Conway's Game of Life in Mips

Jonathan Lyttle

November 27, 2017

1 Program Overview

This program is an implementation of John Conway's Game of Life in Mips, utilizing the Mars bitmap display for visualization. The game is a "zero-player" simulation game consisting of a two-dimensional grid of cells sized 64 by 64, which can either be living or dead, here represented by white and black respectively. The first generation consists of the input to the algorithm and its output becomes the next generation, with births and deaths happening simultaneously. This is then run recursively for all future generations. The rules of the game are:

1. A living cell with fewer than two neighbors dies by underpopulation.
2. A living cell with two or three neighbors lives on to the next generation.
3. A living cell with more than three neighbors dies by overpopulation.
4. A dead cell with exactly three neighbors becomes living by reproduction.

This implementation allows for manually advancing each generation or setting a key to automatically advance with a delay. It features a fade effect between generations to highlight the cells that have died from the previous generation. There are 3 built in pattern presets and a random pattern. The presets are effective in showing the emergent patterns of the game while the random pattern shows the colonies that can typically form.

2 Program Description

The program's first task is to load constants set in the data section of the program into unchanging registers, including the grid width (64), the color chosen for dead cells (black) and living cells (white). These were expressed as constants in the data section rather than loaded as immediates in the program in order to debug and easily

change colors. The program prompts for pattern choice and displays an appropriate error if the user enters an incorrect number.

The array for tracking births and deaths is initialized with a size of 4096 elements (64x64) and every element is set to dead, the color black in hexadecimal (0x000000). Next, depending on the pattern chosen, the program will write out the correct living cells pixel by pixel for the preset (or in the case of random, have a 20% chance of spawning for each cell) to the display. The display memory is treated as an array which is written to after calculating births and deaths. The program writes out the current generation to the console and waits for the user to press space to advance a generation, switch to a different pattern, or press p to auto advance.

When the user advances a generation, the program jumps to the Game of Life section where it enters a loop to process each pixel in the display array. It begins by running the Game of Life Algorithm function with the absolute position of the current pixel as an argument and returns the number of neighbors for that pixel. The algorithm must perform checks to see if the current position is on an edge or a corner so it doesn't try to access memory out of bounds. Since the algorithm uses an absolute position rather than a coordinate system, checking for edges, corners and neighbors is achieved by shifting by the grid size transformed to seek to the correct location. If the algorithm is able to check the desired cell and it is white, it falls through and increments the neighbor counter. If it cannot check a certain edge or side it skips to the next check.

When this algorithm finishes, the result is brought back to the loop where it decides whether the corresponding location in the births and deaths array is living or dead. Dead cells are set as white minus grey (0xEEEEEE) and living are set to white. Finally the DisplayGeneration function is called which writes the values from the births and deaths array to the display unconditionally. After writing to the display, a loop is started which decrements the value of a dying cell down to black, achieving a fade effect.

Several helper functions were developed to help with drawing presets and writing to display memory, including the Draw and GetDisplayAddress functions which were created with the help of another Mips project creating a snake game using the bitmap display (`https://github.com/Misto423/Assembly-Snake`).

3 Testing

Unit testing was done in order to iteratively develop the Game of Life algorithm, display function, and helper functions. Although the finished program assumes the size of the working area is effectively 64 by 64, in order to test the algorithm and game loop it was necessary to test with a smaller grid, which was chosen to be 3 by 3. This ensured that any logic errors were able to be dealt with swiftly. Since the births and deaths array initialized with a smaller size of 9 elements, tracing step by step in Mars was quicker and solved many off-by-one errors and missed branch statements.

For the algorithm, the first case tested was a single pixel in the middle of the grid, with the expected result of the pixel being set as dead (the color black) in births and deaths array and written back to the display. After getting the display function working, the width was increased to the final 64 pixels and a 4 pixel square was placed in the center, with the expected result of the square staying static. Next, the corners and edges of the display were tested again with the same square and with single pixels to make sure they were working correctly. Finally, a pattern was recreated from an online simulation (`https://bitstorm.org/gameoflife/`), the 10 Cell Row pattern, which ended up becoming a preset for the final program.

Tracing issues with the display function was not as straightforward as tracing the algorithm. The display could not be tested in its entirety at the same time as the algorithm because of the nature of the display. Since the display is continuous memory with 64 pixels being the cutoff point for displaying the next line, the 3 by 3 array simply showed up as 9 horizontal pixels. Although this helped in some regard with assuring the algorithm was working, it was not sufficient for testing whether the display function would work over the entirety of the 4096 pixel grid. The best course of action was then making sure the display function worked correctly at small sizes and gradually working up to the full grid size, where the only remaining possible issue would be having an off-by-one error. The bitmap display was tested at 9 pixels, 64 pixels or one full row, half the grid, and finally the full grid. After the display function was working, a loop was added later to transition dying cells from grey to black, achieving a fade effect.

After testing had been completed on the algorithm and display, presets and random permutations were added to finalize the program. The presets were again recreated from the online simulation (`https://bitstorm.org/gameoflife/`) and were tested in manual generation mode against the simulation. The random permutation pattern was tested with 60%, 50%, 40% and 30% spawn rates before settling on 20%. This rate was decided based on the large number of cells that were immediately dying on the first generation for the other rates.

A planned feature of the program was the ability to take a user-specified width as the baseline for the working area; however, the options in the bitmap display only provide support for widths that are multiples of 64 (64, 128, 256, 512 and 1024), and any number provided in-between would have been displayed off-center. Another option provided in the bitmap display is a width and length option of each pixel, where increasing those values decrease the effective working space. Because of this, the program assumes a width of 64. This provided for consistent testing and results.

Lastly a challenge arose in general because of the timing of each generation being set to a certain speed and cannot be faster. This set speed is caused by the fade effect as well as the processing of the alogorithm before writing back to the display. The speed was increased somewhat after the decision to use a set 64 by 64 grid, however it could not be increased any further as-is. The speed is acceptable given the amount of processing and was tested on multiple machines for consistency.