Krisha Sharma

CSE13s

Winter 2022 Long

01/28/2022

Assignment 4 DESIGN.pdf

Description of Program:

The main task in Assignment 4 is to implement the Game of Life in C programming. The Game of Life, also known as Conway's Game of Life was developed by John Horton Conway in 1970. It is a zero-player game, meaning that its evolution is determined by its initial state, requiring no further input.

Files to be included in directory "asgn4":

- `universe.c`
    - Implements the `Universe` ADT.
- `universe.h`
    - Specifies the interface to the `Universe` ADT.
    - This file is provided and *may not* be modified.
- `life.c`
    - Contains the `main()` and *may* contain any other functions necessary to complete the implementation of the Game of Life.
- Makefile
    - `CC = clang` must be specified
    - `CFLAGS = -Wall -Wextra -Werror -Wpedantic` must be specified

    - `make` must build the `life` executable, as it should `make all` and `make life`.
    - `make format` should format all the source code, including the header files
- README.md
    - Text file in Markdown format that describes how to build and run the program, how the program handles erroneous inputs, and any problems encountered while developing the program.
- DESIGN.pdf
    - Describes design for the program thoroughly with pseudocode and visualizations.
- WRITEUP.pdf
    - No WRITEUP is required for this assignment.

General Notes:

- Assignment 4: The Game of Life (also known as Conway's Game of Life) asks us to implement the Game of Life in C programming.
- Understanding the game:
    - The Game of Life should be played on a potentially *infinite* two-dimensional grid of cells that represents a universe.
    - Each cell has two possible states: whether dead or alive.
    - The game progresses through generations, but it can be looked at as "steps in time"
    - There are three main rules for the game:
        - Any *live* cell with two or three live neighbors *survives*.
        - Ant *dead* cell with exactly three live neighbors *becomes a live cell*.

        - All other cells die, either due to loneliness or overcrowding,
- In order to create the universe that this game is played in, an abstraction must be created
- ADT, also known as an *abstract data type*, is something that we will write that will provide the abstraction for a universe, a finite 2-D grid of cells.
    - The gird cannot be infinite because computers work in *finite memory*
    - I will need to create multiple functions that are required for my ADT
        - I need to create the constructor, destructor, accessor, and manipulator functions
        - The universe.h header file is given to us. for the Universe ADT.
- The universe will be abstracted as a `struct` called `Universe`.
    - Use a `typedef` to construct a new type and treat it as opaque, so pretend that you cannot manipulate it directly.
- `Universe.h` declares the new type and `universe.c` defines its concrete implementation
- The `Universe` must contain the following fields: `rows`, `cols`, and a 2-D boolean grid.
        - Since there are 2 states, *alive* and *dead,* then the best choice for representing these states is a bool value.
            - A cell with a `false` in the gird indicates that the cell is dead, likewise, if a cell has the value `true`, that indicates that the cell is alive
- The `Universe` will contain the following functions:
    - `uv_create`: this is the constructor function that creates a Universe.

        - The first two parameters it accepts are `rows` and `cols`, indicating the dimensions of the underlying boolean grid.
        - The last parameter toroidal is a boolean, if the value of `toroidal` is `true`, then the universe is *toroidal*
        - The *return type* of this function is going to be of type `Universe *`, meaning the function should return a pointer to a `Universe`.
        - We will use the `calloc()` function from `<stdlib.h>` to dynamically allocate memory.
    - `uv_delete`: this is the destructor function for the `Universe`
        - This function frees any memory allocated for a `Universe` by the constructor function.
        - This function makes sure that there are no memory leaks.
        - Use `valgrind` to check for memory leaks.
    - `uv_rows`: this function is an accessor function and returns the number of rows in the specified `Universe`.
        - This is possible, but only inside `universe.c`
    - `uv_cols`: this function is an accessor function and returns the number of columns in the specified `Universe`.
    - `uv_live_cell`: this function is a manipulator function and it marks the cell at row `r` and column `c` as *live*.
        - If the specified row and column lie outside the bounds of the universe, nothing changes.

        - Since we are using *bool*, we assume that *true* means live and *false* means *dead*.
    - `uv_dead_cell`: this function marks the cell at row `r` and column `c` as *dead*.
        - Like in `uv_live_cell()`, if the row and column are out-of-bounds, nothing changed.
    - `uv_get_cell`: this function returns the value of the cell at row `r` and column `c`.
        - If the row and column are out-of-bounds, `false` is returned
        - Again, *true* means that the cell is alive.
    - `uv_populate`: this function will populate the Universe with row-column pairs read in from `infile`
        - This function will require `<stdio.h>` since infile is a `FILE *` (`FILE` is defined in the `<stdio.h>` library)
        - The necessary *include* will be supplied in the `universe.h` for use.
    - `uv_census`: this function will return the number of live neighbors adjacent to the cell at row `r` and column `c`.
        - ideally the universe of the game extends to infinity in the $\pm\ x$ and $\pm\ y$ directions; however, we cant have things falling off the edge of our universe.
        - We need to treat our universe (the flat grid) as a flat Earth or a *torus*.
        - If the universe is flat, or non-trodoial, then you should only conside the *valid* neighbors for the count.

        - If the universe is toroidal then you should consider all the neighbors as valid, you simply wrap to the other side.
    - `uv_print`: this function prints out to the `outfile`.
        - A live cell is indicated with the character ‘o’ (a lowercase O) and a dead cell is represented with a character ‘.’ (a period).
        - We will need to use either `fprintf()` or `fputc()` to print out to the specified `outfile`.
        - You cannot print a torus, so you will always be printing out the flattened universe.

