Antoni Wellisz
Juan Collar
PHYS 25000
12 October 2023

Homework 2: Conway's Game of Life

Introduction

The following files are submitted:

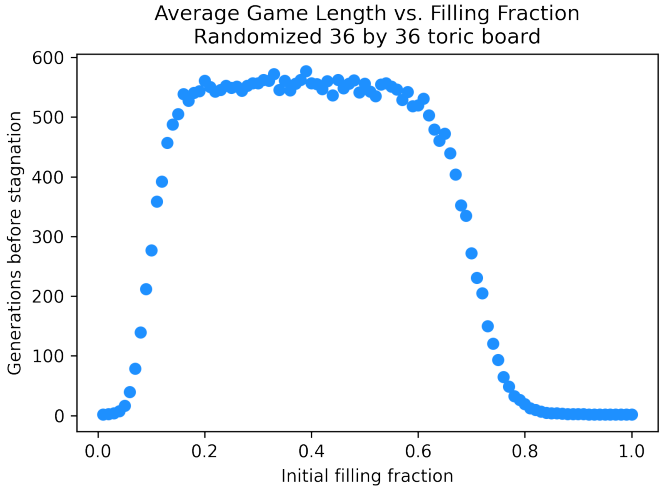
1. `wellisz_antoni_hw2_life.c` is the full-fledged Game of Life with several additional (user-interactive!) features.
2. `wellisz_antoni_hw2_colors.c` is the same as the standard game of life but is more colorful. See details below.
3. `wellisz_antoni_hw2_statistics.c` is used to test how different starting conditions affect the outcome of the game.
4. `wellisz_antoni_hw2_cylic.c` is entirely different and implements cyclic cellular automata in two dimensions as described by Griffeath et al.[1]

It would probably be best programming practice to link several `.c` and `.h` files to properly organize the codebase, but for the purposes of this assignment, each file above is entirely self-contained and only has to be compiled individually to function. (There is therefore a lot of repeated code.)

The code has been tested only on the CS department Linux machines. I'm not 100% sure if the board display will work properly on other systems.

Some Statistics

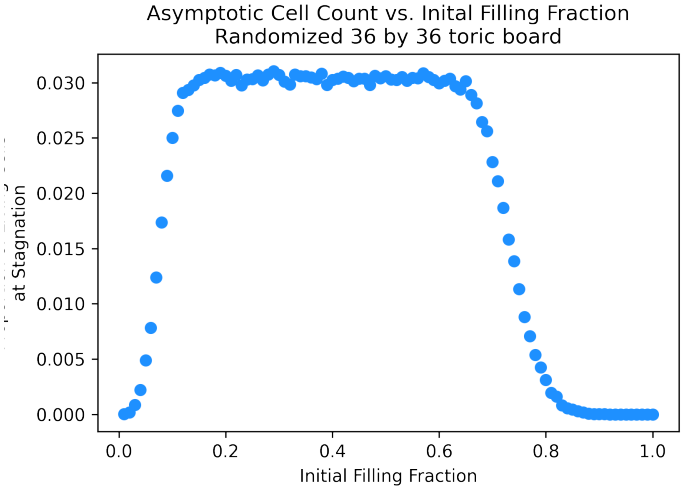
I ran 2,000 iterations of the game of life with random starting conditions (with different seeds) for each filling fraction from 1% to 99% for a total of 200,000 games. The average longevity of the game at each filling fraction is presented below.



As you can see, for a rather large range (about 20% to 60% filling fraction), the average number of generations before the game terminates or cycles is approximately 550 with very low variance.

[1] Fisch, Robert, Janko Gravner, and David Griffeath. Cyclic cellular automata in two dimensions. Birkhäuser Boston, 1991.

Below is presented the asymptotic filling fraction (i.e. the number of living cells on the board at the moment the game stagnates divided by the total number of cells) vs. the initial filling fraction. A similar density function is found.



This result agrees with previous work by Bagnoli et al.[2], who produced the following plot for a 256 by 256 toric board:

