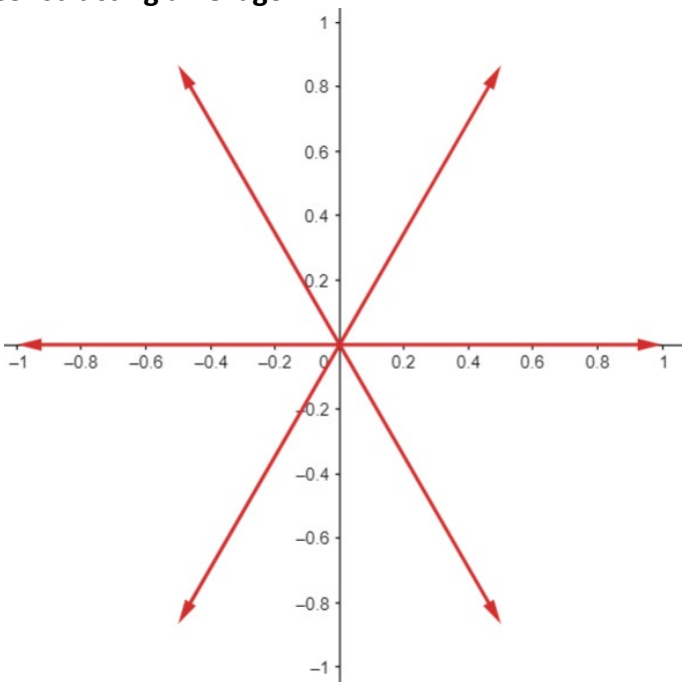
# Hex Grid

In a previous tutorial, we developed an efficient algorithm for detecting whether or not a give point lay within a convex polygon. This time we're going to make use of that tool to replicate a two-player pencil-and-paper game based on a grid of hexagons.

To begin with, we should define a hexagon object. If all our hexagons will be regular, and oriented the same way (so they'll tesselate), it should be possible to define each one simply with a position and a size. But first it'll be helpful to sit down with a pencil and paper so we can work out exactly how to draw a hexagon from a given centre and radius…

**Constructing a hexagon**




If we treat the radius as the distance from the centre to a vertex (which, conveniently, then also corresponds to the side length of the hexagon), we could think of the coordinates of the vertices in terms of vectors equally spaced about a central point.

For simplicity, we initially consider a radius of 1 and a centre of $(0,0)$. A bit of trigonometry gives us the coordinates $(1,0)$, $\left(\frac{1}{2},\frac{\sqrt{3}}{2}\right)$, $\left(-\frac{1}{2},\frac{\sqrt{3}}{2}\right)$, $(-1,0)$, $\left(-\frac{1}{2},-\frac{\sqrt{3}}{2}\right)$, $\left(\frac{1}{2},\frac{\sqrt{3}}{2}\right)$.

Note that scaling up all of these values by $r$ gives a hexagon with radius $r$, and translating by $\begin{pmatrix}x\\y\end{pmatrix}$ moves them to be centred about the point $(x,y)$.

**The Hexagon class**

```
1  class Hexagon{
2    constructor(x, y, r, color){
3      this.p = createVector(x, y);
4      this.r = r;
5      this.color = color;
6      this.getCoords();
7    }
8    getCoords(){
9      this.coords = [];
10     let [x, y, r, rt3] = [this.p.x, this.p.y, this.r, sqrt(3)];
11     this.coords.push(createVector(x + r   , y           ));
12     this.coords.push(createVector(x + r/2, y + rt3 * r/2));
13     this.coords.push(createVector(x - r/2, y + rt3 * r/2));
14     this.coords.push(createVector(x - r   , y           ));
15     this.coords.push(createVector(x - r/2, y - rt3 * r/2));
16     this.coords.push(createVector(x + r/2, y - rt3 * r/2));
17   }
18   display(){
19     noStroke();
20     fill(this.color)
21     beginShape();
22     for (let c of this.coords){
23       vertex(c.x, c.y);
24     }
25     endShape()
26   }
27 }
```

When I first construct my hexagon, in addition to storing the position and the radius as class attributes, I also want to generate the coordinates. I'll make this a separate method in case I ever need to recalculate (eg if at some point I want to move an existing hexagon).

