CS31 Lab 6: Game of Life!

Table of Contents

1. Due Date

- Due 11:59 PM, Tuesday, Nov. 12

Your partner for this lab is: Lab 6 Partners

CS31 Partner Etiquette and Expectations (https://www.cs.swarthmore.edu/~newhall/cs31/resources/partner_expectations.php).

2. Lab Goals:

- Experience 2D arrays, command line arguments, and file I/O in C.
- Gain additional practice with pointers, dynamic memory allocation, and passing pointers to functions.
- Measure the execution time for parts of your program's execution.
- Gain expertise in gdb and valgrind for debugging C programs
- Using a visualization library
- Designing input files to test correctness of your solution

3. Lab Overview

For this lab, you will implement a program that plays Conway's Game of Life. Conway's Game of Life is an example of discrete event simulation, where a world of entities live, die, or are born based based on their surrounding neighbors. Each time step simulates another round of living or dying.

Your world is represented by a 2-D array of values (0 or 1).

If a grid cell's value is 1, it represents a live object; if it is 0, it represents a dead object.

At each discrete time step, every cell in the 2-D grid gets a new value based on the current value of its eight neighbors:

1. A live cell with zero or one live neighbors dies from loneliness.
2. A live cell with four or more live neighbors dies due to overpopulation.
3. A dead cell with exactly three live neighbors becomes alive.
4. All other cells remain in the same state between rounds.

Your 2-D world should be a **torus**, meaning every cell in the grid has **exactly eight neighbors**, even if it is on the edge of the grid. In the torus world, cells on the edge of the grid have neighbors that wrap around to the opposite edge.

For example, the grid locations marked with an 'x' are the eight neighbors of the grid cell whose value is shown as @.

BASH

```
x  @  x  _  _  _  _
x  x  x  _  _  _  _
_  _  _  _  _  _  _
_  _  _  _  _  _  _
_  _  _  _  _  _  _
_  _  _  _  _  _  _
x  x  x  _  _  _  _
```

Conway's Game of Life (http://en.wikipedia.org/wiki/Conway's_Game_of_Life) description from Wikipedia shows some example patterns (such as Blinker, Toad or Beacon) you can use to test the correctness of your solution.

This video shows one solution to an almost idential lab in action: video (https://www.youtube.com/embed/nzrRhEAWYMQ?rel=0) **NOTE: the program's output is slightly different than the output of your's** in that (1) it is not printing out the Round value at the top of the grid at each step, and (2) the exact string printed out for the number of live cells each round is slightly different from your's (see Section 5.6 for an example of how your Round and Live cells should be printed out).

4. Lab Starting Point Code

4.1. Getting Your Lab 6 Lab Repo

Both you and your partner should clone your Lab repo into your `cs31/labs` subdirectory:

1. get your Lab ssh-URL from the CS31 git org (https://github.swarthmore.edu/cs31f19). The repository to clone is named Lab06-user1-user2, where user1 and user2 are the user names of you and your Lab partner.
2. cd into your `cs31/labs` subdirectory:

   BASH

   ```
   $ cd ~/cs31/labs
   $ pwd
   ```

3. clone your repo

   BASH

   ```
   $ git clone [your Lab06-user1-user2 url]
   $ cd Lab06-user1-user2
   $ ls
   Makefile   gol.c       oscillator_output  test_corners.txt
   README.md  oscillator.txt  test_bigger.txt    test_edges.txt
   ```

4.2. Starting Point Code

The files included in your repo are:

- `Makefile` : builds gol executable file
- `README.md` : some notes to you
- `gol.c` : starting point for GOL lab. Your solution goes here.
- `oscilator.txt` : one example input file to gol (you should create more as you test your solution)
- `oscilator_output` : example output from a working solution (see Section 6 for an example of how to use this to see if your output matches mine)
- `test_corners.txt, test_edges.txt` : empty input files to test gol (you should fill in each one with a test that tests the torus correctness of your solution).
- `test_bigger.txt` : an empty input file for you to fill in to test gol on larger worlds (this is useful to compare timings of run mode's 0 and 1, and for testing run mode 2 on some larger worlds)

5. Lab Details

Your program will take **two command line arguments**. The first is the name of a configuration file that specifies how to initialize the game-playing variables (dimensions of the grid, number of iterations, and initial values of the grid cells). The second is an integer flag (0, 1, or 2) that indicates the game's output mode – i.e. how its output should be displayed. The output mode values mean:

- **Mode 0: run gol with no output** (this mode only outputs final game state statistics, including timing information about how long it took to run)
- **Mode 1: run gol with ascii animation** (this mode also includes timing and outputs final game state statistics)
- **Mode 2: run gol with ParaVis animation** (this mode does not including timing or outputing final game state statistics)

Your program should handle badly formed command lines (e.g. print out an error message and exit).