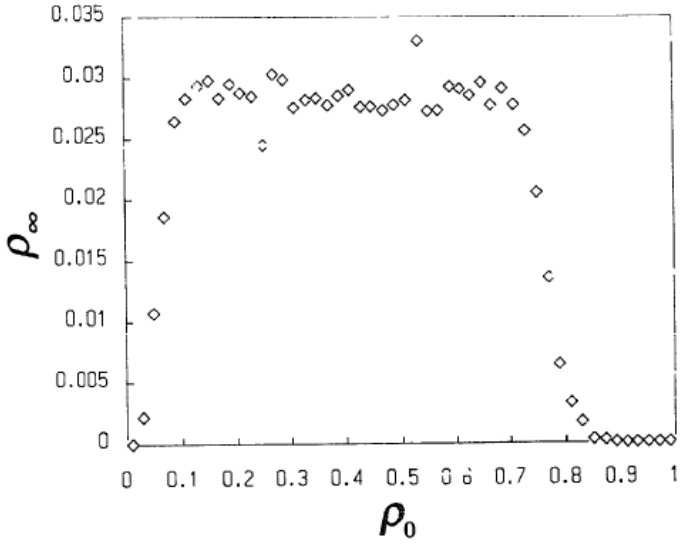
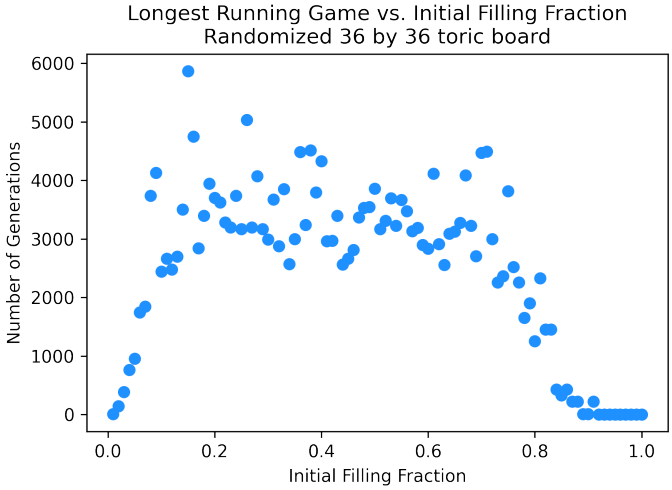


Fig. 3. Asymptotic density of live cells (ρ_∞) versus the initial density (ρ_0).

[2]Bagnoli, Franco, Raúl Rechtman, and Stefano Ruffo. "Some facts of life." Physica A: Statistical Mechanics and its Applications 171, no. 2 (1991): 249-264.

The longest game observed lasted 5865 generations at a filling fraction of just 14%. It is almost certainly true that a longer game is possible on a board of this size, but this analysis is limited by only carrying out 200,000 total games.



Toroidal Board

Implementing a toroidal board involves some modular arithmetic: taking the cell modulo the size of the board means that if one of the nearest neighbors is out of the array bounds, the index will 'wrap around'.

```c
int count_neighbors(int state[SIZE][SIZE], int i, int j) {
    int num_neighbors = 0;

    for(int di = -1; di <= 1; di++) {
        for(int dj = -1; dj <= 1; dj++) {
            // Skip the cell itself
            if(di == 0 && dj == 0) continue;
            // Handling toric boundary
            int ni = (i + di + SIZE) % SIZE;
            int nj = (j + dj + SIZE) % SIZE;
            num_neighbors += state[ni][nj];
        }
    }
    return num_neighbors;
}
```

Stagnation Detection

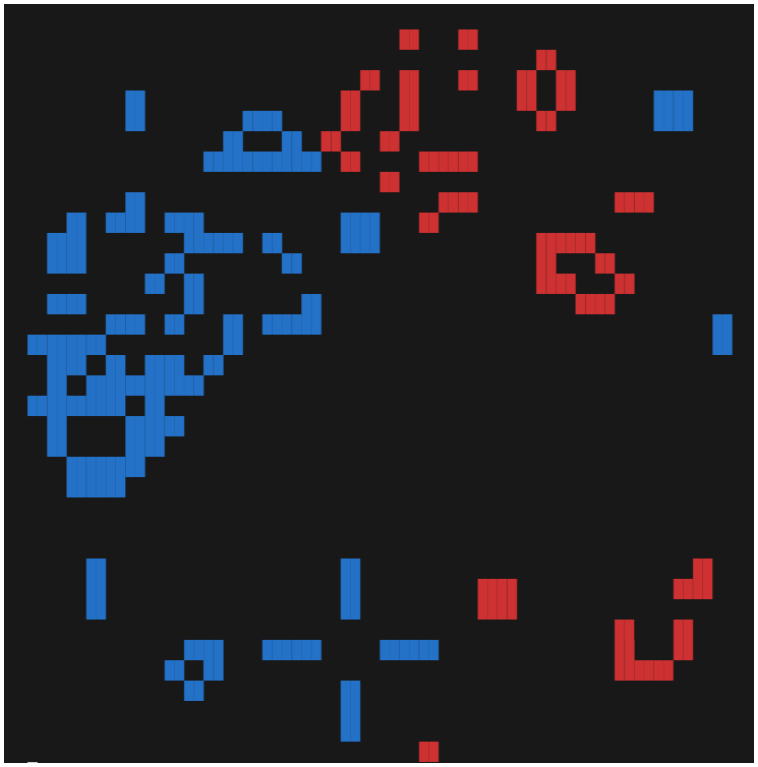
Because the Game of Life is purely deterministic, if a given state is ever reached twice in a game, you are guaranteed to have identified a cycle. For the purposes of this project, I am treating a cycle with a period of less than 2048 generations as stagnation (I chose this number arbitrarily—this takes 16 MB of RAM to store and checking against 2048 hashes of previous states is fast enough). To detect cycles, I used a very basic hash function and stored the hash of each game state in an array. When the program is first run, I initialize a `SIZE` by `SIZE` hash keys array with random 64-bit integer values:

```c
    uint64_t hash_keys[SIZE][SIZE];
    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++) {
            // rand_U64() returns a random uint64_t
            hash_keys[i][j] = rand_U64();
        }
    }
```

