```
ca_plot (generic function with 1 method)
```

Cellular automata

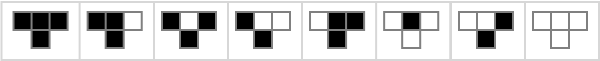
- As powerful computational engines.
- As discrete dynamical system simulators.
- As conceptual vehicles for studying pattern formation and complexity.
- As original models of fundamental physics.

- Complex Behavior
- Emergence
- Self-organization
- Autopoesis

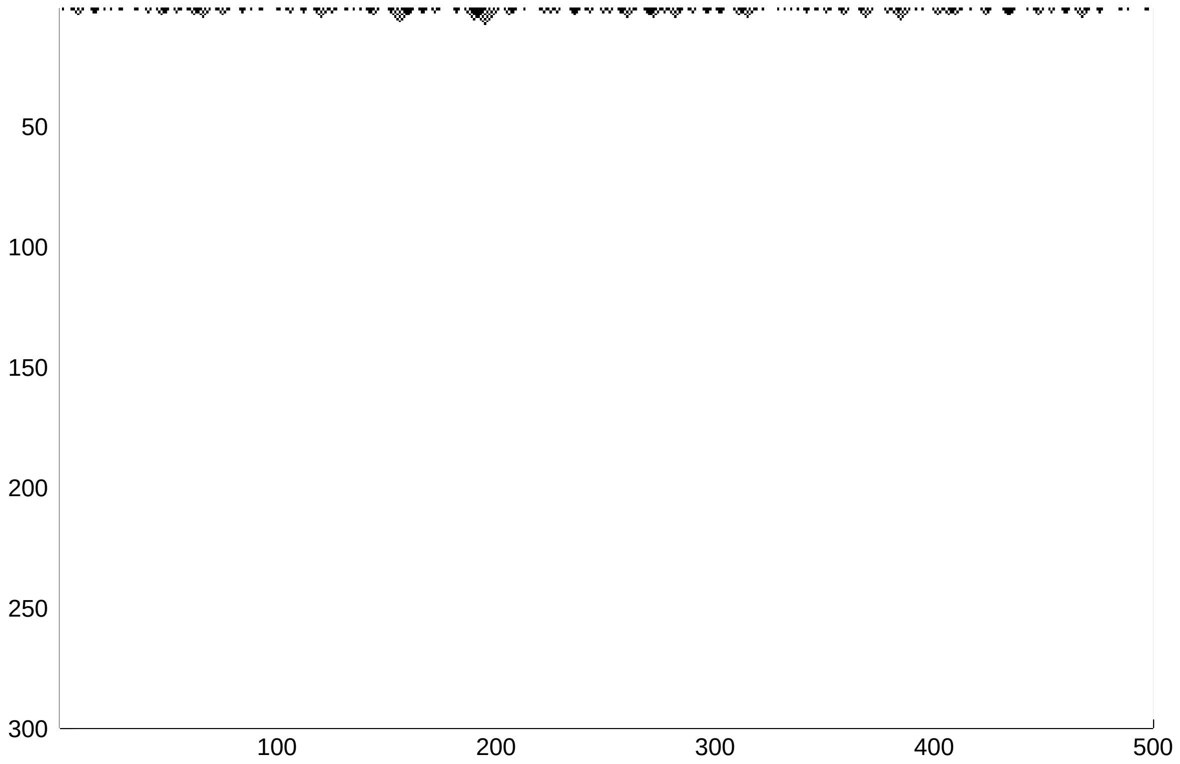
Emergence

- As collective self-organisation.
- As unprogrammed functionality.
- As interactive complexity.
- As incompressible unfolding.

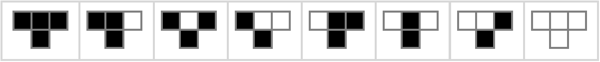
Class1 rules leading to homogenous states, all cells stably ending up with the same value:



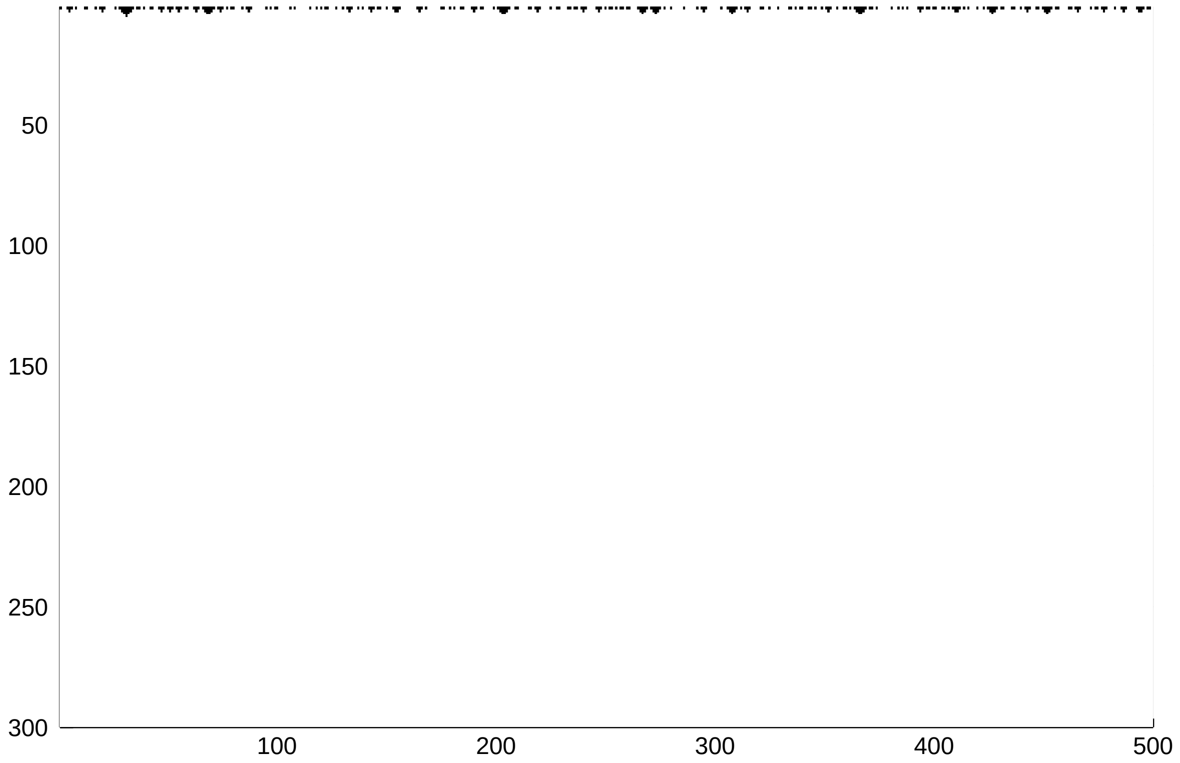
■ 1 | □ 0



```
ca_plot(250, 500, 300)
```

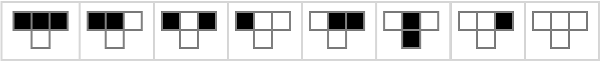


■ 1 | □ 0

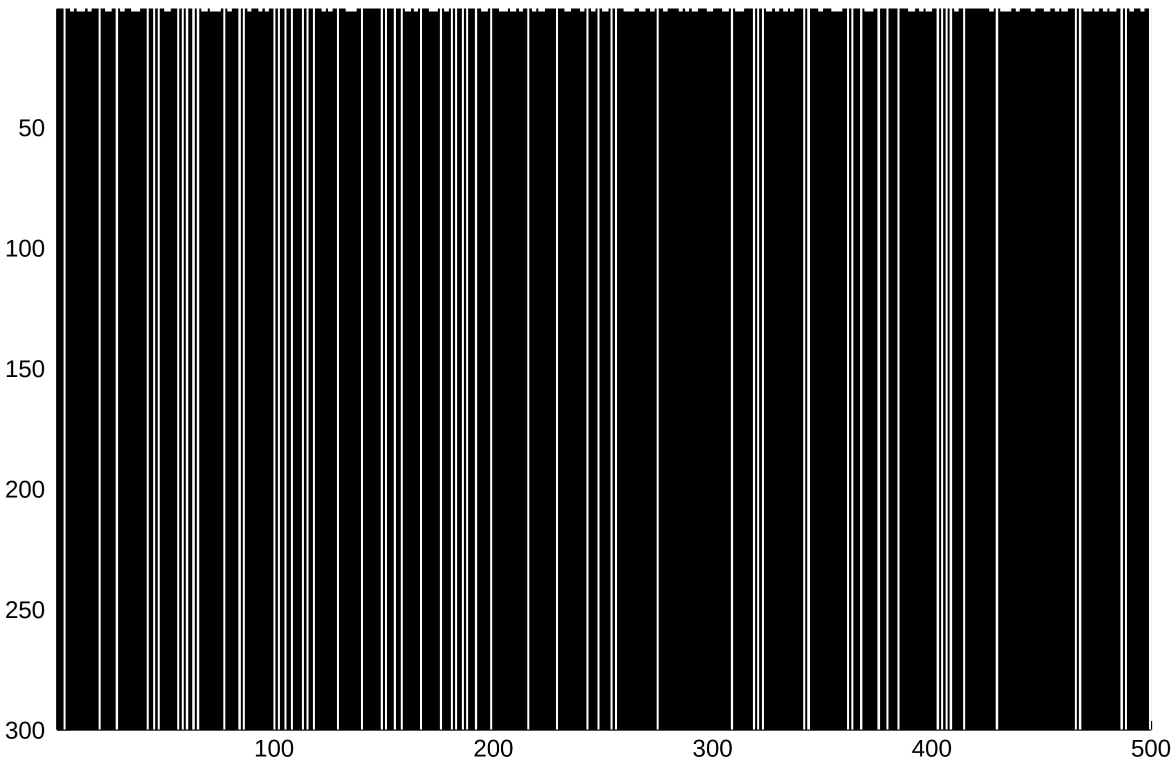


```
ca_plot(254, 500, 300)
```

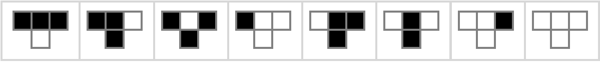
Class2 rules leading to stable structures or simple periodic patterns:



