Assignment 4 The Game of Life

Design Document

Derrick Ko - Winter 2023

1 Overview

Creating Conway's Game of Life

2 universe.c

2.1 Universe *uv_create()

This is the constructor function that creates a Universe. The first two parameters it accepts are the number of 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 of type Universe *, meaning the function should return a pointer to a Universe. You will be using the calloc() function from <stdlib.h> to dynamically allocate memory. For more information on calloc(), read man calloc.

```
uint32_t **matrix = (uint32_t**) calloc(rows, sizeof(uint32_t*));
for (uint32_t r = 0; r < rows; r++) {
  matrix[r] = (uint32_t*) calloc(cols, sizeof(uint32_t));
}
```

Matrix

- allocate memory to the structure
- allocate memory to each row of the structure

2.2 void uv_delete(Universe *u)

This is the destructor function for a Universe. Simply put, it frees any memory allocated for a Universe by the constructor function. Unlike Java and Python, C is not garbage collected. Not freeing allocated memory by the end a program results in a memory leak. Your programs should be free of memory leaks. In the case of multilevel data structures such as a Universe, wemust free the inside first. Imagine getting rid of a water bucket: you should dump out the water before scrapping the bucket. Use valgrind to check formemory leaks – the graders will too!

- free the grid

2.3 uint32_t uv_rows(Universe *u)

Since we will be using typedef to create opaque data types, we need functions to access fields of a data type. These functions are called accessor functions. An opaque data type means that users do not need to know its implementation outside of the implementation itself. This means that it is incorrect to have u->rows outside of universe.c as it violates opacity. This function returns the number of rows in the specified Universe (it is possible, but only inside universe.c).

2.4 uint32_t uv_cols(Universe *u)

Like uv_rows(), this function is an accessor function and returns the number of columns in the specified Universe.

2.5 void uv_live_cell()

The need for manipulator functions follows the rationale behind the need for accessor functions: we need some way to alter fields of a data type. This function simply 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.

- set the cell to be alive at that location which is true

2.6 void 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 is changed.

- set the cell to be dead at that location which is true

2.7 bool 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 the cell is live.

- look at the location of the cell and see if it is alive or dead and return true or false

2.8 bool uv_populate()

The first line of a valid input file for your program consists of the number of rows followed by the number of columns of the universe. Each subsequent line is the row and column of a live cell. You will want a loop that makes calls to fscanf() to read in all the row-column pairs in order to populate the universe. If a pair lies beyond the dimensions of the universe, and return false to indicate that the universe failed to be populated. Return true if the universe is successfully populated. Failure to populate the universe should cause the game to fail with an error message.

- look at each line of the infile to see if it has 2 elements
- if it doesn't make that location alive.
- if there is out of bound error return false.

2.9 uint32_t uv_census()

This function returns the number of live neighbors adjacent to the cell at row r and column c. If the universe is flat, or non-toroidal, then you should only consider the valid neighbors for the count. If the universe is toroidal, then all neighbors are valid: you simply wrap to the other side. Tip: you should calculate the rowand column for each neighbor and apply modular arithmetic if the universe is toroidal.

- loop through all the neighboring cells
- calculate the row and column indices by wrapping around the grid
- if the current location of cell is alive increase count

2.10 void uv_print()

This functions prints out the universe to outfile. A live cell is indicated with the character 'o' (a lowercase O) and a dead cell is indicated with the character '.' (a period). You will want to use either fputc() or fprintf() to print to the specified outfile. Since you cannot print a torus, you will always print out the flattened universe.

3 life.c

3.1 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, either due to loneliness or overcrowding.

3.2 Command-line Options

- -t : Specify that the Game of Life is to be played on a toroidal universe.
- -s : Silence ncurses. Enabling this option means that nothing should be displayed by ncurses.
- -n generations : Specify the number of generations that the universe goes through. The default number of generations is 100.
- -i input : Specify the input file to read in order to populate the universe. By default the input should be stdin.
- -o output : Specify the output file to print the final state of the universe to. By default the output should be stdout.

3.3 Specific step by step

1. Parse the command-line options by looping calls to getopt(). This should be similar to what you did in previous assignments.

2. Use an initial call to fscanf() to read the number of rows and columns of the universe you will be populating from the specified input.

3. Create two universes using the dimensions that were obtained using fscanf(). Mark the universes toroidal if the -t option was specified. We will refer to these universes as universe A and universe B.

4. Populate universe A using uv populate() with the remainder of the input.

5. Setup the *ncurses* screen, as show by the example in §4.

6. For each generation up to the set number of generations:

   (a) If ncurses isn't silenced by the -s option, clear the screen, display universe A, refresh the screen, then sleep for 50000 microseconds.

   (b) Perform one generation. This means taking a census of each cell in universe A and either setting or clearing the corresponding cell in universe B, based off the 3 rules discussed in §2.

   (c) Swap the universes. Think of universe A as the current state of the universe and universe B as the next state of the universe. To update the universe then, we simply have to swap A and B. Hint: swapping pointers is much like swapping integers.

7. Close the screen with endwin().

8. Output universe A to the specified file using uv print(). This is what you will be graded on. We will know if you properly evolved your universe for the set number of generations by comparing your output to that of the supplied program.