For each game state in the main `while` loop, a new hash key is initialized to zero. When iterating through every cell of the grid, if the (i, j) cell is 'alive', that cell's unique random number is hashed into the key by `hashkey ^= hash_keys[i][j]`. (In the case of two competing organisms—see below—one 'tribe' is hashed in as above, and the other tribe is hashed in inversely as `hashkey ^= ~(hash_keys[i][j])`. This likely makes no difference, but there may be edge cases producing false positives for cycles that occur with the same cells but for different tribes). The final key is then added into the next open index of the `hash_table` array. Then, for each generation of the game, I loop through the `hash_table` array to determine whether the current game state has occurred before. (Unfortunately this is not a real hash lookup table—it is likely significantly more efficient to implement a proper hash table, but performance is good enough for the purposes of this assignment).

Tribal Warfare

I implemented an option to play the Game of Life with two competing tribes: the Yangs and the Kohms. In the screenshot below, the Yangs are in blue and the Kohms in red:



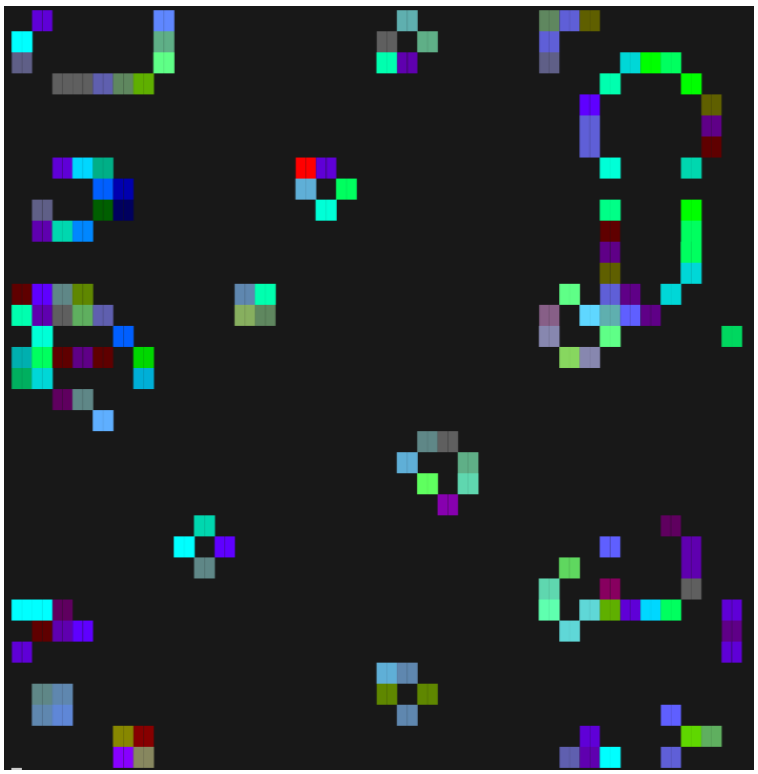
Yangs are encoded in the state array as `+1` and Kohms as `-1`. Dead cells are `0`. When counting neighbors, then, one simply adds the value of the eight surrounding cells (see Toroidal Board above—the code is exactly the same for a single tribe and for two tribes). A negative number of neighbors indicates more surrounding Kohms than Yangs; a positive number indicates more Yangs than Kohms. With this in mind, a slightly altered ruleset is implemented:

1. Each living cell with one or no neighbors dies. This includes the case, for example, of having three friendly neighbors and three hostile neighbors—the two tribes 'cancel out' and the cell dies.
2. Each living cell with four or more neighbors dies.
3. Each living cell with two or three neighbors survives.