Here are some examples of valid command line invocations of the program:

```
# run with config values read from file1.txt and does not print the board
# state as runs (does print out total time and live cells at the end):
./gol file1.txt  0

# run with config file file2.txt and do ASCII animation, printing
# the board state after each step (plus total time and live cells at end)
./gol file2.txt  1

# run with config vales from file3.txt with ParaVis animation
# (does NOT print out total time or live cells at end)
./gol file3.txt 2
```

5.1. File Format

The input file format consists of several lines of ASCII text.

- The first three lines specify the grid dimensions and number of iterations.
- The fourth line lists the number of coordinate pairs that will follow.

- The remaining lines specify `i j` (row index, column index) coordinate values to indicate which grid cells should be initialized to 1 (alive) at startup (all others should be 0 / dead):

```
number of grid rows
number of grid cols
number of iterations to simulate
number of coordinate pairs (set each (i, j) value to 1) that come next
i j
i j
...
```

When reading the list of initially live cells from the input file, the cells will be provided as a coordinate pair: i, j. i represents the row number, j represents the column number. **The origin (0,0) is the top left corner**.

You may assume that the grid will have **at least four rows and four columns**. In other words, you do **not** need to worry about weird cases (e.g., 2x2 grid) in which another cell is both the left and right neighbor at the same time.

5.2. Create your own input files

With the starting point code is one example input file you can use (`oscilator.txt`). You should create your own input files by writing text that conforms to the file input format specification. For example, a file with the following contents generates an interesting pattern that starts in the lower left and walk up to upper right of grid:

```
30
30
100
5
29 1
28 2
27 0
27 1
27 2
```

5.3. Add timing

Additionally, you will add timing code to your program to time just the GOL simulation (the timing should not include the board initialization phase of your code).

> The `gettimeofday` function can be used to instrument your program with timers. Call this function before and after the part of the code you want to time and subtract to get the total execution time. **Note that 1 second is 1,000,000 microseconds.**

`gettimeofday` takes as its first argument the address of a struct timeval. For this lab you should provide NULL as the second argument value.

To time a portion of code, call `gettimeofday` around the code you want to time. For example:

```
struct timeval start_time, stop_time;

ret = gettimeofday(&start_time, NULL);
// code you want to time
ret = gettimeofday(&stop_time, NULL);
```

`gettimeofday` sets the time in the the passed `struct timeval *` as the number of seconds plus and number of microseconds since the Epoch (Jan 1, 1970) (the value of the `tv_sec` field is the number of seconds, and the value of

the `tv_usec` is the number of microseconds). The single current time value can be calculated by adding these two fields together. However, these fields are in different units, so be sure to convert one to one to the other before adding them.

See the man page (`man gettimeofday`) for more information.

5.4. Computing Values at Each time step

One problem you will need to solve is how to update each grid cell value at each time step. Because each grid cell's value is based on its neighbor's current value, you cannot update each cell's new value in place (otherwise its neighbors will read the new value and not the current value in computing their new value).

5.5. Correctness Testing

Part of this lab involves you designing gol input files that are desgined to test, debug, and verify the correctness of different aspects of your gol solution. We suggest that you start with some input files to test small, simple worlds.

- With the starting point code are three test files that you need to implement, use for your own testing, and submit with your solution. You are welcome to, and encouraged to, submit more test files than these three, but these are required:
  1. **test_bigger.txt**: this file should test a larger board size than the oscillator file's board, and should have several patterns on the board that should not interfere with each other. It should be a test of the middle of the board (don't worry about edges or corners here). For example, you could have a few still and/or oscillator patterns on the board in locations that they should not affect each other over time steps. You may want to have a pattern that grows too.
  2. **test_edges.txt**: this file should create a board used for testing and verifying the correctness of updating edge cells (cells on the north, or south, or east, or west edge of the grid, but not including the corner cells). We suggest creating a small to medium size board for this one, and it is fine if this file includes a test for just one of the 4 edges (north, south, east, or west). However, you should test all edges with separate test files like the one you submit as this example, and you are welcome to submit these other test files.
  3. **test_corners.txt**: this file should create a board used for testing the correctness of updating one or more of the 4 corner cells. Like the test_edges.txt file, this does not need to include a test for all 4 corners simultaneously; it can test just one of the corners. You should, however, make sure you have additional files that you use to test the other 3 corners for correctness.

