Rule 90

You may have heard of Conway's 'Game of Life' – probably the most well-known of a family of so-called 'cellular automata', or 'zero-player games'. Such games demonstrate the rich and complex variety that can result from a handful of very simple rules.

In Conway's game of life, cells in a 2D array live or die based on the state of their immediate neighbours. In 'Rule 90', the situation is made even simpler by considering a 1D array.

Live cells are shaded, and dead cells are left blank:

The status of cells in the next generation is determined completely by the status of cells in the previous one. See if you can guess the rule…

Need a clue? For each cell, consider the state of its two neighbours.

The Exclusive Or

In everyday speech, the word 'or' can mean two different things.
Consider this statement made by a student applying to university:

"I need an A in Maths or Physics to get into Warwick or Durham."

The first 'or' is inclusive: the student would satisfy the conditions of the offer by getting an A in Physics, or in Maths, or both. However, the second 'or' is exclusive – you could take up a place at Warwick or at Durham, but not at Warwick *and* Durham simultaneously.

In Maths or computing (or both), we use the inclusive or by default, so we refer to the other one as 'exclusive or', or XOR. For comparison:

OR	XOR	AND
(true) or (true) = true	(true) xor (true) = false	(true) and (true) = true
(true) or (false) = true	(true) xor (false) = true	(true) and (false) = false
(false) or (true) = true	(false) xor (true) = true	(false) and (true) = false
(false) or (false) = false	(false) xor (false) = false	(false) and (false) = false

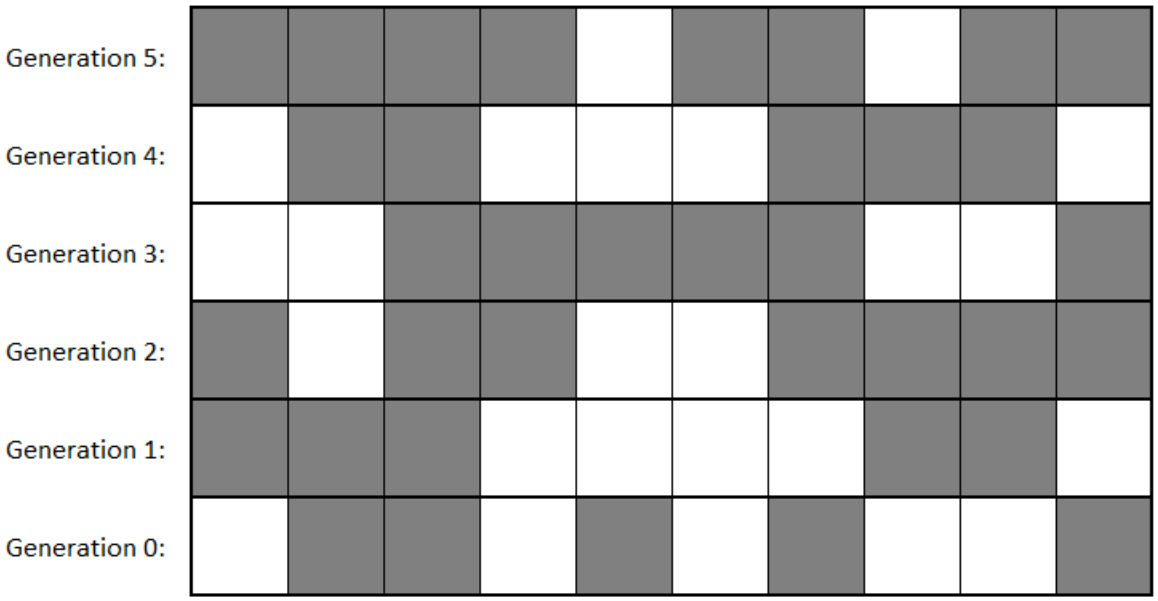
Along with their negation (eg NOR, NAND, XNOR), and a few trivial operations (like 'always true', or 'always the same as the first argument), these cover 12 of the 16 possible arrangements of true and false that you could imagine. Check out the Wikipedia article on truth tables for more details.

