CSD1401 - Assignment 3

Purpose

This assignment will help students develop:

- Familiarity with the C programming language
- Familiarity with the Visual Studios IDE
- Familiarity with the CProcessing engine
- Familiarity with developing an application
- Familiarity with the concept of states
- Familiarity with the concept of double buffers
- Ability to break down and manage large tasks into smaller ones.

Overview

Computer simulations is the usage of the computer to imitate real-word processes. Its application are near-limitless. It can be used to train pilots, test for safety of a car crash, weather forecasting, simulate real-world processes for study and many more. As such, computer simulations used are all around us, serving as an important tool in several fields of study such as mathematics, astrophysics, chemistry, biology, economics, engineering, health care, psychology, social science and business.

Game of Life is one such simulation, also known as a 'cellular automaton'. It is a zero-player game devised by British mathematician John Horton Conway in 1970. It is a 2D grid-based simulation, which is widely used to explore the evolution of ecological communities. Its beauty lies in the simplicity of its rules to create many interesting and dynamic different behaviors, such as a gun.

You can play with Game Of Life online here.

Tasks

Create the Visual Studio Project

1. In the `src` folder, you should see `game.c` and `main.c`
2. Use these files to create a CProcessing Visual Studio Project.

View and understand what's required from the sample program

Open the `sample` folder and run `game.exe`. You should see a window pop out on your screen. Observe what is happening in the window. In this assignment, you will be implementing a similar program to the sample program given to you.

Understanding the rules of Game Of Life

The Game Of Life world is represented with a 2D grid. Each cell on the grid can be 'on', which represents a living cell, or 'off', which represents a dead cell. (some pun intended) . As such, we can think of each cell in the grid as being live or dead.

At each step we take forward in time, the grid will be updated with the following rules:

- Any live cell with fewer than two live neighbors dies (emulates underpopulation).
- Any live cell with two or three live neighbors lives on.
- Any live cell with more than three live neighbors dies (emulates overpopulation).
- Any dead cell with exactly three live neighbors becomes a live cell (emulates reproduction).

Features of our application

Our application has the following features:

- It is filled with a 30 by 30 grid of cells
- It has two main 'states': **pause** and **play** state. By default, the application is in the **pause** state. Users can toggle between both states by pressing any key on the keyboard.
    - **Pause**: In this state, uses can click on each cell to toggle its live or dead state
    - **Play**: In this state, you will 'simulate' the Game Of Life, updating the cells with the rules stated above

Pseudocode and Double Buffer concept

There are many ways to implement this Game Of Life assignment. It would not be uncommon of come up with a very high-level pseudocode for `game_update()` that looks something like this:

```
if any key is pressed:
    toggle pause state
if paused:
    for each cell:
       if mouse is colliding with cell:
           toggle cell state
else:
    // simulation case
    for each cell:
        if cell is alive:
            if cell has less than 2 neighbours:
                cell = dead
            else if cell has more than 3 neighbours:
                cell = dead
        else:
            if cell is dead:
                if cell has exactly 3 neighbours:
                    cell = alive

render all cells
```

The issue with this pseudocode is that in the 'simulation case' part of the code, we are updating the grid as we go through each cell. This will result in the later cells we visit updating based on a partially updated grid. This is wrong. What we want is to update each cell based on the state of the grid *before* any updates to its cells. But our grid state is lost as we update! What should we do?

This brings us to a technique that is similar to one used in computer graphics known as a 'double buffering'. The idea is simple. Instead of having one grid to keep track of the grid state, we have two. Let's call them `grid[0]` and `grid[1]`. As our program simulates each frame, we will update and display one grid while referencing the other:

1. Update `grid[1]` based on `grid[0]`. Render `grid[1]`.
2. Update `grid[0]` based on `grid[1]`. Render `grid[0]`.
3. Update `grid[1]` based on `grid[0]`. Render `grid[1]`.
4. Update `grid[0]` based on `grid[1]`. Render `grid[0]`.
5. ...etc.

This brings us to our `game_update()` pseudocode. The 'displaying grid' is the grid you are going to update and render. The 'reference grid' is the other grid, i.e. the one the 'displaying grid' references to update its cells.

```
figure out which index is the 'reference grid' and the 'displaying grid'.

if any key is pressed:
    toggle pause state
if paused:
    for each cell in 'displaying grid':
       if mouse is colliding with cell of 'displaying grid':
           toggle cell state
else:
    // simulation case
    for each cell in 'displaying grid':
        if cell is alive in 'reference grid':
            if cell has less than 2 neighbours:
                cell = dead
            else if cell has more than 3 neighbours in 'reference grid':
                cell = dead
        else:
            if cell is dead:
                if cell has exactly 3 neighbours in 'reference grid':
                    cell = alive

render all cells in 'displaying grid'
```

Provided constants

You are given these constants to work with. **Do not** change these values in your final submission.

