The Forgetful Point

In this project, we'll experiment with a very simple rule which generates some delightful patterns in the plane. First, put yourself in the shoes of a point in the plane…

The scenario

Imagine you're a forgetful point living in 2D space.
There are 3 destinations you would love to visit: Red City, Greenville and Bluetown.

You set out each day to travel towards whichever one takes your fancy.
You walk halfway to your chosen city, then rest for the night.
The trouble is, you can never remember when you wake up the next day which way you were going, so you just repeat the process, arbitrarily choosing one of the three destinations and walking halfway towards it before stopping to pitch camp.
Of course, forgetfulness isn't your only problem – since you only ever go halfway to whichever destination you choose, even if you continue in the same direction for a few days in a row, you'll never quite reach your target.

A puzzle

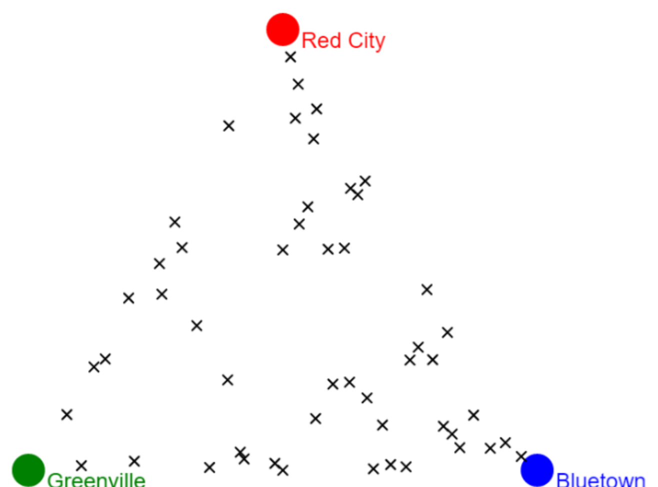
As the days turn into weeks, and the weeks into months, you gradually start to notice that every time you pitch a tent, there are signs of your previous camp-sites nearby.
This seems particularly odd since during your daily walks you often pass through large open spaces with no sign of old camp-sites whatsoever.

Can you work out why there are certain areas you never seem to stop for the night?

What's so special about them?

Try running a simulation by hand before going any further, to see if you can get a feel for what's going on.

How many 'dead zones' can you find?



Making the cities

Let's start by making the main set-up. We'll need three equally spaced cities, along with names and colours. I'd like to be able to move these around later (we'll get to that), so I'm going to future-proof them by writing them as custom objects. Remember, you can write your class code in the main *sketch.js* document, or make a new file to keep things tidy. Just remember to add a reference to any additional files in *index.html*.

```
class City{
  constructor(x, y, col, name=""){
    this.p = createVector(x, y);
    this.col = col;
    this.name = name;
  }
  display(){
    noStroke();
    fill(this.col);
    circle(this.p.x, this.p.y, 15);
    text(this.name, this.p.x + 10, this.p.y + 10);
  }
}
```

We will create cities by passing coordinates, a colour and, optionally, a name label.

Cities will be displayed by default with a circle and their name label to one side.

We'll return to this code shortly to enable clicking and dragging to change the location of cities.

Making the forgetful point

The forgetful point can have his own class, too. Although he might forget where he's going from one day to the next, I want to keep track of his old campsites, so we'll make an array specially for that.

```
class Camper{
  constructor(x, y){
    this.p = createVector(x, y);
    this.sites = [];
    this.col = "■black";
  }
  display(showTrail=true){
    cross(this.p, this.col);
    if (showTrail){
      for (let site of this.sites){
        let [p, col] = site;
        cross(p, col);
      }
    }
  }
  move(cities){
    this.sites.push([this.p.copy(), this.col]);
    let city = random(cities);
    this.col = city.col;
    this.p = p5.Vector.add(this.p, city.p).mult(0.5);
  }
}
```

The *constructor* just sets up the initial position of our camper, with an initial colour and an empty array for storing campsites.

The *display* method draws a cross at the camper's current position (we'll put off writing the *cross* code for the moment), then, if a trail is required, it loops through the campsites and draws a cross at each of those as well.

The *move* method is where the rule is applied – a random city (from those passed into the function) is selected as the next destination, and after the current campsite is marked (added to the *sites* array) we update our position with the midpoint of our current position and the destination's position.

```
function cross(p, col, r=2){
  stroke(col);
  line(p.x - r, p.y - r, p.x + r, p.y + r);
  line(p.x - r, p.y + r, p.x + r, p.y - r);
}
```

And, finally, a quick *cross* function. Note that this takes an optional *r* argument, so we can modify the size later. You could even build in a way to draw just a single pixel (using *point*) if *r* is sufficiently small, to save on memory.