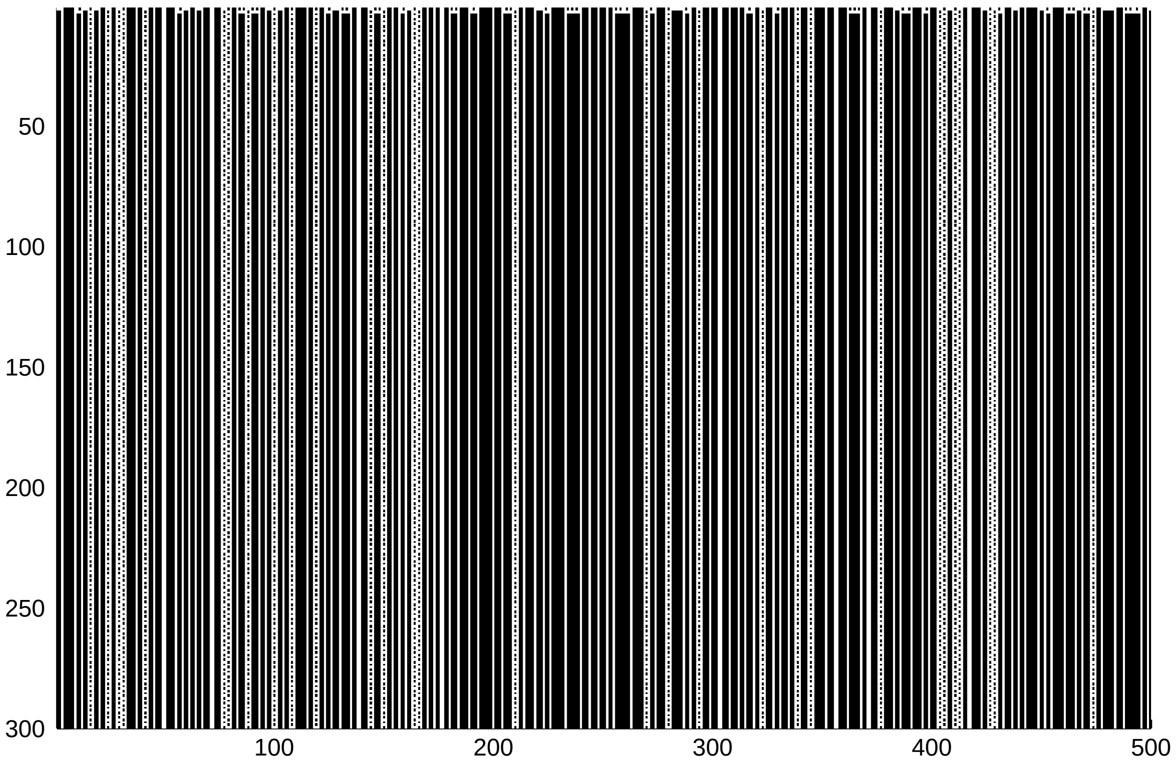
■ 1 | □ 0



```
ca_plot(4, 500, 300)
```

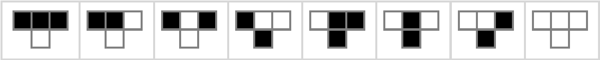


■ 1 | □ 0

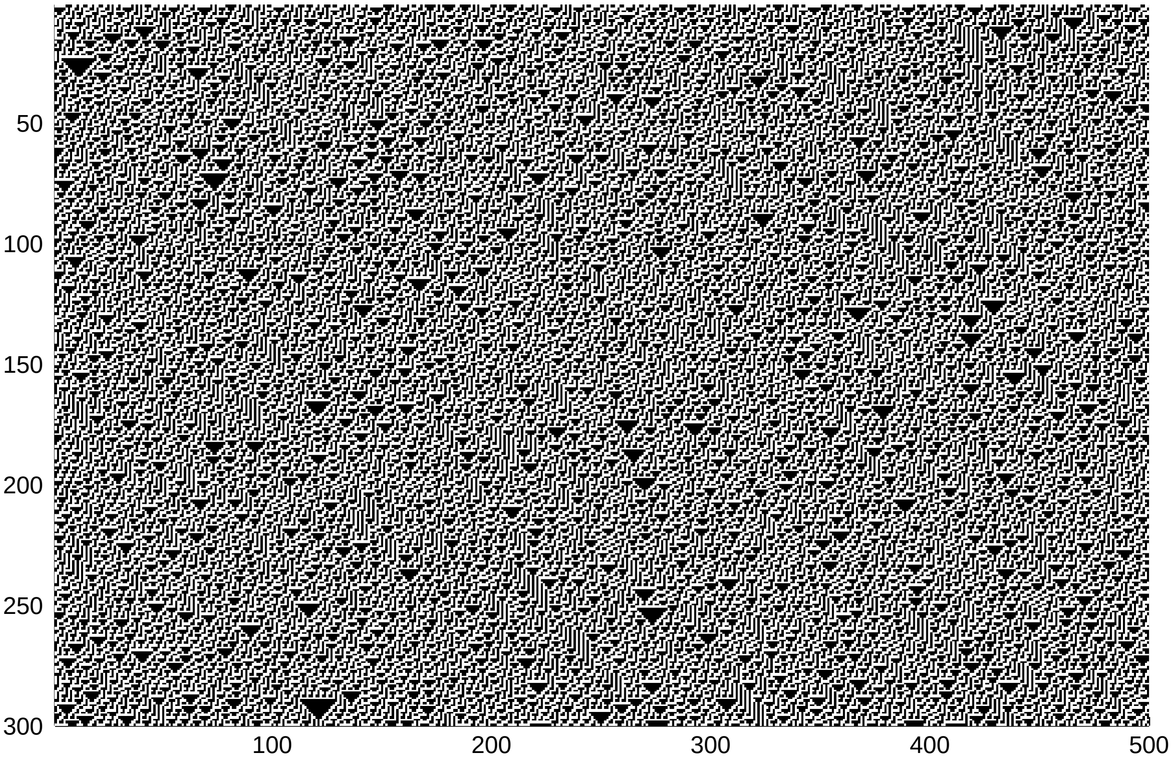


```
ca_plot(108, 500, 300)
```

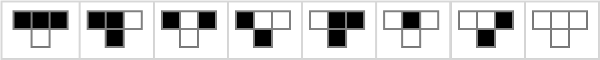
Class3 rules leading to seemingly chaotic, non-periodic behavior:



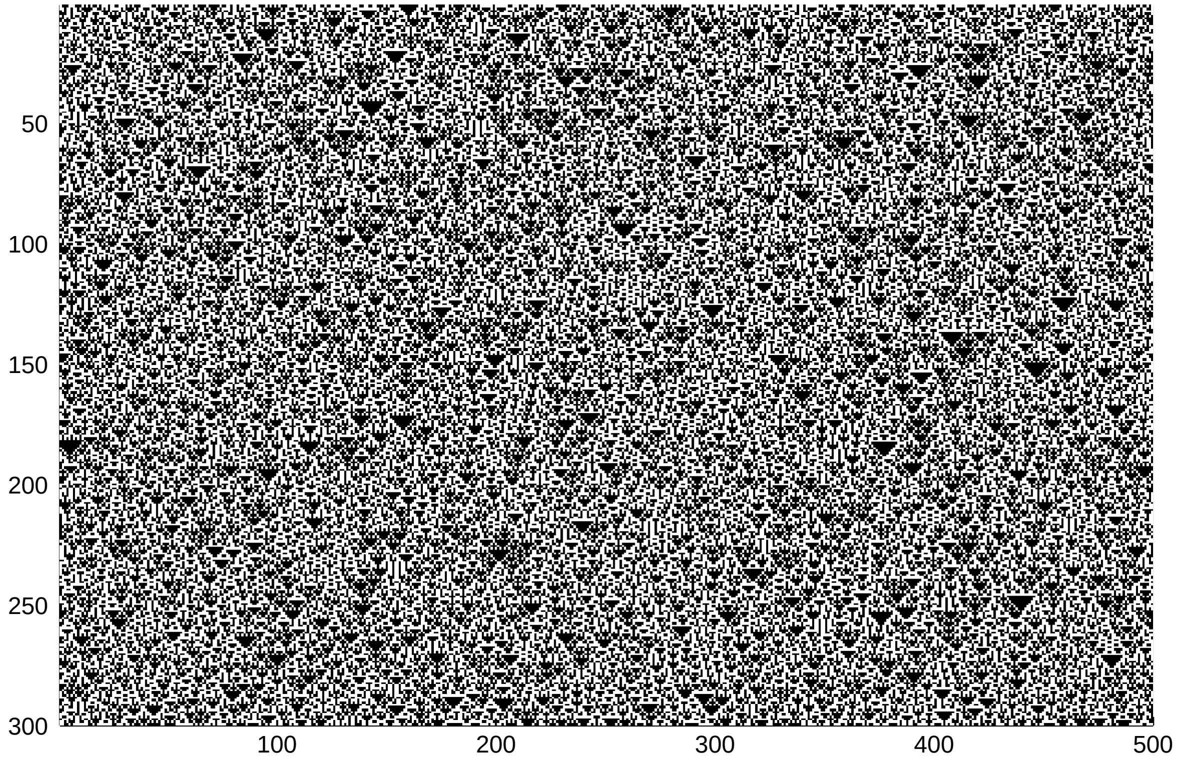
■ 1 | □ 0



```
ca_plot(30, 500, 300)
```

