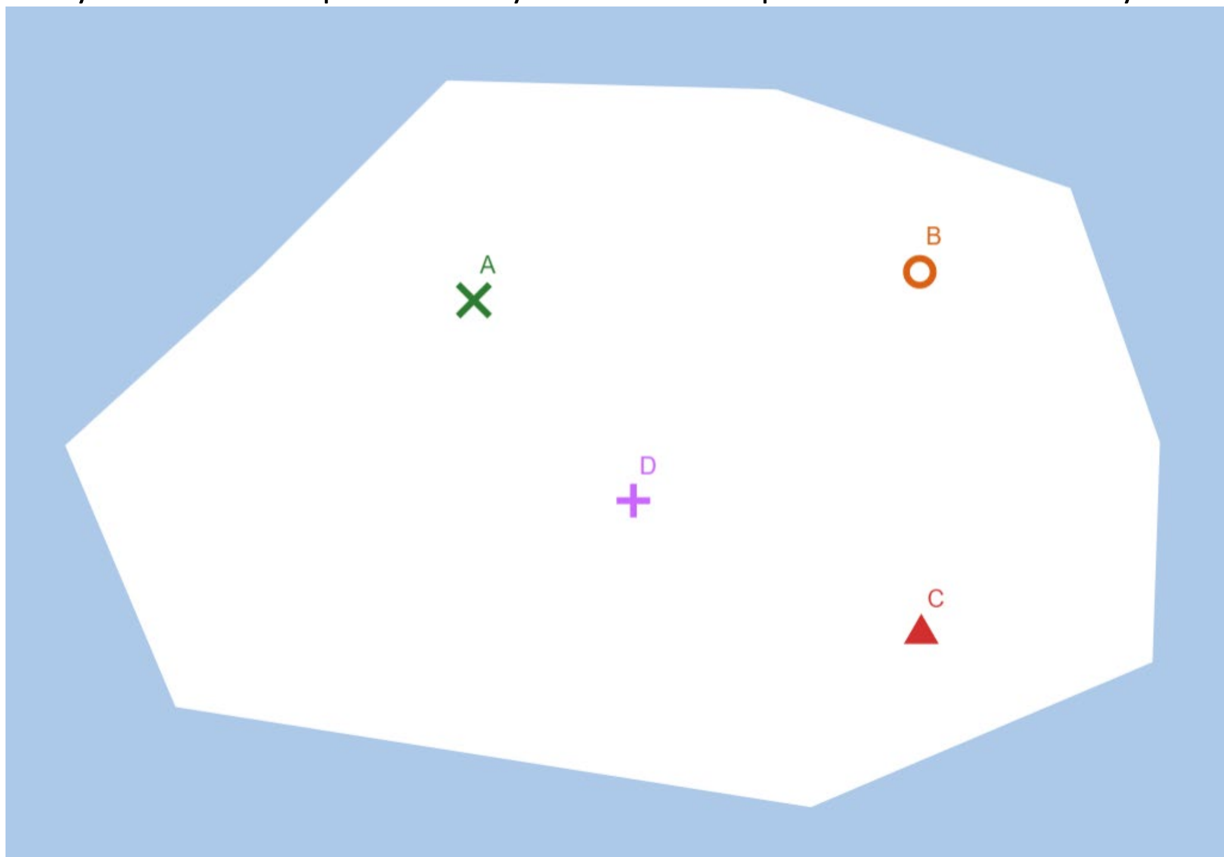
Voronoi Diagram

p5.js

This pattern occurs in cracking clay, in the formation of bubbles, in the arrangement of cells in a leaf, sections in a dragonfly's wing or patterns on a giraffe or a tortoise. Known as a Voronoi diagram, each cell corresponds to the region closest to a given point (or 'site').

Imagine the territory controlled by competing forces. If each force is located in a single city, and can only lay claim to territory beyond their city if they are the closest city to that territory, what would the shape of the corresponding city states look like?

