Design Document - The Game of Life

Lev Teytelman

January 29, 2022

1 Program Description

This program simulates the Game of Life for a certain number of generations before printing the output to the standard output or a file.

2 Included Files

- universe.c - The source file containing the implementation of the universe struct and its related functions.
- universe.h - The header file with the struct typedef and function declarations for inclusion in other files.
- life.c - The file containing the main function, along with the logic for flag arguments and displaying the steps with `ncurses`.
- Makefile - The logic to preprocess, compile, and link the program files with one command.
- README.md - A short description of the program and its abilities and features.
- DESIGN.pdf - This document, showcasing ideas and code structure.

3 Structure

This program will take command-line flags and arguments (with none mandatory). These flags will allow the user to set whether the universe is toroidal, whether the generations run silently without printing, the number of generations (default 100), and the input/output for the program. After parsing the flags, the program will create two Universe structs using the first two numbers in the input as the width and height for both. It will then populate the live cells on the field by converting pairs of numbers into coordinates for the 2D array grid and setting those cells to be alive. After setting up the universe, the program will run the specified number of generations, either printing each iteration with `ncurses` and adding a delay, or running the program ”silently” - without showing each generation and without a delay. Finally, it will convert the universe to text and output it to the standard output or save it to a file, and free the memory used for the universe.

4 Pseudocode

Managing memory with these more complex data structures will be a challenge; every aspect of the struct will need to have memory allocated for it, and therefore freed. I will need to make sure that I `malloc` or `calloc` all of the memory needed before using it, and ensure that I `free` it afterwards, similarly to the example on the assignment document:

```
// allocate (rows) pointers
grid = alloc(rows)
// allocate (columns) pointers for each row
for row in range(rows):
    grid[row] = alloc(cols)
}
Universe u = alloc(Universe)
// use universe here
for row in range(rows):
    free(grid[row])
free(grid)
free(u)
```

One of the more interesting challenges for this assignment is ensuring coordinates are properly normalized for toroidal universes, as all integers between $[-1, rows]$ are valid for the row index, and all integers between $[-1, columns]$ are valid for the column index. Therefore, since the rows and columns are stored in 32-bit unsigned integers, the range can only be stored with 64-bit signed integers. The function to normalize the coordinates will need to take `int64_t`s and will look something like this, where `a` is the number to normalize and `b` is the upper value of the range to normalize ($[0, b)$):

```
def normalize(int64_t a, int64_t b) {
    return (a + b) % b;
}
```

Since negative numbers do not become positive when the modulus is taken, this will have to be used. When using this function to check neighbors on the board, we will also need to incorporate the predicate of the board being toroidal before we normalize:

```
if (!toroidal && !verify(u, row, col)) {
    return;
}
```

This will ensure the census function won't consider out-of-bounds coordinates if the universe does not wrap around (only checking for out-of-bounds when the universe is not toroidal). After ensuring that only valid spots are being checked, we can finally check the cell itself with the normalized coordinates using

```
    uv_get_cell(u, normalize(row, u_rows), normalize(col, u_cols));
```

the result of which will then be used to increment some sort of counter. For the game itself, the program will need to have a function called `advance` in which the conditions of the game are checked, looking something like this:

```
def advance(Universe u):
    for (r in uv_rows(u)):
        for (c in uv_cols(u)):
            alive = uv_get_cell(u, r, c)
            neighbors = uv_census(u, r, c)
            if (alive && (neighbors == 2 || neighbors == 3)):
                uv_live_cell(u, r, c)
            else if (!alive && neighbors == 2):
                uv_live_cell(u, r, c)
            else:
                uv_dead_cell(u, r, c)
```

I decided to split the conditions for cells that become alive into two separate if statements, as having one was getting too long and convoluted.