Chapter 6. Autonomous Agents

"This is an exercise in fictional science, or science fiction, if you like that better."
—Valentino Braitenberg

6.1 Forces from within

Believe it or not, there is a purpose. Well, at least there's a purpose for the first five chapters of this book. We could stop right here; after all, we've looked at several different ways of modeling motion and simulating physics. Angry Birds, here we come!

Still, let's think for a moment. Why are we here? The *nature* of code, right? What have we been designing so far? Inanimate objects. Lifeless shapes sitting on our screens that flop around when affected by forces in their environment. What if we could breathe life into those shapes? What if those shapes could live by their own rules? Can shapes have hopes and dreams and fears? This is what we are here in this chapter to do—develop *autonomous agents*.

The term ***autonomous agent*** generally refers to an entity that makes its own choices about how to act in its environment without any influence from a leader or global plan. For us, "acting" will mean moving. This addition is a significant conceptual leap. Instead of a box sitting on a boundary waiting to be pushed by another falling box, we are now going to design a box that has the ability and "desire" to leap out of the way of that other falling box, if it so chooses. While the concept of forces that come from within is a major shift in our design thinking, our code base will barely change, as these desires and actions are simply that—*forces*.

Here are three key components of autonomous agents that we'll want to keep in mind as we build our examples.

- **An autonomous agent has a *limited* ability to perceive environment.** It makes sense that a living, breathing being should have an awareness of its environment. What does this mean for us, however? As we look at examples in this chapter, we will point out programming techniques for allowing objects to store references to other objects and therefore "perceive" their environment. It's also crucial that we consider the word ***limited*** here. Are we designing a all-knowing rectangle that flies around a Processing window aware of everything else in that window? Or are we creating a shape that can only examine any other object within 15 pixels of itself? Of course, there is no right answer to this question; it all depends. We'll explore some possibilities as we move forward. For a simulation to feel more "natural," however, limitations are a good thing. An insect, for example, may only be aware of the sights and smells that immediately surround it? For a real-world creature, we could study the exact science of these limitations. Luckily for us, we can just make stuff up and try it out.

- **An autonomous agent processes the information from its environment and calculates an action.** This will be the easy part for us, as the action is a force. The environment might tell the agent that there's a big scary-looking shark swimming right at it, and the action will be a powerful force in the opposite direction.

- **An autonomous agent has no leader**. This third principle is something we care a little less about. After all, if you are designing a system where it makes sense to have a leader barking commands at various entities, then that's what you'll want to implement. Nevertheless, many of these examples will have no leader for an important reason. As we get to the end of this chapter and examine group behaviors, we will look at designing collections of autonomous agents that exhibit the properties of complex systems— intelligent and structured group dynamics that emerge not from a leader, but from the local interactions of the elements themselves.

In the late 1980s, computer scientist Craig Reynolds (http://www.red3d.com/cwr/) developed algorithmic steering behaviors for animated characters. These behaviors allowed individual elements to navigate their digital environments in a "lifelike" manner with strategies for fleeing, wandering, arriving, pursuing, evading, etc. Used in the case of a single autonomous agent, these behaviors are fairly simple to understand and implement. In addition, by building a system of multiple characters that steer themselves according to simple locally based rules, surprising levels of complexity emerge. The most famous example is Reynolds's "boids" model for "flocking/swarming" behavior.

6.2 Vehicles and Steering

Now that we understand the core concepts behind autonomous agents, we can begin writing the code. There are many places we could start. Artificial simulations of ant and termite colonies are fantastic demonstrations of systems of autonomous agents (for more, I encourage you to read *Turtles, Termites, and Traffic Jams* by Mitchel Resnick). However, we want to start by examining agent behaviors that build on the work we've done in the first five chapters of this book: modeling motion with vectors and driving motion with forces. And so it's time to rename our Mover class that became our Particle class once again. This time we are going to call it ***Vehicle***.

```
class Vehicle {

  PVector location;
  PVector velocity;
  PVector acceleration;
                             $$ What else do we need to add?
```

In his 1999 paper "Steering Behaviors for Autonomous Characters", Reynolds uses the word "Vehicle" to describe his autonomous agents, so we will follow suit.

Why Vehicle?

In 1986, Italian neuroscientist and cyberneticist Valentino Braitenberg described a series of hypothetical vehicles with simple internal structures in his book *Vehicles: Experiments in Synthetic Psychology*. Braitenberg argues that his extraordinarily simple mechanical vehicles manifest behaviors such as fear, aggression, love, foresight, and optimism. Reynolds took his inspiration from Braitenberg, and we'll take ours from Reynolds.

Reynolds describes the motion of *idealized* vehicles (idealized because we are not concerned with the actual engineering of such vehicles, but simply assume that they exist and will respond to our rules) as a series of three layers—Action Selection, Steering, and Locomotion.

1. **Action Selection.** A Vehicle has a goal (or goals) and can select an action (or a combination of actions) based on that goal. This is essentially where we left off with autonomous agents. The vehicle takes a look at its environment and calculates an action based on a desire: "I see a zombie marching towards me. Since I don't want my brains to be eaten, I'm going to flee from the zombie." The goal is to keep one's brains and the action is to flee. Reynolds's paper describes many goals and associated actions such as: seek a target, avoid an obstacle, and follow a path. In a moment, we'll start building these examples out with Processing code.