Rule 90

In case you hadn't guessed by now, the rule is: a cell in the next generation will be alive precisely when exactly one (but not both) of its current neighbours is alive: (left) XOR (right)

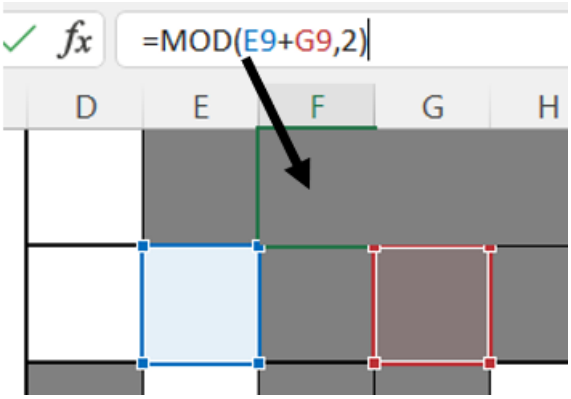
A time-space diagram

The space-time continuum in which we live gives physicists a real headache when studying relativity because we can't visualise four dimensions. However, if 'space' is just 1D, there's no problem with representing time as a second spatial dimension. We can just shunt the previous generation lower down the screen whenever a new generation is formed:



An alternative to XOR

Another way to think about the rule is by considering live cells to have a value of 1, and dead cells a value of 0. Then we just need a mathematical way to map (0,1) to 1, (1,0) to 1, (0,0) to 0 and (1,1) to 0. Addition modulo 2 should do the trick, since whenever the sum is odd, we want an answer of 1, and whenever it's even we want an answer of 0. This makes it easy enough to simulate in a spreadsheet:



And in p5.js?

We'll need a way to store the current generation, generate the next generation, and to keep track of previous generations. Each generation can be stored as an array of 0s and 1s, and to display them on the screen we can just draw squares coloured accordingly.

A class-less approach

The implementation I'm going to use here doesn't depend on classes. This keeps thing simpler for the time being, but bear in mind if you decide you want to create multiple copies of our automata on the screen at once, or better future-proof your code for subsequent modifications, you may find it helpful to refactor later using classes.

The grid

The simplest way to store the data will be an array of arrays, where the first array corresponds to the top row, or the most recent generation. Let's start by setting up some parameters to determine the size of our cells, populating the grid with 0s.

```
let grid = [];
let across = 10;
let down = 10;

function setup() {
  createCanvas(400, 400);
  for (let j=0; j<down; j++){
    let row = [];
    for (let i=0; i<across; i++){
      row.push(0);
    }
    grid.push(row);
  }
}
```

For now, I'll limit the population to 10 cells across, and track up to 10 generations. This will keep the animation quick and make it easier to check visually that the code is behaving as expected.

Note that I'm nesting my loops in such a way that my *grid* object (an array of arrays) will contain rows (arrays of zeroes).

To access, say, the i^{th} cell in the current population I would use *grid[0][i]*.

Displaying the grid

To make sure everything works as expected, I'm going to make sure I can view the whole array.

```
function draw() {
  background(220);
  displayGrid();
}

function displayGrid(){
  noStroke();
  for (let j=0; j<down; j++){
    for (let i=0; i<across; i++){
      if (grid[j][i] == 1){fill(0)}
      else {fill(255)}
      let x = map(i, 0, across, 0, width);
      let y = map(j, 0, down, 0, height);
      rect(x, y, width / across, height / down);
    }
  }
}
```

In *draw* I've made a reference to a new function, *displayGrid*, which is defined below.

This uses a similar loop to the one used to generate the grid. It then sets the fill colour based on the contents of the grid at that position, and determines the actual pixel location of the box to be drawn by mapping the *i* and *j* values to the full width and height of the canvas.

The size of each box is determined by the number across or down compared to the width & height.

Note: you can check that this is displaying the grid as expected by putting an extra line in *setup*:

```
    grid.push(row);
  }
  grid[0][4] = 1;
}
```

Array methods

I now need a function that calculates the values of the next generation, based on the current one, and shunts the rows down to make room for the new one.

For this, I'm going to have to manipulate some arrays, so it's time for a quick review of JavaScript array methods.

We use *slice* to create a copy of an array, so that we don't unwittingly modify the original array.

We use *push* and *pop* to add or remove elements from the end of an array (*pop* also returns this element).

