Project - Game of Life

CSE/IT 113

Conway's Game of Life

Your project is to write a program that simulates Conway's Game of Life. As the game is best displayed with graphics, you will be using Simple Media Direct Layer (SDL) (`http://www.libsdl.org/`) to display the game on your screen. The SDL code for the graphics is provided for you and you must use the provided code. Your job is to implement the backend for the game, not the visual part of the game.

Conway's game is a cellular automata (`http://en.wikipedia.org/wiki/Cellular_automaton`) game invented in 1970. The game takes place on an infinite 2-d grid, in which each space of the grid represents a cell. A set of rules is applied to each cell of the current generation in order to determine what cells live or die in the next generation.

For more information on the Game of Life see `http://www.conwaylife.com/wiki/Conway%27s_Game_of_Life` and `http://www.conwaylife.com` for general information. If you are unfamiliar with the game, reading this material will help you greatly in programming this project.

Rules of the Game of Life

From conwaylife.com, the rules of the game (really a simulation) are:

"The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, live or dead. Every cell interacts with its eight neighbors, which are the cells that are directly horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

- Any live cell with fewer than two live neighbors dies, as if by needs caused by under population.
- Any live cell with more than three live neighbors dies, as if by overcrowding.
- Any live cell with two or three live neighbors lives, unchanged, to the next generation.
- Any dead cell with exactly three live neighbors cells will come to life.

The initial pattern constitutes the 'seed' of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed – births and deaths happen simultaneously, and the discrete moment at which this happens is sometimes called a tick. (In other words, each generation is a pure function of the one before.) The rules continue to be applied repeatedly to create further generations."

Modeling the game of life

To model the game of life, you first create two grids (multidimensional arrays or a matrix) of size $width \times height$, where width is the width of the screen (in pixels) divided by the sprite size, and height is the height of the screen (in pixels) divided by the sprite size (see below).

It is important to note that your matrix is defined in the opposite way of how you would normally think: the width of the screen is stored in the rows of the matrix, while the height of the screen is stored in the columns of the matrix. The program will seg fault if you create a matrix with dimensions $height \times width$. You have been warned.

At any given time, one grid, say A, contains the current generation and the other grid, say B, contains the next generation.

To start the model, you initialize A with a pattern of cells that are alive. The initial pattern determines how quickly the ecosystem lives or dies. Some initial patterns produce ecosystems that last a number of generations, while other patterns die off quickly.

To calculate the generations, you loop through the current generation grid (A) determining if each cell is going to live or die in the next generation based upon the status of its 8 neighbors. You write the results to the next generation grid (B). At the next time step (generation) the roles of the grids reverse. You now scan grid B for who lives and dies and write to A. You continue on this manner indefinitely, until the game of life has reached some equilibrium or the pattern dies out.

To display the cells on the screen you pass the current generations grid to the provided function `sdl_render_life`:

```
/**
 * draw the game of life on the screen
 * @param struct sdl_info_t sdl_info for SDL parameters
 * @param unsigned char **life the grid of alive or dead cells
 * @remarks The background is black. Make sure init_sdl_info()
 * is called before using this function.
 */
void sdl_render_life(struct sdl_info_t *sdl_info,
                     unsigned char **life);
```

Determining the grid

Obviously, you cannot model an infinite two-dimensional grid with a machine. Your grid is based on three factors: your screen dimensions, the size of your sprite, and how you model the boundaries of the grid.

The screen dimensions are the width and height of a 2-d grid in pixels. For example, 640 x 480, 800 x 600, 1024 x 768, 300 x 400, etc. Note: these are (width x height dimensions). The width and height determine the size of the rectangle you can draw to.

As pixels are too small to see, you will use a sprite, an image made of pixels, to model cells on the screen. The size of a sprite is the width and height of the cell (in pixels) as displayed on the screen. As our sprite is a square, a size of 2 corresponds to a cell of 2 x 2 pixels; a size of 4 is a cell of 4 x 4 pixels; a size of 8 is a cell of 8 x 8 pixels; a size of 16 is a cell of 16 x 16 pixels. Valid sizes are 2, 4, 8, and 16.

You have to adjust the height and width of the grid based on the sprite size. To determine the sprite_width (w_s) and the sprite_height (h_s) you use the relationships:

$$w_s = \frac{width\,of\,screen}{sprite\,size}$$

$$h_s = \frac{height\,of\,screen}{sprite\,size}.$$

You create a matrix of $w_s \times h_s$ To hold the cells. **Important: you are creating a matrix with w_s rows and h_s columns and not the other way around.** To allocate space for the matrix you use an array of pointers to pointers. For example, to create a matrix of integers with dimension $rows \times cols$:

```
/* create a multi-dimensional array */
int **init_matrix(int rows, int cols)
{
        int i;
        int j;
        int **a;
        /* allocate rows */
        a = malloc(rows * sizeof(int *));
        if(!a)
                return NULL;
        for(i = 0; i < rows; i++) {
                /* allocate cols for each row */
                a[i] = malloc(cols * sizeof(int));
                if (!a[i]) {
                        for (j = 0; j < i; j++)
                                free(a[j]);
                        free(a);
                        return NULL;
                }
        }
        return a;
}
```

You ultimately need to create two matrices of size $w_s \times h_s$ to hold the alternating generations.