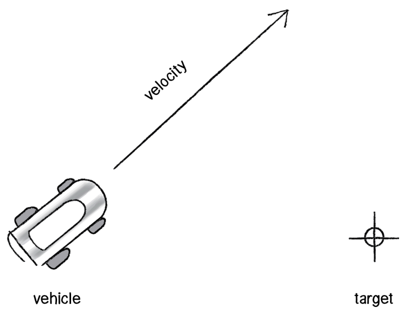
2. **Steering**. Once an action has been selected, the vehicle has to calculate its next move. For us, the next move will be a force; more specifically, a *steering* force. Luckily, Reynolds has developed a simple steering force formula that we'll use throughout the examples in this chapter: *Steering Force = Desired Velocity minus Current Velocity*. We'll get into the details of this formula and why it works so effectively in the next section.

3. **Locomotion**. For the most part, we're going to ignore this third layer. In the case of fleeing zombies, the locomotion could be described as "left foot, right foot, left foot, right foot, as fast as you can." In our Processing world, however, a rectangle or circle or triangle's actual movement across a window is irrelevant given that it's all an illusion in the first place. Nevertheless, this isn't to say that you should ignore locomotion. You will find great value in thinking about the locomotive design of your vehicle and how you choose to animate it. The examples in this chapter will remain visually bare, and a good exercise would be to elaborate on the animation style —could you add spinning wheels or oscillating paddles or shuffling legs?

Ultimately, the most important layer for you to consider is #1 -- *Action Selection*. What are the elements of your system and what are their goals? In this chapter, we are going to look at a series of steering behaviors (i.e. actions): seek, flee, follow a path, follow a flow field, flock with your neighbors, etc. It's important to realize, however, that the point of understanding how to write the code for these behaviors is not because you should use them in all of your projects. Rather, these are a set of building blocks, a foundation from which you can design and develop vehicles with creative goals and new and exciting behaviors. And even though we will think literally in this chapter (follow that pixel), you should allow yourself to think more abstractly (like Braitenberg). What would it mean for your vehicle to have "love" or "fear" as its goal, its driving force? Finally (and we'll address this later in the chapter) you won't get very far by developing simulations with only one action. Yes, our first example will be "seek a target." But for you to be creative—to make these steering behaviors *your own*—it will all come down to mixing and matching multiple actions within the same vehicle. So view these examples not as singular behaviors to be emulated, but as pieces of a larger puzzle that you will eventually assemble.

6.3 The Steering Force

We can entertain ourselves by discussing the theoretical principles behind autonomous agents and steering as much as we like, but we can't get anywhere without first understanding the concept of a steering force. Consider the following scenario. A "Vehicle" moving with velocity desires to seek a target.



Its goal and subsequent action is to seek the target in the above figure. If you think back to Chapter 2, you might begin by making the target an "attractor" and apply a gravitational force that pulls the vehicle to the target. This would be a perfectly reasonable solution, but conceptually it's not what we're looking for here. We don't want to simply calculate a force that pushes the Vehicle towards its target; rather, we are asking the Vehicle to make an intelligent decision to steer towards the target based on its perception of its state and environment (i.e. how fast and in what direction is it currently moving). The vehicle should look at how it desires to move (a vector pointing to the target), compare that goal with how quickly it is currently moving (its velocity), and apply a force accordingly.

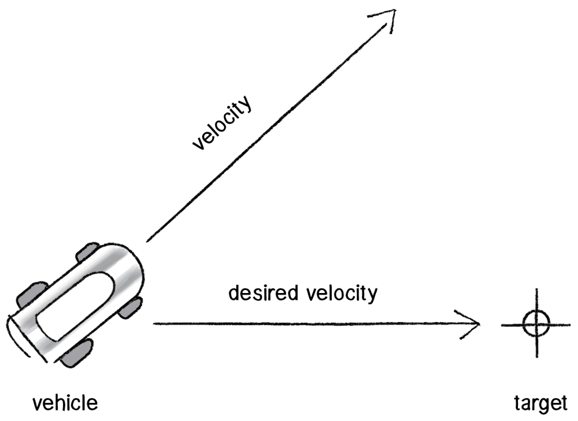
STEERING FORCE = DESIRED VELOCITY - CURRENT VELOCITY

Or as we might write in Processing:

```
PVector steer = PVector.sub(desired,velocity);
```

In the above formula, velocity is no problem. After all, we've got a variable for that. However, we don't have the desired velocity; this is something we have to calculate. Let's take a look at Figure X again. If we've defined the vehicle's goal as "seeking the target", then its desired velocity is a vector that points from its current location to the target location. Assuming a PVector target, we then have:

```
PVector desired = PVector.sub(target,location);
```



But this isn't particularly realistic. What if we have a very high-resolution window and the target is thousands of pixels away? Sure, the vehicle might desire to teleport itself instantly to the target location with a massive velocity, but this won't make for an effective animation. What we really want to say is:

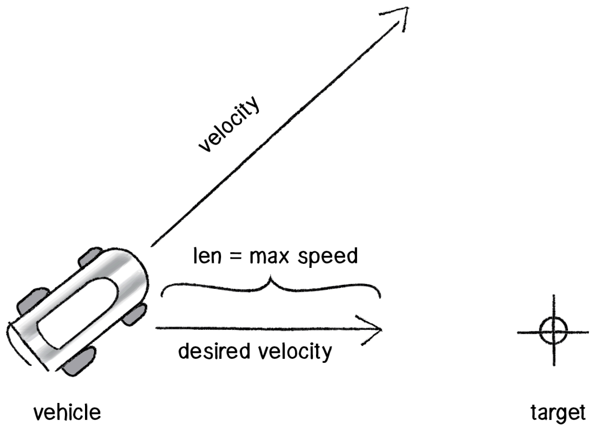
The vehicle desires to move towards the target at maximum speed.