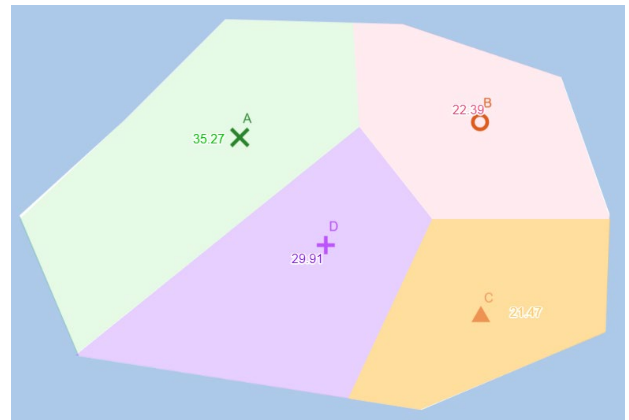
Give it a try with this example: who do you think ends up with the most territory?



(spoilers over the page)

One way to define these cells is in terms of the perpendicular bisectors between each pair of points: the area shaded green in the diagram opposite is simultaneously closer to A than B, closer to A than D and closer to A than C (although that final inequality ends up being irrelevant in this case).

One of the simplest ways, however, involves considering the problem from the perspective of an arbitrary point somewhere on the map: which of the sites is it closest to? Then colour accordingly.



JS object literals

Let's begin by making some arbitrary sites. These will have position and colour, but other than that are pretty basic, so a full-on class definition is probably overkill. This is a good opportunity to learn about 'object literals': JavaScript objects that are defined quickly and easily, with much of the flexibility of custom objects without the overhead of a class.

```
let sites = [];
let numSites = 5;

function setup() {
  createCanvas(400, 400);
  for (let i=0; i<numSites; i++){
    let site = {x: random(width),
                y: random(height),
                col: color(random(255), random(255), random(255))}
    sites.push(site);
  }
}

function draw() {
  background(220);
  noStroke();
  for (let site of sites){
    fill(site.col);
    circle(site.x, site.y, 20);
  }
}
```

We declare an empty array, then loop through adding our object literals, which hold just three parameters: *x*, *y* and *col*: position and colour.

Note the format: a variable name and its value, separated by a colon, followed by a comma and then the next "key-value pair". It may remind you

For now, the draw loop simply runs through each site and displays a circle on the screen to show its position.

You could also add code to allow addition of sites by clicking the mouse. Remember to include *return false* so that you don't get strange behaviour from touchscreens.

```
function mousePressed(){
  let site = {x: mouseX, y: mouseY,
              col: color(random(255), random(255), random(255))}
    sites.push(site);
  return false;
}
```

Looping through the pixel array

Recall (from the Mandelbrot project) that we can loop through the pixel array, accessing every pixel individually, and set its colour. First, some housekeeping:

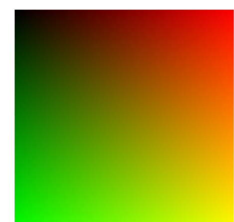
```
function setup() {
  createCanvas(400, 400);
  pixelDensity(1);
  loadPixels();
```

To keep things simple, even for HD displays, let's set the pixel density to 1, so that a 400 by 400 canvas will have 160000 pixels. Then use *loadPixels* so that we can access and modify them later.

```
function draw() {
  background(0);
  for (let j=0; j<height; j++){
    for (let i=0; i<width; i++){
      let index = 4*(j*width + i);
      let r = map(i, 0, width, 0, 255);
      let g = map(j, 0, height, 0, 255);
      let b = 0;
      pixels[index+0] = r;
      pixels[index+1] = g;
      pixels[index+2] = b;
      pixels[index+3] = 255;
    }
  }
  updatePixels();
}
```

I'm no longer displaying the sites in *draw*. Instead, I'm looping through every pixel by running a loop within a loop. The *index* value takes into account the fact that the pixel array is four times as long as the number of pixels with the first four values corresponding to the red, green, blue and alpha values of the first pixel, etc.

Don't forget to run *updatePixels()* at the very end, so that the changes get applied to the canvas.