Universe pseudo-code:

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

uv_create pusedocode:

```
Make a pointer to allocate memory big enough for one
universe.
Allocate a column of pointers to rows.
Allocate the actual rows by column
Set and update the rows
Set and update the grids
```

```
        Update the toroidal bool
        Update the 2d array grid to equal malloc of u pointer
        Return the universe
```

uv_delete pusedocode:

```
    free what is inside the universe
```

uv_rows pusedocode:

```
    return u->rows;
```

uv_cols pusedocode:

```
    return u->cols;
```

uv_live_cell pusedocode:

```
    if r >= 0 and r < uv_rows(u)
        if c >=0 and c < uv_cols(u)
            u->grid[r][c]= true, mark the cell at row r and
            column c as alive
```

uv_dead_cell pusedocode:

```
    if r >= 0 and r < uv_rows(u)
        if c >=0 and c < uv_cols(u)
            u->grid[r][c]= false, mark the cell at row r and
            column c as dead
```

uv_get_cell pusedocode:

```
    if r < 0 or c < 0 or r >= uv_rows(u) or c >= uv_cols(u)
        return false
    return u->grid[r][c]
```

uv_populate pusedocode:

```
inatilize rows
inatilize cols
while infile has two arguments, rows and cols
     if rows <= uv_rows(u) and cols <= uv_cols(u)
          uv_live_cell(u, rows, cols)
     else
          return false
```

uv_census pusedocode:

```
initialize a live neighbor counter
inatilize a dead neighbor counter
inatialize the increment at 1
for i = 1, i < 8 (the number of possible neighbors), i++
     if the universe does not wrap around
          while the increment is on the first neighbor (i = 1)
               change row and column value as needed
               increase the increment by 1
               if the current cell is dead
                    increase the dead cell count
                    increase the increment by one and move on to
                    the next neighbor
               if the currect cell is alive
                    increase the live neighbor count
```

```
                    increase the increment by one and move on to
                    the next neighbor
          while the increment is on the second neighbor (i = 2)
               (repeat everything again until i = 8 and all the
               neighbors are checked)
          return n;
```

generations pusedocode:

```
generations for loop pusedocode Audrey Ostrom
three for loops both universes are the same size, A is
populated, B starts empty
     for (int i = 0; ___ ; i++) {
          for (rows) {
               for (cols) {
                    take a census; feed through uv_census
                    check the results and see how many neighbors
                    its has
                    mvprintw(ROW, col, "o"); // where this goes
                    depends on whether the cell is marked as
                    live or dead
                    make the cells in univese B dead or alive
                    based on the neighbors
          refresh();
          usleep(DELAY);
```

Error Handling:

- I ran into an error while trying to make the toroidal part of my `uv_census` function. I was using while loops to increment through the neighbors; however, I forgot to actually increase the initial increment, and thus it was stuck in an infinite loop.
- I ran into a segmentation fault core dump error while finishing up my `uv_census`. The error was in my `uv_create` and it was because I while I inatilized grid in the Universe struct I forgot to actually pass it through u and thus I ran into an error. Brain pointed this out when I attended his section on 02/02/2022 and helped me fix my error.
- While trying to test my `uv_census` function, i ran into another problem where my census function was returning 0 as the live neighbor count, even though it should have been returning 2.
    - In order to fix this I implemented test cases in my census function where I created a dead cell counter and made an if statement to check if the cell was dead. If the cell was dead then it would increase the dead cell counter, print an test statement, and move on to the next cell. I did this for three of the cells and then had my census function return the dead cell count rather than the live cell neighbor count, and found out that it returned 3.
    - I then knew that the problem wasn't in my `uv_census` function or my `uv_get_cell` function, but rather most likely in my `uv_populate`.

Citations:

- Professor Long is cited throughout this assignment for the following:
    - Pusedocode in the assignment 4 PDF document

    - Tomai (Elmer) on discord in the CSE13s - Winter 2022 - Professor Long server
- Throughout this assignment, I am doing high-level pseudocode collaboration with my sister Twisha Sharma (tvsharma). We are bouncing ideas off of each other and generally talked out the best ways to go about implementing the Game of Life.
    - We also specifically had high-level pseudocode collaboration in `uv_census` after watching Eugene's section video and learning how to do toroidal with helper functions such as `uv_prev` and `uv_next`.
    - Twisha and I also had high level discussions and pusedocode collaboration when we started ncurses and the generations for loop. We both attended Audrey Ostrom's tutoring session on 02/02/2022 and during that time Audrey provided pusedocode for the generations loop
- I watched the video recording of Eugene's section on 1/28/2022. The section was focused on Assignment 4 and during it, Eugene covered many important topics that were relevant to the assignment. I implemented my toroidal function similar to the way he did during his section.
    - Eugene is specifically cited for my `uv_prev` function and `uv_next` function which is based on the ones he gave as an example during the section on 1/28/2022
- I attended Ben's section on 2/1/2022 and during it, he helped me understand how uv_populate and uv_census work in terms of the game as well as how `fscanf()` takes in arguments.
- I attended Brian's tutoring session on 02/02/2022 and during it he helped me debug my segmentation core dump error. He had me run valgrind and add `-g` in my `Makefile` under `(CFLAGS)` that way valgrid would specify what line the errors were on.

- I attended Omar's office hours on 02/03/2022 for help debugging my segmentation fault (core dumped) error. We endedup finding out that the error was because when I was initially attempting to swap universes, I was doing so in a way similar to parallel assignment, where I created a third empty universe that was later on deleted causing the memory leak. Omar corrected this by letting me know that I only had to create a temporary universe rather than an empty universe.