In other words, the vector should point from location to target and with a magnitude equal to maximum speed (i.e. the fastest the vehicle can go.) So first, we need to make sure we add a variable in our Vehicle class to store maximum speed.

```
class Vehicle {
  PVector location;
  PVector velocity;
  PVector acceleration;
  float maxspeed;                  $$ Maximum speed
```

Then, in our desired velocity calculation, we scale according to maximum speed.

```
PVector desired = PVector.sub(target,location);
desired.normalize();
desired.mult(maxspeed);
```

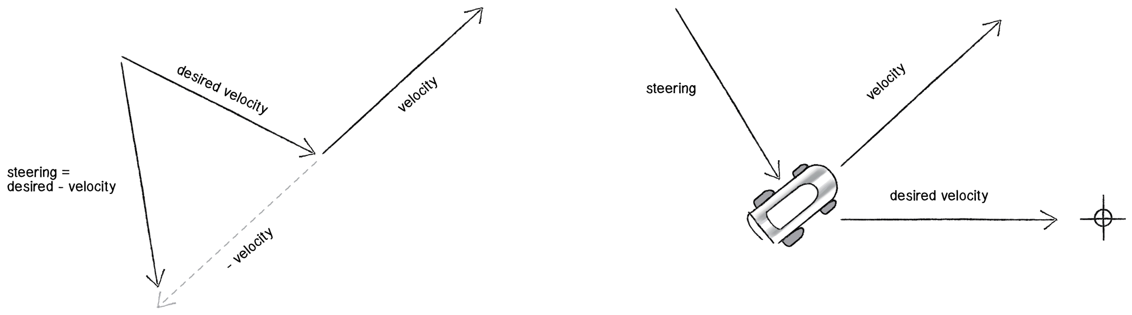


Putting this all together, we can write a function called ***seek()*** that receives a PVector target and calculates a steering force towards that target.

```
void seek(PVector target) {
  PVector desired = PVector.sub(target,location);
  desired.normalize();
  desired.mult(maxspeed);          $$ Calculating the desired velocity to target at max speed

  PVector steer = PVector.sub(desired,velocity);
                                   $$ Reynolds formula for steering force
  applyForce(steer);               $$ Using our physics model and applying the force
}                                  to the object's acceleration
```

Note how in the above function we finish by passing the steering force into ***applyForce()***. This assumes that we are basing this example on the foundation we built in Chapter 2. However, you could just as easily use the steering force with Box2D's ***applyForce()*** function or toxiclibs' ***addForce()*** function.

So why does this all work so well? Let's see what the steering force looks like relative to the vehicle and target locations.



[ALL THREE VECTORS NEED TO MATCH IN EACH ILLUSTRATION]

Again, notice how this is not at all the same force as gravitational attraction. Remember one of our principles of autonomous agents: An autonomous agent has a *limited* ability to perceive its environment. Here is that ability, subtly embedded into Reynolds's steering formula. If the vehicle weren't moving at all (zero velocity) desired minus velocity would be equal to desired. But this is not the case. The vehicle is aware of its own velocity and its steering force compensates accordingly. This creates a more active simulation, as the way in which the vehicle moves towards the targets depends on the way it is moving in the first place.

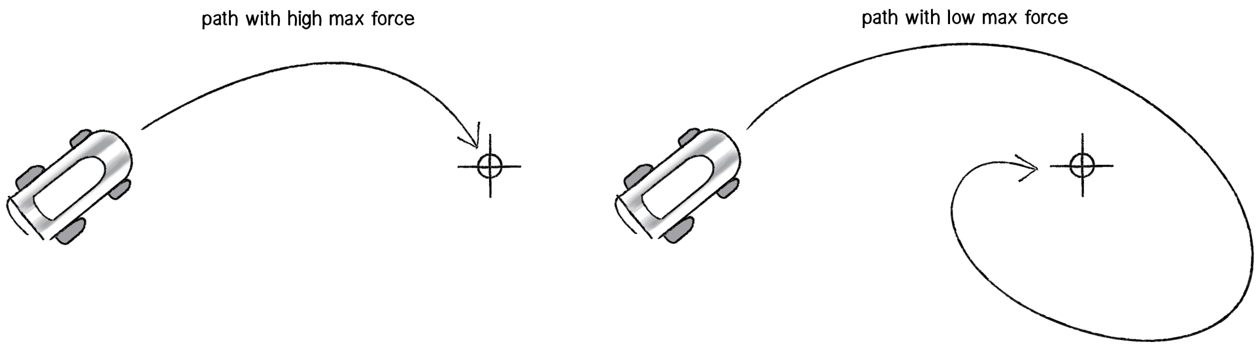
In all of this excitement, however, we've missed one last step. What sort of vehicle is this? Is it a super sleek race car with amazing handling? Or a giant Mack truck that needs a lot of advance notice to turn? A graceful panda, or a lumbering elephant? Our example code, as it stands, has no feature to account for this variability in steering ability. Steering ability can be controlled by limiting the magnitude of the steering force. Let's call that limit the "maximum force" (or "maxforce" for short). And so finally, we have:

```
class Vehicle {
  PVector location;
  PVector velocity;
  PVector acceleration;
  float maxspeed;       $$ Maximum speed
  float maxforce;       $$ Now we also have maximum force
```

followed by:

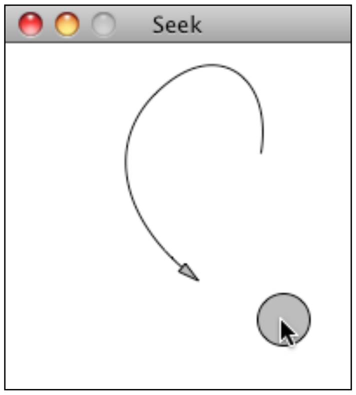
```
  void seek(PVector target) {
    PVector desired = PVector.sub(target,location);
    desired.normalize();
    desired.mult(maxspeed);
    PVector steer = PVector.sub(desired,velocity);

    steer.limit(maxforce);   $$ Limit the magnitude of the steering force

    applyForce(steer);
  }
```

Limiting the steering force brings up an important point. We must always remember that it's not actually our goal to get the Vehicle to the target as fast as possible. If that were the case, we would just say "location equals target" and there the vehicle would be. Our goal, as Reynolds puts it, is to move the vehicle in a "lifelike and improvisational manner." We're trying to make it appear as if the vehicle is steering its way to the target, and so it's up to us to play with the forces and variables of the system to simulate a given behavior. For example, a large maximum steering force would result in a very different path than a small one. One is not inherently better or worse than the other; it depends on your desired effect. (And of course, these values need not be fixed and could change based on other conditions. Perhaps a vehicle has health: the higher the health, the better it can steer.)



[LOW MAX FORCE SHOULD BE A BIT "WIDER"]

Here is the full Vehicle class, incorporating the rest of the elements from the Chapter 2 "Mover" object.



Example 6-1: Seeking a Target

```
class Vehicle {

  PVector location;
  PVector velocity;
  PVector acceleration;
  float r;                    $$ Additional variable for size
  float maxforce;
  float maxspeed;

  Vehicle(float x, float y) {
    acceleration = new PVector(0,0);
    velocity = new PVector(0,0);
    location = new PVector(x,y);
    r = 3.0;
    maxspeed = 4;             $$ Arbitrary values for maxspeed and force; try varying these!
    maxforce = 0.1;
  }

  void update() {             $$ Our standard "Euler integration" motion model
    velocity.add(acceleration);
    velocity.limit(maxspeed);
    location.add(velocity);
    acceleration.mult(0);
  }

  void applyForce(PVector force) {   $$ Newton's second law; we could divide by mass if we wanted
    acceleration.add(force);
  }

  void seek(PVector target) {        $$ Our seek steering force algorithm
    PVector desired = PVector.sub(target,location);
    desired.normalize();
    desired.mult(maxspeed);
    PVector steer = PVector.sub(desired,velocity);
    steer.limit(maxforce);
    applyForce(steer);
  }

  void display() {                             $$ Vehicle is a triangle pointing in
    float theta = velocity.heading2D() + PI/2; the direction of velocity; since it is drawn
    fill(175);                                 pointing up, we rotate it an additional 90
    stroke(0);                                 degrees
    pushMatrix();
    translate(location.x,location.y);
    rotate(theta);
    beginShape();
    vertex(0, -r*2);
    vertex(-r, r*2);
    vertex(r, r*2);
    endShape(CLOSE);
    popMatrix();
  }
```

Exercise: Implement a "fleeing" steering behavior (desired vector is inverse of "seek").

Exercise: Implement seeking a moving target, often referred to as "pursuit." In this case, your desired vector won't point towards the object's current location, rather its "future" location as extrapolated based on its current velocity. We'll see this ability for a Vehicle to "predict the future" in later examples.

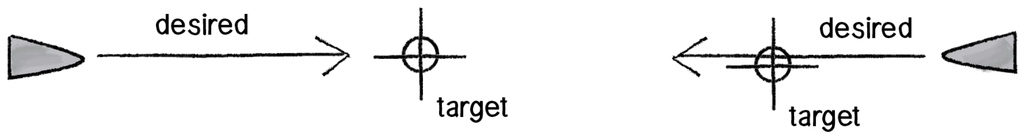
Exercise: Create a sketch where a Vehicle's maximum force and maximum speed do not remain constant, but rather vary according to environmental factors.

6.4 "Arrive"

After working for a bit with the seeking behavior, you probably are asking yourself, "What if I want my vehicle to slow down as it approaches the target?" Before we can even begin to answer this question, we should look at the reasons behind why the seek behavior causes the vehicle to fly past the target so that it has to turn around and go back. Let's consider the brain of a seeking vehicle. What is it thinking?

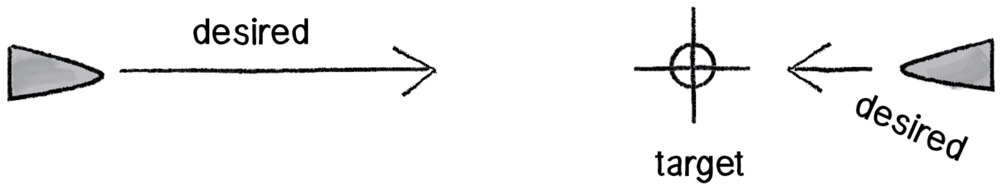
Frame 1: I want to go as fast as possible towards the target!
Frame 2: I want to go as fast as possible towards the target!
Frame 3: I want to go as fast as possible towards the target!
Frame 4: I want to go as fast as possible towards the target!
Frame 5: I want to go as fast as possible towards the target!
etc.

The Vehicle is so gosh darn excited about getting to the target that it doesn't bother to make any intelligent decisions about its speed relative to the target's proximity. Whether it's far away or very close, it always wants to go as fast as possible.



