.

```
bool uv_populate(Universe *u, FILE *infile) {
    For lines 2-n in the input file…
        Read the rows and columns using fscanf in variables r and c.
        If r and c are out of bounds:
            Print an error message and return false.
```

```
                Set the cell value to true.


        return(true);
}
```

10. Count and return the number of live neighbors.

Input Parameters:

- u: Universe *: Universe
- r: uint32_t: Row index
- c: uint32_t: Column index

Returns: uint32_t: Number of live neighbors

```
uint32_t uv_census(Universe *u, uint32_t r, uint32_t c) {
        uint32_t num_live_neighbors = 0;
```

When r/c are 0 or u->row, the cell lies on the boundary of the grid. If the universe is toroidal, we need to “wrap” to find all the neighbors. If the universe is flat however, we need to ignore the neighbors that will fall off the grid. In other cases, we want to consider the 8 neighbors around the cell. Moreover, since we are using uint32_t, we can't compare it to -1. Further if we subtract 1 from 0, we will get 0xffffffff. So, we’re setting the range for determining neighbors here. Toroidal universes are handled as a special case when determining neighbors.

```
    If (r - 1), (r + 1), (c - 1), or (c + 1) are out of bounds:
        Handle it using modular arithmetic.
    If we repeat the above process for columns, the four corner elements will be counted
    twice. Therefore, we add checks for them.
    For row from (r - 1) to (r + 1)...
        For col from (c - 1) to (c + 1)...
            If the cell is live:
                ++num_live_neighbors;
    return(num_live_neighbors);
}
```

11. Print the universe. Print an "o" for a live cell, and a "." for a dead cell.

Input Parameters:

- u: Universe *: Universe to print
- outfile: FILE *: File to print the output to

Returns: void

```
void uv_print(Universe *u, FILE *outfile) {
    For row from 0 to (u->rows - 1):
        For col from 0 to (u->cols - 1):
            If the cell is dead:
                Print "." to outfile.
            Else:
                Print “o” to outfile.

        Print “\n” to outfile.
}
```