4. Each dead/empty cell with `+3` neighbors is populated with a new Yang; each empty cell with `-3` neighbors is populated with a Kohm. For example, if an empty cell is surrounded by 4 Yangs (`+4`) and 1 Kohm (`-1`), it will be populated with a Yang.

Colorful Life

In `wellisz_antoni_hw2_color.c`, the game board is initialized with random colors. Any new cell created in a formerly empty cell takes on the average color of its three parent cells. There are 256 standard terminal colors, as described here. The result, perhaps unsurprisingly, is that the game board quickly gets dominated by cyans, browns, and purples, which occupy the middle region of this 8-bit colorspace.



Other Small Features

FEN String Input

It is common in chess to use what is called Forsyth-Edwards Notation (FEN) to describe a particular board position. A FEN string defines a particular game position in a single line of ASCII characters, encoding information about piece positions, en passant opportunities, castling permissions, etc. I adapted a simpler version of this notation to more easily enter the initial game state without having to manually enter in coordinate pairs (i, j) for every living cell.

In this version of FEN strings, 'y' denotes a Yang cell, 'k' denotes a Kohm cell, and a number 1 to 9 denotes that number of empty cells. A '/' indicates that the rest of the row is empty. For example, for a glider in the top left corner of a 36 by 36 board, the corresponding FEN string would be `1y/2y/yyy`.

```
void parse_fen(char *fen, int (*state)[SIZE]) {
    assert(fen != NULL);
    int count, cell;
    int i = 0;
    int j = 0;
    while(*fen) {
```

```
        cell = 0;
        count = 1;
        // Switch on current character
        switch (*fen) {
            case 'y': cell = YANG; break;
            case 'k': cell = KOHM; break;
            case '1':case '2':case '3':case '4':case '5':case '6':case '7':case '8':case '9'
    :
                // subtract ASCII values from characters to get integer val
                count = *fen - '0';
                break;
            case '/': // at a new row
                j = 0; // go to start of row (1st column)
                i++; // increase row
                fen++;
                continue;
            default: // something wrong
                fprintf(stderr, "FEN parse error!\n");
                exit(0);
        }
        for (int k = 0; k < count; k++) {
            if (cell != 0) state[i][j] = cell;
            j++; // increase column
        }
        // Read next character
        fen++;
    }
}
```

Built-in Presets

The addition of FEN string parsing to populate the game board also makes it easy to include user-selectable presets. E.g., the initial condition defined by the FEN string `///////////////95yy/95yy1y1yy/991yy` (several `/` in a row indicates several empty rows on the game board) leads to very interesting symmetrical patterns with a duration of 302 generations. Even such a sparse initial condition would be a pain to enter manually, one cell at a time.

Cyclic Cellular Automata

Cyclic cellular automata were developed by David Griffeath. As described in this paper, I implemented two-dimensional cyclic cellular automata which follows one simple rule: each type ζ eats *every* neighboring type that it can. The game board is initialized with random cells of (in this case) seven types. The types are organized into a cyclic 'food chain' of sorts, where each type can 'eat' exactly one other type. Stated more formally, this game is determined recursively by the rule

$$\begin{aligned} \zeta_{t+1}(x) &= \zeta_t(y) \quad \text{if } \zeta_t(y) - \zeta_t(x) = 1 \bmod N \text{ for some } y \text{ such that } \|y - x\| = 1 \\ &= \zeta_t(x) \quad \text{otherwise} \end{aligned}$$

where $\zeta(x)$ is some cell and $\zeta(y)$ is any neighboring cell. This implementation uses the von Neumann neighborhood to detect nearest neighbors—each cell has exactly 4 nearest neighbors (above, below, left, right).

Here, as in most implementations, the seven types are represented by seven colors: red, orange, yellow, green, blue, indigo, and violet (where red eats orange, orange eats yellow, etc., and violet eats red). When the game starts, the board is entirely randomized, and the vast majority of cells have very little food (very few neighbors they can eat and turn into their own color). However, a few active areas become what Griffeath describes as 'critical droplets', nuclei of future growth.

