Developer Documentation

Overview

The program starts in the MENU state, the input processing thread is created. The program enters a(n infinite) loop, which switches to the case given by the MenuState of the program. The functionalities of the states are explained in the Data Structures part of this documentation. main.c is responsible for handling the main program loop, the handling of the input, and drawing to the screen. board.c contains the definitions of the functions related to the creation, deletion, and handling of the board, and file related operations. The state.txt file stores the state which will be loaded before the simulation, and set state changes this file based on the user selected width+height and alive cells. The program consists of the following files: main.c, board.c, board.h, state.txt and the external dependecies (included in the solution) econio.c, econio.h. The users can create their own save files, which will be stored in the "SAVES" folder, from where the program tries to load the save files. (A few examples are included in the solution.)

Data Structures

enum MenuState

Represents the different states of the menu:

- MENU
  - The user can enter and return to from the functionalities of the program from here
- SETSTATE
  - The user can change write a new board configuration to state.txt
- LIFE
  - Once started, starts the simulation with the state loaded from state.txt
- LIFESTEP
  - Same as 'LIFE', but the user must step trough the simulation manually
- LOAD
  - Copies the content of a save file from the folder "SAVES" to state.txt
- SAVE
  - Copies the content of state.txt to a file in the "SAVES" folder
- QUIT
  - Deallocates dynamically allocated memories, exits the thread processing the character input, and quits the program
- ERRORMESSAGE
  - The program enters this state whenever an error occurs(, which can be handled)
  - @errorString gets printed to this screen, so the user knows what went wrong (for example a file is missing from somewhere)

enum KeyState

Represents the different key states which are used for control. The @key variable in main.c stores a single KeyState. The InputThread only registers new input, once the current input is processed on the main thread, and the key is set to the KEY_EMPTY state:

- KEY_EMPTY: empty state, indicates that the previous input was processed
- KEY_UP: the 'w' key
- KEY_DOWN: the 's' key
- KEY_LEFT: the 'a' key
- KEY_RIGHT: the 'd' key
- KEY_ENTER: the 'ENTER' key
- KEY_ESCAPE: the 'ESCAPE' key

struct InputThreadParams

Used to pass pointers to variables to the input thread:

- enum MenuState* state
- enum KeyState* key

struct Board

Represents the game board. Only one static instance (board) is created in board.c, this way the data is encapsulated outside of this file (in main.c). The functions declared in board.h are used to manipulate the data of the board AS IF it were the public functions of a class. Functions only defined in board.c are helper functions used by the functions declared in board.h, as if it were the private functions of the board "class".

- int height: logical height of the board
- int width: logical width of the board
- int generation: current iteration of the simulation
- char* data: Stores the state of the board. 0 is a dead cell, 1 is a live cell. It's a dynamically allocated 2D array with the size (width+2)x(height+2), because the border is stored as well as 0 cells, to make the algorithm for the next state nicer. The data is stored in one "chunk" on the heap, and the element specified by the row and column can be accessed with appropriate indexing.
- char* next_data: The next state of the board is stored here. After all the calculations are done, this becomes the state which is displayed.

Algorithms

Game of life algorithm

1. Initialization: The function starts by iterating over each cell in the current board (data), excluding the border/padding cells. It uses nested loops to access each cell by its row and column indices.
2. Neighbor Counting: For each cell, the function counts the number of live neighbors. This involves checking the eight surrounding cells. (In practice, for a cell @i,k it

loops trough the 3x3 adjacent grid, then subtracts the value of the cell @i,j).

3. Applying Rules: Based on the number of live neighbors, the function applies the rules of Conway's Game of Life and saves the result in next_data, to not interfere with further calculations:
   - Underpopulation: If a live cell has fewer than two live neighbors, it dies
   - Overpopulation: If a live cell has more than three live neighbors, it dies.
   - Survival: If a live cell has two or three live neighbors, it remains alive.
   - Reproduction: If a dead cell has exactly three live neighbors, it becomes alive.
4. Update State: After processing all cells, the function swaps the data and next_data pointers. This makes the next_data (which now contains the updated state) the current state for the next generation.
5. Generation Increment: Finally, the function increments the generation counter to reflect the advancement of the simulation.