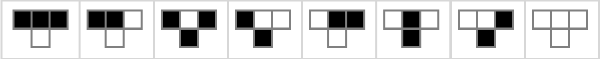


■ 1 | □ 0

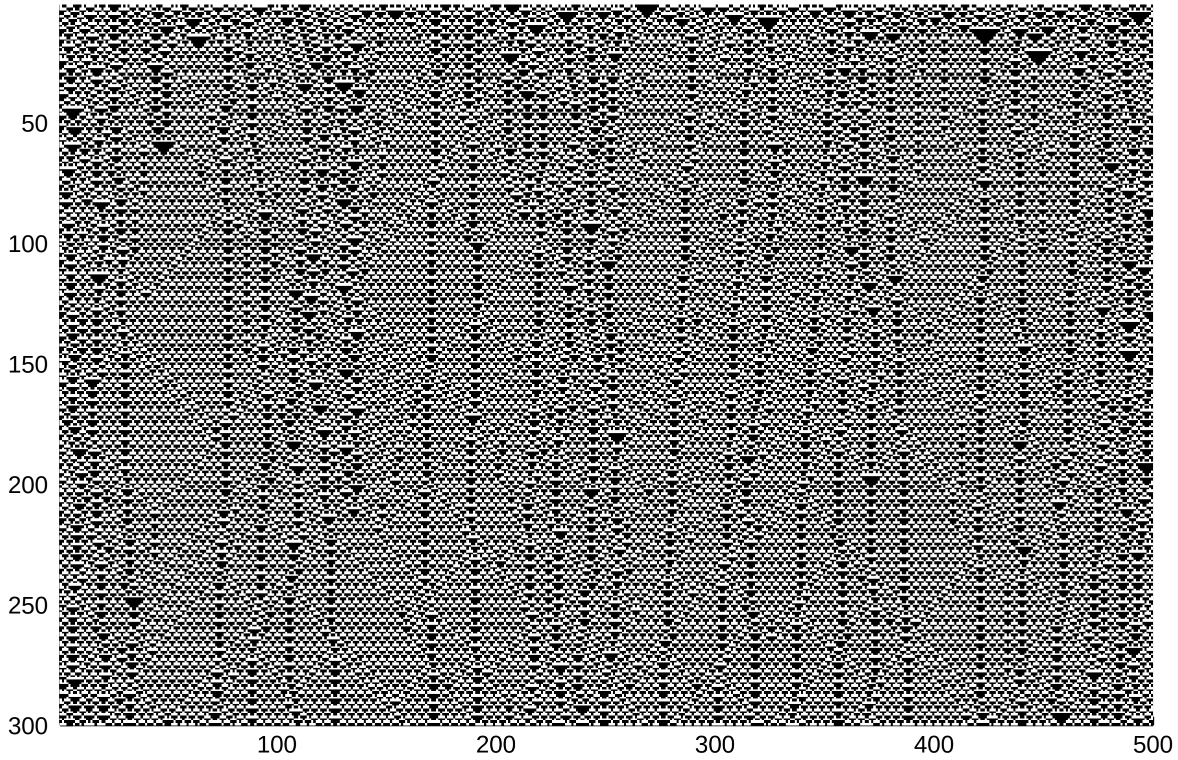


```
ca_plot(90, 500, 300)
```

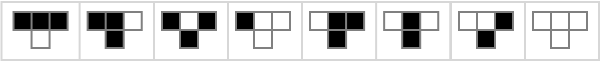
Class4 rules leading to complex patterns and structures propagating locally in the lattice:



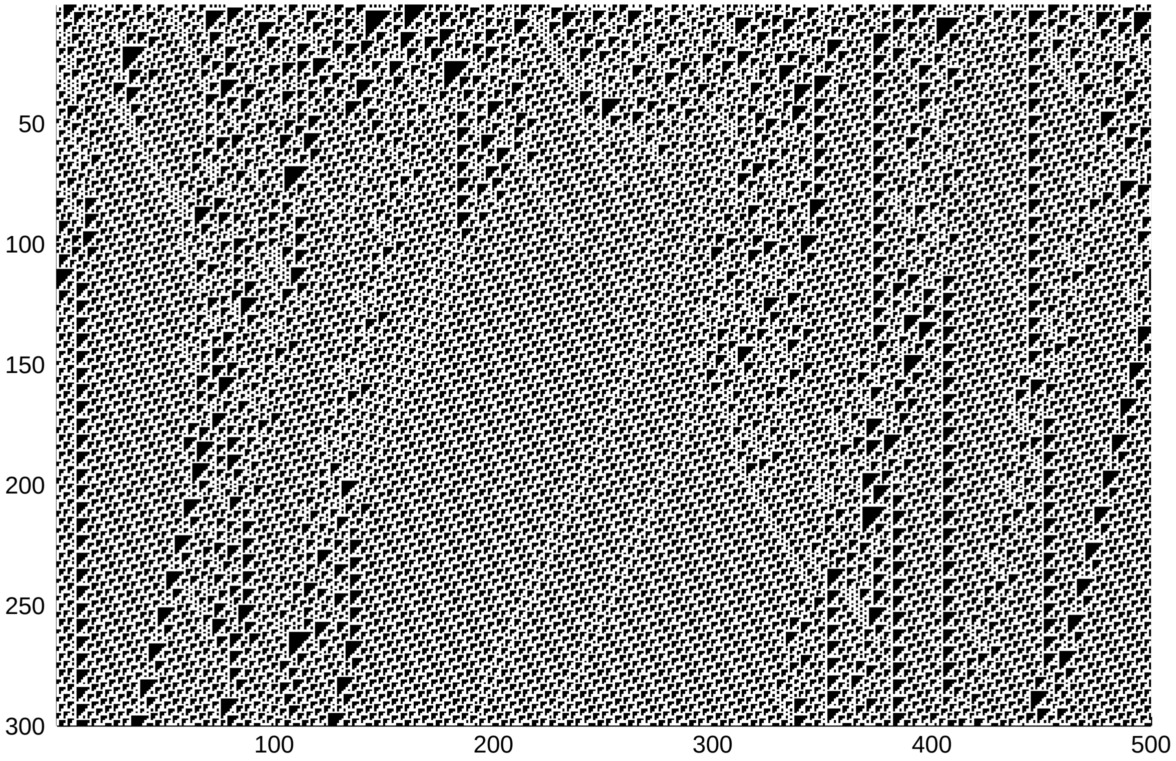
■ 1 | □ 0



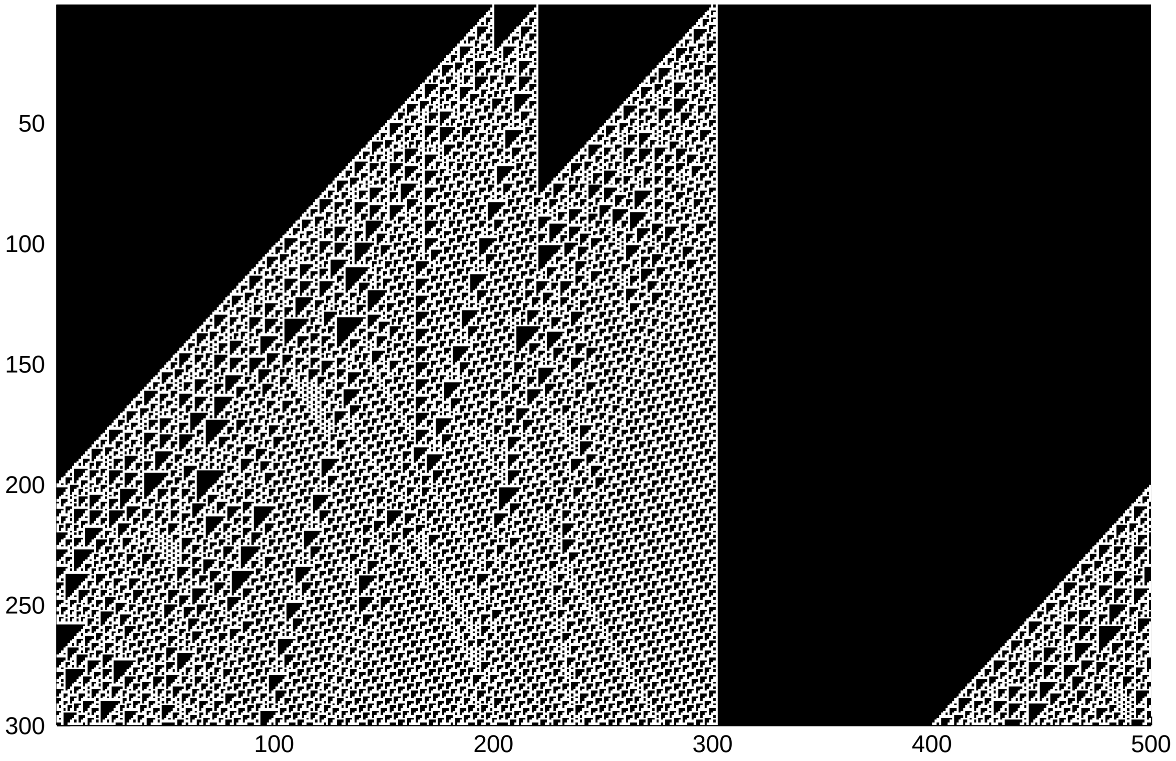
```
ca_plot(54, 500, 300)
```



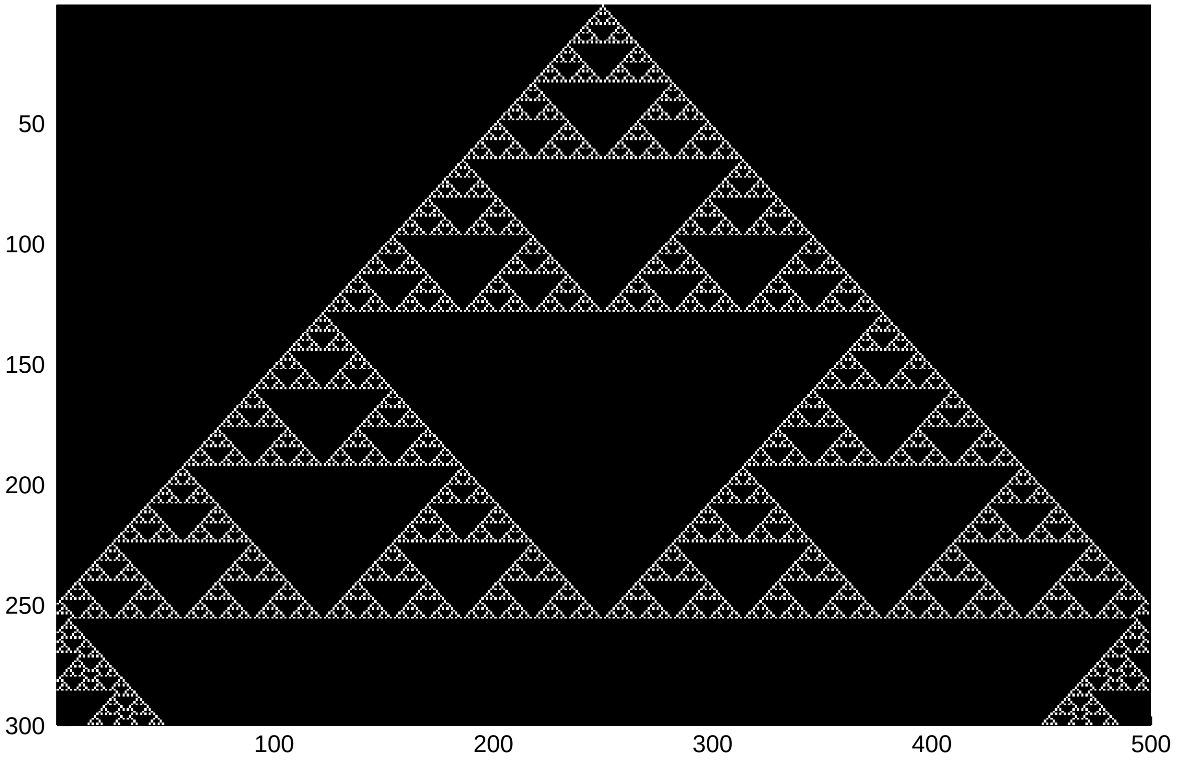
■ 1 | □ 0



```
ca_plot(110, 500, 300)
```



```
ca_plot(110, 500, 300, rnd_init = false, seed_at = [200, 220, 300, 302])
```



```
ca_plot(90, 500, 300, rnd_init = false)
```

```
draw_life_ca (generic function with 1 method)
```

On the edge of chaos

Perhaps the most exciting implication [of CA representation of biological phenomena] is the possibility that life had its origin in the vicinity of a phase transition and that evolution reflects the process by which life has gained local control over a successively greater number of environmental parameters affecting its ability to maintain itself at a critical balance point between order and chaos. (Langton 1990: 13)

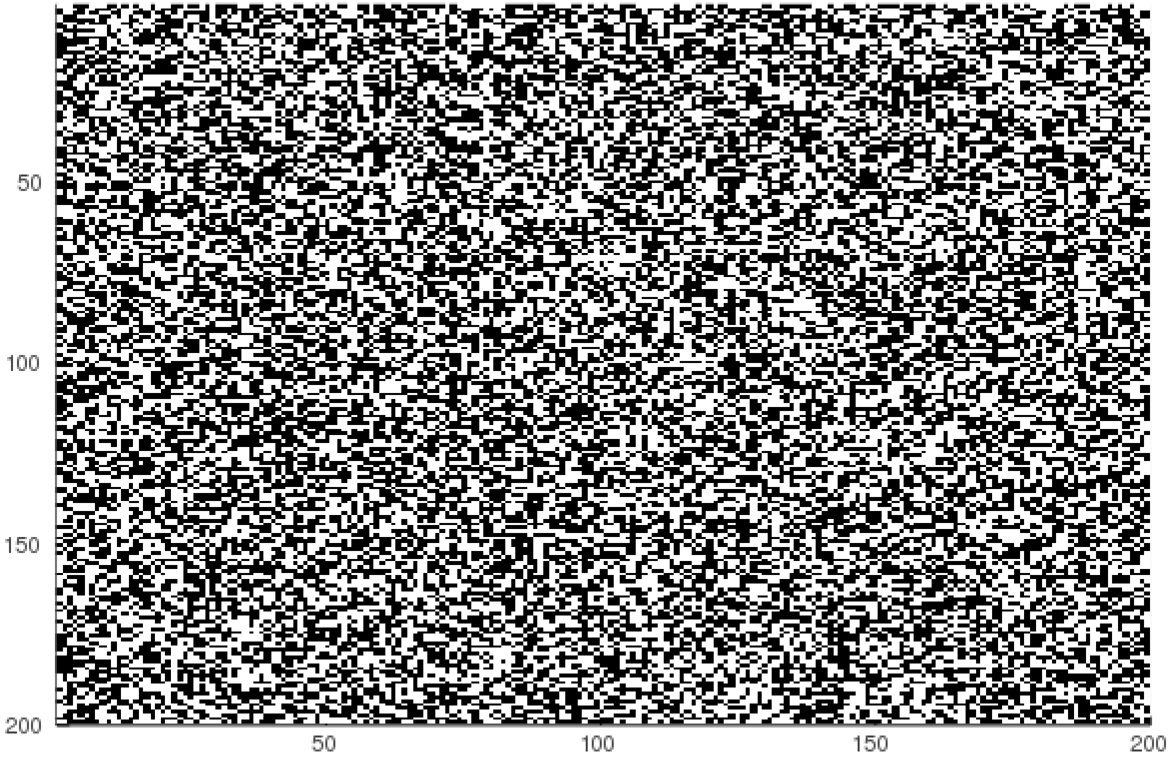
	110	111	108	106	102	126	78	46	228
000	0	*1*	0	0	0	0	0	0	0
001	1	1	*0*	1	1	1	1	1	1
010	1	1	1	*0*	1	1	1	1	1
011	1	1	1	1	*0*	1	1	1	1
100	0	0	0	0	0	*1*	0	0	0
101	1	1	1	1	1	1	*0*	1	1
110	1	1	1	1	1	1	1	*0*	1
111	0	0	0	0	0	0	0	0	*1*
Class	4	2	2	3	3	3	1	2	1