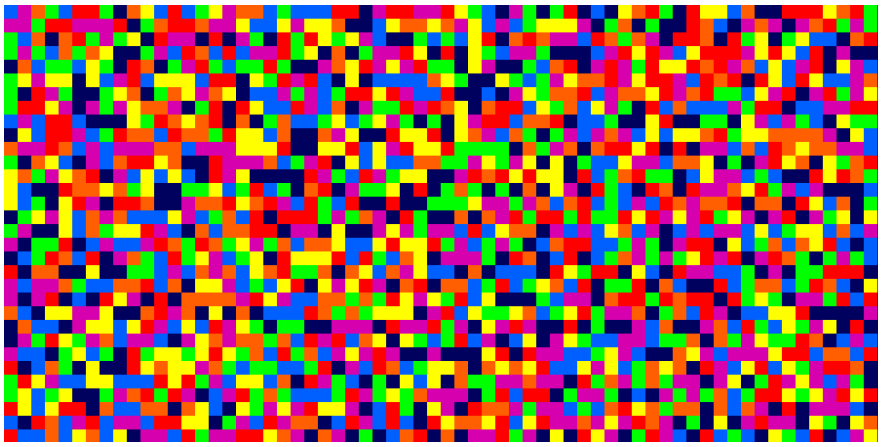


Figure 1: The very first frame of the game with an entirely randomized board.

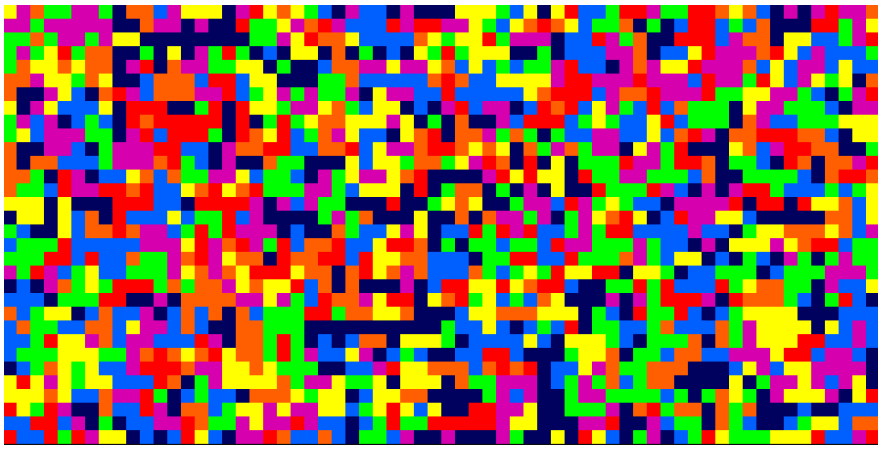


Figure 2: Generation 3: droplets begin to form.

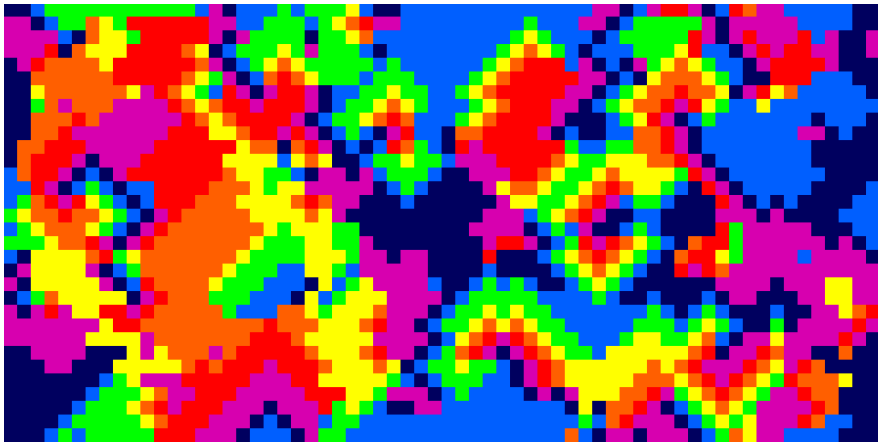


Figure 3: After a few generations, large domains of like type begin to appear.

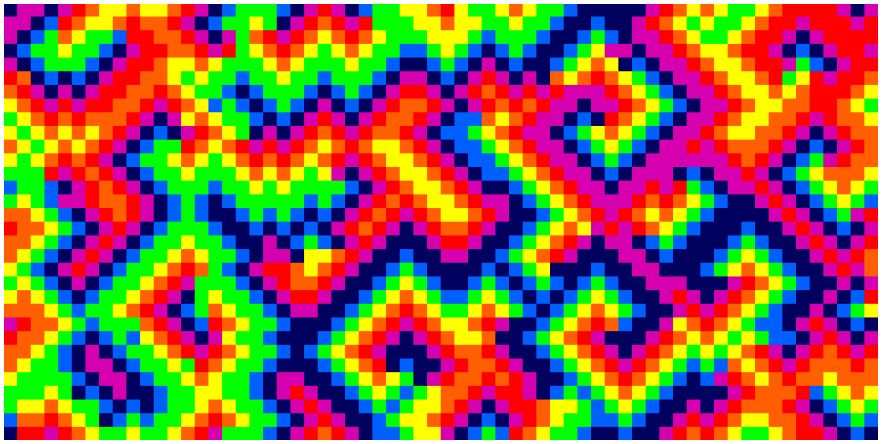


Figure 4: Quickly after that, the system reaches a metastable state of rapidly repeating spirals.

