Table of content

Project details

- **Project Title**: Game of life
- **Student Name**: Harith Zaid Al-Safi
- **Student ID:** 201416467

Project Design

The project will aim to simulate Conway's game of life using the SDL library. According to Izhikevich (2015) Conway's game of life is an example of cellular automation such that it shows a n infinite two dimensional grid with cells having two states either alive or dead. The project will visualise a 2-D representation of cellular automation through the game of life. The game is shown as a grid with alive and dead cells. The cells will be moving across the grid based on Conway's set of rules. The program will have the following directory structure :

```
├── CMakeLists.txt
├── egypt-1.10
│   └── ..
├── includes
│   ├── graphic.h
│   ├── logic.h
│   ├── misc.h
│   └── main.h
├── out
│   ├── final.txt
│   └── initial.txt
├── README.md
├── scripts
│   ├── build.sh
│   ├── egypt.sh
│   └── SDL.sh
├── src
│   ├── graphic.c
│   ├── logic.c
│   ├── main.c
│   └── misc.c
├── tests
│   ├── test_graphic.c
│   ├── test_misc.c
│   └── test_logic.c
└── unity
    └── ..
```

The program will take the initial configuration from `initial.txt` as it will have the following format:

```
number_of_rows number_of_columns rate_of_simulation pixels_per_cell
```

When the simulation starts, the user will be able to `mouse left click` on dead cells to assign them as alive, `mouse right click` on alive cells to assign them as dead. The simulation will start after pressing the `Enter` key, pauses when pressing `space` and restarts when pressing `r` . After some iterations, the final state of the world will be shown in a file called `final.txt` as it will have the following output:

```
alive_cells dead_cells number_of_rows number_of_columns
shape_of_the_wolrd
```

More details on how to run the program, use the bash scripts in `scripts/` , other keyboard shortcuts and the configuration files in `src/` will be mentioned in `README.md`. As the bash scripts will be around installing SDL and compiling the program in addition to the unit testing.

The program will have some limitation as it will have some restriction such that the world is finite and any cell on the edge of the world is considered dead. Not to mention that since using `C` programming language is not very friendly with external libraries compared to `C++`. The key modules of the project are:

Module	Description
`graphic.c`	contains graphic functions that will run the game
`logic.c`	contains logical functions that sets the game rules and functionality
`main.c`	contains the main loop for running game
`misc.c`	contains functions that use both `logic.c` and `graphic.c` functions

Test plan

Testing the program will go along two steps, using unity for test programs and also visually assessing the behaviour of the program. Each module will have its own test plan such that they are compiled with the `cmake` list.

The tests will mainly revolve around integer assertions, Boolean assertions and changing the program state . The program state is a variable called `game_state` such that its an enumerated data type containing the different states that the program can be in. Functions containing `void` aim to change the `extern` variables of the program as a way of declaring that they are doing the required functionality. Each test will simulate a mini version of the original program such as declaring certain variables and testing the changes that occur to them. Those tests will contain the most significant functions however functions are subjected to change and so are the tests.

- The following is the test plan for all the modules
  - **Logic** (`test_logic.c`)
    - `init load_file()`
      - **Return**: return value of `a`
      - **Expected behaviour** :
        - `a == 1`
        - Loads everything from a file into `row`, `col`, `rate` , `pixels`
      - **Checked exceptions**:
        - `a == 0` if there was an error in opening the file
        - `a == -1` if given file had `row` and `col` more than `maxdim`
      - **Assertions**:
        - `a` is equal to 1
        - `row`, `col` `pixels` == 1st 3 columns of `initial.txt`
        - `rate` == last column of `initial.txt`
        - `grid[0][0]` is not a `NULL` pointer
        - `grid[0][0]->current_state` is `1`
    - `init store_file()`
      - **Return**: returns value `a`
      - **Expected behaviour** :
        - `a == 1`
        - prints `alive`, `dead`, `pop`, and the shape of the world to `final.txt`
      - **Checked exceptions**: `a == 0` if there was an error in opening the file
      - **Assertions**: `a` is equal to 1
    - `int get_neighbours(int x, int y)`
      - **Return**: returns value `a`
      - **Expected behaviour** : `a > 0` when we have alive neighbours
      - **Checked exceptions**: `a = 0` when we have dead neighbours
      - **Assertions**: `a` is a positive integer and matches the actual number of neighbours
    - `void reshape_grid()`
      - **Return**: returns null
      - **Expected behaviour** : updates the cell states based on Conway's set of rules
      - **Checked exceptions**: N/A
      - **Assertions**:
        - `grid[0][0]->current_state == 1` as it is accompanied by the alive cells `grid[0][1]` and `grid[1][0]`
        - `alive`, `dead` matches the actual number of alive and dead cells (calls `get_populations()`)
    - `void undo_reshape()`
      - **Return**: returns null
      - **Expected behaviour** : undo the reshape operation
      - **Checked exceptions**: N/A