In some cases, this is the desired behavior (if a missile is flying at a target, it should always travel at maximum speed). However, in many other cases (a car pulling into a parking spot, a bee landing on a flower), the Vehicle's thought process needs to consider its speed relative to the distance from its target. For example:

Frame 1: I'm very far away, I want to go as fast as possible towards the target!
Frame 2: I'm very far away, I want to go as fast as possible towards the target!
Frame 3: I'm somewhat far away, I want to go as fast as possible towards the target!
Frame 4: I'm getting close, I want to go more slowly towards the target!
Frame 5: I'm almost there, I want to go very slowly towards the target!
Frame 6: I'm there, I want to stop!

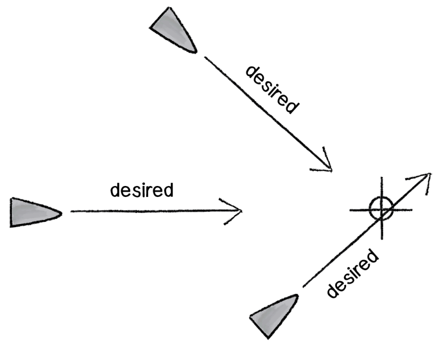


[THESE TWO ILLUSTRATIONS SHOULD MATCH EXACTLY, ONLY DIFF IS THE SHORTER DESIRED VECTOR]

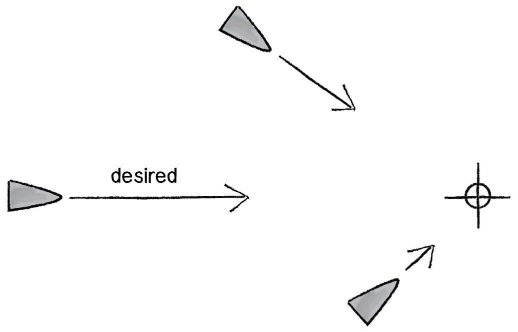
How can we implement this "arriving" behavior in code? Let's return to our ***seek()*** function and find the line of code where we set the magnitude of the desired vector.

```
PVector desired = PVector.sub(target,location);
desired.normalize();
desired.mult(maxspeed);
```

In the above example, the magnitude of the desired vector is always "maximum" speed.



What if we instead said the desired velocity is equal to half the distance?



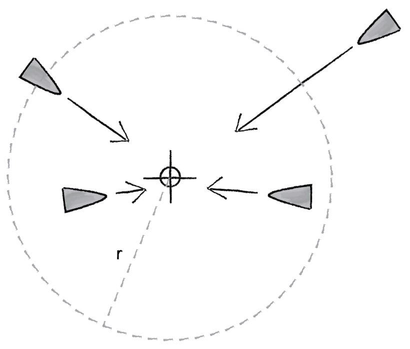
[THESE SHOULD ALSO 100% MATCH EXCEPT VECTORS]

```
PVector desired = PVector.sub(target,location);
desired.div(2);
```

While this nicely demonstrates our goal of a desired speed tied to our distance from the target, it’s not particularly reasonable. After all, 10 pixels away is rather close and a desired speed of 5 is rather large. Something like a desired velocity with a magnitude of 5% of the distance would work much better.

```
PVector desired = PVector.sub(target,location);
desired.mult(0.05);
```

Reynolds describes a more sophisticated approach. Let’s imagine a circle around the target with a given radius. If the Vehicle is within that circle it slows down—at the edge of the circle its desired speed is maximum speed, and at the target itself its desired speed is 0.



In other words, if the distance from the target is less than r, the desired speed is between 0 and maximum speed mapped according to that distance.

Example 6.x: Arrive Steering Behavior

```
  void arrive(PVector target) {
    PVector desired = PVector.sub(target,location);

    float d = desired.mag(); $$ The distance is the magnitude of the vector pointing from
                             location to target
    desired.normalize();
    if (d < 100) {                             $$ If we are closer than 100 pixels
      float m = map(d,0,100,0,maxspeed);       $$ Set the magnitude according to how close
      desired.mult(m);
    } else {
      desired.mult(maxspeed);                  $$ Otherwise, proceed at maximum speed
    }

    PVector steer = PVector.sub(desired,velocity);   $$ The usual steering = desired - velocity
    steer.limit(maxforce);
    applyForce(steer);
  }
```

The arrive behavior is a great demonstration of the magic of "desired minus velocity." Let's examine this model again relative to how we calculated forces in earlier chapters. In the "gravitational attraction" examples, the force always pointed directly from the object to the target. It strong or weak, but it always pointed at the target, the exact direction of the desired velocity.

[NEW DIAGRAM?]

The steering function, however, says: "I have the ability to perceive the environment." The force isn't based just on the desired velocity, but it is based on the desired velocity relative to the current velocity. Knowing your current velocity is only something that things that are alive can do. A box falling off a table doesn't know it's falling. A cheetah chasing its prey, however, knows it is chasing.

The steering force, therefore, is essentially a manifestation of the current velocity's ***error***. I'm supposed to be going this fast in this direction, but I'm actually going this fast in this other direction. My error is the difference between where I want to go and where I am currently going. Taking that error and applying it as a steering force results in more dynamic, life-like simulations. With gravitational attraction, you would never have a force pointing away from the target no matter how close, but with arriving via steering if you are moving too fast towards the target, the error would actually tell you to slow down!