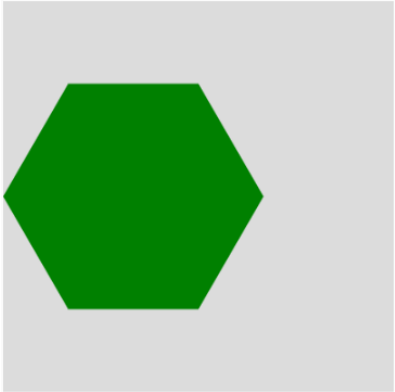
To make it easier to write out the coordinates, I create dummy variables *x*, *y*, *r* and *rt3* which correspond to the values I'll be referring to often. It should also slightly improve memory usage since I won't be recalculating $\sqrt{3}$ four times or pulling data from a vector a dozen times.

Display makes use of the *beginShape()* and *endShape()* built-in methods and our array of coordinates.

**Checking everything works**

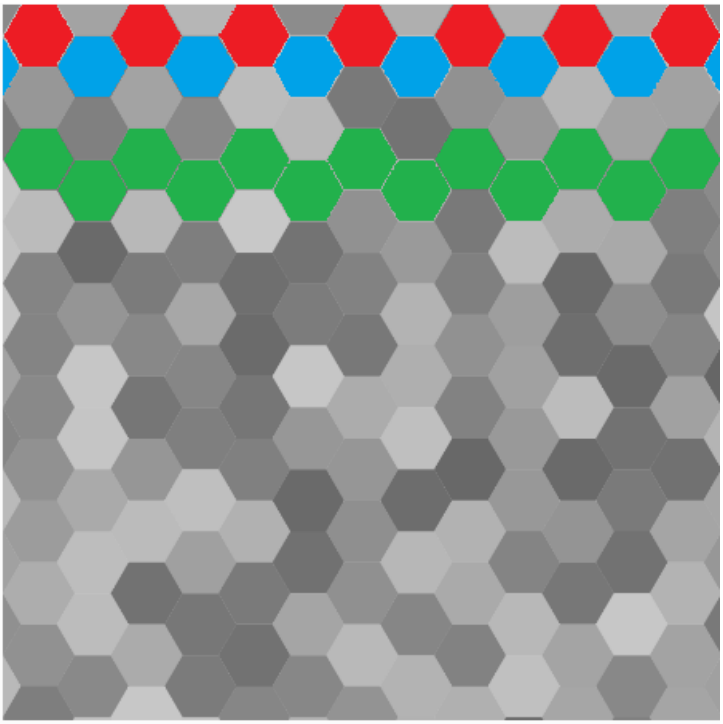
Before we get too far into our code, it's worth making a hexagon or two to check that the code behaves the way it should. *Remember, if you store your Hexagon class code in a separate file that you need to reference it in index.html along with sketch.js.*

```
let hexagons = [];

function setup() {
  createCanvas(400, 400);
  hexagons.push(new Hexagon(width/3, height/2, width/3, "green"))
}

function draw() {
  background(220);
  for (let hex of hexagons){
    hex.display();
  }
}
```



**Making a whole bunch of them**

One of the nice things about hexagons is how well they tesselate. But since they're not squares, our usual approach of nesting a couple of loops to iterate through rows and columns doesn't work. Or does it…?



While an attempt to define rows horizontally and columns vertically is doomed to failure (or at least massive coding headaches – trust me, I've tried), our two directions need not be perpendicular. For a start, we'd need to either skip over gaps (like the attempt at horizontal rows in red or blue) or wiggle up and down (like the attempt shown in green lower down).

