Vector Quadrilateral

In this project, we'll learn how to make a point on the screen that we can click and drag around with the mouse, and use them to create random quadrilaterals. We'll also investigate a neat result involving midpoints which you can prove using vectors.

A movable point

I can create a circle on the screen simply by using the *circle* command, and I could even have it appear wherever the mouse happens to be using *circle(mouseX, mouseY, 20)*. But to make a circle that can be picked up and dragged, then put down again, etc, I'm going to want my own class of object.

```
class MovablePoint{
  constructor(x, y){
    this.x = x;
    this.y = y;
    this.selected = false;
  }
  display(){
    noStroke();
    fill(0);
    circle(this.x, this.y, 20);
  }
  mouseOver(){
    return (this.x - mouseX)**2 + (this.y - mouseY)**2 < 10;
  }
}
```

The name of my object is capitalized, and the *constructor* function simply stores the position of the object as well as a 'flag' variable that will be used to keep track of whether it is currently selected or not.

In *display*, I draw the circle at its current position, and the *mouseover* helper function will be used to determine if a given *MovablePoint* is clicked on.

Trying it out

Before we get too far ahead of ourselves, we should make some of these points and check that they display as expected.

```
let pts = [];
let n = 4;

function setup() {
  createCanvas(400, 400);
  for (let i=0; i<n; i++){
    pts.push(new MovablePoint(random(width), random(height)));
  }
}

function draw() {
  background(220);
  for (let i=0; i<n; i++){
    pts[i].display();
  }
```

I'll make a global variable *pts*: an array in which to store my *MovablePoint* objects. The value *n* may change later, so instead of hard-coding 4 into my code, I'll set it up to make it easier to change later.

My points are assigned randomly, anywhere on the screen, then each time the draw loop runs, each one is displayed.

The only problem with this, as a test, is that it won't show me if the *mouseOver* method is doing its job. Let's make the points look a bit different when the mouse hovers over them.

```
display(){
  noStroke();
  fill(0);
  if (this.mouseOver()){
    fill(128);
  }
  circle(this.x, this.y, 20);
}
mouseOver(){
  return (this.x - mouseX)**2 + (this.y - mouseY)**2 < 10**2;
}
```

This code runs in the *display* method within the *MovablePoint* class, making the circle appear grey instead of black whenever the mouse is hovering over it.

Straight away, I can detect the mistake I made – the dots do change colour if I'm close enough to the centre of them, but I have to be very close. My *mouseOver* method compared the *square* of the distance to the radius, instead of to the *square* of the radius.

You could also have the points displayed slightly larger when the mouse is over them, or even increase the 'detection radius' so that the mouse doesn't have to be directly over the point to select it.

Click and Drag

P5 can determine whether the mouse is pressed using the built-in variable *mouseIsPressed*, and we can have it run a block of code once on a mouse click using the built-in function *mousePressed()*. Consider how we might combine these two things, along with the *selected* flag variable in the *MovablePoint* class code to produce the kind of behaviour we want. Specifically, we need the following:

- If you click and drag a point, it will update its position to match that of the mouse.
- When you release the mouse, the dragged point will be released, staying where it was dropped.
- When you click and drag a point, only that point and no other will be dragged. Consider the edge case of two overlapping points, where the mouse pointer is close enough to both – we want to select only one of them (preferably the one on top).

In the *MovablePoint* class:

```
moveTo(x, y){
  this.x = x;
  this.y = y;
}
```

This is a quick helper function to make it easy to reposition my points. Although for the time being I'll be using the mouse's position, a more general use function may be handy later.

The *mousePressed* function:

```
function mousePressed(){
  for (let i=n-1; i>-1; i--){
    if (pts[i].mouseOver()){
      pts[i].selected = true;
    }
  }
  return false;
}
```

This code loops *backwards* through the array of points, so that the most recent to be displayed will be the first to be tested. This should help us ensure that, where there's an overlap conflict, the point most visible will be the one we assume the user intends to select. If they want the other one, they can drag the top one out of the way first!
The *return false* at the end is just to ensure predictable behaviour on a touchscreen for all browsers.

In the *draw* loop:

```
function draw() {
  background(220);
  for (let i=0; i<n; i++){
    pts[i].display();
    if (pts[i].selected){
      pts[i].moveTo(mouseX, mouseY);
    }
  }
}
```

Selecting points won't do much good unless we do something with them once they're selected. This code checks to see if a given point is selected, and if so, moves it to correspond to the mouse's position.

Play around with the sketch for a bit, and see if you can work out which of the bullet-point conditions it doesn't yet meet. See if you can work out how the code above would need to be changed to fix each problem.