I've mapped *r* and *g* values to *i* and *j* for now, so you can check it really works:

The distance function

Every pixel is now being individually referenced. We just need its colour to match that of the closest site. For this, we'll need a function or two…

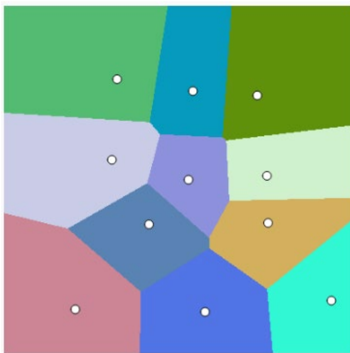
```
function sqDist(i, j, site){
  return (i - site.x)**2 + (j - site.y)**2
}

function bestColor(i, j){
  let col;
  let lowest = (width + height)**2;
  for (let site of sites){
    let d = sqDist(i, j, site);
    if (d < lowest){
      col = site.col;
      lowest = d;
    }
  }
  return col;
}
```

Any code that requires us to loop through every single pixel is necessarily going to have some performance issues, so the more efficient we can make this the better. A simple improvement we can build in at this stage is to compare the squared distance to each site rather than the actual distance.

The *bestColor* function starts with a (very high) upper bound and no defined colour, then loops through every site updating the minimum distance and switching allegiance until it reaches the end. Essentially a 'winner-stays-on' situation.

```
for (let j=0; j<height; j++){
  for (let i=0; i<width; i++){
    let index = 4*(j*width + i);
    let col = bestColor(i, j);
    pixels[index+0] = red(col);
    pixels[index+1] = green(col);
    pixels[index+2] = blue(col);
    pixels[index+3] = 255;
  }
}
```

Back in the *draw* loop, we can get the best colour for any given pixels simply by running our *bestColor* function. Note that *red*, *green* and *blue* extract the three parameters so that we can assign them to the appropriate places in the pixel array. Note that the fourth parameter, *alpha* corresponds to the opacity of the colour, hence set to maximum.



You should find that, although it may take a beat or so, this code can cope well with adding new points through mouse clicks, and can be run at 600 by 600 pixels without crashing!

Note that I've added back in the site locations – if you do this, just make sure the code that draws them runs *after* the pixel array is updated (otherwise they'll all be overwritten).

Other distance functions…

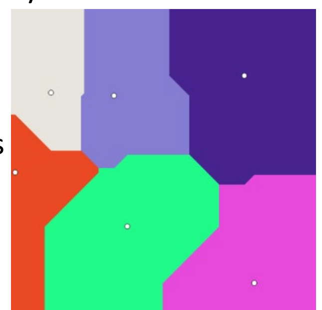
In mathematics, a 'metric' is defined as 'a function that gives a distance between any pair of points in the set'. You might think that 'distance' is not something that's up for discussion, but what you understand by distance is know as the 'Euclidean distance', calculated with Pythagoras: $\sqrt{(x-x_1)^2+(y-y_1)^2}$. What if, instead, you were assigning territory in New York, where the grid system of roads makes the distance from A to B equal to the sum of the horizontal and vertical displacement (rather than the root of the sum of their squares)? The formula for this so-called 'city-block metric' is $|x-x_1|+|y-y_1|$.
We can alter our own metric (aka 'distance function') really easily, and immediately observe the effect on the resulting Voronoi diagram:

```
function metric(i, j, site){
  return abs(i - site.x) + abs(j - site.y);
}

function bestColor(i, j){
  let col = sites[0].col;
  let lowest = metric(i, j, sites[0]);
  for (let site of sites){
    let d = metric(i, j, site);
    if (d < lowest){
      col = site.col;
      lowest = d;
    }
  }
  return col;
}
```

I've renamed *sqDist* since it won't necessarily be that any more. Note that *dist* is a protected name since it's a built-in *p5* function, so I'm using *metric* instead.

To avoid any issues if I mess with this function any further, I've modified *bestColor* and *lowest* so that they are defined before I loop through the sites.

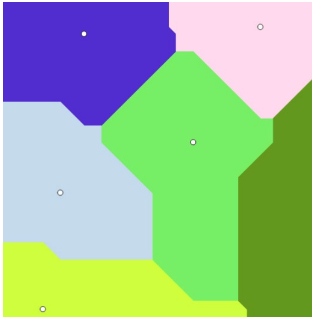
Notice how the boundary is made up of vertical, horizontal and 45° lines. This is the kind of shape a school catchment area might be within an American city.