Before reading ahead, see if you can find two directions where a single step of uniform distance in either of the two directions will take you to another hexagon…

And while you're at it, try to describe the directions you find in terms of vectors. They'll probably involve $\sqrt{3}$ just like the coordinates of a single hexagon do.

**The right perspective (or one of them at least)**

We can move down and right, like the blue and pink hexagons.

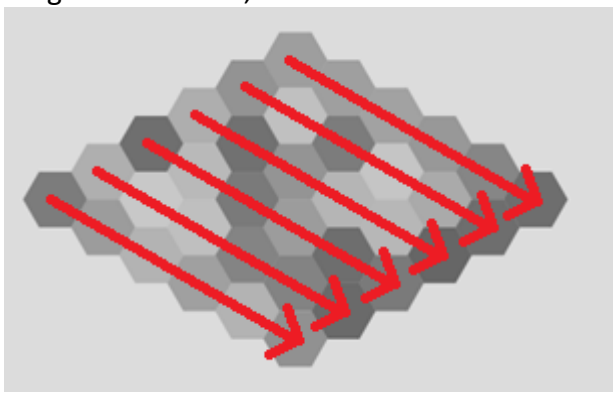
Each step along here would be $\begin{pmatrix} \frac{3r}{2} \\ \frac{r\sqrt{3}}{2} \end{pmatrix}$

Or we can move down and left, like the green and yellow hexagons.

Each step in this direction would be $\begin{pmatrix} -\frac{3r}{2} \\ \frac{r\sqrt{3}}{2} \end{pmatrix}$

Notice that in both cases the step length is exactly $r\sqrt{3}$, and the angle is $120°$ from vertical.

Using these vectors, we can start to see how our 2D array of vectors might form a grid of sorts:



Consider this grid. Although it isn't made of nice perpendicular rows and columns, we can still think of it as a 2 by 2 grid, with rows moving down and to the right. Each time we move to a new row, we move down and to the left.

Treating the top middle hexagon's position as $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$, can you work out a general expression for the position vector of the hexagon in 'row' $i$ and 'column' $j$?

If we consider the top middle hexagon above to be in position $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$, then add the down-right vector for each increment of $i$, and the down-left vector for each increment of $j$, we get:

$$\begin{pmatrix} \frac{3r}{2} \\ \frac{r\sqrt{3}}{2} \end{pmatrix} i + \begin{pmatrix} -\frac{3r}{2} \\ \frac{r\sqrt{3}}{2} \end{pmatrix} j = \begin{pmatrix} \frac{3r}{2}(i-j) \\ \frac{r\sqrt{3}}{2}(i+j) \end{pmatrix}$$

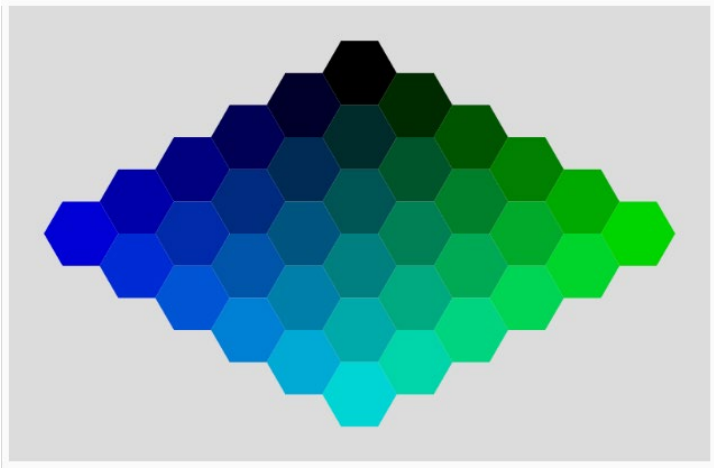
All that remains is to work out a suitable radius based on the width of the canvas. Note that, because of the offset nature of the hexagons, the above rhombus with six hexagons to a side is actually 17 radiuses across. Measuring along the long diagonal we have six full diameters and five edges in between. In general, that will be $n \times 2r + (n-1) \times r$, or just $(3n-1)r$. Vertically, we have six hexagons, but side to side rather than corner to corner, which corresponds to $(\sqrt{3}n)r$. We could use this to scale the size of the canvas as well as determining what radius to choose for a given number of hexagons.

```
let hexagons = [];
let n = 6;
let w = 400;
let border = 20;

function setup() {
  let r = (w - 2 * border) / (3 * n - 1);
  let h = 2 * border + sqrt(3) * n * r;
  createCanvas(w, h);
  for (let j=0; j<n; j++){
    for (let i=0; i<n; i++){
      let x = width/2 + 1.5 * r * (i - j);
      let y = border + 0.5 * sqrt(3) * r * (i + j + 1);
      let g = map(i, 0, n, 0, 255);
      let b = map(j, 0, n, 0, 255);
      let col = color(0, g, b);
      hexagons.push(new Hexagon(i, j, x, y, r, col));
    }
  }
}
```