Colours

To better visualise and understand the behaviour, I've incorporated colour-coding. Since the cities are uniquely coloured, when I log the site of the latest camp, I also store the colour of the city we were travelling to at the time of stopping. This means that we'll see not only a map of campsites, but also which ones correspond to which destinations.

Running the simulation

```
let cities = [];
let camper;

function setup() {
  createCanvas(500, 400);
  cities.push(new City(60, height - 20,
                      "■green", "Greenville"));
  cities.push(new City(width - 60, height - 20,
                      "■blue", "Bluetown"));
  cities.push(new City(width/2, 20,
                      "■red", "Red City"));
  camper = new Camper(width/2, height - 20);
}
```

In the main sketch, I'll need an array of cities, which I create in manually determined positions here (you can use $(r\cos\theta, r\sin\theta)$ to form a perfect equilateral triangle if you prefer), and a *camper* who is initially placed halfway between the two lower cities.

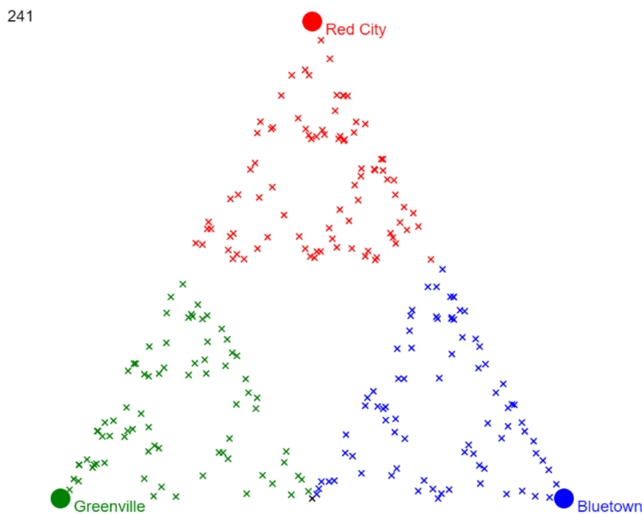
```
function draw() {
  background(255);
  for (let city of cities){
    city.display();
  }
  camper.move(cities);
  camper.display();
}
```

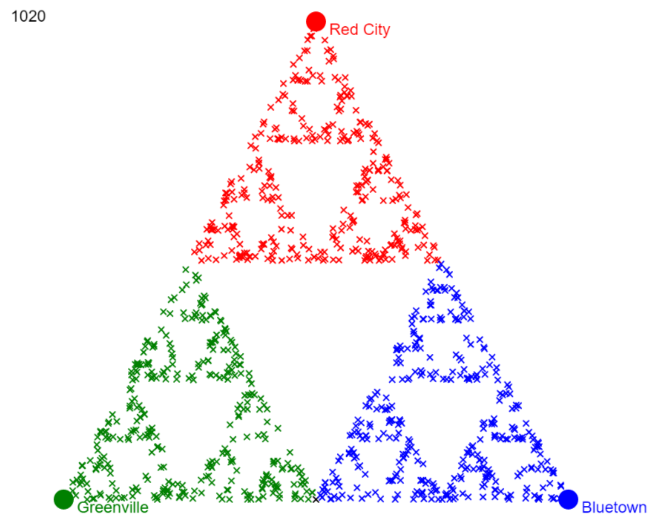
The *draw* function simply displays every city, moves the camper (passing him the cities array so he can make his choice), then displays him (by default also showing each campsite).

The result

After only a couple of hundred iterations, we can already see a roughly triangular gap in the middle. (use *camper.sites.length* to extract the number of sites if you also want to display it on the screen).

After 1000 or so, the result is even more striking:





Speeding things up

To get a clearer picture without waiting ages, we can make is to update *camper* multiple times per frame:

Add a *speed* variable to the top of your sketch:

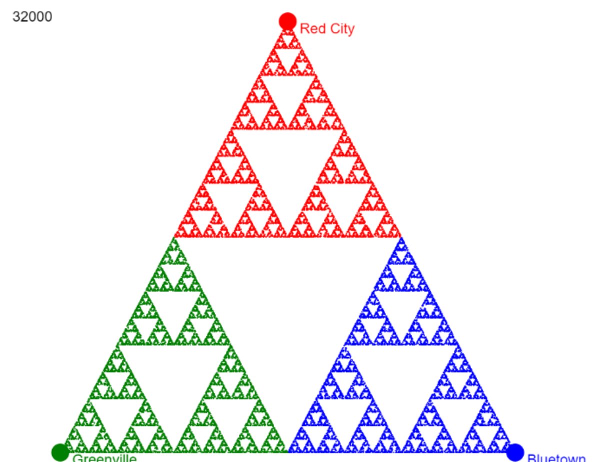
```
let speed = 20;
```

And in *draw*, embed the move in a loop:

```
for (let i=0; i<speed; i++){
  camper.move(cities);
}
```

You might also want to tweak the *cross* function:

```
function cross(p, col, r=0){
  stroke(col);
  if (r == 0){
    point(p.x, p.y);
  }
  line(p.x - r, p.y - r, p.x + r, p.y + r);
  line(p.x - r, p.y + r, p.x + r, p.y - r);
}
```



Why?

Why are dead zones? Why are there *no* red campsites below a certain line? If you start in the triangle, a step halfway to Red City will always take you to somewhere within the top quarter. Once in the top quarter, any further steps in that direction will take you to the top quarter of the top quarter, and so on.

Changing the goalposts

What happens if we change the position of the cities? We could of course just manually alter the coordinates, but it's pretty easy to build in click-and-drag capabilities (and could come in handy in other coding projects you tackle in the future). Returning to the *City* class, we start by defining a helper method:

```
mouseOver(){
  let mouse = createVector(mouseX, mouseY);
  return p5.Vector.sub(mouse, this.p).mag() < 20;
}
```

This will allow us to detect directly from within the city whether the mouse is within a certain distance of the city. I've set it to 20 pixels, which is fairly generous – the mouse need not be touching the city, just fairly close.

We can test out our function by slightly changing how cities are displayed when the mouse is nearby:

```
display(){
  noStroke();
  fill(this.col);
  let size = 15;
  if (this.mouseOver()){
    size = 20;
  }
  circle(this.p.x, this.p.y, size);
  text(this.name, this.p.x + 10, this.p.y + 10);
}
```

Provided the mouse is near the city, it'll now show up with a slightly larger circle. And as soon as the mouse pointer is too far away, the circle reverts to its regular size.

To drag a city, the best way is to incorporate a *selected* attribute. That way, even if the mouse is moved quickly, it won't 'lose' the city simply by getting too far away. When the mouse is clicked, if a city is nearby it will become 'selected'. A city which is selected will automatically change its position to match the mouse's position while the mouse is still pressed, and cease to be selected once the mouse is released.

```
  this.selected = false;
}
display(){
  if (this.selected && mouseIsPressed){
    this.p = createVector(mouseX, mouseY);
  }
  else{
    this.selected = false;
  }
```

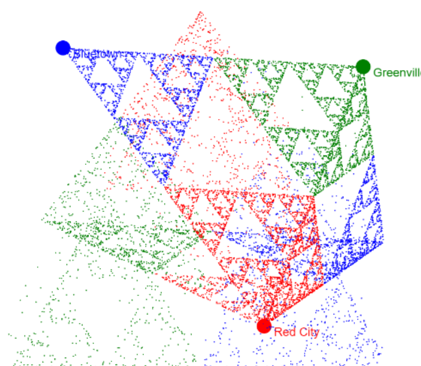
I set *this.selected* to *false* at the tail end of the *constructor* method code, and then use conditionals at the start of *display* (too lazy to make a whole different *update* method) to move the position if the city is selected and the mouse is still down. If the mouse is up, no city should be selected.

And finally, while the set-up above makes for some interesting visuals, you will probably want to build in a reset for the camper's trail whenever a city is moved.

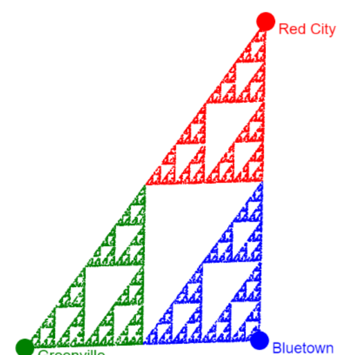
```
function draw() {
  background(255);
  for (let city of cities){
    city.display();
    if (city.selected){
      camper.sites = [];
    }
  }
```

A side-effect of clearing the campsite memory like this is that if you click and hold a city you'll see only the current camper's location.

Without reset:



With reset:



Challenges: Modify your code to work with 5 cities.

What happens if you go $\frac{3}{5}$ of the way instead of $\frac{1}{2}$?