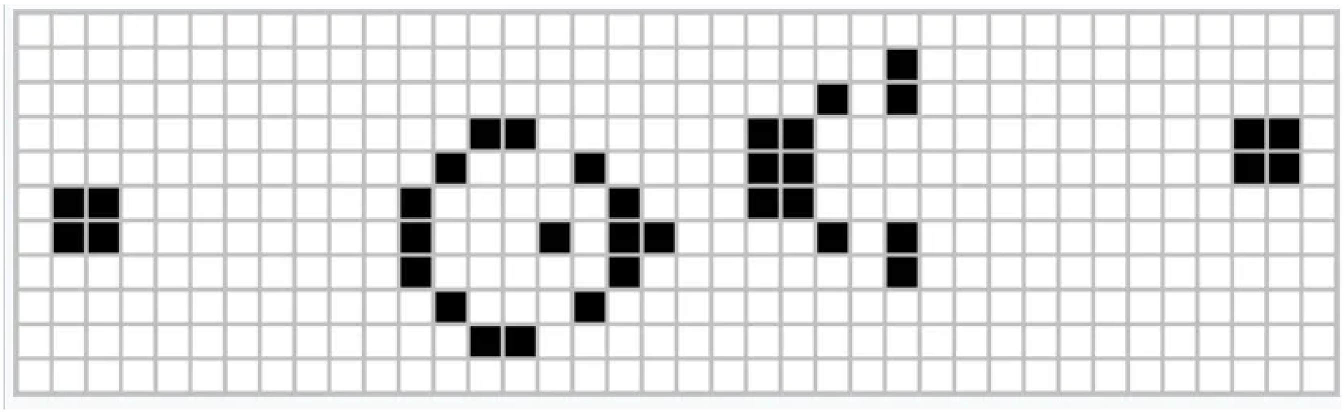
Creating Conway's Game of Life in C++

A step-by-step guide to creating a memory-efficient game

Aleksa Zatezalo · Follow
Published in Better Programming
6 min read · Feb 21

Listen Share More



Source

Conway's Game of Life

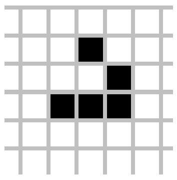
Conway's game of life is a cellular automation game simply known as Life. It was designed by Mathematician John Horton Conway in 1970 and is a Zero Player Game, meaning its outcome depends on its initial state. It has four basic rules, which are as follows:

1. Any live cell with fewer than two live neighbors dies, as if by underpopulation.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by overpopulation.

4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

The glider

The glider with is defined by five cells located at (x, y), (x+1, y), (x+2, y), (x+2, y+1) and (x+1, y+2) coordinates, respectively. It glides across the screen diagonally and can collide with other gliders to produce more complex patterns. If gliders collide correctly, they can be used to create And, Or, and Not Gates making for a Turing Complete Computer with theoretically infinite memory and computing power. Because of this, it is considered emblematic of the hacker community.



Source: https://conwaylife.com/wiki/Glider

The version we will create

The version of Life we will be creating is coded in C++. It randomly populates the board with living cells and allows you to watch the development of the board in real time. Our version of life was created to be as memory efficient as possible. The cell is represented by 5 bits. The first bit represents a living or dead status. The subsequent 4 bits will represent the existence of top, bottom, left, right, and corner neighbors. `1` (or true) represents living status and the existence of a specific neighbor, while `0` (or false) represents the neighbors' absence and the dead state. The whole program will be coded in one file.

SDL Boilerplate

We will start this project by creating our main function, which instantiates a window using C++'s SDL Graphics package. SDL stands for Simple DirectMedia Layer. It is a cross-platform development library designed to provide low-level access to audio, keyboard, mouse, joystick, and graphics hardware. It can be used to make animations and video games. We will be creating a screen with the dimensions of `SCREEN_HEIGHT` and `SCREEN_WIDTH`, titled "Conway's game of life." Aside from line 16 (`SDL_Delay(1000)`), which delays the refreshing of each screen by 1,000 milliseconds. All the code used is boilerplate and can be seen below:

```
int main(int argc, char * argv[]) {
    // SDL Boilerplate
    SDL_Init(SDL_INIT_VIDEO);
    window = SDL_CreateWindow("Conway's game of life", SDL_WINDOWPOS_UNDEFINED, SDL_WINDOWPOS_U
    SCREEN_WIDTH, SCREEN_HEIGHT, SDL_WINDOW_SHOWN);

    surface = SDL_GetWindowSurface(window);
    // SDL Event Handler
    SDL_Event e;

    // Rendering Loop
    bool quit = false;
    while (!quit){
        while (SDL_PollEvent(&e) != 0)
            if (e.type == SDL_QUIT) quit == true;
        SDL_Delay(1000);
        // Update frame buffer
        SDL_UpdateWindowSurface(window);
    }

    // Clean up SDL
    SDL_DestroyWindow(window);
    // Quit
    SDL_Quit();
    return 0;
}
```