In addition to these, you will want to create other input files to test correctness and features of your solution, and also just for fun!

5.6. Example Output

Here is an example of what you might see from different runs. We're printing '@' for live cells and '.' for dead ones because it is easier to see than '0' and '1' characters.

The first example shows the end board configuration from a run that is initialized to run from the oscillator.txt config file. And, the second is the same configuration, but run with 0 as the second parameter (**notice the time difference between the two**):

BASH

```
$ ./gol oscillator.txt 1

Round: 19
 . . . . . . . . . . . . . . . . . . . .
 . . . . . . . . . . . . . . . . . . . .
 . . . . . . . . . . . . . . . . . . . .
 . . . . . . . . . . . . . . . . . . . .
 . . . . . . . . . . . . . . . . . . . .
 . . . . . . . . . . . . . . . . . . . .
 . . . . . . . . . . . . . . . . . . . .
 . . . . . . . . . . . . . . . . . . . .
 . . . . . . . . . @ . . . . . . . . . .
 . . . . . . . . . @ . . . . . . . . . .
 . . . . . . . . . @ . . . . . . . . . .
 . . . . . . . . . . . . . . . . . . . .
 . . . . . . . . . . . . . . . . . . . .
 . . . . . . . . . . . . . . . . . . . .
 . . . . . . . . . . . . . . . . . . . .
 . . . . . . . . . . . . . . . . . . . .
 . . . . . . . . . . . . . . . . . . . .
 . . . . . . . . . . . . . . . . . . . .
 . . . . . . . . . . . . . . . . . . . .
 . . . . . . . . . . . . . . . . . . . .
Live cells: 3


Total time: 2.145 seconds
After 19 rounds on 20x20, number of live cells is 3


$ ./gol oscillator.txt 0
Total time: 0.001 seconds
After 19 rounds on 20x20, number of live cells is 3
```

5.7. Lab Starting Point Code and what you need to add

You will implement your solution in the `gol.c` file.

There is a start of the definition of a `struct gol_data`. You should use the fields we have already defined for you in this struct, but you will also need to add more for your solution.

The `main` function in `gol.c` is almost complete (see the `TODO` comments for just the parts you need to add). It contains the main control flow of your program:

1. Initalize gol game playing state from command line arguments (the input file, and run/output mode values). There is a call from `main` to the function `init_game_data_from_args` that you will need to fill in with this functionality.
2. Call your the gol game playing function `play_gol` in different ways depending on the run mode specified. You need to implement the core gol game playing functionality in the `play_gol` function. However, the main control flow for calling `play_gol` in different ways based on the run mode is already implemented for you in `main`. It does the following:
   a. if `OUTPUT_NONE (mode 0)`: calls `play_gol`, and prints out final statistics (time and total live cells).
   b. if `OUTPUT_ASCII (mode 1)`: calls `play_gol`, and print out final state of the board, and final stats (time and total live cells). It also calls the `print_board` function to print out the game board. `print_board` is almost completely implemented for you (and don't change what it prints out). See the `TODO` notes about the missing part you need to add.
   c. if `OUTPUT_VISI (mode 2)`: calls the ParaVisi library functions to start the animation (`init_pthread_visi`, `get_animation_buffer`, `connect_animation`, and `run_animation`), and does NOT print out final statistics at

the end.

3. Clean up any program state before exit.

There are calls to `printf` to print out final game statitics in `main`. You **SHOULD NOT** change these calls. You will need to add code to `main` to perform timing for run modes (0 and 1). Otherwise, most of the code you add is to functions `play_gol` and `init_game_data_from_args`, and in other addtional game playing and intialization functions you write as part of good modular code design.

6. Lab Requirements

- Use the `main` function's main control flow that is already filled in for you (see the `TODO` and `NOTE` in comments in `gol.c` for more details):
  - Use the fields already defined in the `struct gol_data`. You will also need to add fields to this struct to implement your solution.
  - Implement the main gol game playing functionality in the `play_gol` function that is part of the starting point code.
  - Implement the gol game initialization functionality in the `init_game_data_from_args` function that is part of the starting point code.
  - Complete the `print_board` function.
- Your program must take two command line arguments: the config file name and an integer to determine the output mode:

  0. OUTPUT_NONE mode: Report the time at the end, but don't print the board to the terminal or graphically.
  1. OUTOUT_ASCII mode: Report the time at the end and print the board at each iteration (call `system("clear")` at the start of each round to get the animation part of this).
  2. OUTPUT_VISI: Display gol gram graphically, using ParaVis visualization library.
- When run in OUTPUT_VISI and OUTPUT_ASCII mode, your program will want to contain calls to `usleep` to slow down the animation. When run in `OUTPUT_NONE` mode, your program should not call `usleep`; it should run as fast as possible.
- When running in OUTPUT_ASCII mode (1), you should print the board using the provided print_board function's formatting. That is, you **should NOT make changes to the `fprintf(…)` calls that are already in `gol.c`.**

  You are welcome to add your own `printf()` calls though. You can verify that your output conforms to the expected format by comparing your output against mine with the `diff` command:

  BASH
  ```
  # Run your program and save all the fprintf output to a file named output.txt
  ./gol oscillator.txt 1 2> my_output

  # Compare your output against mine
  diff -u -s my_output.txt oscillator_output
  ```

  If formatted correctly, you'll see: `"Files my_output and oscillator_output are identical"`. If not, you'll see output that compares how they differ. Feel free to contact your instructor(s) for help with output formatting. This bit of the lab is here for ease of grading purposes. When your and my output differ, it is sometimes helpful to view the differences side-by-side in vim. You can do this by running `vimdiff` on the two ouput files:

  BASH
  ```
  vimdiff my_output oscillator_output
  ```

- The space for the GOL board(s) should be dynamically allocated on the heap according to the grid dimensions

specified in the input configuration file. See the example from Week 9 (https://www.cs.swarthmore.edu/~newhall/cs31/f19/WeeklyLabs/wlab09.php) as well as from Chapter 2 of the textbook and Arrays in C (http://web.cs.swarthmore.edu/~newhall/unixhelp/C_arrays.html) for more information. **Use the option that does a single malloc of a contiguous chunk of N*M. Do not use the `int**` (or `char**`) approaches for this assignment**.

- Your gol game playing should play for the specified number of rounds read from the input file. At the end of each round, it should set `total_live` to the number of live cells in the world and perform a visi step — or not — based on the run mode.
- Other than the `total_live` global variable that should be updated each round to the total number of live cells in the world, your solution should not use global variables. Instead, the board(s) and other variables should be passed to the functions that need them using the provided `gol_data` struct. **If you want a function to change the value of an argument, remember you need to pass that argument as a pointer**.
- Your program should have timing code in your solution around the GOL main computation (don't include grid initialization). When run in mode OUTPUT_NONE or OUTPUT_ASCII, your program will print out the total time for the GOL simulation at the end of the simulation. Use `gettimeofday` before and after the section of code you wish to time and subtract to compute the time duration.
- Use the C FILE interface (`fopen`, `fscanf`, `fclose`, etc.) for file I/O. You should detect and handle all errors, calling `exit(1)` if the error is unrecoverable (after printing out an error message, of course).

This means that any time you call a function that returns a value, there should be a check for error return values and they should be handled: do not just assume that every function call and every C library or system call is always successful!

- Your solution should be well-commented and apply good top-down design, dividing functionality into functions. **Consider the big steps: initializing the game board, updating one round of play, printing the game board, etc.**
  - `main` represents the high-level steps with calls to high-level functions that perform each of the big steps.
  - These functions in turn should use helper functions to implement well-defined pieces of functionality.
  - As a rule of thumb, a function should not be longer than 100 - 150 lines. There are exceptions, but if you are writing a function that is getting really long, break it up into several parts, each implemented by a separate helper function.
- For full credit, your solution should be correct, robust, and free of valgrind errors (when run in modes OUTPUT_ASCII or OUTPUT_NONE). You are NOT responsible for running valgrind in the OUTPUT_VISI mode (the ParaVisi visualization mode)
- All `TODO` comments in the starting point code should be removed in your final submission. These are notes to you for what you need to add into the starting point code and where you need to add it.

7. Tips

- Implement and test incrementally!
- Use gdb as you incrementally implement and test to find and fix bugs as you go. Use valgrind as you go to catch memory access errors as you make them.
- As you test and debug, you may find it useful to use config files with small numbers of iterations and to comment out the call to system("clear") so you can examine the results of every intermediate iteration.
- When reading the list of initially live cells from the input file, the cells will be provided as a coordinate pair: i, j. i represents the row number, j represents the column number. **The origin (0,0) is the top left corner**.