- `GOL_GRID_COLS` - The amount of cells in a grid's column. This is set to 30.
- `GOL_GRID_ROW` - The amount of cells in a grid's row. This is set to 30.
- `GOL_GRID_BUFFERS` - The amount of grids we use in our application. Since we are using this for the 'double buffer' technique, it is set to 2.
- `GOL_ALIVE` - The cell value that represents a live cell. This is set as 1.
- `GOL_DEAD` - The cell value that represents a dead cell. This is set as 0.
- `TRUE` - Convenient representation of 'true'. This is set as 1.
- `FALSE` - Convenient representation of 'false'. This is set as 0.

Provided variables

You are also given these variables that you **must use and modify** to indicate the various states of the application.

- `gIsPaused` - An integer used to indicate if the application is paused or not. The value of `TRUE` indicates that it is paused. The value of `FALSE` indicates that it is **not** paused.

- `gGrids` - A triple array of integers representing our grids.

  In our application, a single grid is normally represented with a double array of integers representing cells with the following format:

  ```
  /* This represents 1 grid of (GOL_GRID_ROWS * GOL_GRID_COLS) amount of cells
  */
  int gGrid[GOL_GRID_ROWS][GOL_GRID_COLS];
  ```

  Since we are managing multiple of such grids, we extend this by having an array of `GOL_GRID_BUFFERS` grids.

  ```
  /* This represents GOL_GRID_BUFFERS grids of (GOL_GRID_ROWS * GOL_GRID_COLS)
  amount of cells */
  int gGrid[GOL_GRID_BUFFERS][GOL_GRID_ROWS][GOL_GRID_COLS];
  ```

  What we end up is a triple array of integers that represents all the grids we will be using.

  Each integer in the grid represents a cell, whose value should either be `GOL_ALIVE` or `GOL_DEAD`

  Below are some examples for clarity on the expected usages of this variable.

  ```
  /* Set grid 0's cell at position at row 1 and column 1 to be 'live' */
  gGrid[0][1][1] = GOL_ALIVE;

  /* Set grid 0's cell at position at row 2 and column 3 to be 'live' */
  gGrid[0][2][3] = GOL_ALIVE;

  /* Set grid 1's cell at position at row 5 and column 2 to be 'dead' */
  gGrid[1][5][2] = GOL_DEAD;
  ```

Initialization of these variables is already done in `game_init()`.

Do not change the initialization of these variables in your final submission.

Provide your own variables and constants

Aside from the variables and constants provided above, you are free to provide your own variables and initialize them in `game_init()`. As a hint, you might want to come up with one or two variables to indicate which is the 'reference grid' and the 'displaying grid'.

Reference `cprocessing.h` for functions that can help you

In order to perform some of code above, you will need to use functions provided in `cprocessing.h`.

You have learnt some functions in the previous assignment. Below are some additional functions in there that you will need to use to complete the assignment and some example code snippets:

- `CP_System_GetWindowWidth` and `CP_System_GetWindowHeight`: Returns the width and height of the application window respectively.

```
/* Draws a 10 diameter circle in the middle of the window */
float x = (float)CP_System_GetWindowWidth() * 0.5f;
float y = (float)CP_System_GetWindowHeight() * 0.5f;
CP_Graphics_DrawCircle(x, y, 10);
```

- `CP_Input_KeyTriggered`: Checks if a keyboard key is pressed. Returns 1 if it is and 0 otherwise.

```
/* Checks if any key is pressed */
if (CP_Input_KeyTriggered(KEY_ANY)) {
    /* Case whereany key is pressed */
}
```

Check your work

Check your program against the `Processing_Sample.exe` executable provided in the `sample` folder and compare their behaviors.

You do not have to worry about details that are not specified in this documents like:

- The color of a live and dead cell. As long as they are distinctly different, it's good enough.

Once you are done, proceed to the **Deliverables** section to submit your assignment.

Grades

For this assignment, grades will be roughly be given as follows:

What was achieved	Grade
Game of Life with pause state	A
Game of Life without pause state	C
Game of Life not even functional	F

Grades in between can also be given (eg. D and B) depending on quality of implementation.

A+ will be considered for students who achieved A **and** have a clean submission (check **Deliverables** section below).

Deliverables

1. Zip your completed `game.c` using the convention `<DigiPen student id>_3.zip`. For example: If my DigiPen student e-mail is `cheng.dingxiang@digipen.edu`, my DigiPen student id is `cheng.dingxiang`. My zipped folder will hence be named `cheng.dingxiang_3.zip`
2. Submit `<DigiPen student id>_3.zip` file to the appropriate assignment submission link in Moodle.
3. After submitting, do a sanity check by re-downloading the file that you submitted and ensure that it is indeed the file that you submitted.