

C++ Programming Methods

Assignment 3, Game of Life **compensation**

Bas Terwijn <b.terwijn@uva.nl>

This is the compensation assignment to compensate for deficiencies in “Assignment 3, Game of Life”. We will still create a Game of Life simulator (see [this simulator](#) as an example), but we will make the simulator more efficient so that it uses less memory and runs faster. We will do this by using a `std::unordered_set<Coord>` to only keep track of alive cells.

Install the Simulator

A skeleton Graphical User Interface (GUI) for our simulator is already provided. It’s your job to complete it. Install the GUI using:

Install Linux/Mac

Install the following packages with your package manager:

- `cmake`
- `libsdl2-dev`
- `libsdl2-gfx-dev`

Then compile from within the “assignment2” directory using:

```
mkdir build
cd build/
cmake ..
make
```

Install Windows

Install [git](#) and [Visual Studio Studio Community 2022](#) and in the Visual Studio Installer install Workload “Desktop development with C++” (includes CMake).

Install the SDL2 graphics library with [vcpkg](#) by using:

```
git clone https://github.com/Microsoft/vcpkg.git
cd vcpkg
.\bootstrap-vcpkg.bat
.\vcpkg.exe integrate install
.\vcpkg.exe install sdl2 sdl2-ttf --triplet x64-windows
```

Then in Visual Studio Studio Community 2022 open the “assignment3” directory and build the project.

Conway's Game of Life

In Conway's Game of Life the world consists of a grid of cells that each can be dead or alive. At each time step the following rules are applied that determine the state of each cell in the next time step. A cell's state in the next time step depends on the number of life cells in the current time step in its 8 surrounding neighbor cells:

- a life cell with less than 2 life neighbors dies (underpopulation)
- a life cell with 2 or 3 life neighbors lives on
- a life cell with more than 3 life neighbors dies (overpopulation)
- a dead cell with 3 life neighbors becomes alive (reproduction)

Complete the Simulator

Complete the simulator by adding at least the classes **World**, **Random** and **Menu**, each in its own header file. Use encapsulation when writing these classes to hide the implementation details of a class behind its methods/interface. Change/rewrite/move parts of the 'game_of_life.cpp' as required.

World Class

In file 'World.h' create a new World class that represents the world as grid of cells and implements the Game of Life rules and other things that have to do with the world. Use a `std::unordered_set<Coord>` to represents the world where 'Coord' is a coordinate of each alive cell. All other cells are assumed to be dead. This avoids having to use any memory of the dead cells. Additionally in each time step we will only have to update the state of cells that are close to alive cells, so this can make the simulator run much faster (or use less computing resources) as no computations need to be done in areas with only dead cells. See the `std::unordered_set<Coord>` example in the provided 'set_example.cpp' file.

Random Class

In file 'Random.h' create a Random class that implements your own random number generator that you use to randomize the world. Do not use existing C++ random number generators for the implementation. Instead search the web or think about how you can create a random number generator from scratch, for example by just using some simple arithmetic. Add a way to optionally "seed" your Random class so that it doesn't produce the same set of random values each time you run the program, for example by using a value base on the current time to initialize the random number generator.

Menu Class

In file 'Menu.h' create a new Menu class that holds the following buttons and handles button presses (the first three buttons are already partially provided in 'game_of_life.cpp', better move them to 'Menu.h').

- **Randomize**, randomizes the world by setting the state of each cells to dead/alive.
- **Clean**, sets the state of each cell to dead.
- **Load**, asks the user for a filename and loads the pattern in this file into the world at the position the GUI is looking at
- **One**, moves the grid one timestep forward in time using the Game of Life rules
- **Go**, moves the grid 100 timesteps forward in time using the Game of Life rules and shows each timestep
- **Parameters**, opens a new text menu (similar to **Load**) where the user can set parameters:
 - probability of a cell being alive when randomizing the world (default: 0.5)
 - number of timesteps that **Go** moves the world forward in time (default: 100)
 - simulation speed, the time used to show each timestep before showing the next timestep when **Go** is pressed
 - add more parameters when needed

Pattern Files

Different pattern files are provided to test your implementation of the rules. For example pattern “gliderGun.txt” should over time produce a stream of moving objects. Pattern “toad.txt”, “beacon.txt”, “pulsar.txt” should produce a simpler result. A pattern file consists of a small 2-dimensonal grid where dead cells are represented by a dot or whitespace (‘.’ or ‘ ’) character and living cells are represented by any other character except a new-line character which represents the start of a new line. When a pattern file is loaded it overwrites the cells in the grid where the GUI is looking at.

Submission

Submit your solution before the deadline to canvas. Add to the solution:

- your name, student number and the name of the assignment
- references to the source of any algorithm or code that you did not create yourself
- operating system and compiler that was used to test the code

Submit your solution by zipping the assignment3 directory with all the required files for your Game of Life simulator. But first **remove** the “build” directory (when grading we will create our own “build” directory) and any other files that are not necessary to keep the zip file small. Unzip your zip file in a temporary directory and test to make sure your simulator compiles using the install instructions above (otherwise we cannot grade it).