Finally, you have determine the behavior of cells at the edge of the screen. **You will code three types of edges: the hedge, the torus, and the Klein bottle.**

The hedge – you add one row to the top and bottom and one column to the left and right side of your grid. All cells are dead in the newly added rows and columns. This allows the cells at the edges of your screen to be calculated the same way as interior cells. This doesn't lead to interesting patterns as cells die quickly along the edges, but is great for testing/development.

The *torus* – you fold the left and right edges of your grid together and the top and bottom edges together to create a torus (`http://en.wikipedia.org/wiki/Torus#Flat_torus`. The top row of cells use the bottom row of cells in its calculations and vice versa; and similarly cells in the left and right edges you each other in their calculations. This leads to a much more interesting patterns as the cells move across screen edges.

The Klein bottle – The Klein bottle joins the top and bottom edges of your grid (`http://en.wikipedia.org/wiki/Klein_bottle` –**read this**) and then the left edge is twisted before being joined to the right edge. Essentially, this means the upper left edge is joined to the lower right edge, and the lower left edge is joined to the upper right edge.

The easiest way to model the torus is through modular arithmetic. If you find you are writing complex if logic to do this, come talk to me or a TA as there is a better way. And to create a Klein bottle, think about how you can model its behavior via a torus and a reflection about a horizontal line through the center of the screen.

Initializing the Grid - File I/O

You will initialize the grid by reading in a ConwayLife's Life 1.06 formatted file. You should be able to place the pattern anywhere on the screen. So you also need to set an initial coordinate for the pattern. The initial coordinate (x_0, y_o) is relative to the upper left hand corner of the screen, which is considered (0, 0) in computer graphics. Unlike math, computer graphics flips the direction you move for a positive y-value – you move downward. For instance the point (10, 20) is located by starting at the origin moving 10 pixels to the right and 20 pixels down. A negative number for the y-value, would mean that you would move upward. For the point (10, -20), you would move 10 pixels to the right and 20 pixels up, which would be a point off the screen. The notation may seem strange at first, but notice how the coordinate system corresponds nicely with array index notation used in C and other programming languages for multi-dimensional arrays.

You must be able to read in the ConwayLife's file format Life 1.06 for the life pattern. The file format is available online at `http://www.conwaylife.com/wiki/Life_1.06`. Place all cells relative to the initial coordinate and movements mimic computer graphic conventions. A value of 0 0 in the file corresponds to the initial point x_o, y_o. Remember, a positive x-value moves right and a negative x-value moves left. A positive y-value moves down and a negative y-value moves up.

For the torus and Klein bottle, your cells should "wrap around" as you might choose a pattern and a initial coordinate that places the pattern outside the size of your screen. For example, if you are modeling a torus edge and a cell is outside the left edge of your screen, place it on the right side of the screen.

Command Line Arguments

Your program will get input from a number of command line arguments. This makes the program much more user-friendly.

usage: life -w 640 -h 480 -r 100 -g 150 -b 200 -s 2 -f glider_106.lif -o 150,200 -e torus

-w width of the screen argument 640, 800, 1024, etc.

-h height of the screen argument 480, 600, 768, etc.

-e type of edge. Values are hedge, torus or klein (strings)

-r the red color value, an integer between [0, 255]

-g the green color value, an integer between [0, 255]

-b the blue color value, an integer between [0, 255]

-s size of the sprite. Valid values are 2, 4, 8, or 16 only. An integer.

-f filename, a life pattern in file format 1.06.

-o x,y the initial x,y coordinate of the pattern found in the file. No space between the x and y.

-H help, print out usage information and a brief description of each option. Note the capitalization, non standard, but -h is already taken.

Extra credit options (see below)

-P filename, a life pattern in file format 1.06/1.05

-Q filename, a life pattern in file format 1.06/1.05

-p x,y the initial coordinate of pattern P

-q x,y the initial coordinate of pattern Q

As Linux is a (mostly) POSIX compliant system, you can use the function `getopt` found in `unistd.h`. This will simplify command line processing. For details, read `man -s3 getopt`. Note: you are using the POSIX version of getopt, not GNU's so no long options.

Make sure you have default values and error check all argument values. For instance, a user might enter 270 for red. You can fix this without causing the program to terminate. The program should work without any options if you have set all options to a default value.

Installing SDL

The details of installing SDL 2 (**make sure your version is 2.x**) depend on which version of Linux you are running. For Ubuntu users:

```
sudo apt-get install libsdl2-dev
```

For other distros, you are on your own.

Or you can install from source, `http://www.libsdl.org/download-2.0.php`.