Functions

main.c

DESCRIPTION:
Stores all text file names found in SAVES in an array
INPUT/OUTPUT:
@fileNames is an array of strings to store the file names, caller must deallocate the memory allocated for the strings
Returns the number of files found, which is the number of strings allocated in the @fileNames array
int listTextFiles(char* fileNames[MAX_SAVE_FILES]);

DESCRIPTION:
Draws the borders of the board
void DrawBorders();

DESCRIPTION:
Draws the cells to the screen
void DrawBoard();

board.h

DESCRIPTION:
Deallocates data, and next_data
void FreeBoard();

DESCRIPTION:
Calls NewBoard with height=-1 and width=-1 to load the state from state.txt
INPUT/OUTPUT:
@errorstring is printed at the error screen if an error occurs
int NewBoardFromState(char* errorString);

DESCRIPTION:
Creates, initializes data and next_data (with dead cells / 0-s) to store the new board then loads cells from state.txt if height=-1 and width=-1
Automatically deallocates the previous memory (if needed)

INPUT/OUTPUT:
@errorstring is printed at the error screen if an error occurs
and returns 0 if board initialization was successful
int NewBoard(char* errorString, int height, int width);

DESCRIPTION:
Writes a new state to state.txt
INPUT/OUTPUT:
@width is the width of the board, @height is the height of the board, @cellData is the board data (without padding)
Returns 0 if board initialization was successful
int SetState(char* errorString);

DESCRIPTION:
Copies state.txt to a new or existing file named @filename
INPUT/OUTPUT:
@filename must be a c type string
Returns 0 if board initialization was successful
int SaveState(char *filename, char* errorString);

DESCRIPTION:
Copies an existing file named @filename to state.txt
INPUT/OUTPUT:
@filename must be a c type string
int LoadState(char *filename, char* errorString);

DESCRIPTION:
Advances the simulation
void UpdateBoard();

INPUT/OUTPUT:
Returns the character at [@row][@column] of the board data without the padding row and column
char Get(int row, int column);

INPUT/OUTPUT:
Sets the character at [@row][@column] of the board next_data without the padding row and column
void Set(int row, int column, char value);

DESCRIPTION:
Used for setting the initial state
INPUT/OUTPUT:
Sets the character at [@row][@column] of the board data without the padding row and column
void Start_Set(int row, int column, char value);

INPUT/OUTPUT:
Returns the height of the board
int Get_Height();

INPUT/OUTPUT:
Returns the width of the board
int Get_Width();

INPUT/OUTPUT:
Returns the generation number
int GetGeneration();

INPUT/OUTPUT:
Returns the population
int GetPopulation()

board.c

DESCRIPTION:
Used by the UpdateBoard function which needs to access the board with padding
Allows extended indexing from -1 to M, -1 to N
INPUT/OUTPUT:
Returns the character at [@row][@column] of the board with the padding row and column
char GetExt(int row, int column);

DESCRIPTION:
Copies the content of the source file to the destination file
INPUT/OUTPUT:
@source is the path to the source file, @dest is the path to the destination file
@errorString is printed at the error screen if an error occurs
Returns 0 if the copy was successful
int CopyFile(char *source, char* dest, char* errorString)

Testing documentation

Manual test cases

- All functionalities, navigation work for the expected input
  - running the simulation
  - running the simulation after setting a state (then check state.txt)
  - saving a state
  - loading a state, then running it
  - quitting
- Try to go out of bounds when navigating the MENU or files in the LOAD state
- Enter invalid dimensions for setting the state
- Try to go out of bounds when setting the state
- After deleting state.txt, start the simulation and try to create a save file
  - check if the error message is correct
- Delete the "SAVES" folder
  - the user can't load files (no files are shown)
  - the user can't save files (error opening SAVES//filename.txt)

Testing result: All test cases passed