Documentation

Conway's Game of Life

Implementation

Our implementation follows an object-oriented approach with two main classes: *World Class* (`World.h, World.cpp`), which manages the cellular automaton's state and evolution logic, and *CLI Class* (`Cli.h, Cli.cpp`), which handles user interaction and command parsing.

File Organization

Our project follows a modular C++ structure with separation of headers, implementation files and a CMake build system.

```
game-of-life/
|-- CMakeLists.txt              # CMake configuration
|-- main.cpp                    # Application entry point
|-- World.h                     # World class declaration
|-- World.cpp                   # World class implementation
|-- Cli.h                       # Cli class declaration
|-- Cli.cpp                     # Cli class implementation
\-- build/                      # Build directory (created)
```

Build Instructions

1. **Configure project**:

   ```
   mkdir build
   cd build
   cmake ..
   ```

2. **Compile**:

   ```
   make
   ```

3. **Run**:

```
./main
```

4. **Testing different optimization levels**:

```
cmake -DCMAKE_CXX_FLAGS="-O2" ..
-O0  # No optimization (debugging)
-O1  # Basic optimization
-O2  # More optimization
-O3  # Aggressive optimization
```

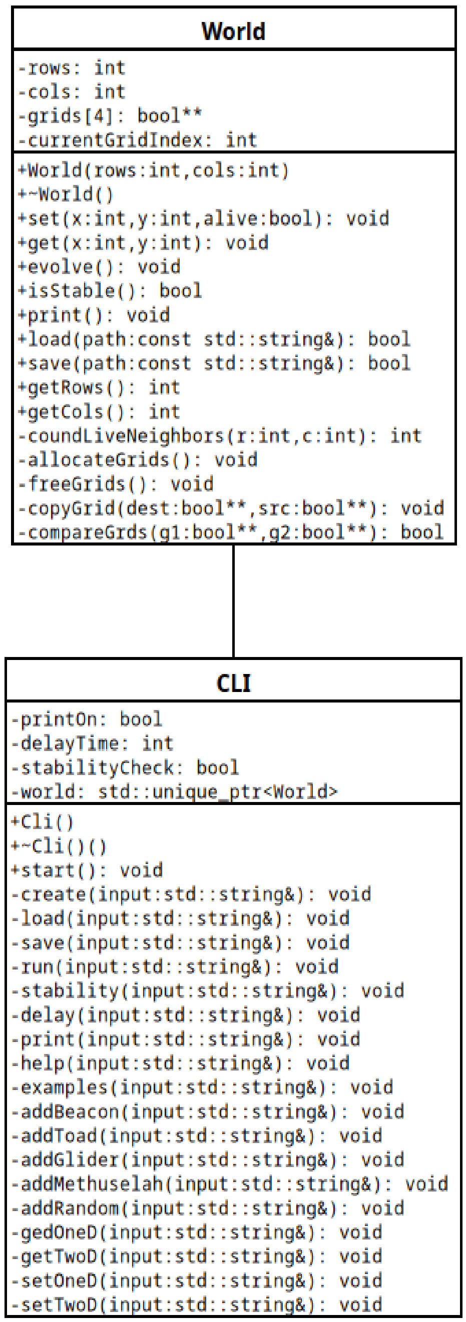

Figure 1: UML Class Diagram of Conway's Game of Life Implementation

Starting cell of multi-cell figures at the grid position (x,y)

All multi-cell patterns use the top-left corner as their anchor point (x, y), marked with a red X in the diagrams below. When placing patterns using commands like `glider 5 5`, the pattern is positioned such that its top-left cell aligns with grid coordinates $(5, 5)$. The toroidal grid behavior ensures that patterns wrap around the world boundaries. For example, placing a *glider* at $(width - 1, height - 1)$ will cause parts of the pattern to appear at the opposite edges of the grid.

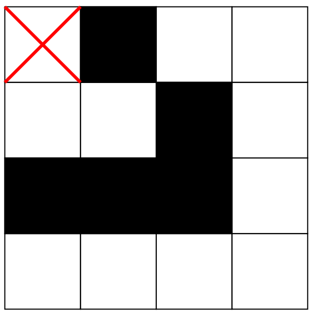

Figure 2: Glider

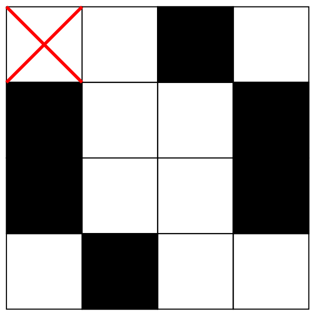

Figure 3: Toad

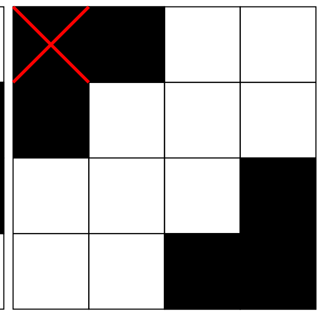

Figure 4: Beacon

Methuselah

We chose one of the smallest methuselahs, the **R-pentomino**, as our pattern. A small configuration (only 5 cells) that takes over 1,100 generations to stabilize. The anchor point (red X) marks the top-left position (x, y) where the pattern is placed.