[NEW DIAGRAM]

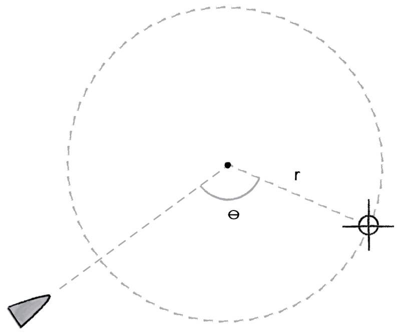
6.5 Your Own Desires

The first two examples we've covered—seek and arrive—boil down to calculating a single vector for each behavior: the ***desired*** velocity. And in fact, every single one of Reynolds's steering behaviors follows this same pattern. In this chapter, we're going to walk through several more of Reynolds's behaviors—flow field, path-following, flocking. First, however, I want to emphasize again that these are *examples*—demonstrations of common steering behaviors that are useful in procedural animation. They are not the be-all and end-all of what ***you*** can do. As long as you can come up with a vector that describes a vehicle's ***desired*** velocity, then you have created your own steering behavior.

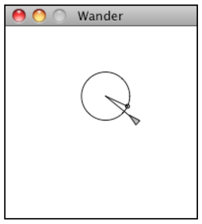
Let's see how Reynolds defines the desired velocity for his wandering behavior.

"Wandering is a type of random steering which has some long term order: the steering direction on one frame is related to the steering direction on the next frame. This produces more interesting motion than, for example, simply generating a random steering direction each frame." —http://www.red3d.com/cwr/steer/Wander.html

For Reynolds, the goal of wandering is not simply random motion, but rather a sense of moving in one direction for a little while, wandering off to the next for a little bit, and so on and so forth. So how does Reynolds calculate a desired vector to achieve such an effect?



The above diagram illustrates how the vehicle predicts its future location as a fixed distance in front of it (in the direction of its velocity), draws a circle with radius *r* at that location, and picks a random point along the circumference of the circle. That random point moves randomly around the circle in each frame of animation. And that random point is the vehicle's target, its desired vector pointing in that direction.



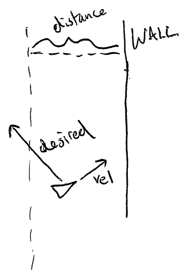
Exercise: Write the code for the wander behavior. Use polar coordinates to calculate the vehicle's target along a circular path.

Sounds a bit absurd, right? Or, at the very least, rather arbitrary. In fact, this is a very clever and thoughtful solution—it uses randomness to drive a vehicle's steering, but constrains that randomness along the path of a circle to keep the vehicle's movement from appearing totally random and jittery.

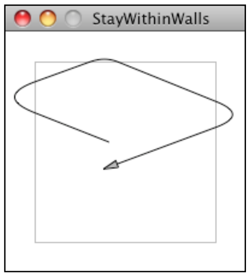
But the seemingly random and arbitrary nature of this solution should drive home the point I'm trying to make—these are made-up behaviors inspired by real-life motion. You can just as easily concoct some elaborate scenario to compute a desired velocity yourself. And you should.

Let's say we want to create a steering behavior called "stay within walls." We'll define the desired velocity as:

If a Vehicle comes within a distance d *of a wall, it desires to move at maximum speed in the opposite direction of the wall.*



If we define the walls of the space as the edges of a Processing window and the distance *d* as 25, the code is rather simple.



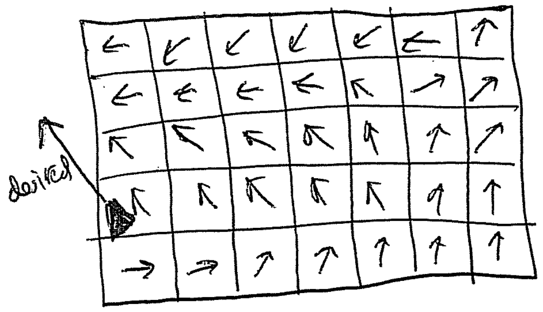
Example 6.x: Stay within bounds steering behavior

```
if (location. x > 25) {
  PVector desired = new PVector(maxspeed,velocity.y);
                $$ Make a desired vector that retains the y direction of
                the vehicle but points the x direction directly away from
                window's left edge
  PVector steer = PVector.sub(desired, velocity);
  steer.limit(maxforce);
  applyForce(steer);
}
```

Exercise: Come up with your own arbitrary scheme for calculating a desired velocity.

6.6 Flow Field

Now back to the task at hand. Let's examine a couple more of Reynolds's steering behaviors. First, flow field following. What is a flow field? Think of your Processing window as a grid. In each cell of the grid lives an arrow pointing in some direction—you know, a vector. As a Vehicle moves around the screen, it asks, "Hey, what arrow is beneath me? That's my desired velocity!"



Reynolds's flow field following example has the vehicle predicting its future location and following the vector at that spot, but for simplicity's sake, we'll have the Vehicle simply look to the vector at its current location.

