CSE 13S: Assignment 4 Design Doc

Johnny Li

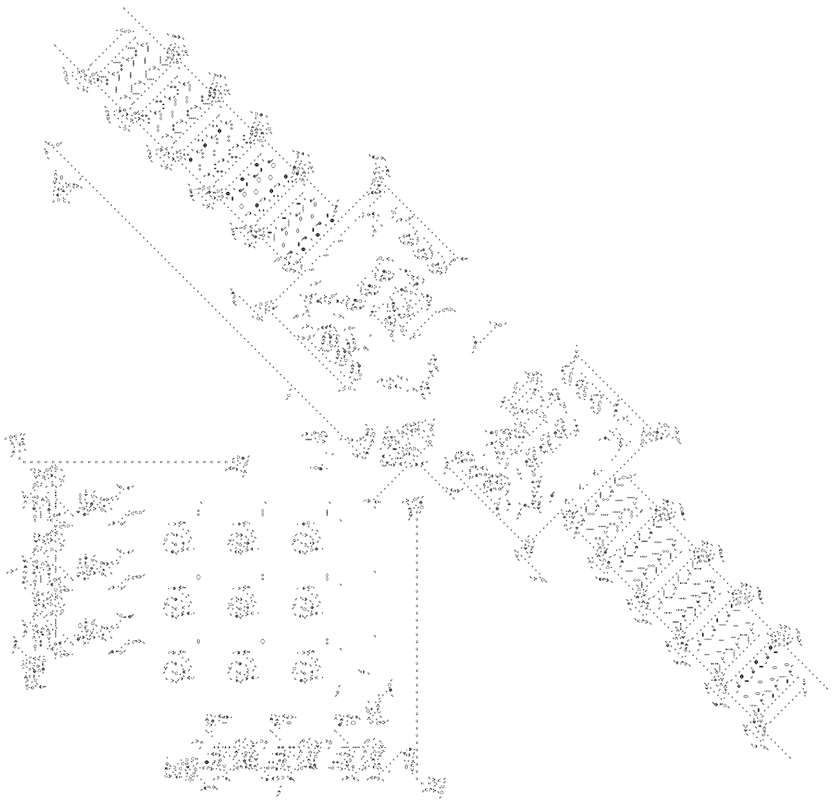
February 12, 2023

1 Introduction

John Conway was a prominent British mathematician who was active in the theory of finite groups, knot theory, number theory, combinatorial game theory, and coding theory. He also made many contributions to recreational mathematics where he made his most notable one lay: The Game of Life. Conway first conceived this idea in 1970 to observe how something so complex such as life can evolve from a simple initial state. The concept builds upon ideas first put forth by the famous Hungarian Physicist and Mathematician John von Neumann. Conway's game involves a two-dimensional grid in which each square cell interacts with its neighbors according to a simple set of 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.

Over successive generations, these simple rules – in conjunction with the initial state – can give rise to surprisingly complex states. Since The Game of Life is a zero-player game, one needs only to interact with the game to set its initial state; once the game begins, it requires no further input and the player can simply observe how it evolves through generations. Interestingly enough, Conway's Game of Life is also Turing Complete since a Universal Turing Machine can be built from a pattern that acts like a finite-state machine connected to two counters.



Example of a Turing Machine implemented in The Game of Life

Our goal was to implement Conway's Game of Life with a minimum of two files – universe.c and life.c. The former containing various functions which is called from a main function implemented in life.c; however, like the previous assignment, life.c can include various functions not directly mentioned in the asgn4.pdf which we deem necessary to the function of our program. Additionally, we'll be constructing our first abstract data type (ADT) in this assignment with an abstract called universe which is described in the universe.h file. The various functions in universe.c are described in further detail in the Universe section directly below.

2 Universe

An abstract data type is a mathematical model for data type defined by its behavior from the user's view. Our ADT, named Universe, abstracts a finite, 2-dimensional grid. With the functions below, we'll be implementing various constructor, destructor, accessor, and manipulator functions – whose headers are derived from the provided universe.h file. For our purposes, universe.h declares the new type and universe.c defines its concrete implementation although the former shouldn't be modified. Furthermore, a type def is be used to construct a new type which is treated as opaque; meaning we shouldn't manipulate it directly although we technically can since C doesn't enforce opacity.