Game of life

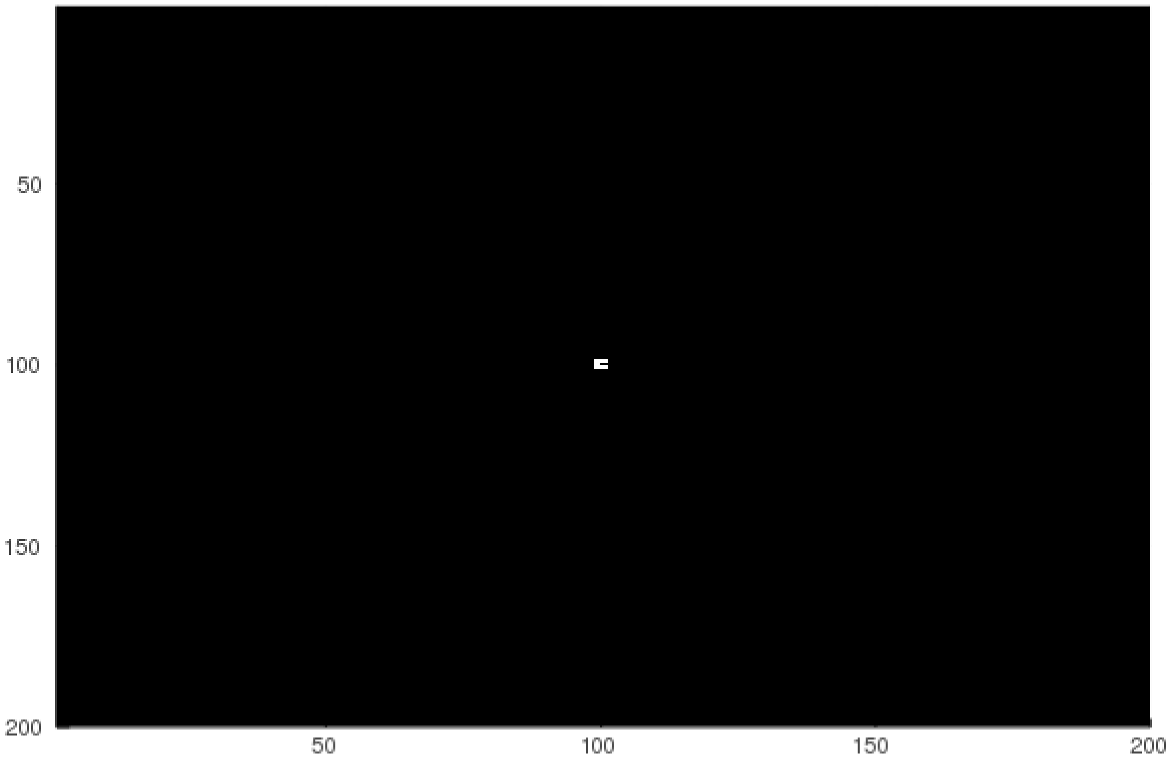
Life’s transition rule goes as follows. At each time step t exactly one of three things can happen to a cell:

- Birth: If the cell state at t−1 was 0 (dead), the cell state becomes 1 (alive) if exactly three neighbors were 1 (alive) at t−1;

- Survival: If the cell state at t−1 was 1 (alive), the cell state is still 1 if either two or three neighbors were 1 (alive) at t−1;
- Death: If the cell state at t−1 was 1 (alive), the cell state becomes 0 (dead) if either fewer than two or more than three neighbors were 1 (alive) at t−1 (cells can die of “loneliness” or “overpopulation”).

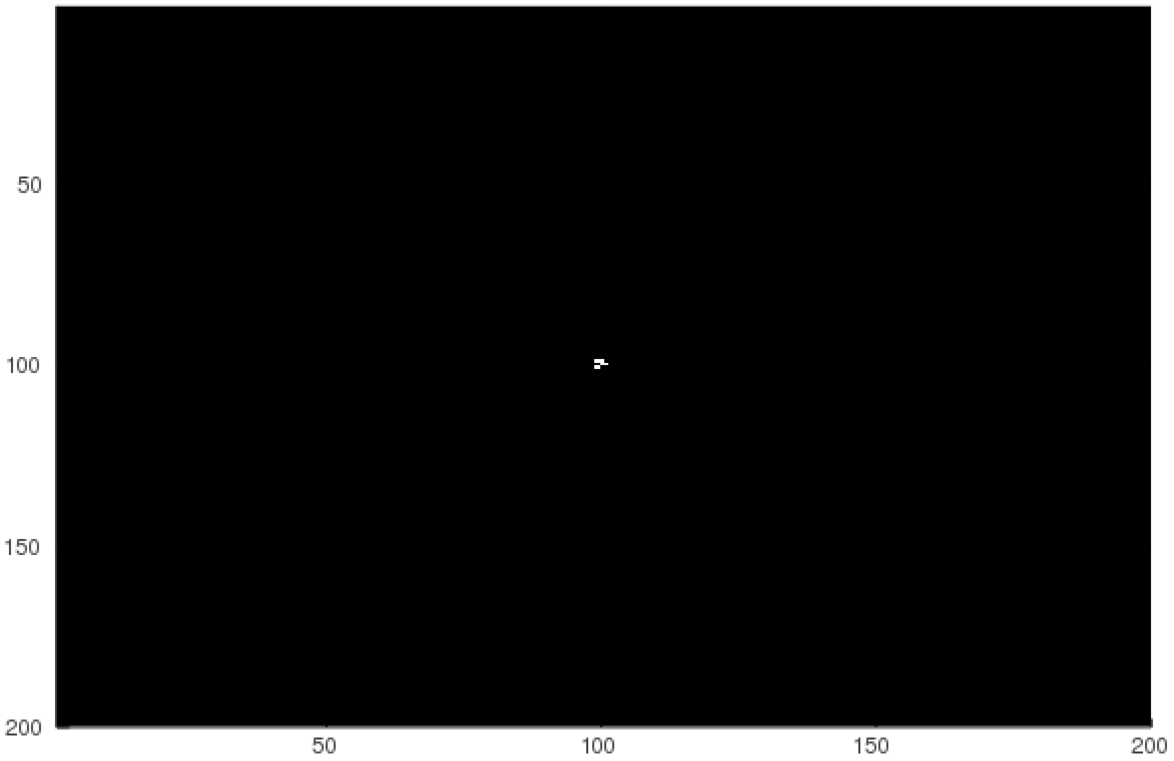


```
draw_life_ca(steps = 500, random_init=true)
```

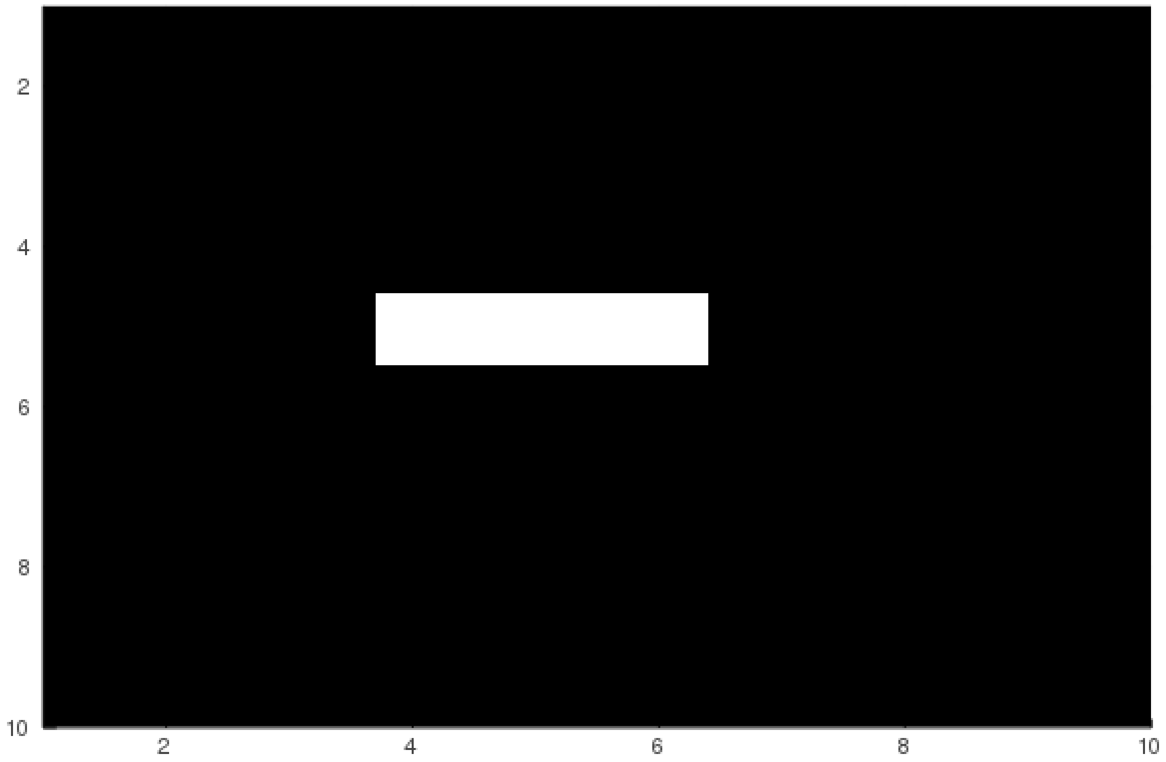


```
draw_life_ca(pattern = [
        true true true;
```

```
        true  false false;
        true true true
])
```