If you want to learn more about SDL check out the SDL Wiki: `https://wiki.libsdl.org/FrontPage`

gl.c, life.c, life.h, sdl.o, Makefile, conwaylife.tar.gz

You are given the following files in `cse113_life.tar.gz`. You need to use these files in your project. Do not write your own SDL routines for code you turn in. You project will not be graded if you do not use the provided `sdl.o` file and `sdl.h` for the graphics.

`gl.c` – this is the main game. It includes calls to SDL to setup the bitmap so you can draw to the screen. Places are marked as to where you need to insert code. Currently, it has the function `sdl_test`, which lets you test your SDL installation.

`life.c` – This is where you place you game of life functions (initialization, calculation of living and dead cells, etc)

`life.h` – the header file for life.c

`sdl.h` – the header file for sdl.o. You don't need to change or add to this.

`sdl.o` – an object file for 64 bit machines

`sdl32.o` – an object file for 32 bit machines. If you are compiling on a 32 bit machine (run the command `uname -a` to determine what type of system you have). Rename `sdl32.o` to `sdl.o` so the `Makefile` works correctly.

`Makefile` – the makefile

`conwaylife.tar.gz` – a tar file of Life patterns in Life 1.05 and Life 1.06 format.

Once you have installed SDL 2, run `make -k all` to compile. This will create an executable named `life`. Run it and you should see a screen with a object twisting in space.

sdl_test

To test your SDL installation, a function `sdl_test` is provided in the file `gl.c`. The function declaration for `sdl_test` is:

```
void sdl_test(struct sdl_info_t *sdl_info, int m, int n)
```

You can change the m and n parameters, which change what is displayed on the screen.

At the beginning of main, you can change the width, height, sprite_size parameters to change the screen size and the size of the sprite that is displayed. To get a feel for what the parameters do you should try various values, recompile, and see what happens.

Also, notice that line above `sdl_test` is commented out. `SDL_GetTicks()`, returns clock ticks in milliseconds. Taking the modulus of this with a number ≥ 1 will delay the rendering of the bitmap to the screen. Try it.

Once you have `sdl_test` working correctly, you can move onto coding the game of life.

Comments and Doxygen

You will use Doxygen to generate HTML pages for your comments.

You will need to install Doxygen for your system and read the manual at:

`http://www.stack.nl/~dimitri/doxygen/manual.html`

The getting started section tells you how to quickly get up and working with Doxygen. You will create HTML pages. Make sure you set `EXTRACT_ALL` to `YES` in your `Doxyfile.`

Binaries are available for Ubuntu and other distros.

As you have been writing Doxygen style comments (@-tags) since the beginning of class, there is nothing special you need to do as far as commenting the code except to set Doxygen up correctly.

Use the following Doxygen commands for the header of each source code (*.c) file:

```
@file
@brief
@details
@author
@date
@todo
@bug
@remark (if needed)
```

Only the `@remark` section is optional. For functions, make sure you include a brief description and document all parameters and non-void return types.

For the header files (*.h), the Doxygen commands for the header are:

```
@file
@brief
@author
@date
```

Do not document the function declarations in the header file.

Doxygen Milestone

We will not grade the project unless you meet with your grader to go over your Doxygen html files. This gives you a chance to write all the functions you will need and provide in the @details section of your source code files the big picture of your code.

Valgrind and freeing memory

Make sure you free all memory you allocated. To do this you add free calls to the `while (SDL_PollEvent(&event))` loop. Add free calls to all cases that `return(0)`.

To check for memory leaks with Valgrind you need to run it with the following command options.

```
$ valgrind --show-reachable=yes --leak-check=full
  --log-file=out.valgrind ./life
```

This will dump valgrind's output to a file named `out.valgrind`. You want to use a file as the output will be quite large. Open the file and scan the file for errors in function calls you wrote. There will be SDL function errors. This is normal. You are looking for errors in the code you wrote, not in the SDL library.

Milestones

1. Make sure `sdl_test` works correctly.
2. Have your Doxygen html files checked off.
3. Implement the hedge, torus, and Klein bottle (in that order) versions of the game of life **without** command line options. You should be able to manipulate the width, height, color, size, filename, and initial location all by manually changing variables and recompiling. Gliders are a good initial pattern to use for testing.
4. Implement the command line options.

Extra Credit

Add the ability to read in Conway:life's file format Life 1.05. Details are at `http://www.conwaylife.com/wiki/Life_1.05`. Files in 1.05 format have the string "105" in their filename. You just need to add logic to the file command line options to distinguish between "105" and "106" files and process accordingly.

Add the ability to initialize your grid with more than one Life pattern. Add the ability to initialize it with three different patterns from files and at different locations. Add this as a command line option. The patterns could also be the same, two that are the same and one different, and all three are different.

Submission Guidelines

Submit a tarball of your project named
`cse113_firstname_lastname_life.tar.gz`
before the deadline.

Do not forget to meet with your grader to have the various milestones checked off.