Interactive vector motion

To finish out this section, let's try something a bit more complex and a great deal more useful. We'll dynamically calculate an object's acceleration according to a rule stated in Algorithm #3 — *the object accelerates towards the mouse.*

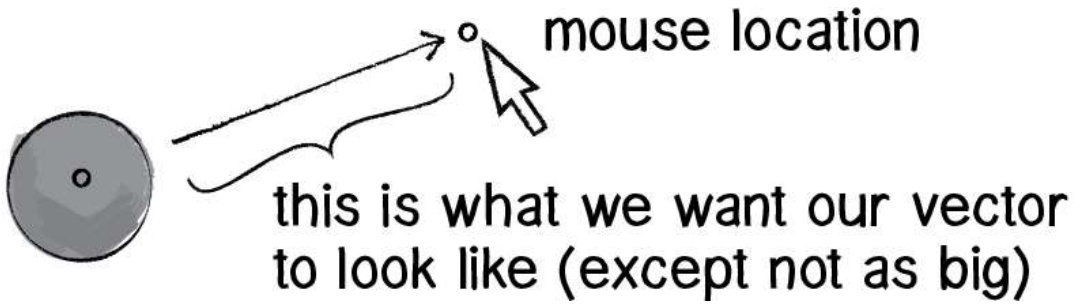

Diagram of mouse acceleration vector

Anytime we want to calculate a vector based on a rule or a formula, we need to compute two things: ***magnitude*** and ***direction***. Let's start with direction. We know the acceleration vector should point from the object's location towards the mouse location. Let's say the object is located at the point `(x,y)` and the mouse at `(mouseX,mouseY)`.

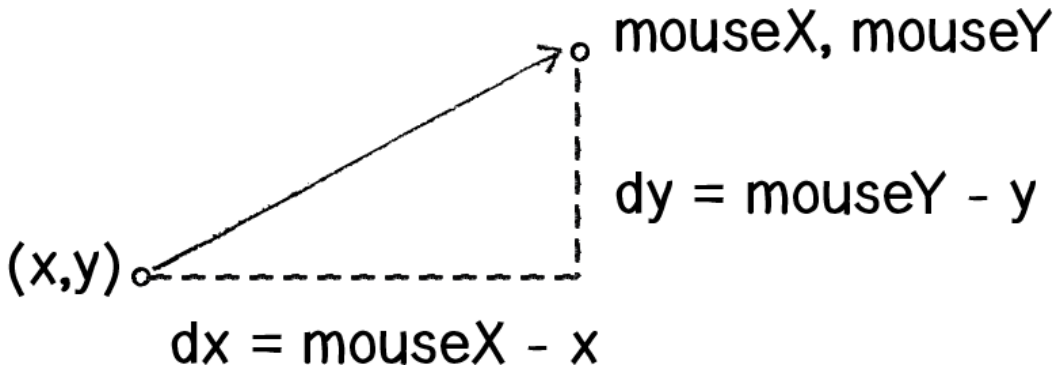

Diagram of dx, dy

In that diagram, we see that we can get a vector `(dx,dy)` by subtracting the object's location from the mouse's location.:

- `dx = mouseX - x`
- `dy = mouseY - y`

Let’s rewrite the above using `PVector` syntax. Assuming we are in the `Mover` object definition and thus have access to the object’s `PVector` location, we then have:

```
var mouse = new PVector(mouseX, mouseY);
// Look! We’re using the static sub() because we want a completely new
PVector
var dir = PVector.sub(mouse, location);
```

We now have a `PVector` that points from the mover’s location all the way to the mouse. If the object were to actually accelerate using that vector, it would appear instantaneously at the mouse location. This does not make for good animation, of course, and what we want to do now is decide how quickly that object should accelerate toward the mouse.

In order to set the magnitude (whatever it may be) of our acceleration PVector, we must first ___ that direction vector. That’s right, you said it. *Normalize*. If we can shrink the vector down to its unit vector (of length one) then we have a vector that tells us the direction and can easily be scaled to any value. One multiplied by anything equals anything.

```
var anything = ??;
dir.normalize();
dir.mult(anything);
```

To summarize, we take the following steps:

1. Calculate a vector that points from the object to the target location (mouse)
2. Normalize that vector (reducing its length to 1)
3. Scale that vector to an appropriate value (by multiplying it by some value)
4. Assign that vector to acceleration
   Here's what the program looks like, with those steps fully implemented:

```
var Mover = function() {
  this.position = new PVector(width/2, height/2);
  this.velocity = new PVector(0, 0);
  this.acceleration = new PVector(0, 0);
};

Mover.prototype.update = function() {
  var mouse = new PVector(mouseX, mouseY);
  var dir = PVector.sub(mouse, this.position);
  dir.normalize();
  dir.mult(0.5);
  this.acceleration = dir;
  this.velocity.add(this.acceleration);
  this.velocity.limit(5);
  this.position.add(this.velocity);
};

Mover.prototype.display = function() {
  stroke(0);
  strokeWeight(2);
  fill(127);
  ellipse(this.position.x, this.position.y, 48, 48);
};

Mover.prototype.checkEdges = function() {

  if (this.position.x > width) {
```

```
      this.position.x = 0;
    } else if (this.position.x < 0) {
      this.position.x = width;
    }

    if (this.position.y > height) {
      this.position.y = 0;
    } else if (this.position.y < 0) {
      this.position.y = height;
    }
};

var mover = new Mover();

var draw = function() {
    background(255, 255, 255);

    mover.update();
    mover.checkEdges();
    mover.display();
};
```

You may be wondering why the circle doesn't stop when it reaches the target. It's important to note that the object moving has no knowledge about trying to stop at a destination; it only knows where the destination is and tries to go there as quickly as possible. Going as quickly as possible means it will inevitably overshoot the location and have to turn around, again going as quickly as possible towards the destination, overshooting it

again, and so on and so forth. Stay tuned; in later sections we'll learn how to program an object to ***arrive*** at a location (slow down on approach).

This example is remarkably close to the concept of gravitational attraction (in which the object is attracted to the mouse location). Gravitational attraction will be covered in more detail in the next section. However, one thing missing here is that the strength of gravity (magnitude of acceleration) is inversely proportional to distance. This means that the closer the object is to the mouse, the faster it accelerates.

Let's see what this example would look like with an array of movers (rather than just one).

```
var Mover = function() {
  this.position = new PVector(random(width), random(height));
  this.velocity = new PVector(0, 0);
  this.acceleration = new PVector(0, 0);
};

Mover.prototype.update = function() {
    var mouse = new PVector(mouseX, mouseY);
    var dir = PVector.sub(mouse, this.position);
    dir.normalize();
    dir.mult(0.2);
    this.acceleration = dir;
    this.velocity.add(this.acceleration);
    this.velocity.limit(5);
    this.position.add(this.velocity);
};
```

```
Mover.prototype.display = function() {
  stroke(0);
  strokeWeight(2);
  fill(127);
  ellipse(this.position.x, this.position.y, 10, 10);
};

Mover.prototype.checkEdges = function() {

  if (this.position.x > width) {
    this.position.x = 0;
  }
  else if (this.position.x < 0) {
    this.position.x = width;
  }

  if (this.position.y > height) {
    this.position.y = 0;
  }
  else if (this.position.y < 0) {
    this.position.y = height;
  }
};

var movers = [];

for (var i = 0; i < 20; i++) {
```

```
    movers[i] = new Mover();
}

draw = function() {
    background(255, 255, 255);
    for (var i = 0; i < movers.length; i++) {
        movers[i].update();
        movers[i].display();
    }
};
```