Deselecting points

In the current iteration of our code, even after we lift our finger off the mouse button, we can't shake off the moving point. We could use an additional test in *draw* to switch off the *selected* flag in the case where the mouse button is seen to no longer be down.

```
function draw() {
  background(220);
  for (let i=0; i<n; i++){
    pts[i].display();
    if (pts[i].selected){
      if (mouseIsPressed){
        pts[i].moveTo(mouseX, mouseY);
      }
      else{
        pts[i].selected = false;
      }
    }
  }
}
```

If a particular point is selected, we then go on to check if the mouse button is currently down. If it is, we move it as before, but if not, we deselect that point.

Much better. But keep playing – there's more amiss. What undesirable behaviour do we still need to fix?

One at a time please

```
function mousePressed(){
  for (let i=n-1; i>-1; i--){
    if (pts[i].mouseOver()){
      pts[i].selected = true;
      return false;
    }
  }
  return false;
}
```

While the points can now be picked up and dropped as needed, it is possible to select more than one. If, for instance, two points overlap, and we click on both at once, both will be selected, and from that point on will only ever appear in the same spot. The easiest fix is for the *mousePressed* function, which sets the *selected* flag, to give a return value as soon as it's selected one point. Since our loop is running backwards, this should select only the one in front, then terminate.

Drawing a shape

P5 has a built-in *quad* function for drawing quadrilaterals, by passing in eight ordinates – the coordinates of all four points. But if we want more flexibility (eg if we change the number of points) we should use a more general function. Because I plan on doing this with different sets of points shortly, I'll make a dedicated function for it, but we'll make use of the built-in *beginShape* and *endShape* functions.

In the *draw* loop:

```
function draw() {
  background(220);
  stroke("red");
  drawShape(pts);
  for (let i=0; i<n; i++){
    pts[i].display();
```

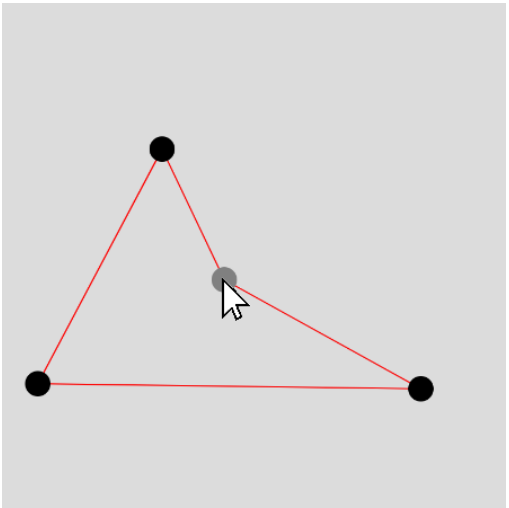
Often it's easier to work out where and how a function is to be used before creating it. That way, you'll have a clearer sense of what it should do, and what kind of return value (if any) it should generate.

Our *drawShape* function:

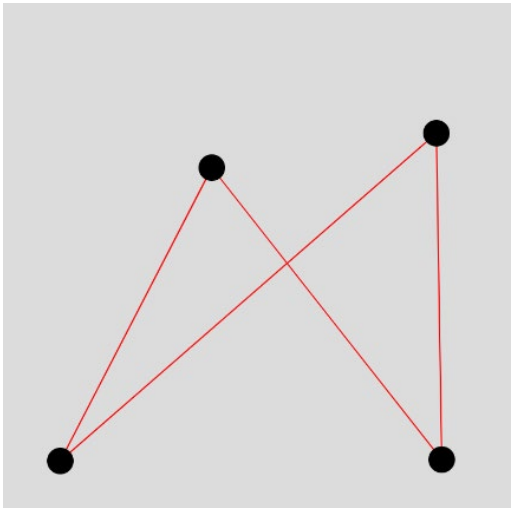
```
function drawShape(pts){
  noFill();
  beginShape();
  for (let p of pts){
    vertex(p.x, p.y);
  }
  endShape(CLOSE);
}
```

This new function uses *beginShape* and *endShape* to bracket a loop which goes through all the points and plots a vertex. The key word *CLOSE* as an argument in *endShape* ensures that a final line is drawn between the last vertex and the very first.

Note that *pts* here is a local variable, so I can reuse the function *drawShape* later if I have a different set of points. All they will need is x and y attributes, so they don't necessarily have to be *MovablePoint* objects.



Now you should have a four-sided shape with movable corners. Note that, in addition to slightly unorthodox shapes like the concave quadrilateral to the left, you could also generate self-intersecting quadrilaterals like the one on the right.



Constructing midpoints

Next, I want to find the midpoints of each of those line segments. I can loop through my list of points, finding the halfway mark between points 0 and 1, then points 1 and 2, then points 2 and 3, and finally points 3 and 0 (first to last). The simplest way to do this involves the modulo operator, %, where an expression like n % 4 gives the *remainder* (or 'residue') when n is divided by 4. For any integer n, this will always return one of four values: 0, 1, 2 or 3.