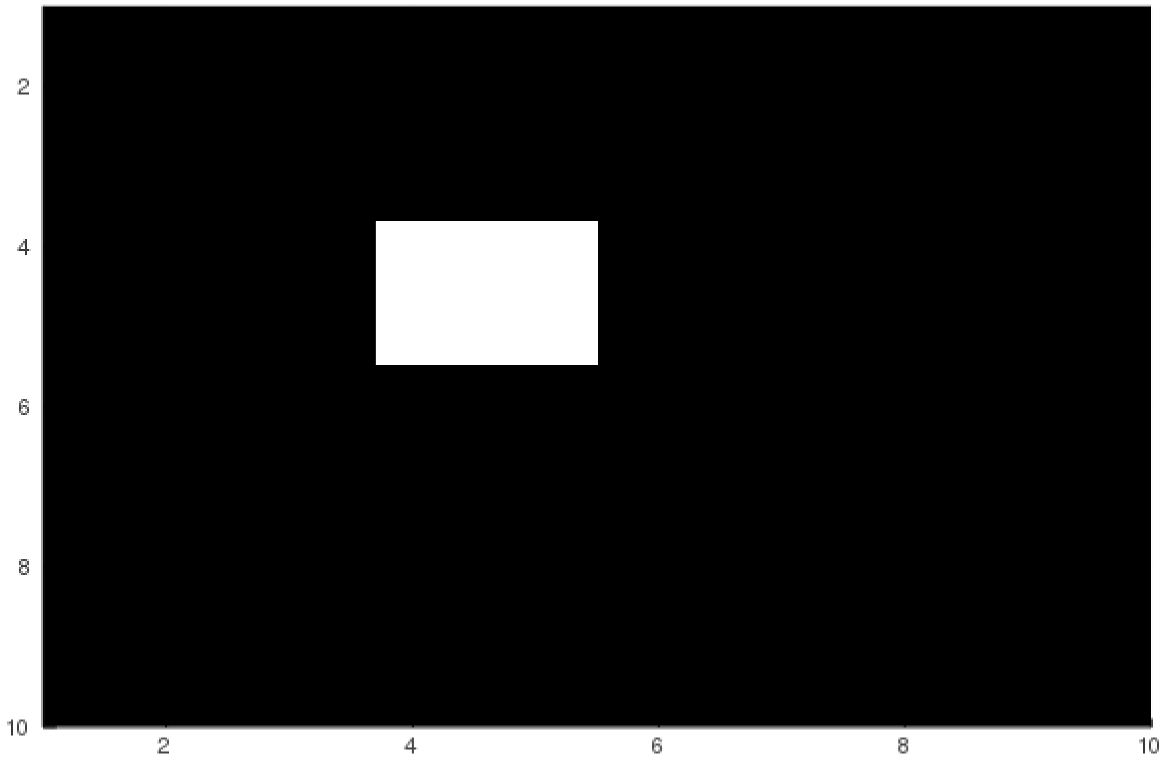


```
draw_life_ca(pattern = [
        true true false;
        false  true true;
        true false false
])
```

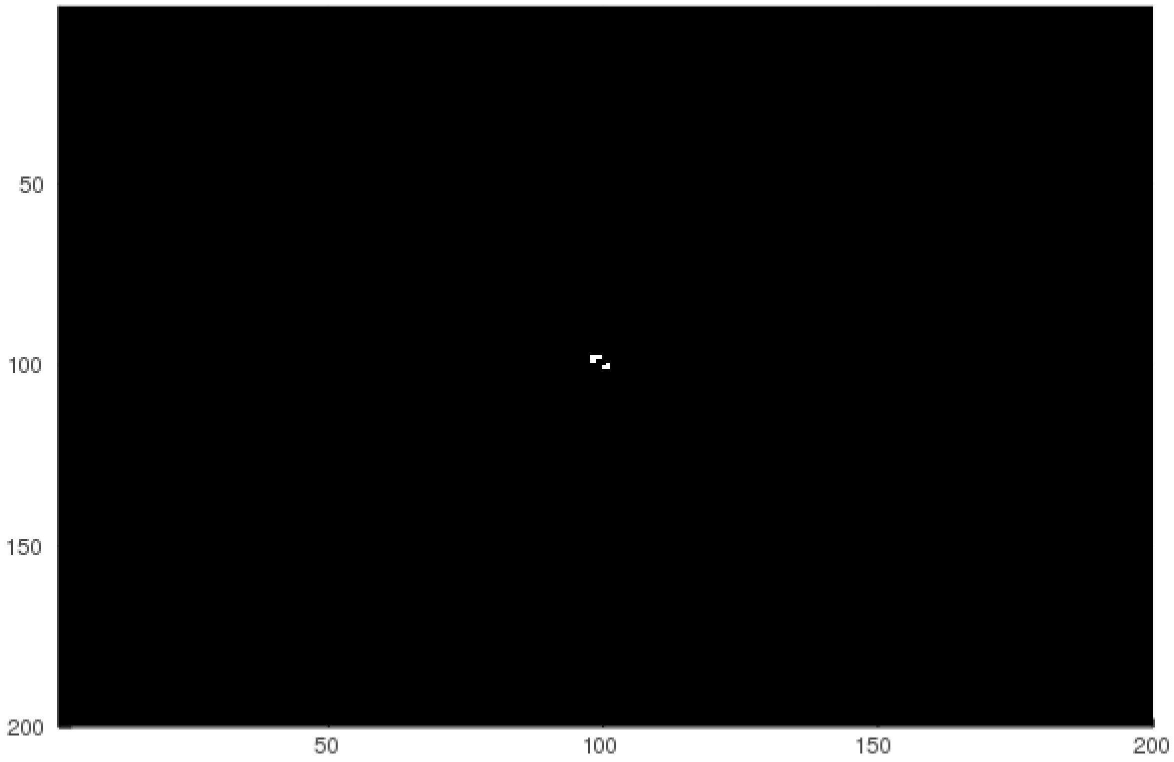


```
draw_life_ca(
    grid_size = (10,10),
    pattern = [
        false false false
```

```
        true  true true;
        false false false
    ])
```



```
draw_life_ca(
    grid_size = (10, 10),
    pattern = [
        true true false;
        true  true false;
        false false false
    ])
```



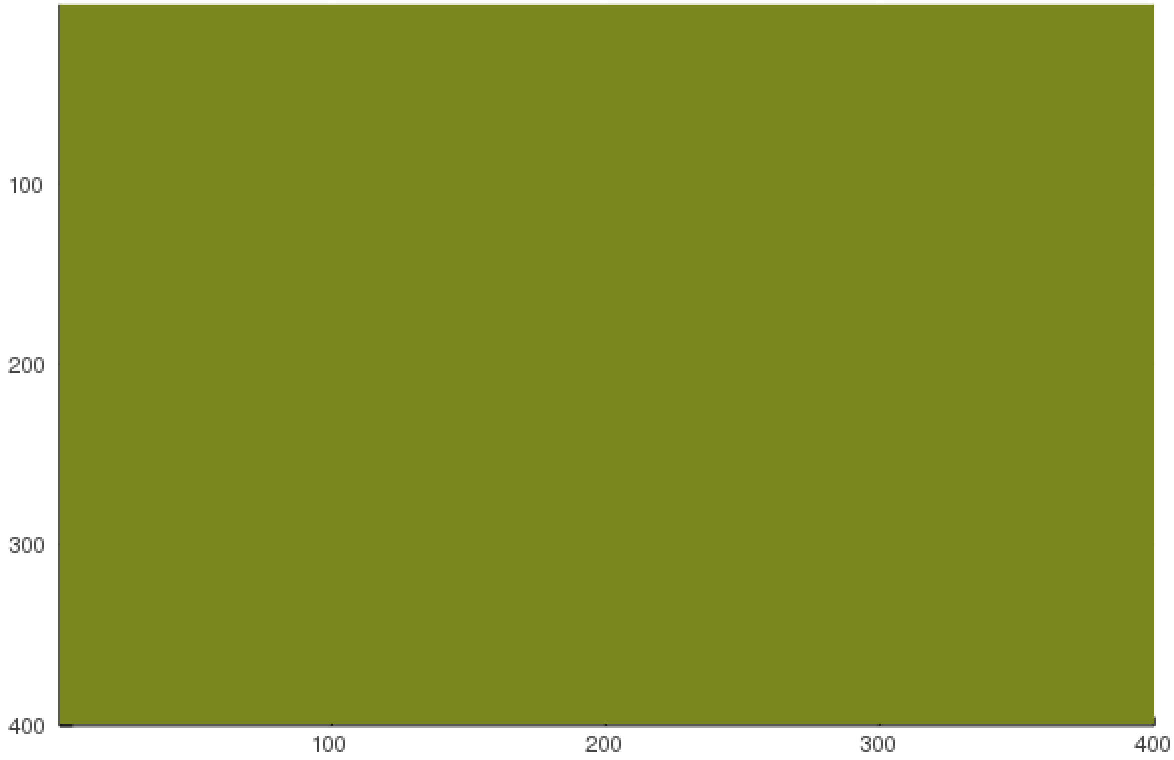
```
draw_life_ca(
    pattern = [
```

```julia
        true true false false ;
        true false false false;
        false false false true;
        false false true true
    ])
```

```
hex2rgba (generic function with 1 method)
```

```
draw_fire (generic function with 1 method)
```