- Recall that passing pointers to a function allow you to to "return" more than one value from a function, since that function's execution can have "side effects" that modify the value that's pointed to.
- When debugging your code, it is helpful to run your program in OUTPUUT_ASCII mode (in mode 1), so that you can see what your program is doing. You also may want to use input config files with a small number of iterations for testing, and different config files for testing specific functionality. You can also comment out the calls to `system("clear");` to get rid of the animation part of the ASCII animation. Doing this will allow you to scroll up the terminal window to to see the board after each iteration.
- You do not need to read all of a file's contents at once. Your program can read part of the file contents, do something else, and then later read more of the file contents. If you do this, you may need to pass the FILE * handle to the open file to several functions, so think about where you want to open the file in your code to make this easy to do.
- `usleep` : pauses the program for some number of micro seconds. This is useful in print mode to slow down your program after each step so that you can see what it is doing. About .2 seconds is a good amount to sleep:

```
usleep(100000);  // sleep for 100,000 micro seconds (0.1 seconds)
```

- `system("clear");` clears the text on the terminal so that the next output is at the top of the window (useful for printing the grid at each timestep to the same window location).
- `atoi` converts a string to an integer. For example, `int x = atoi("1234")` , assigns x the int value 1234. This is useful in your command line parsing functions for converting command line arguments that are read in as strings to their numeric values. See the man page for `atoi` for other similar functions to convert stings to other numeric types (`man atoi`).
- CNTRL-C will kill your program if it seems to be stuck in an infinite loop.
- ParaVis display: One thing to note about the ParaVisi display is that it displays coordiante (0,0) at the bottom left of the visualiztion, and coordinate (rows-1, 0) at the top left. This means you will need to perform some calculation on the row index into the color3 array to "flip" the visualization horrizontally to match your view of it (so the ascii and the ParaVis display the world in the same way for the same input file).

8. Extra Challenge

After completing a well-designed, well-commented, correct solution to the required parts of this assignment, as an extra challenge you can see how your program compares speed-wise, and see if you can tune your program to beat my implementation: Speed Challenge

The extra challenge isn't worth any points, but is just for fun. Even if you do not do the extra challange, you may want to read it over to learn a bit more about running timed experiments, and about gcc compiler optimization flags.

9. Submitting

Here are the commands to submit your solution (from one of you or your partner's `cs31/labs/Lab06-user1-user2` directory:

BASH

```
$ git add gol.c *.txt
$ git commit -m "Lab 6 completed"
$ git push
```

And of course, if you want to submit any nifty starting point world config files you come up with, please do:

BASH

```
$ git add coolworld.txt
$ git commit -m "cool world"
$ git push
```

If you have difficulty pushing your changes, see the "Troubleshooting" section and "can't push" sections at the end of the Using Git for CS31 Labs (https://www.cs.swarthmore.edu/~newhall/cs31/resources/labsetup.php) page. And for more information and help with using git, see the git help page (https://www.cs.swarthmore.edu/help/git/).

10. Handy Resources

- Class piazza page (https://piazza.com/swarthmore/fall2019/cpsc31/) for questions and answers about lab assignment
- **C Debugging**: gdb Guide (https://www.cs.swarthmore.edu/~newhall/unixhelp/howto_gdb.php), Valgrind Guide (https://www.cs.swarthmore.edu/~newhall/unixhelp/valgrind.php), Chapter 3 of the textbook (https://diveintosystems.cs.swarthmore.edu)
- **C Programming**: C programming resources (https://www.cs.swarthmore.edu/~newhall/unixlinks.html#Clang), C code style guide (https://www.cs.swarthmore.edu/~newhall/unixhelp/c_codestyle.html)
- Week 9 in-lab examples (https://www.cs.swarthmore.edu/~newhall/cs31/f19/WeeklyLabs/wlab09.php): using ParaVisi, command line arguments, file I/O.
- man and Manual pages (https://www.cs.swarthmore.edu/~newhall/unixhelp/man.html) documentation for libraries and commands
- **Git**: CS31 github org (https://github.swarthmore.edu/cs31f19), Git help (https://www.cs.swarthmore.edu/help/git/), Using Git for CS31 Labs (https://www.cs.swarthmore.edu/~newhall/cs31/resources/labsetup.php)
- **Misc**: Some Useful Unix Commands (https://www.cs.swarthmore.edu/~newhall/unixhelp/howto_unixCommands.html), vi (and vim) quick reference (https://www.cs.swarthmore.edu/~newhall/unixhelp/viquickref.pdf), make and makefiles (https://www.cs.swarthmore.edu/~newhall/unixhelp/howto_makefiles.html)

Last updated 2019-11-12 17:34:13 EST