Snowflake Generator

This video by Veritasium (https://youtu.be/ao2Jfm35XeE) explains some of the physics behind how snowflakes are generated. Apparently it all boils down to temperature and humidity, with the particular structure of a water molecule dictating the crystalline nature of the snowflake, while the presence of water in the air allows for condensation, initially on a genesis point like a speck of dust, and then onto one another.

While much of this process is a bit beyond a short project, we can simulate the broad brushstrokes, cheat a little to generate the required symmetry, and build our very own snowflake generator. Being formed through random collisions of molecules should mean that every time we run the program we get a different snowflake.

Moving specks

We'll start with the specks which will condense to form ice crystals. A basic class to start with that will allow some freedom of movement, an element of randomness and a way to display the results:

```
class Speck{
  constructor(x, y){
    this.p = createVector(x, y);
    this.v = p5.Vector.random2D().mult(maxSpeed);
    this.color = color(random(255), random(255), random(255));
  }
  update(){
    this.v.rotate(random(-maxTurn*PI/180, maxTurn*PI/180));
    this.p.add(this.v);
    if (abs(this.p.x) > width/2){this.p.x *= -1}
    if (abs(this.p.y) > height/2){this.p.y *= -1}
  }
  display(){
    noStroke();
    fill(this.color);
    circle(this.p.x, this.p.y, 2*waterRadius);
  }
}
```

In order to give us enough freedom to modify key parameters as we develop our program, I'm making up new variables as I go along. Values such as *maxSpeed* and *maxTurn* will need to be declared in the global scope, but once they are, they will be accessible by any *Speck* object.

Note that I'm using a lazy 'wrap-around' model for specks that go off the screen. More on that later…

In lieu of acceleration, paths are altered through random rotation of the velocity vector (effectively this means a random acceleration perpendicular to the path, so that the direction – but not the magnitude – of the velocity can change.

Playing catch-up

After writing the bare bones of our class code, we now need to go through the rest of our program catching up with any variables that need declaring, etc:

Global variables are defined at the top:

```
let maxSpeed = 5;
let maxTurn = 15;
let waterRadius = 5;

let specks = [];
let numSpecks = 100;

class Speck{
  constructor(x, y){
```

I include *translate* to move the origin to the centre of the screen:

```
function draw() {
  background(0);
  translate(width/2, height/2);
  for (let i=0; i<specks.length; i++){
    specks[i].update();
    specks[i].display();
  }
}
```

In *setup* we create a bunch of new specks and store them in an array:

```
function setup() {
  createCanvas(400, 400);
  for (let i=0; i<numSpecks; i++){
    let x = random(-width/2, width/2);
    let y = random(-height/2, height/2);
    specks.push(new Speck(x, y));
  }
}
```

For ease of editing I've made the units of *maxTurn* degrees rather than radians.

Condensation

We now need particles to exhibit the behaviour of condensing, and hence becoming fixed in place:

```
 9 class Speck{
10   constructor(x, y){
11     this.frozen = false;
12     this.p = createVector(x, y);
13     this.v = p5.Vector.random2D().mult(maxSpeed);
14     this.color = color(random(255), random(255), random(255));
15   }
16   condense(){
17     this.frozen = true;
18   }
19   update(){
20     if (this.frozen){return false}
21     this.v.rotate(random(-maxTurn*PI/180, maxTurn*PI/180));
22     this.p.add(this.v);
23     if (abs(this.p.x) > width/2){this.p.x *= -1}
24     if (abs(this.p.y) > height/2){this.p.y *= -1}
25   }
26   display(){
27     noStroke();
28     fill(this.color);
29     let d = 2*waterRadius
30     if (this.frozen){d = 2*iceRadius}
31     circle(this.p.x, this.p.y, d);
32   }
33 }
```

Line 11 has been added, defining the *this.frozen* variable (they are in motion by default).

Line 16 is a new method, which can be run to cause the particle to condense (and hence become fixed in place).

When the *update* method is run, in the case where the speck is frozen, the function does nothing (returning a value terminates the function without carrying out the remaining commands).

Finally, in *display*, we fix the radius based on a (new) global variable, *iceRadius*, in case we want to change the size of the particles depending on their nature at a later date:

```
3 let waterRadius = 5;
4 let iceRadius = 10;
```

The genesis point

```
function setup() {
  createCanvas(400, 400);
  let genesis = new Speck(0, 0);
  genesis.condense();
  specks.push(genesis);
  for (let i=0; i<numSpecks; i++){
    let x = random(-width/2, width/2);
    let y = random(-height/2, height/2);
    specks.push(new Speck(x, y));
  }
}
```

Before anything can happen, we need a genesis point. The simplest way is probably just to place a speck in the centre of the screen (recall, that's now (0, 0) thanks to the *translate* function in *draw*), and force it to condense.