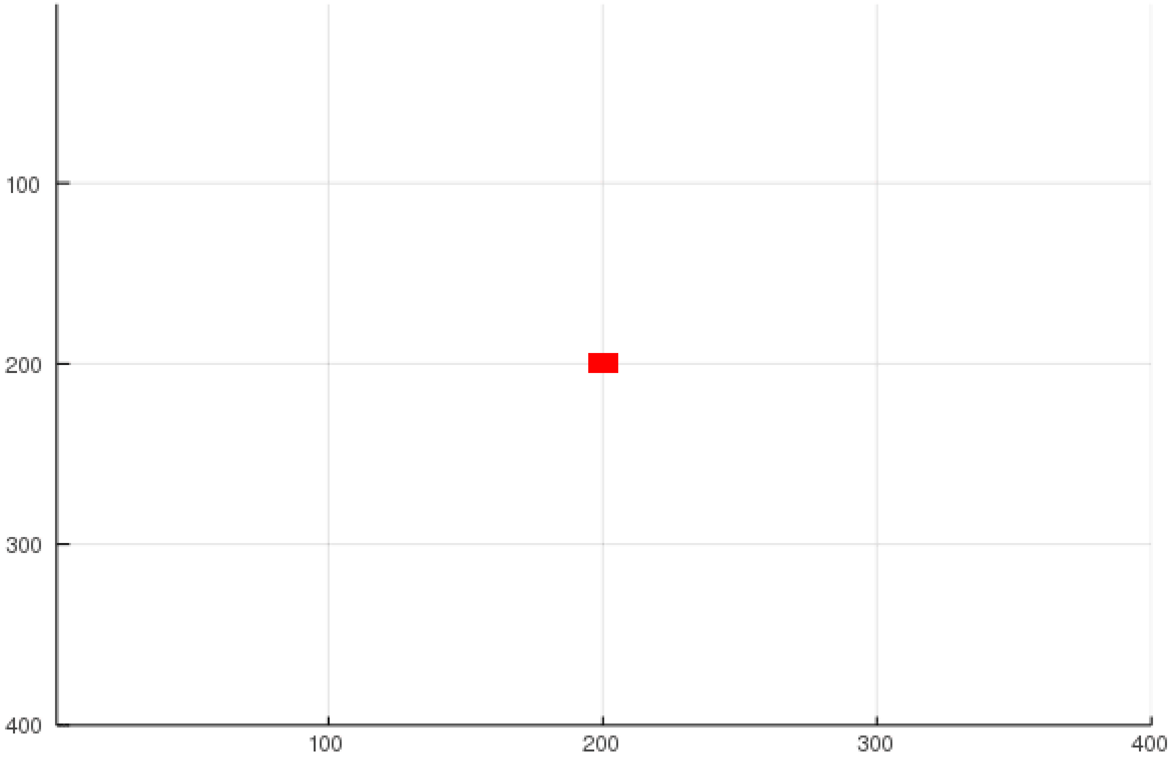
```julia
function draw_fire(;
        steps = 200,
        size = (400, 400)
    )
    DEAD, ALIVE, BURNING = 1, 2, 3

    neighbors_rule = let prob_combustion=0.0001, prob_regrowth=0.01
        Neighbors(Moore(1)) do neighborhood, cell
            if cell == ALIVE
                if BURNING in neighborhood
                    BURNING
                else
                    rand() <= prob_combustion ? BURNING : ALIVE
                end
            elseif cell == BURNING
                DEAD
            else
                rand() <= prob_regrowth ? ALIVE : DEAD
            end
        end
    end

    # Set up the init array and output (using a Gtk window)
    init = fill(ALIVE, size...)
    output = ArrayOutput(init; tspan=1:steps)

    # Run the simulation, which will save a gif when it completes
    sim!(output, neighbors_rule)

    s_colors = Dict(
        DEAD => hex2rgba(0x000000),
        BURNING => hex2rgba(0xFF4500),
        ALIVE => hex2rgba(0x7A871E)
    )

    animation = @animate for i in 1:steps
        plot(map(s -> s_colors[s], output[i]))
    end

    gif(animation, fps=10)
end
```



```
draw_fire()
```

```
draw_diff (generic function with 1 method)
```

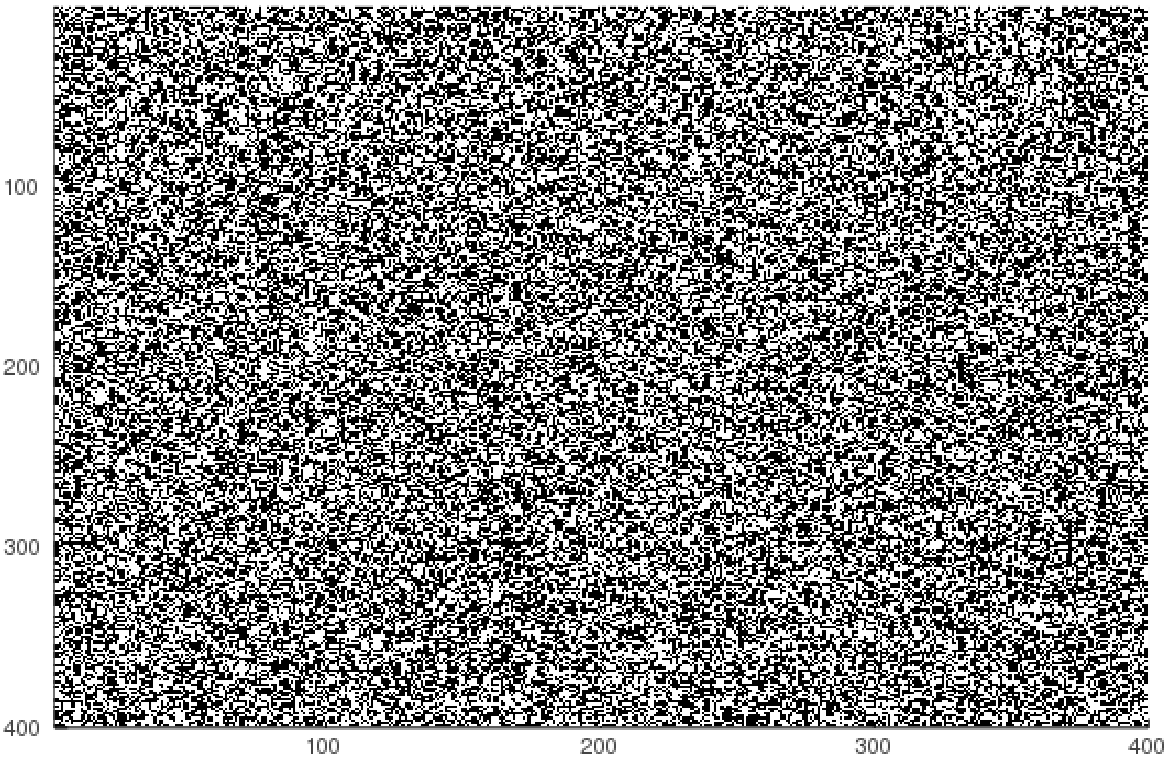
```
function draw_diff(;
        steps = 300,
        size = (400, 400)
    )

    neighbors_rule = Neighbors(VonNeumann(1)) do neighborhood, cell
            sum(neighborhood) / 4.
    end


    # Set up the init array and output
    init = zeros(Float64, size...)
    init[195:205, 195:205] .= 1000.
    output = ArrayOutput(init; tspan=1:steps)

    # Run the simulation, which will save a gif when it completes
    sim!(output, neighbors_rule)

    vox_log(x) = log(0.1 + x)

    animation = @animate for i in 1:steps
        plot(map(v -> RGBA(1., 0., 0., vox_log(v) / vox_log(1000.)), output[i]))
    end

    gif(animation, fps=20)
end
```