I'm going to make a standalone function whose job it is to generate midpoints for a given list of points (or point-like objects such as my *MovablePoint* – all it should require is x and y attributes).

The midpoint function

```
function createMidpoints(pts){
  let midpoints = [];
  for (let i=0; i<pts.length; i++){
    let a = createVector(pts[i].x, pts[i].y);
    let b = createVector(pts[(i+1)%4].x, pts[(i+1)%4].y);
    let midpoint = a.add(b).mult(0.5);
    midpoints.push(midpoint);
  }
  return midpoints;
}
```

This function makes use of p5's *Vector* class, generating vectors *a* and *b*, the start and end of each line segment. Then, by adding the two and halving the result, we construct a point exactly halfway between them. These vectors are then pushed into an array which is returned.

Joining the dots

```
function draw() {
  background(220);
  stroke("red");
  drawShape(pts);
  let midpoints = createMidpoints(pts);
  stroke("blue");
  drawShape(midpoints);
  for (let i=0; i<n; i++){
    pts[i].display();
```

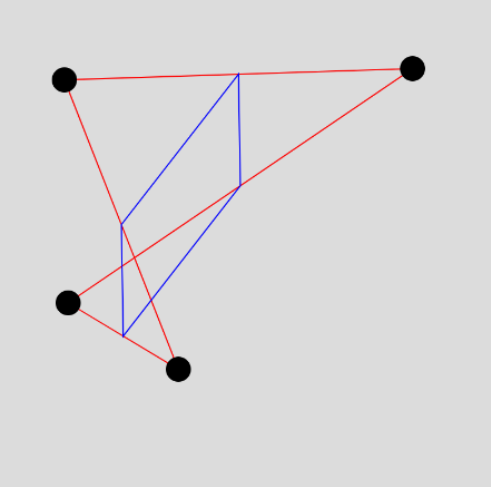
The reason this function is called in *draw* rather than *setup* is that when we change the position of the movable points, we also need to recalculate the midpoints.

With our dedicated functions, it's quick and easy to draw the shape formed by linking the midpoints.

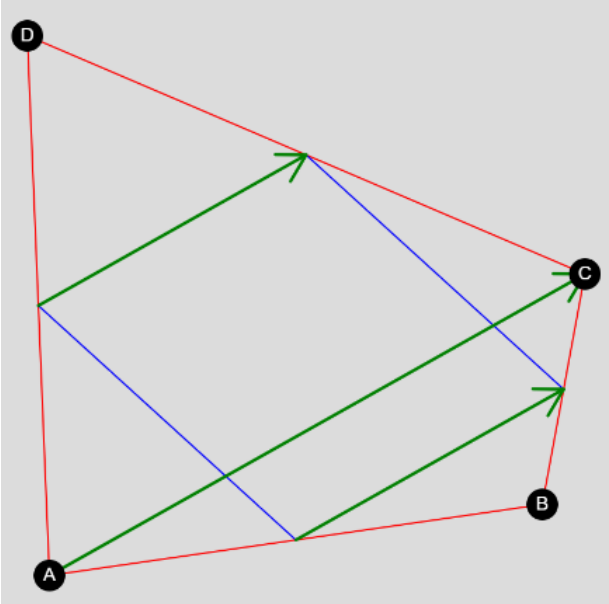
Try playing around with the position of the corners.

What do you notice?

Curiously enough, the blue quadrilateral appears to be a perfect parallelogram even if the original quadrilateral is not simple and convex.
The opposite of simple is self-intersecting – ie, the boundary lines intersect other than at a corner, making 'inside' and 'outside' hard to define.
The opposite of convex is concave – ie there's at least one corner that bows inwards, forming a reflex interior angle).



Why is the shape formed by connecting the midpoints always a parallelogram?



This result is provable using ordinary coordinate geometry – label the points (x_1, y_1), (x_2, y_2) etc, and compute the midpoints, find the distance and gradient between each pair, etc.

But a much neater trick is to consider the *vector* that corresponds to a journey across the shape.

Note that $\overrightarrow{AC} = \overrightarrow{AB} + \overrightarrow{BC}$, and $\overrightarrow{AC} = \overrightarrow{AD} + \overrightarrow{DC}$.

The vector from the midpoint of AB to the midpoint of BC must be equivalent to $\frac{1}{2}\overrightarrow{AB} + \frac{1}{2}\overrightarrow{BC}$.

Since this is also $\frac{1}{2}\left(\overrightarrow{AB} + \overrightarrow{BC}\right)$, it must be $\frac{1}{2}\overrightarrow{AC}$.

The same reasoning applies to the other pair of midpoints, and since both vectors are equal, they have the same **length** and same **direction**, which is precisely the condition required for opposite sides of a parallelogram.

Bonus: can you modify your code to give your movable points labels? And how would you make arrows?