As you might have guessed, `SCREEN_HEIGHT` and `SCREEN_WIDTH` are constants placed at the top of the file. I used 100 and 100 for height and width, respectively, but you can use any other values. The software requires a few constants and imports seen below:

```cpp
#include <SDL2/SDL.h>
#include <memory>
#include <ctime>

// SDL Window and Surface for pixel manipulation
SDL_Window *window = NULL;
SDL_Surface *surface = NULL;



// Width and height of cell size in pixels
unsigned int CELL_SIZE =4;
unsigned  CELLMAP_WIDTH = 100;
unsigned CELLMAP_HEIGHT = 100;

// Screen length and width
unsigned int SCREEN_WIDTH = CELLMAP_WIDTH * CELL_SIZE;
unsigned int SCREEN_HEIGHT = CELLMAP_HEIGHT * CELL_SIZE;
```

Our next step involves creating a function to draw each cell.

Drawing the Cell

Our function, `DrawCell`, will take three variables, the `x` and `y` coordinates of the cell and its color, as seen below. To populate the cell, we will be converting the location of the cell into its location on a unidimensional array through the formula Y*WIDTH+X. This happens on row 4. We will use a nested for loop to iterate over the length and width of the cell and set each of its pixels to the specified color. This happens from line 4 to line 9.

```
void DrawCell(unsigned int x, unsigned int y, unsigned int colour) {
    Uint8* pixel_ptr =  (Uint8*)surface->pixels + (y * CELL_SIZE * SCREEN_WIDTH + x * CELL_SIZE

    for (unsigned int i = 0; i < CELL_SIZE; i++) {
        for (unsigned int j = 0; j < CELL_SIZE; j++) {
            *(pixel_ptr + j * 4) = colour;
            *(pixel_ptr + j * 4 + 1) = colour;
            *(pixel_ptr + j * 4 + 2) = colour;
        }

        pixel_ptr += SCREEN_WIDTH * 4;
    }
}
```

Creating the Cell Map

To create our screen, we need to represent it logically. This will be done through our `CellMap` class which has four functions: `SetCell`, which sets the state and color of the cell (makes it living and its color white), `ClearCell` which kills the cell and makes it black, `Init` , which initializes the array, `CellState`, which checks the status of the cell (living or dead), and `NextGen`, which takes the board to its next generation.

```
class CellMap {
    public:
        CellMap(unsigned int width, unsigned int height);
        ~CellMap();
        void SetCell(unsigned int x, unsigned int y);
        void ClearCell(unsigned int x, unsigned int y);
        void Init();
        int CellState(unsigned int x, unsigned int y);
        void NextGen();
    private:
        unsigned char *cells;
        unsigned char *temp_cells;
        unsigned int w, h;
        unsigned length;
};
```

Our `CellMap` class will contain five variables. Two pointers, `cells` and `temp_cells`, will point to our cells in memory while `w` (for width), `h` (for height), and `length` (for `w*h`) will represent the dimensions of our cell mapping.

Next, we will create the functionality to set and clear the cell.

SetCell and ClearCell

Once we convert our cell map to a unidimensional array, as we did previously, our `SetCell` function will set the first bit of the cell to 1, representing the alive state. It will use the `w` and `h` variables defined in our `CellMap` class to calculate offsets which will be used to point to the first bit of all of the cell's neighbors (Top-Left, Top, Top-Right, Left, Right, Bottom-Left, Bottom, and Bottom-Right), which of course represent their living status. This can be seen on lines 24–31. Through this process, each cell is “aware” of its living neighbors. This can be seen in the code snippet below:

```cpp
1   void CellMap::SetCell(unsigned int x, unsigned int y) {
2       unsigned char *cell_ptr = cells + (y*w + x);
3       int xleft, xright, yabove, ybelow;
4
5       *(cell_ptr) |= 0x01; // Set the first bit as 1, 'on'
6
7       if (x==0)
8           xleft = w -1;
9       else
10          xleft = -1;
11      if (x== (w-1))
12          xright = -(w-1);
13      else
14          xright = 1;
15      if (y==0)
16          yabove = length -w;
17      else
18          yabove = -w;
19      if (y==(h-1))
```

```cpp
25      *(cell_ptr + yabove) += 0x02;
26      *(cell_ptr + yabove + xright) += 0x02;
27      *(cell_ptr + xleft) += 0x02;
28      *(cell_ptr + xright) += 0x02;
29      *(cell_ptr + ybelow + xleft) += 0x02;
30      *(cell_ptr + ybelow) += 0x02;
31      *(cell_ptr + ybelow + xright) += 0x02;
32  }
```

SetCell.cpp hosted with ❤ by GitHub view raw

Our `ClearCell` function is almost identical except lines 23–31, which decrement by `0x02`, indicating to all the cells' neighbors that it died.