von Neumann and Moore Neighborhoods

The above game only considers the four nearest neighbors that directly border the cell, and each cell only requires one neighboring cell of the next type in the food chain in order to be eaten. The von Neumann neighborhood can be generalized to an arbitrary Manhattan distance d away from the center P:

		2		
	2	1	2	
2	1	P	1	2
	2	1	2	
		2		

The von Neumann neighborhood contains $d^2 + (d+1)^2$ total cells. Additionally, we can change the rules slightly: instead of only requiring one neighbor to be up one level in the food chain for a given cell to be eaten, we can count the number of neighbors (within the von Neumann neighborhood of a cell) and only allow the cell to be eaten if the number of neighbors that can eat it exceeds a certain threshold. The below figures demonstrate the behavior for different neighborhood sizes and cell count thresholds.

We can also count cells within the Moore neighborhood around a cell, which is simply a square centered at

the cell containing $(2d+1)^2-1$ cells. (For $d=1$, this includes the 8 nearest neighbors of the standard Game of Life). This also leads to some pretty patterns. In general, a balance must be struck: if the neighborhood is large, the threshold cannot be too high, or large regions will take over and cover the entire board; if the neighborhood is large, the threshold can't be too small, or no large patterns will form. The same is true for a small neighborhood with a large threshold.

Figure 5: Midgame behavior for von Neumann neighborhood distance = 2, threshold = 2

Figure 6: Asymptotic behavior for von Neumann neighborhood distance = 2, threshold = 2

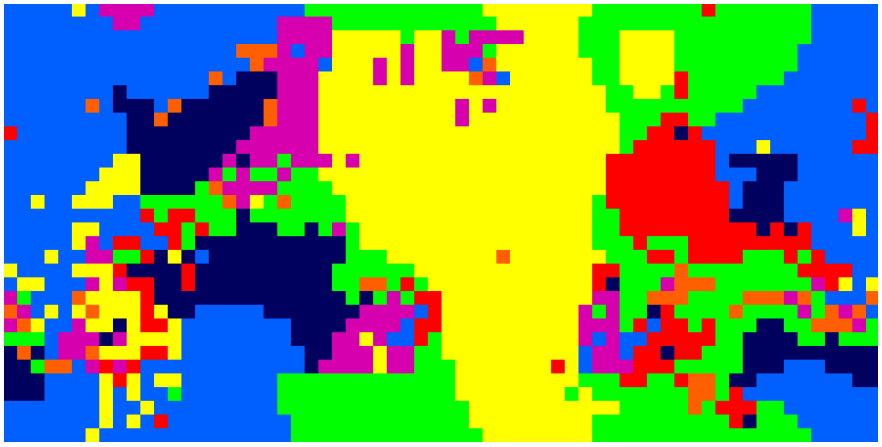


Figure 7: Midgame behavior for von Neumann neighborhood distance = 2, threshold = 3. The structure of the board is not as cyclic and tends to consist of several slowly growing regions.

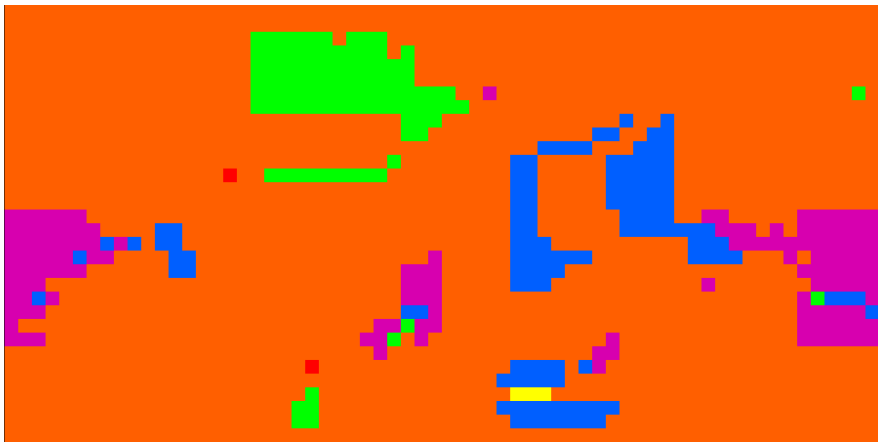


Figure 8: Asymptotic behavior for von Neumann neighborhood distance = 2, threshold = 3. Eventually, the large regions coalesce and the game fully stagnates.