Make 'em stick

The particles now need to be taught to interact. We need a way to check to see if any speck is close enough to a frozen one to condense. We should think about the efficiency of the distance function: potentially checking the absolute distance first before using Pythagoras for a more exact measure would allow us to rule out most specks very quickly, hopefully speeding up execution time and allowing us to scale up our program to more specks.

```
freezeNeighbours(){
  for (let other of specks){
    if (other != this && !other.frozen){
      if (abs(this.p.x - other.p.x) + abs(this.p.y - other.p.y) < bond){
        if (p5.Vector.sub(this.p, other.p).mag() < bond){
          other.condense();
        }
      }
    }
  }
}
```

A new method within *Speck*, *freezeNeighbours* will find any non-frozen speck in the vicinity and freeze them. I'm using yet another global variable *bond* to decide how close they should be in order to affect one another:

```
let iceRadius = 10;
let bond = 10;
```

We will make another big saving on efficiency here by only running this method on the frozen specks: there's no point in interrogating every speck if we only care about the interaction when one of the molecules is frozen. We've also take care here of making those nearby specks condense. If we develop a more nuanced approach (eg nearby specks condense with different probabilities depending on some environmental factor), we can build that in to this method later.

Draw

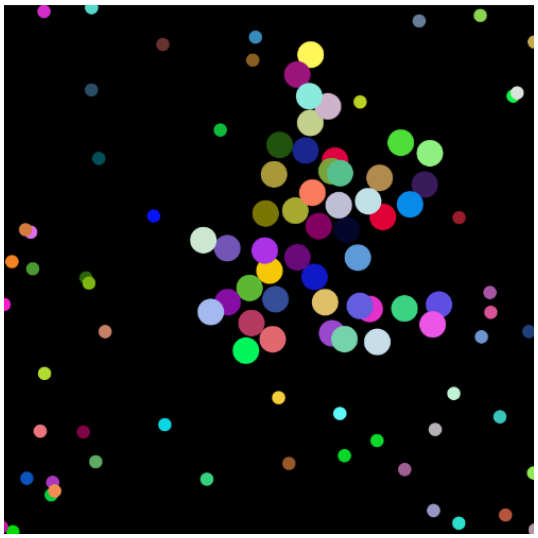
Next, we build in the checking in the *draw* loop, so that frozen specks run this *freezeNeighbours* method:

```
function draw() {
  background(0);
  translate(width/2, height/2);
  for (let i=0; i<specks.length; i++){
    if (specks[i].frozen){specks[i].freezeNeighbours()}
    specks[i].update();
    specks[i].display();
  }
}
```

This is just one line: before updating and displaying each spark, we check to see if it's frozen, and if so, it attempts to freeze its neighbours.

Where's the snowflake??

Granted, although this should now behave more or less as we planned, it doesn't look much like a snowflake.



In reality, the reason snowflakes exhibit the characteristic six-fold rotational and reflection symmetry is partly because, on the microscopic scale, conditions (temperature, humidity) are near enough identical all around the genesis point. The other factor is the molecular bond, which gives a nice $120°$ angle between adjacent molecules. If we could simulate billions of specks, and if we could build in a similar bond structure, we could probably make snowflakes the way God intended. However, if we simply want the *appearance* of a snowflake, we can force the symmetry through transformations…

Rotations and reflections

The lazy method (letting p5 do the heavy lifting):

```
function transformedSpecks(speck){
  let c = speck.color;
  let unit = createVector(1, 0);
  let p = speck.p.copy();
  let q = p.copy().reflect(unit);
  let reflected = new Speck(q.x, q.y);
  reflected.condense();
  reflected.color = c;
  let newSpecks = [reflected];
  for (let i=0; i<5; i++){
    p.rotate(PI/3);
    q.rotate(PI/3);
    let pSpeck = new Speck(p.x, p.y);
    let qSpeck = new Speck(q.x, q.y);
    pSpeck.condense();
    qSpeck.condense();
    pSpeck.color = c;
    qSpeck.color = c;
    newSpecks.push(pSpeck);
    newSpecks.push(qSpeck);
  }
  return newSpecks;
}
```

This is a new function whose job it is to generate and return new specks. First, we make a copy of the speck's position vector, and its reflection, reflecting in the unit vector $\begin{pmatrix}1\\0\end{pmatrix}$. This gives mirror symmetry in the x-axis.

Then, iterating 5 times, we rotate both the original vector and its reflection, generating the remaining 5 parts of the 6-part rotational symmetry required.

Note the additional setting of colour: if you leave this out, every new speck generated will have a random colour. If you ensure all new specks created in this way match the colour of the original, we'll more clearly see the symmetry.