CellState

The next function, `CellState`, will perform a simple AND operation on the first bit of the cell pointer returning `1` (or True) if the cell is alive and `0` (or False) if the cell is dead. It will be leveraged in our cell map initialization later on.

```cpp
int CellMap::CellState(unsigned int x, unsigned int y) {
    unsigned char *cell_ptr = cells + (y*w) + x;

    return *cell_ptr & 0x01;
}
```

CellState.cpp hosted with ❤ by GitHub view raw

NextGen

The `NextGen` function will be called in our main loop seen earlier in this article. After each 1000 millisecond delay, it will parse through all living cells using a for loop, updating its neighbor info and status. Our `NextGen` function will operate on the `CellMap`'s `temp_cells` array, which will act as a copy of the `cell_ptr` such that the cell map's initial state is considered for each cell whenever the `NextGen` function is called.

We will start by skipping past all dead cells with no color and are not alive, meaning their bitwise representation is `0`. This is seen in lines 9 through 18, which increment the cell pointer each time the value pointed to is 0.

We will then perform a right shift on the cells' bitwise representation, which will remove the cells living or dead status, leaving us with a number representing the number of the cells living neighbors. Cells with two neighbors will remain alive, while cells with three will "become alive." This is seen on lines 20 through 30 below. This same process is done for every row on the cell map.

```cpp
void CellMap::NextGen() {
    unsigned int x, y, live_neighbours;
    unsigned char* cell_ptr;

    memcpy(temp_cells, cells, length);

    cell_ptr = temp_cells;

    for (int y=0; y < h; y++){
        x = 0;
        do {
            // Skipping
            while (*cell_ptr == 0) {
                cell_ptr++;

                if (++x >= w) goto NextRow;

            }

            live_neighbours = *cell_ptr >> 1;
            if (*cell_ptr & 0x01) {
                if ((live_neighbours != 2) && (live_neighbours != 3)){
                    ClearCell(x, y);
                    DrawCell(x, y, 0x00);
                } else {
                    if (live_neighbours == 3) {
                        SetCell(x, y);
                        DrawCell(x, y, 0xFF);
                    }
                }

            }

            cell_ptr++;
        } while(++x < w);
    NextRow:;

    }
}
```

Completing the Code

To complete this code, we must create the `CellMap`'s `Init` method. This function will set the status to living for a random subset of cells, as seen from lines 1 through 15. In our main method, we will initialize the cell map and run the `nextGen` function (lines 28 and 36, respectively) in an infinite while loop. `CellMap.Init` is only to be called once.

```cpp
void CellMap::Init() {
    unsigned int seed = (unsigned)time(NULL);

    srand(seed);

    unsigned int x, y;

    for (int i = 0; i < length * 0.5; i++) {
        x = rand() % (w-1);
        y = rand() % (h-1);

        if (!CellState(x, y))
            SetCell(x, y);
    }
}

int main(int argc, char * argv[]) {
    // SDL Boilerplate
    SDL_Init(SDL_INIT_VIDEO);
    window = SDL_CreateWindow("Conway's game of life", SDL_WINDOWPOS_UNDEFINED, SDL_WINDOWPOS_U
    SCREEN_WIDTH, SCREEN_HEIGHT, SDL_WINDOW_SHOWN);

    surface = SDL_GetWindowSurface(window);
    // SDL Event Handler
    SDL_Event e;

    CellMap map(CELLMAP_WIDTH, CELLMAP_HEIGHT);
    map.Init();

    // Rendering Loop
    bool quit = false;
    while (!quit){
        while (SDL_PollEvent(&e) != 0)
            if (e.type == SDL_QUIT) quit == true;
        SDL_Delay(1000);
        map.NextGen();
        // Update frame buffer
        SDL_UpdateWindowSurface(window);
    }

    // Clean up SDL
    SDL_DestroyWindow(window);
    // Quit
    SDL_Quit();
    return 0;
}
```

The complete code for this project can be found below. I recommend trying to finish this yourself before "cheating."

GitHub - AleksaZatezalo/ConwaysLife

Conway's game of life, simply known as Life is a cellular automation game simply known as Life. It was designed by...

github.com

A note on compilation

As you all (hopefully, LOL) know, C++ is a compiled language. The SDL library is not standard like `<memory>` or `<ctime>`, meaning you have to link it manually. I was able to do that using this g++ command below:

```
g++ -o life main.cpp -lSDL2
```

End Notes: A Genius at Play

In creating this article, I noticed John Horton Conway had a biography written. I intend on reading it at some point. I thought it would be interesting to share. I am not affiliated with or sponsored by the author in any way, but you can find a link to its Amazon page here.:

Genius At Play: The Curious Mathematical Mind Of John Horton Conway

Genius At Play: The Curious Mathematical Mind Of John Horton Conway: Roberts, Siobhan: 9781620405932: Books - Amazon.ca

www.amazon.ca

Hacking   C Programming   Conways Game Of Life   Programing   Hacking Culture