We use *shift* and *unshift* to add or remove elements from the start of an array (*shift* also returns this element).

```
let arr = [1, 2, 3];

let arrCopy = arr.slice();
// arrCopy is [1, 2, 3]

let n = arr.length;
// n is 3

arr.push(4);
// arr is now [1, 2, 3, 4]

let last = arr.pop();
// last is 4, arr is now [1, 2, 3]

let first = arr.shift();
// first is 1, arr is now [2, 3]

arr.unshift(0);
// arr is now [0, 2, 3]
// arrCopy is still [1, 2, 3]
```

The next generation

```
function nextGeneration(){
  let current = grid[0].slice();
  current.unshift(0);
  current.push(0);
  let newGen = [];
  for (let i=1; i<current.length - 1; i++){
    newGen.push((current[i-1] + current[i+1]) % 2);
  }
  grid.unshift(newGen);
  grid.pop();
}
```

First, I make a copy of the first row using *slice*. Then I tack zeroes to the beginning and end so that the end points will still have neighbours. The new generation array, *newGen*, is created using $(a + b)\%2$, looping through all but the first and last elements of *current*.
Finally, *unshift* is used to add the *newGen* array to the start of *grid*, and *pop* is used to remove the oldest generation from the end of *grid*.

Trying it out

```
function draw() {
  background(220);
  nextGeneration();
  displayGrid();
}
```

…

```
let grid = [];
let across = 50;
let down = 50;
```

…

```
function setup() {
  createCanvas(400, 400);
  frameRate(4);
```

All we have to do now is add a call to *nextGeneration* in the *draw* loop to have our program iterate once every frame.

Given the speed of iteration when we only have a small number, you may want to increase the resolution by changing *across* and *down*…

…and if it's still too quick, you can use *frameRate* in *setup* to slow things down to a manageable speed.

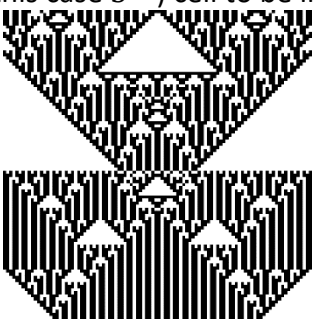
Making patterns

If you've done this right, even starting from a single live cell should yield a recognizable fractal pattern, but you can also create some very pretty shapes by setting, say, every third cell to be live:

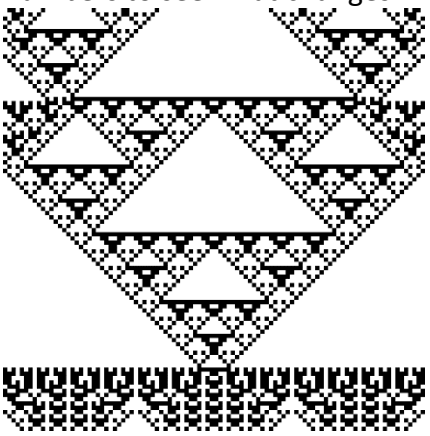
```
function setup() {
  createCanvas(400, 400);
  for (let j=0; j<down; j++){
    let row = [];
    for (let i=0; i<across; i++){
      row.push(0);
    }
    grid.push(row);
  }
  makeRegular(3);
}

function makeRegular(n){
  for (let i=0; i<across; i+=n){
    grid[0][i] = 1;
  }
}
```

The only change here is the call to *makeRegular* at the end of *setup*, followed by a simple function which sets every n^{th} (in this case 3^{rd}) cell to be live:



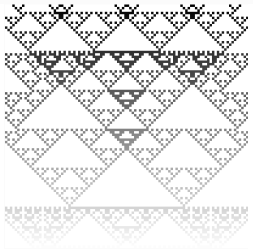
Try this out with different numbers to see what changes:



Fade out

Adding in an extra line to map the colour of a live cell to the vertical height:

```
function displayGrid(){
  noStroke();
  for (let j=0; j<down; j++){
    let col = map(j, 0, down, 0, 255);
    for (let i=0; i<across; i++){
      if (grid[j][i] == 1){fill(col)}
```



Speed run

If you increase the number of cells across and down, you may find that animation speed suffers. Since this is primarily down to how quickly the browser can render images to the screen, you can speed things up by having the *draw* loop compute several generations every frame before displaying:

```
let speed = 5;
```

…

```
function draw() {
  background(220);
  for (let i=0; i<speed; i++){
    nextGeneration();
  }
  displayGrid();
}
```