An instance of a Universe must contain the following fields: rows, cols, and a 2-D Boolean grid – grid. Since all squares in the grid can only have two values, dead or alive, their state is be represented by a Boolean with false for dead and true for alive. The Universe can also be toroidal in which case our 2-dimensional grid simulates a 3-dimensional shape wrapping around itself. So a square going past the top of the grid wraps around and appear to the bottom and vice versa.

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

The constructor function that creates Universe that accepts three parameters. The first two parameters are the grid's row and columns (rows and cols respectively) which indicate the dimensions of the grid while the third parameter is a Boolean to indicate if the universe is toroidal (true for toroidal and false for non-toroidal). The function should return a pointer to a Universe. Additionally, calloc is used in this function to dynamically allocate the appropriate amount of memory for a Universe. To do this, a column of pointers are allocated to rows, and then the actual rows are allocated themselves and vice versa for the columns as well.

```
void uv_delete(Universe *u)
```

The destructor function that deallocates or frees memory which has been allocated for a Universe by the previous constructor function. Unlike other languages like Java or Python, C is not garbage collected and unlike C++, there is no built-in destructor function so we have to manually implement our destructor function. Not freeing the allotted memory causes a memory leak which is something we intially experienced in Assignment 3 but is a major component of this assignment since we're working significantly more with memory. There must be an order to how memory is freed, especially with multilevel data structures such as a Universe. The contents inside an array must be freed first before the array itself can be; otherwise, we wouldn't be able to access the contents anymore to free memory which would cause memory leaks.

```
uint32_t uv_rows(Universe *u)
```

Since type is used to create opaque data types, accessor functions are needed to be created in order to access those data fields. An opaque data type means that users do not need to know its implementation outside of the implementation itself; somewhat akin to the idea behind Object-Oriented Programming (OOP) in which data is hidden to the user. For example, having "u-¿rows" outside of universe.c would violate opacity. uv_rows is an accessor function that returns the number of rows in the specified Universe.

```
uint32_t uv_cols(Universe *u)
```

Just like the uv_rows function above, uv_cols is an accessor function that returns the number of columns in the specified Universe; most likely through similar methods.

```
void uv_live_cell(Universe *u, uint32_t r, uint32_t c)
```

The last function type after constructor, destructor, and accessor: a manipulator function allows for the manipulation or alteration of the fields of data types. The uv_live_cell 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. As stated previously, a Boolean is used to represent a cell's state with true for alive and false for dead.

```
void uv_dead_cell(Universe *u, uint32_t r, uint32_t c)
```

The adverse of the previous uv_live_cell function, uv_dead_cell marks a 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.

```
bool uv_get_cell(Universe *u, uint32_t r, uint32_t c)
```

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.

```
bool uv_populate(Universe *u, FILE *infile)
```

This function populates a Universe with row-column pairs read from infile. The first line of the file contains the width and height of the Universe, and the following lines contain row-column pairs of live cells. A loop iterates through the entire infile, reading each row-column pair using the fscanf() function. If a pair does not occupy a valid location in the Universe, uv_populate returns false, causing the game to fail and print an error message. If the Universe is successfully populated, uv_populate returns true.

```
uint32_t uv_census(Universe *u, uint32_t r, uint32_t c)
```

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 only the valid neighbors for the count are considered. Otherwise, if the universe is toroidal, then all neighbors are valid, including those on the opposite end since the Universe would wrap around. To do this, we plan to calculate the row and column for each neighbor and apply modular arithmetic if the universe is toroidal.

```
void uv_print(Universe *u, FILE *outfile)
```

This function simply prints out the Universe to a file named ”outfile”. A live cell is indicated with the character 'o' (lowercase O) and a dead cell is indicated with the character '.' (period). This is accomplished through the use of functions fputc() or fprintf() to read and write to the file. Since we cannot print a torus, a flattened universe is always be printed instead.

3 Life

The life.c file contains the main function from which everything is called. Like Assignment 3, life.c can also contain various other functions not mentioned in asgn4.pdf which are necessary to the functionality of our program. Although for my implementation, it doesn't have any functions other than the main function although I did entertain the idea throughout the process.

3.1 Command-Line Options

When run, life.c accepts the following 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.

4 Sources

- The C Programming Language
- Several functions in universe.c were provided by TA John Yu and subsequently used in their original form or altered to various extents.
- Various Stack Overflow posts were used to help debug and/or to gain inspiration.
- Various manual pages were used to used to help debug and/or to gain inspiration.