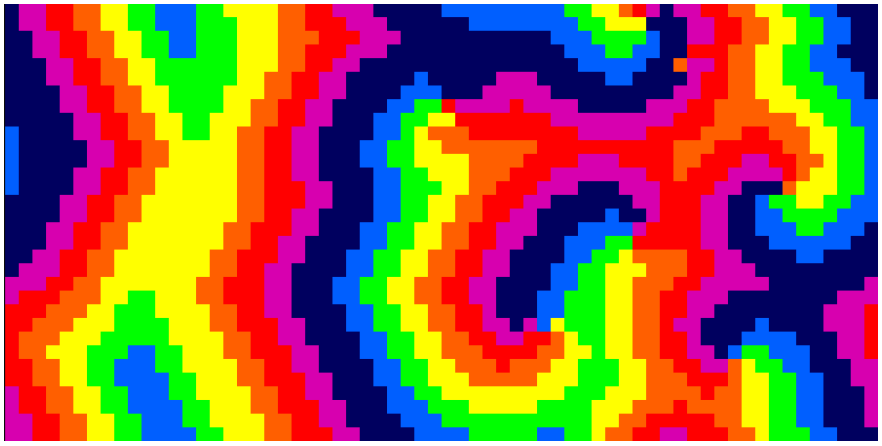


Figure 9: Asymptotic behavior for neighborhood distance = 3, threshold = 3

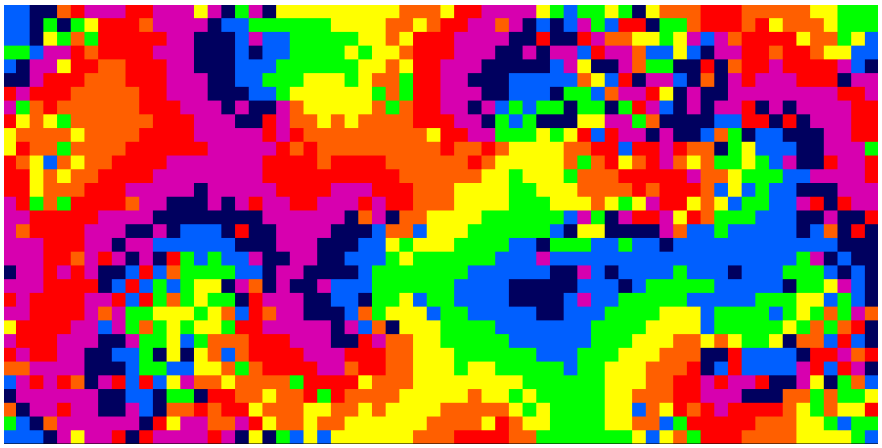


Figure 10: Asymptotic behavior for von Neumann neighborhood distance = 4, threshold = 4. This board is metastable (i.e. structurally it stays the same but the colors (types) cycle rapidly), but large-scale patterns are not as clean for these settings.

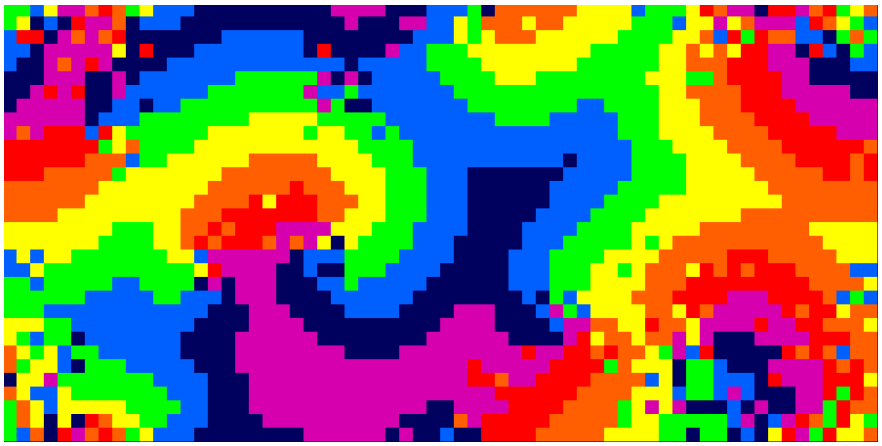


Figure 11: Asymptotic behavior for neighborhood distance = 5, threshold = 6. A von Neumann neighborhood up to distance 5 contains $5^2 + (5+1)^2 = 61$ cells, so a threshold of 5 results in a meaningless mess since any cell is essentially guaranteed to have 5 eatable neighbors. A threshold of 6 creates and retains some large-scale structure.

