ECE École d'Ingénieurs Paris, France
Department of Embedded Systems

Advanced C Programming
by **Aakash SONI**

Project – The Embedded Game of Life

Introduction

Imagine a world where tiny organisms live on a checkerboard grid. Each square can either be empty (dead) or contain a living organism (alive). At each tick of the clock (i.e. one generation) all organisms simultaneously follow the same simple rules: some survive, some die, and new ones are born.

This is **Conway's Game of Life**, invented in 1970. Despite its simplicity, the game has fascinated mathematicians, computer scientists, and hobbyists for decades because of the surprising complexity it produces. Patterns that look random can stabilize into still shapes, start oscillating forever, or even crawl across the grid like little creatures.

In this lab, you will bring this world to life in C. But here's the twist: you are not just writing a playful simulation. You are programming as if you were targeting a small **embedded system**. Think of a microcontroller with a tiny amount of memory and a strict timing budget. This means you must write your code with discipline: **memory-efficient data structures, predictable execution times, and precise control over behavior**.

For more information about the game, refer to Wikipedia at
http://en.wikipedia.org/wiki/Conway's_Game_of_Life
and youtube at
https://www.youtube.com/watch?v=CgOcEZinQ2I

Step 1 – Understanding the rules

The rules of the Game of Life, as described by Conway, are simple:

1. A dead cell with exactly **three live neighbors** becomes alive.
2. A live cell with **two or three live neighbors** survives.
3. All other cells die or remain dead.

> **Think about it:** Why do you think "3 neighbors" is the magic number for birth? What happens if a cell has too many neighbors? What real-world phenomena might this remind you of (e.g., overpopulation, loneliness)?

Example 1: Stability

Take a 2×2 block of live cells:

```
. . . .
. * * .
. * * .
. . . .
```

(Alive = *, Dead = .)

What happens next? Count carefully. Each cell in the block has exactly 3 neighbors, so all survive. No dead cell outside the block has exactly 3 neighbors, so no new cells are born. This block remains **unchanged forever**. It's called a *still life*.

Example 2: Oscillation

Now try a line of three cells:

```
. . . . .
. * * * .
. . . . .
```

(Alive = *, Dead = .)

After one generation, the line flips vertically. After another, it flips back horizontally. This pattern repeats endlessly. This is called a *blinker*.

> **Question:** How would you detect an oscillator like the blinker in your program? Can you think of a way to prove that the pattern has entered a cycle?

Step 2 – Representing the world

If you were coding this on a powerful desktop with gigabytes of RAM, you might store your world as a 2D array of `int` or `char`. But in embedded systems, every byte counts.

In this project, you must represent the grid as a **bit-packed array**:

- Each cell uses exactly one bit.
- That means you can fit 64 cells into a single `uint64_t`.

> **Hint:** Practice thinking in binary. To set the 5th cell in a row, you might shift a `1` left by 5 and OR it into place. To test if a cell is alive, you might AND with a bitmask.

You also have a **strict memory budget of 64 KiB** for the entire world. This forces you to calculate: *Given a width and height, how many bits will I need?*

> **Think about it:** What happens if the user requests a world that is too large for your memory budget? Should your program fail gracefully?

Step 3 – Evolving the world

At the heart of your program is the function that advances the world by one generation. To do this, you will need:

- `get_cell_state(x, y)` – Is this cell alive or dead?
- `num_neighbors(x, y)` – How many neighbors are alive?
- `get_next_state(x, y)` – Apply the rules of Life.
- `next_generation()` – Compute the entire next world from the current one.

> **Hint:** Don't overwrite the world while you're still using it to compute neighbors! Use a second buffer (a "shadow world") to hold the next generation.

Expected outcome:

- Try evolving a blinker. You should see it flip orientation every generation.
- Try evolving a block. It should remain unchanged.