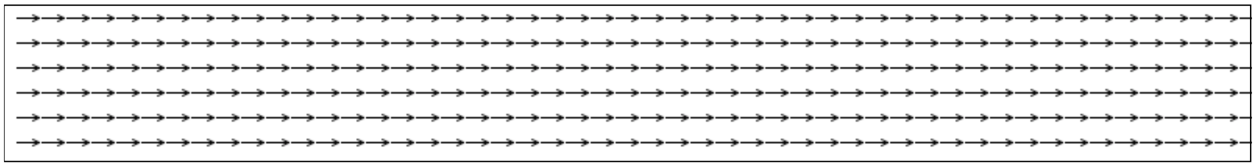
Before we can write the additional code for our Vehicle class, we'll need to build a class that describes the flow field itself, the grid of vectors. A two-dimensional array is a convenient data structure in which to store a grid of information. (If you are not familiar with 2D arrays, I suggest reviewing this online Processing tutorial: http://processing.org/learning/2darray/). The 2D array is convenient because we reference each element with two indices, which we can think of as columns and rows.

```
class FlowField {

  PVector[][] field;    $$ Declaring a 2D array of PVectors
  int cols, rows;       $$ How many columns and how many rows in the grid?
  int resolution;       $$ Resolution of grid relative to window width and height in pixels
```

Notice how we are defining a third variable called "resolution" above. What is this variable? Let's say we have a Processing window that is 200 pixels wide by 200 pixels high. We could make a flow field that has a PVector object for every single pixel, or 40,000 PVectors (200 * 200). This isn't terribly unreasonable, but in our case, it's overkill. We don't need a PVector for every single pixel; we can achieve the same effect by having one, say, every ten pixels (20 * 20 = 400). We use this resolution to define the number of columns and rows based on the size of the window divided by resolution:

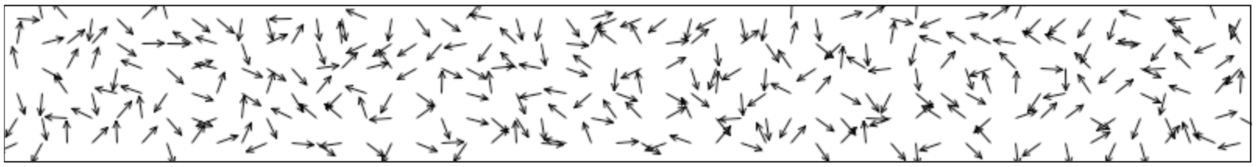
```
FlowField() {
  resolution = 10;
  cols = width/resolution;       $$ Total columns equals width divided by resolution
  rows = height/resolution;      $$ Total rows equals height divided by resolution
  field = new PVector[cols][rows];
}
```

Now that we've set up the flow field's data structures, it's time to compute the vectors in the flow field itself. How do we do that? However we feel like it! Perhaps we want to have every vector in the flow field pointing to the right.



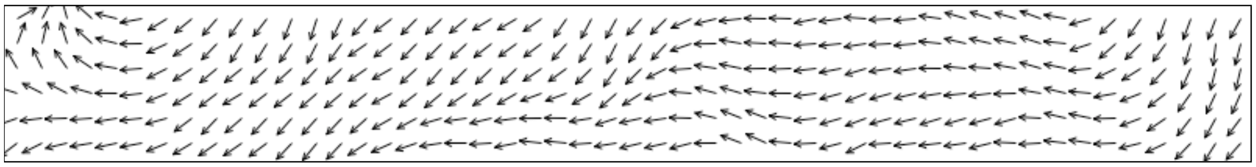
```
for (int i = 0; i < cols; i++) {         $$ Using a nested loop to hit every column
  for (int j = 0; j < rows; j++) {       and every row of the flow field

     field[i][j] = new PVector(1,0);     $$ Arbitrary decision to make each vector point to right
  }
}
```

Or a random vector.



```
 for (int i = 0; i < cols; i++) {
    for (int j = 0; j < rows; j++) {

      float theta = random(TWO_PI);
      field[i][j] = new PVector(cos(theta),sin(theta));    $$ A random PVector
    }
  }
```

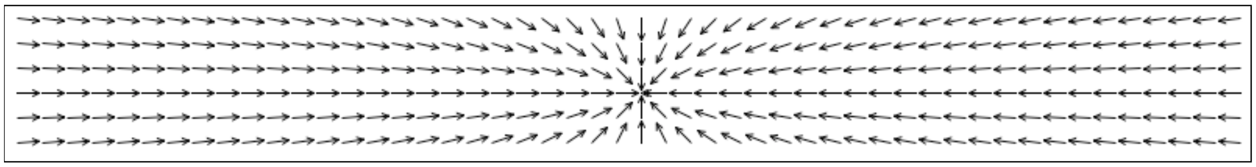
What if we use 2D Perlin noise (mapped to an angle)?



```
  float xoff = 0;
  for (int i = 0; i < cols; i++) {
    float yoff = 0;
    for (int j = 0; j < rows; j++) {
      float theta = map(noise(xoff,yoff),0,1,0,TWO_PI);    $$ Noise
      field[i][j] = new PVector(cos(theta),sin(theta));
      yoff += 0.1;
    }
    xoff += 0.1;
  }
```

Now we're getting somewhere. Flow fields can be used for simulating various effects, such as an irregular gust of wind or the meandering path of a river. Calculating the direction of your vectors using Perlin noise is one way to achieve such an effect. Of course, there's no "correct" way to calculate the vectors of a flow field; it's really up to you to decide what you're looking to simulate.