The hardcore method (coding the transformations using 2 by 2 matrices):

```
function reflectVector(p){
  return createVector(p.x, -p.y);
}

function rotateVector(p){
  return createVector((p.x - sqrt(3)*p.y)/2, (sqrt(3)*p.x+p.y)/2);
}
```

The main function above would be similar, but instead of relying on p5's built-in vector functions for rotation and reflection, you might be more efficient hard-coding your own.

After all, we only need reflection in $y = 0$ and rotation by $\frac{\pi}{3}$ radians, so this could be more efficient.

Adding the transformed specks

```
function draw() {
  background(0);
  translate(width/2, height/2);
  for (let i=0; i<specks.length; i++){
    if (specks[i].frozen){
      let newlyFrozen = specks[i].freezeNeighbours()
      for (let i=0; i<newlyFrozen.length; i++){
        let newSpecks = transformedSpecks(newlyFrozen[i]);
        for (let s of newSpecks){
          specks.push(s);
        }
      }
    }
    specks[i].update();
    specks[i].display();
  }
}
```

In order to make use of our *transformedSpecks* function, we'll need to know which specks have been newly frozen. Since this occurs in the *draw* loop, we'll create a new array called *newlyFrozen* and find a way to populate it when we run the *freezeNeighbours* method. Then for any newly frozen speck, we'll add the 11 additional specks that our transformation function generates to our *specks* array.

Note that we'll need to adapt the *freezeNeighbours* method slightly to not only freeze its neighbours, but also return an array of them so that they can have their mirror images and rotations added to the pattern.

```
freezeNeighbours(){
  let neighbours = [];
  for (let other of specks){
    if (other != this && !other.frozen){
      if (abs(this.p.x - other.p.x) + abs(this.p.y - other.p.y) < bond){
        if (p5.Vector.sub(this.p, other.p).mag() < bond){
          other.condense();
          neighbours.push(other);
        }
      }
    }
  }
  return neighbours;
}
```

Only a few extra lines of code required here: make an empty array called *neighbours*, then push any newly frozen sparks into it at the same time as the *other.condense()* line is run.
And finally, return this array.

Turning a kaleidoscope into a snowflake

The range of colours can be helpful while you're getting the symmetry right, but now it's time to turn them all white:
Replace the colour in the *Speck constructor* function here:

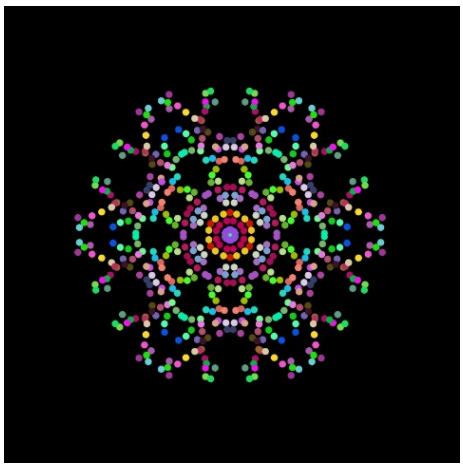
```
class Speck{
  constructor(x, y){
    this.frozen = false;
    this.p = createVector(x, y);
    this.v = p5.Vector.random2D().mult(maxSpeed);
    this.color = color(random(255), random(255), random(255));
  }
```

with the greyscale version:

```
    this.color = color(random(255));
```

This gives a colour between black and white. For brighter ice particles, tweak a little more:

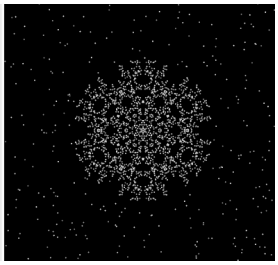
```
    this.color = color(random(128, 255));
```



All that remains now is to tinker with the parameters and see what effect you can produce…

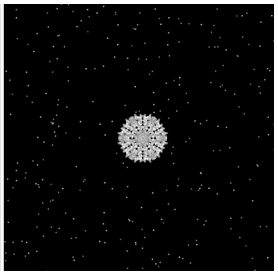
```
let maxSpeed = 5;
let maxTurn = 15;
let waterRadius = 1;
let iceRadius = 1;
let bond = 5;

let specks = [];
let numSpecks = 500;
```



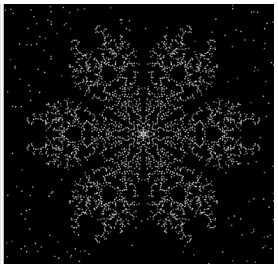
```
let maxSpeed = 3;
let maxTurn = 15;
let waterRadius = 1;
let iceRadius = 1.5;
let bond = 2;

let specks = [];
let numSpecks = 500;
```



```
let maxSpeed = 3;
let maxTurn = 60;
let waterRadius = 1;
let iceRadius = 1;
let bond = 5;

let specks = [];
let numSpecks = 500;
```



What's next?

A good next step is to have specks keep track of the speck they condense on, so that we could perhaps draw in lines to show the connections. Then maybe dispense with circular dots altogether.