I have also included a *border* variable so we can have some spacing between the sides.

The only other change to *x* and *y* is to add *width/2* to the *x* value and an extra helping of $\frac{\sqrt{3}}{2}r$ to the *y* value so that the top middle hexagon starts in the right place.

I've also built in a colour that changes depending on *i* and *j* so that I can check that my loops are behaving as predicted. I'm going to modify my Hexagon code as well to accept and store the *i* and *j* values for later use, too…



We should now have a pretty array of hexagons, nicely spaced and centred, with an appropriately scaled canvas containing them.

Now let's see if we can make them change colour when clicked.

I also like the idea of having them *temporarily* change colour when I hover the mouse pointer over them, so I can see clearly which will change.

We're going to need our *pointInPolygon* code from the previous project here. I suggest you put it in its own separate .js file, and don't forget to reference it in *index.html* so that the project can use the code.

```
function pointInPolygon(pointVector, vertexVectors){
  let vectors = []; // vectors from point to each vertex
  for (let i=0; i<vertexVectors.length; i++){
    vectors.push(p5.Vector.sub(vertexVectors[i], pointVector));
  }
  let values = []; // sign of each value corresponds to direction of turn from one vector to the next
  for (let i=0; i<vectors.length; i++){
    let u = vectors[i];
    let v = vectors[(i+1)%vectors.length]; // including last to first vector
    values.push(u.x*v.y-v.x*u.y);
  }
  for (let i=0; i<values.length - 1; i++){
    if (values[i] * values[i+1] < 0){ // sign change indicates direction change, hence point outside polygon
      return false
    }
  }
  return true;
}
```

Once this is imported, I can reference it as easily as this from within my *Hexagon* class:

```
containsPoint(p){
  return pointInPolygon(p, this.coords);
}
mouseOver(){
  let m = createVector(mouseX, mouseY);
  return this.containsPoint(m);
}
```

**Listening for the mouse**

```
display(){
  if (this.mouseOver()){
    if (mouseIsPressed){
      this.color = "red";
    }
    fill("red");
  }
  else{
    fill(this.color);
  }
```

When I display a given hexagon, I'll choose the fill colour based on where the mouse is. If the mouse is over the hexagon, it should be coloured red. If, in addition, the mouse button is pressed, that means this hexagon is currently being clicked on, and in that case I want to not only fill red for this frame, but to permanently change the hexagon's colour to red as well.

If not, revert to showing the hexagon with its own saved colour.

**Extension ideas**

- Use the *i* and *j* stored values for a hexagon to collate a list of its immediate neighbours. Use this to simulate weed spreading across a pond, or fire spreading across a prairie.

- A version of electric shuffleboard (look it up: it is as cool as it sounds) awards points to players based on how many hexagonal cells are 'owned' by their puck. Build in a way for a hexagon to determine whose puck is nearest, and be coloured in accordingly.