Step 4 – Boundaries of the world

The Game of Life was originally imagined on an infinite grid. But your grid will have edges. What happens at the edges? You must implement **four possible boundary conditions**:

1. **Edge** – Anything outside the grid is dead.
2. **Toroidal** – The grid "wraps around" like Pac-Man. The left edge connects to the right, and the top connects to the bottom.
3. **Mirror** – Coordinates beyond the edge reflect back into the grid.
4. **Alive rim** – Outside the grid, all cells are considered alive.

Question: How do you think a glider behaves differently on each boundary type? Try predicting before you simulate.

Step 5 – Real-time constraints

Embedded systems often have to meet deadlines. For example, a screen refresh must complete in time, or the user sees glitches.

To simulate this, your program must evolve one generation (for example a **320×240 grid) in about 16.7 ms** (to match 60 Hz refresh rate).

You should measure and report:

- The **average time** per generation.
- The **worst-case time**.
- The **jitter** (difference between worst and average).

Hint: Use `clock_gettime()` to measure. Run 1000 generations and collect statistics.

Expected outcome:
Your report should contain these numbers. If your program is too slow or too fast, think about how you could optimize ?

Step 6 – Input and Output

Input

Your program must load an initial configuration from a text file,

where `*` = alive and `.` = dead.

Example (`glider.txt`):

```
 *
. .
  *
..
***
```

Output

- Display each generation on the terminal **in place** using ASCII (i.e., show one grid that updates on screen rather than scrolling many screens).
- At the end, save the final grid into a text file.

Expected outcome:
Start with a glider and run with toroidal boundaries. Watch as the glider walks diagonally across the grid and reappears on the other side.

Step 7 – Command line interface

Your program should be executed like this:

```
./lifegame --width 320 --height 240 --gens 500 --boundary torus  --in glider.txt --out result.txt --target-hz 60
```

- `--width`, `--height`: dimensions of the grid.
- `--gens`: how many generations to simulate.
- `--boundary`: edge, torus, mirror, or rim.
- `--in`: input file.
- `--out`: output file.
- `--target-hz`: target refresh rate.

Step 8 (Bonus) – Parallelism

For students who want to go further:

Modern CPUs have multiple cores. Can you use them to make the Game of Life faster?

- Add a `--threads N` option.
- Split the grid into N horizontal slices.
- Each thread computes its slice.

  But beware: neighbors at the slice borders still matter!

Question: How will you make sure two threads agree on the values at the borders? What happens if one thread updates before the other?

Expected outcome:
On a multicore machine, your parallel version should run faster than the single-threaded version. However, this may not be the case if the threading overhead is larger than the computation speed up.

Deliverables

- Code:
  - Code files : `.c` and `.h`
  - For simplified compilation : `Makefile`.
  - Do not forget to write relevant comments in your code.

- **Report** (4–8 pages):

  - Explain your design choices.
  - Present your timing results (step 5).
  - Show examples of interesting patterns.
  - **Do not copy paste your code in the report.**
  - Explain the implemented logic through pseudo-code, algorithm and/or flow diagrams.
  - Explain the project outcome and observations along with the screenshots.
  - Do not forget to answer the questions introduced in different steps.
  - Provide the commands required to use your Makefile and to execute the output file.
  - Reflect on what you learned about thinking "like an embedded engineer."

Important

You are welcome to discuss the project with other students, provided that all work turned in must be your own. If you do discuss your work with other students or you use AI tools, don't forget to give them a credit by listing their names in the acknowledgement part of your report.
In summary, when you are turning in an assignment with your name on it, what you turn in must be your work, and yours alone. Cheating will not be tolerated.

Closing thoughts

The Game of Life teaches us that simple rules can lead to unexpected complexity. In this project, you will not only see these patterns emerge, but also learn to **think under constraints**, just like embedded engineers must do every day.

Watch closely: sometimes your patterns will grow, sometimes they will vanish, and sometimes they will surprise you by repeating endlessly. Ask yourself at each step: *Why did this happen? What rule caused it? How would the outcome change under a different boundary condition?*

That curiosity is the real goal of this lab.

Happy Coding