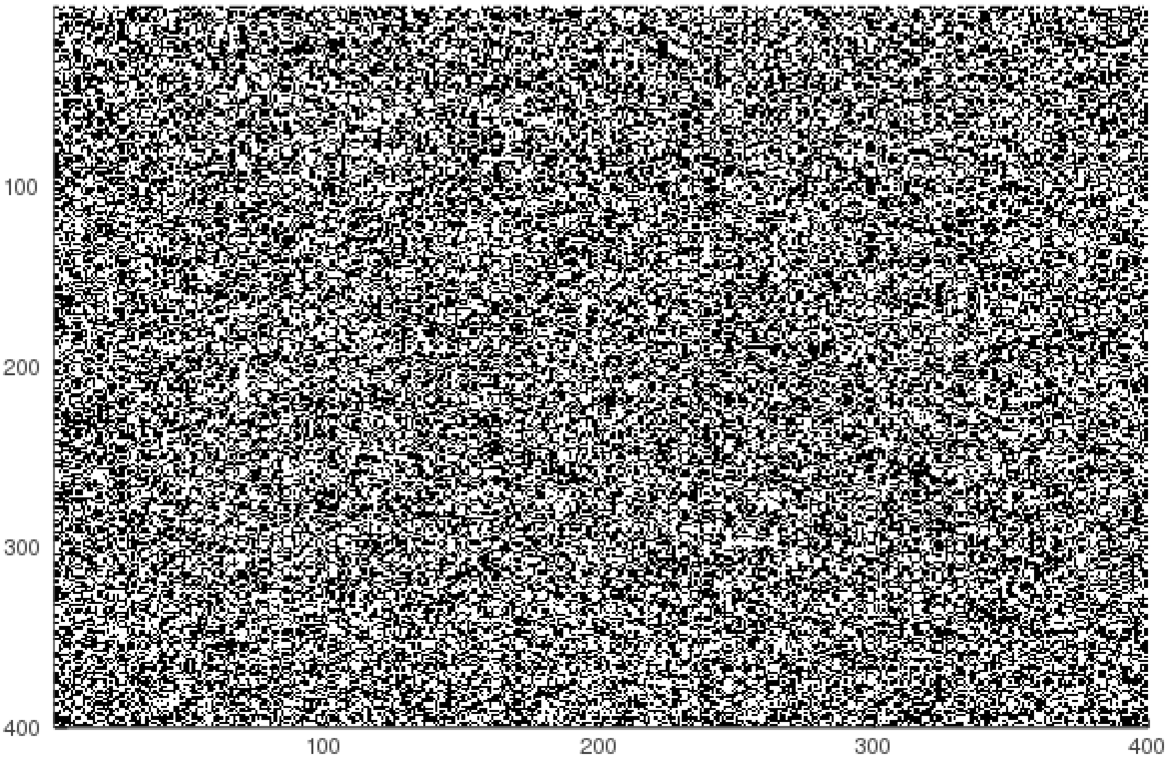
```
draw_diff()
```

```
draw_majority (generic function with 1 method)
```

```
function draw_majority(;
        r = 1,
        steps = 200,
        grid_size = (400, 400)
    )

    neighbors_rule = Neighbors(Moore(r)) do neighborhood, cell
        s = reduce(neighborhood, init = Int(cell)) do s, c
            s += c ? 1. : -1.
        end
        s > 0 ? true : false
    end


    # Set up the init array and output
    init = rand(Bool, grid_size...)
    output = ArrayOutput(init; tspan=1:steps)

    # Run the simulation, which will save a gif when it completes
    sim!(output, neighbors_rule)


    animation = @animate for i in 1:steps
        plot(Colors.Gray.(output[i]))
    end

    gif(animation, fps=10)
end
```



```
draw_majority()
```

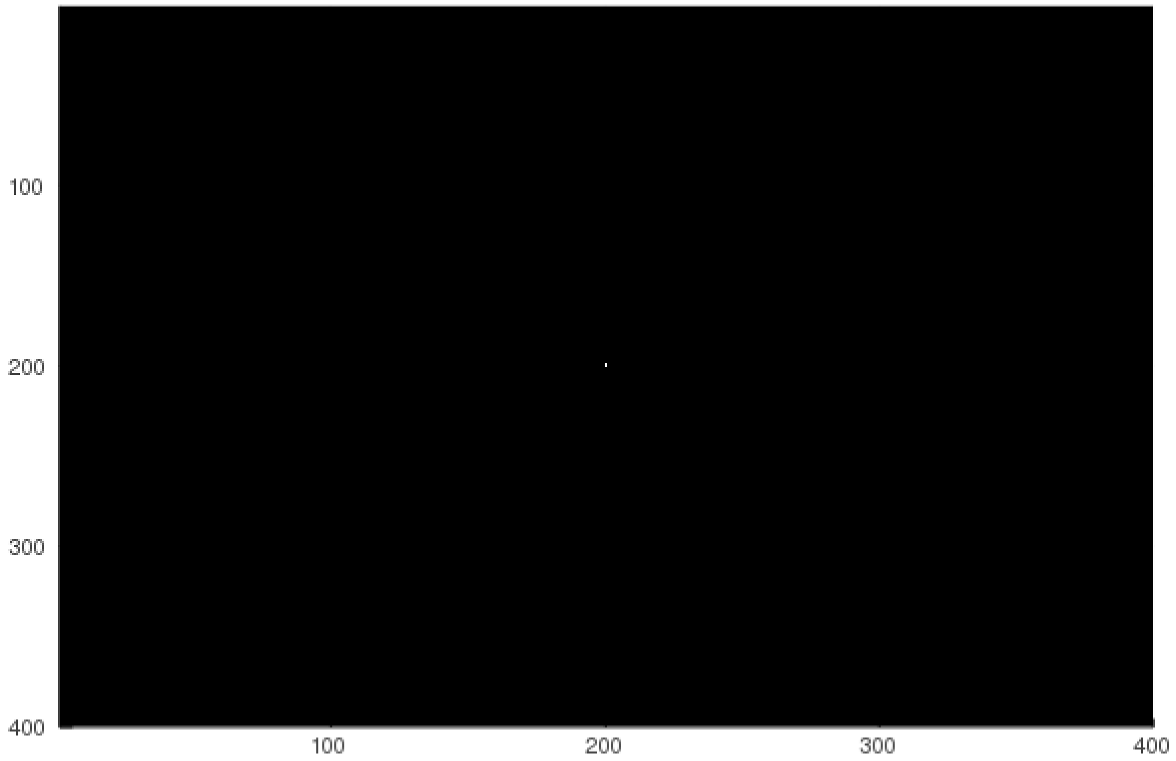


```
draw_majority(r=4)
```

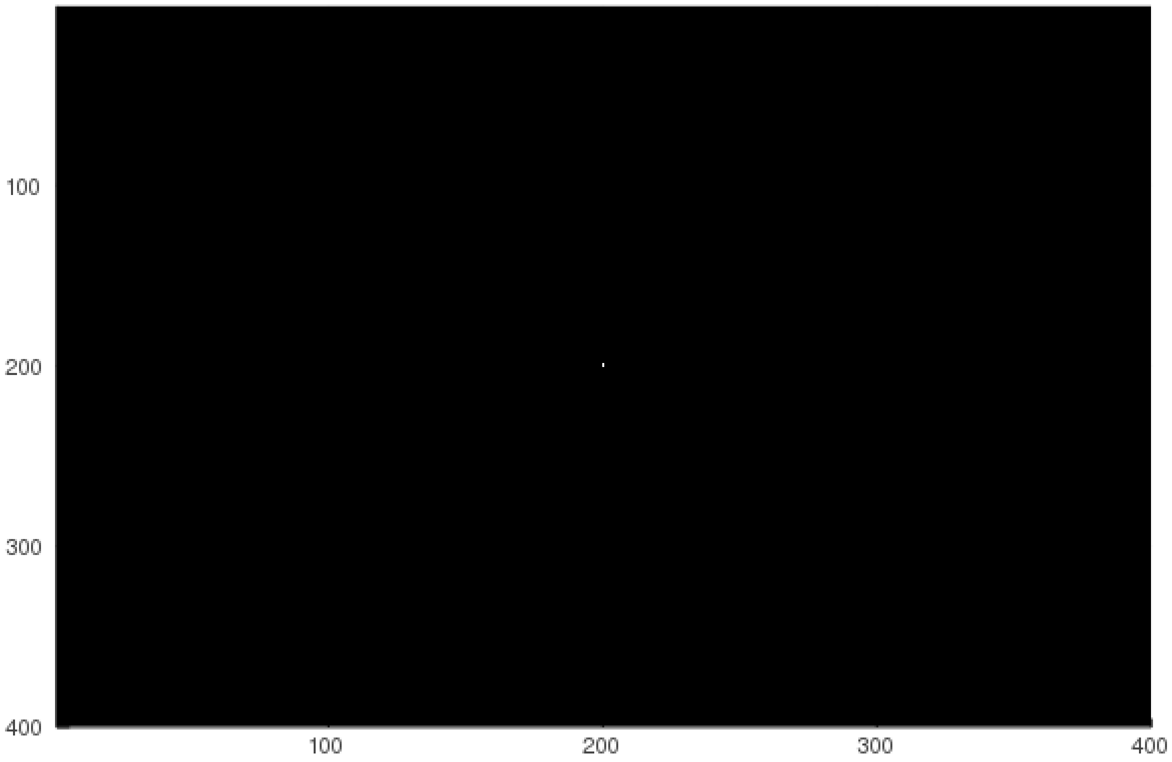
draw_parity (generic function with 1 method)

```
function draw_parity(;
        r = 1,
        steps = 200,
        grid_size = (400, 400),
        random_init = false,
        pattern = [false true false;
                   true  false true;
                   false true false]
```

```
    )

    neighbors_rule = Neighbors(Moore(r)) do neighborhood, cell
        reduce(neighborhood, init = cell) do s, c
            s = xor(s, c)
        end
    end

    # Set up the init array and output
    if random_init
        init = rand(Bool, grid_size...)
    else
        init = zeros(Bool, grid_size...)
        s = size(pattern)
        off = div.(s, 2)
        c = div.(grid_size, 2)
        init[c[1] - off[1]:c[1] - off[1] + s[1]-1,
             c[2]-off[2]:c[2]-off[2]+s[2]-1] .= pattern
    end

    output = ArrayOutput(init; tspan=1:steps)

    # Run the simulation, which will save a gif when it completes
    sim!(output, neighbors_rule)


    animation = @animate for i in 1:steps
        plot(Colors.Gray.(output[i]))
    end

    gif(animation, fps=10)
end
```



```
draw_parity()
```



```
draw_parity(r=2)
```