- **Pattern**: R-pentomino
- **Cells**: 5 live cells in 3×3 bounding box
- **Evolution**: Complex long-term behavior (1,103 generations)
- **Anchor**: Top-left corner at command coordinates

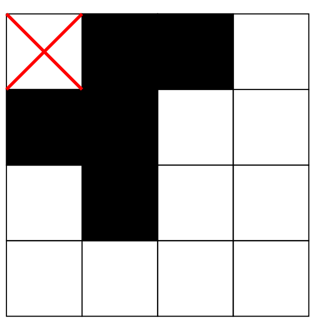

Figure 5: Methuselah

CLI Features

Help System

Our CLI implementation includes a comprehensive help system accessible via the `help` or `h` commands. This system provides users with complete command reference and usage guidance.

Help Command Output Structure:

```
==================================================
COMMAND REFERENCE
==================================================

WORLD MANAGEMENT:
create <width> <height>    Create new world
load <unix-path>           Load world from file
save <unix-path>           Save world to file

SIMULATION CONTROL:
run <generations>          Run simulation
print <0|1>                Toggle live display
delay <milliseconds>       Delay between generations
stability <0|1>            Auto-stop when stable

CELL OPERATIONS:
set <x> <y> <0|1>          Set cell state
set <position> <0|1>       Set cell by 1D index
get <x> <y>                Get cell state
get <position>             Get cell by 1D index

PATTERNS:
glider <x> <y>             Add glider pattern
toad <x> <y>               Add toad pattern
beacon <x> <y>             Add beacon pattern
methuselah <x> <y>         Add methuselah pattern
random <count>             Add random patterns

APPLICATION:
help, h                    Command reference
example, ex                Usage examples
quit, q, exit, :q          Exit program
==================================================
```

Examples System

The `examples` or `ex` command provides practical usage scenarios to help users understand how to combine commands effectively.

Examples Command Output Structure:

```
==================================================
USAGE EXAMPLES
==================================================

EXAMPLE 1: Basic simulation with visualization
create 30 30       # Create 30x30 world
glider 5 5         # Add a glider
toad 15 10         # Add a toad oscillator
print 1            # Enable live display
delay 100          # 100ms between frames
run 50             # Run 50 generations

EXAMPLE 2: Fast simulation with stability check
create 40 40
random 8           # Add 8 random patterns
print 0            # Disable display for speed
stability 1        # Stop when world becomes stable
run 1000           # Run up to 1000 generations

EXAMPLE 3: Manual cell creation
create 20 20
set 5 5 1          # Create custom pattern
set 6 6 1
set 7 5 1
set 7 6 1
set 7 7 1
print 1
run 20

EXAMPLE 4: Save and load workflow
create 25 25
glider 0 0
beacon 10 10
save my_pattern.txt
# ... later ...
load my_pattern.txt
run 100
==================================================
```

Exercise 1.4: Optimization Level and Compilation Settings

The `print` to console and `is_stable` checks were disabled. The simulation ran with the file `p67_snark_loop.txt` for $n = 2000$ generations.

Results

Setting	Time [ms]
`-O0`	73
`-O1`	73
`-O2`	74
`-O3`	75
`Debug`	446
`Release`	75

Discussion

Optimization levels `-O0` – `-O3` show negligible performance differences (73–75 ms). The workload likely benefits little from compiler optimizations.

In contrast, switching from `DEBUG` (446 ms) to `RELEASE` (75 ms) greatly improves speed, as `DEBUG` mode includes extra checks and disables most optimizations.

Conclusion

Optimization level has minimal effect, while compilation mode strongly influences performance.

Exercise 1.5: Simulation Time per Grid Size

The `print` to console functionality and `is_stable` check were disabled. The simulation was compiled in `RELEASE` mode with optimization level `-O3`. A blank world was used with a methuselah pattern placed at coordinates $(0, 0)$. Each simulation ran for $n = 100$ generations.

Results

Grid Size (x, y)	Time [ms]
(10, 10)	0
(20, 20)	0
(100, 100)	4
(1000, 1000)	355
(10000, 10000)	35514

Discussion

The simulation time increases rapidly with grid size. Small grids complete almost instantly, while large grids require significantly more time. This trend suggests an approximately quadratic scaling with the number of cells ($O(x \cdot y)$), as each generation updates the entire grid.

Plot

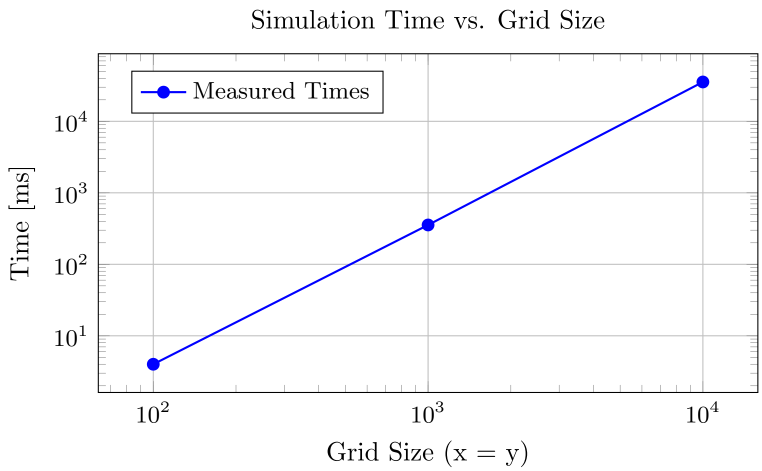

Figure: Simulation time as a function of grid size (log–log scale).