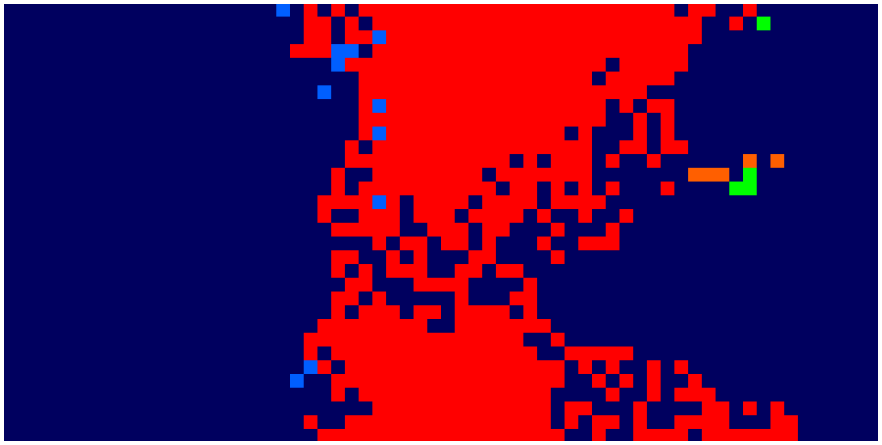


Figure 12: Asymptotic behavior for von Neumann neighborhood distance = 6, threshold = 10.

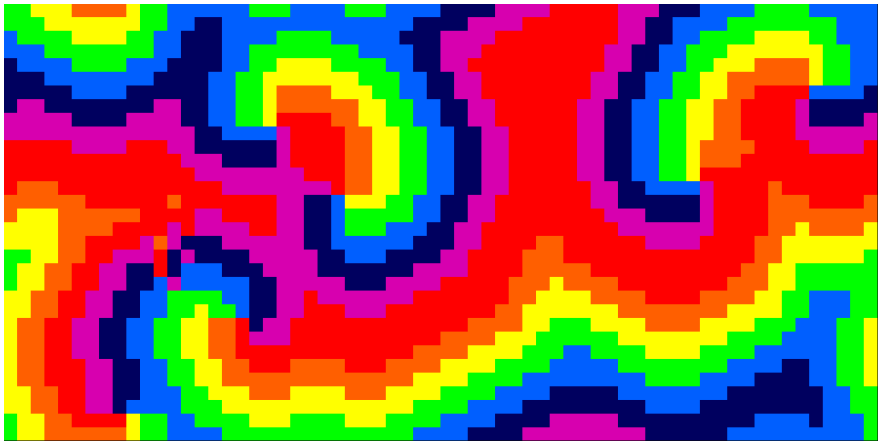


Figure 13: Asymptotic behavior for Moore neighborhood distance =2, threshold = 3.

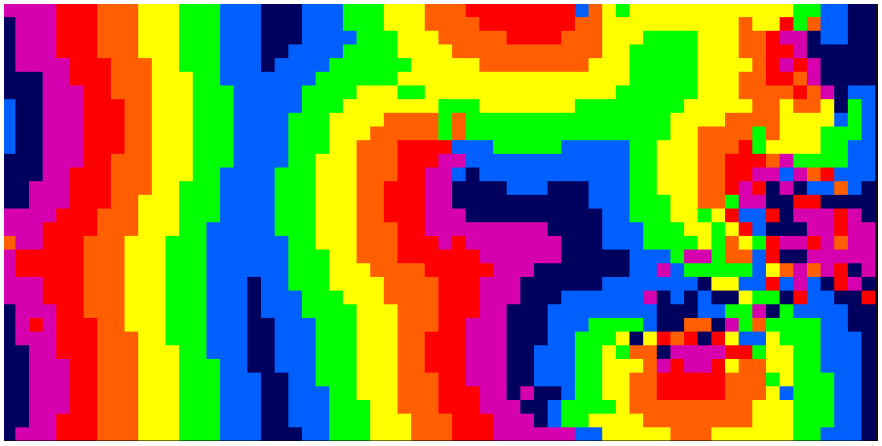


Figure 14: Asymptotic behavior for Moore neighborhood distance = 3, threshold = 5