            - **Assertions**:
                - `gird[2][2]->current_state` was `0` before calling `reshape_grid()` and then becomes `1` and after calling `undo_reshape()` it should becomes `0` again
                - this also checks if `reset_grid()` is working as it is called first
        - `int restore_stored_file()`
            - **Return**: returns value `a`
            - **Expected behaviour** :
                - `a == 1` when everything goes fine
                - loads the previous file state into the current grid
            - **Checked exceptions**:
                - `a == -1` when `initial.txt` doesn't match `final.txt`
                - `a == 0` when `final.txt` couldn't be opened
            - **Assertions**: `gird[0][0]->current_state` is `1` after calling `reset_gird()` then calling `restore_stored_file()` as `grid[0][0]->current_state == 1` in `final.txt`
    - **Graphics** (`graphics.c`)
        - `int initialize()`
            - **Return**: returns value `a`
            - **Expected behaviour** : `a == 1` if starting SDL goes correct
            - **Checked exceptions**: `a == 0` if starting SDL doesnt go as planned
            - **Assertions**: `a == 1` is asserted as true
        - `void display_grid()`
            - **Return**: returns null
            - **Expected behaviour** : shows the `grid` in a window
            - **Checked exceptions**: N/A
            - **Assertions**: changing few cells to alive and seeing if they are drawn correctly
    - **Misc** (`misc.c`)
        - `void poll_event()`
            - **Return**: returns null
            - **Expected behaviour** :
                - if `state == start`
                    - changes the corresponding cell based on mouse click coordinates
                    - pressing `o` restores previous `final.txt` file
                    - pressing `Enter` key changes the `state` to `running`
                - pressing `r` changes `state` to `start`
                - pressing `space` key does the following
                    - changes the `game_state` of the program to `pause` if it was `running`
                    - changes the `game_state` back to `running` if it was `pause`
                    - pressing `right-arrow` moves the program by 1 iteration forward
                    - pressing `left-arrow` moves the program by 1 iteration backward

        - pressing `exit` in SDL window changes `game_state` to `quit`
    - **Checked exceptions**: N/A
    - **Assertions**:
        - pressing `space`, `game_state == pause` becomes `true` if it was `game_state == running` and the opposite
        - pressing `enter`, `game_state == running` becomes `true`
            - pressing `right-arrow` calls `reshape_grid()`
            - pressing `left-arrow` calls `undo_reshape()`
        - pressing `r`, `game_state == start` becomes `true` and calls `reset_grid()` + `reset_alive_cells()`
        - pressing `X` button on window, `game_state == quit` becomes `true`
        - `grid[x][y]->current_state == 1` becomes `true` if it was `false` when clicking on its coordinates

Project schedule

- Week counting starts from 3^{rd} of March 20201

Week	Description
week-1	Adjusting the repository directory structure Adding some bash build scripts for SDL, egypt and cmake Creating `CMakeLists.txt`
week-2	Writing the header files `main.h` and `logic.h` with function prototypes in order to start writing the report
week-3	writing `graphic.h` and `misc.h` Finishing up the report Start with logic module (`logic.c`)
week-4	Finish logic module Write unit test for logic module (`test_logic.c`) Start with graphics module (`graphic.c`)
week-5	Finish graphics module Write unit test for graphics module (`test_graphic.c`) Start with misc module (`misc.c`)
week-6	Finish up misc module Write unit test for misc module (`test_misc.c`) Start with the main module (`main.c`)
week-7	Finish main module Finish up the `README` file Clean up the directory and adjust code organisation Testing the program as a user (running build scripts, compiling and running the program)
week-8	Finalise everything and submit the work

Bibliography

- Izhikevich, E., Conway, J. and Seth, A., 2015. *Game of Life*. [online] scholarpedia. Available at: http://www.scholarpedia.org/article/Game_of_Life#See_Also [Accessed 23 March 2021].

- SDL: http://www.libsdl.org/
- Cmake: https://cmake.org/
- Egypt: https://www.gson.org/egypt/
- Unity: http://www.throwtheswitch.org/unity