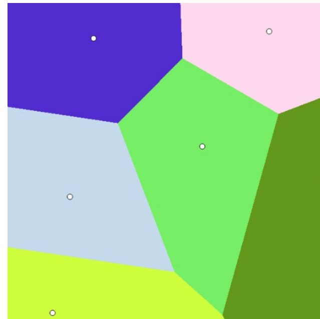
More curves!

We can, of course, go the other way. Instead of defining distance according to the sum of the absolute distances or the sum of the squares, what about the sum of the cubes, or the fourth powers?

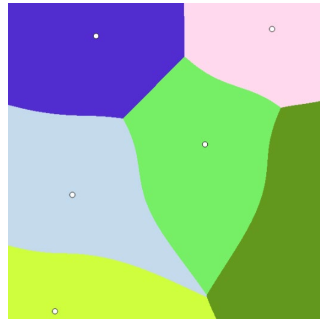
$d = |x - x_1| + |y - y_1|$



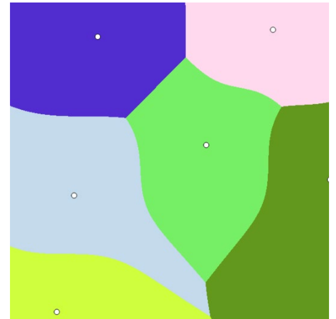
$d = |x - x_1|^2 + |y - y_1|^2$



$d = |x - x_1|^3 + |y - y_1|^3$



$d = |x - x_1|^4 + |y - y_1|^4$



Making the most of the pixel array

Working with every single pixel directly is somewhat inefficient (you might consider mathematical approaches to finding the set of polygons for the traditional tiling using the standard 2D Euclidean metric for a more efficient approach). However, it does have the advantage that we can generate more sophisticated patterns, incorporating blending instead of simply solid colours. What if, in addition to computing which site any given pixel is closest to, we worked out how close that pixel was to its nearest boundary? We could then represent each region using a gradient colour.

Keeping track of 'second place'

```
function bestColor(i, j){
  let col = sites[0].col;
  let lowest = metric(i, j, sites[0]);
  let second = lowest;
  for (let site of sites){
    let d = metric(i, j, site);
    if (d < lowest){
      col = site.col;
      second = lowest;
      lowest = d;
    }
    else if (d < second){
      second = d;
    }
  }
  let closeness = (second - lowest) / (second + lowest);
  return [col, closeness];
}
```

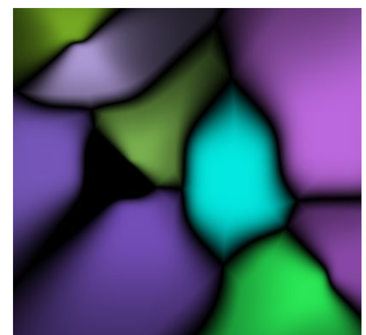
My *bestColor* function has been adapted to find not only the closest site, but also to keep track of the next closest, and return a measure of 'closeness': the difference between the two relative to the total distance between the sites. If a new closest site is found, the previous closest is relegated to second place, *second*, and in addition to checking for a new best, we check for a new second best too.

My *closeness* measure is not quite equivalent to the fraction of the distance from the closest to the second closest. It's close to 0 if the pixel is nearly equidistant from two sites, and close to 1 if the next closest is a long way away compared to the closest, but if, say the second closest is 90 units away compared to 10, we get $\frac{90-10}{100} = 0.8$, not 0.9. You can experiment with different measures here just like we played around with the distance metric. Recall that the values we're comparing here will change depending on the distance metric we use, so as you change those parameters, the shading patterns will also change.

Making the blended colour

```
for (let j=0; j<height; j++){
  for (let i=0; i<width; i++){
    let index = 4*(j*width + i);
    let [col, sf] = bestColor(i, j);
    pixels[index+0] = red(col)*sf;
    pixels[index+1] = green(col)*sf;
    pixels[index+2] = blue(col)*sf;
    pixels[index+3] = 255;
  }
}
```

All that's needed here, within the *draw* loop, is to amend the line that extracts the best colour to also extract our closeness measure (called *sf* for scale factor), and then use it to scale the brightness accordingly.



The image shown is a combination of this shading process and the $(x - x_1)^4 + (y - y_1)^4$ metric.