Exercise: Write the code to calculate a PVector at every location in the flow field that points towards the center of a window.



```
PVector v = new PVector(____________,____________);
v.______________();
field[i][j] = v;
```

Now that we have a two-dimensional array storing all of the flow field vectors, we need a way for a Vehicle to look up its desired vector from the flow field. Let's say we have a vehicle that lives at a PVector: its location. We first need to divide by the resolution of the grid. For example, if the resolution is 10 and the vehicle is at (100,50), we need to look up column 10 and row 5.

```
int column = int(location.x/resolution);
int row = int(location.y/resolution);
```

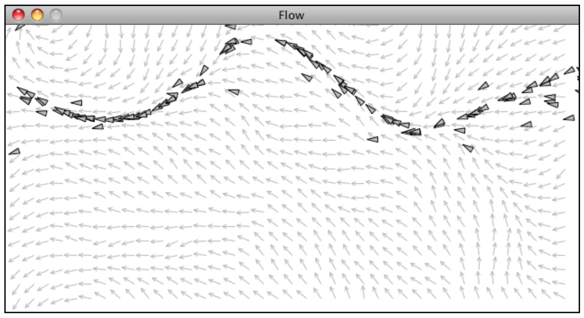
Because a vehicle could theoretically wander off the Processing window, it's also useful for us to employ the ***constrain()*** function to make sure we don't look outside of the flow field array. Here is a function we'll call ***lookup()*** that goes in the FlowField class—it receives a PVector (presumably the location of our vehicle) and returns the corresponding flow field PVector for that location.

```
  PVector lookup(PVector lookup) {
    int column = int(constrain(lookup.x/resolution,0,cols-1));  $$ Using constrain()
    int row = int(constrain(lookup.y/resolution,0,rows-1));
    return field[column][row].get();
  }                    $$ Note the use of get() to ensure we return a copy of the PVector
```

Before we move on to the Vehicle class, let's take a look at the FlowField class all together.

```
class FlowField {

  PVector[][] field;                 $$ A flow field is a two-dimensional array of PVectors
  int cols, rows;
  int resolution;

  FlowField(int r) {
    resolution = r;
    cols = width/resolution;         $$ Determine the number of columns and rows
    rows = height/resolution;
    field = new PVector[cols][rows];
    init();
  }

  void init() {                      $$ In this example, we use Perlin noise to seed the vectors
    float xoff = 0;
    for (int i = 0; i < cols; i++) {
      float yoff = 0;
      for (int j = 0; j < rows; j++) {
        float theta = map(noise(xoff,yoff),0,1,0,TWO_PI);
        field[i][j] = new PVector(cos(theta),sin(theta));
        yoff += 0.1;               $$ Polar to cartesian coordinate transformation to get x and y
                                      components of the vector
      }
      xoff += 0.1;
    }
  }

  PVector lookup(PVector lookup) {        $$ A function to return a PVector based on a location
    int column = int(constrain(lookup.x/resolution,0,cols-1));
    int row = int(constrain(lookup.y/resolution,0,rows-1));
    return field[column][row].get();
  }
}
```

So let's assume we have a FlowField object "flow". Using the ***lookup()*** function above, our vehicle can then retrieve a desired vector from the FlowField object and use Reynolds's rules (steering = desired minus velocity) to calculate a steering force.



Example 6-x: Flow Field Following

```
class Vehicle {

  void follow(FlowField flow) {
    PVector desired = flow.lookup(location);   $$ What is the vector at that spot
                                                  in the flow field?

    desired.mult(maxspeed);
    PVector steer = PVector.sub(desired, velocity);  $$ Steering is desired minus velocity
    steer.limit(maxforce);
    applyForce(steer);
  }
```

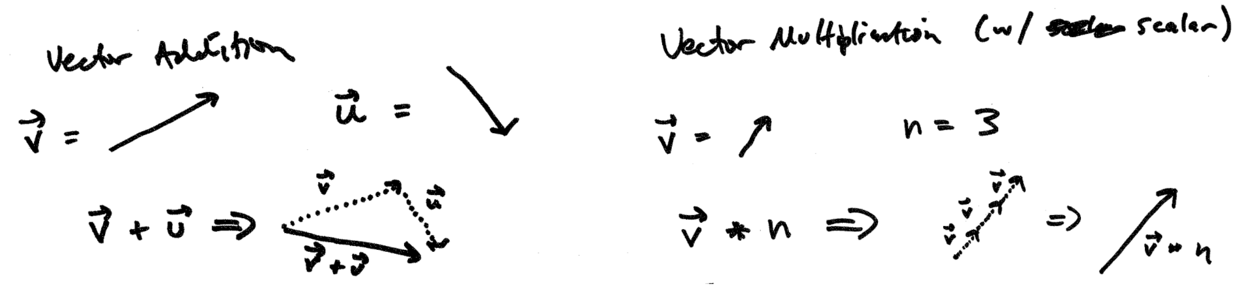
Exercise: Adapt the flow field example so that the PVectors change over time (hint: try using the 3rd dimension of Perlin noise!)

Exercise: Can you seed a flow field from a PImage? For example, try having the PVectors point from dark to light colors (or vice versa).

6.7 The Dot Product

In a moment, we're going to work through the algorithm (along with accompanying mathematics) and code for another of Craig Reynolds's steering behaviors: path following (see: http://www.red3d.com/cwr/steer/PathFollow.html). Before we can do this, however, we have to spend some time learning about another piece of vector math that we skipped in Chapter 1—the dot product. We haven't needed it yet, but it's likely going to prove quite useful for you (beyond just this path-following example), so we'll go over it in detail now.

Remember all the basic vector math we covered in Chapter 1? Add, subtract, multiply and divide?



Notice how in the above diagram vector multiplication involved multiplying a vector by a scalar value. This makes sense; when we want a vector to be twice as large (but facing the same direction), we multiply it by 2. When we want it to be half the size, we multiply it by 0.5.

However, there are two other *multiplication-like* operations with vectors that are useful in certain scenarios—the dot product and the cross product. For now we're going to focus on the dot product, which is defined as follows. Assume vectors **A** and **B:**

A = (ax,ay)
B = (bx,by)

THE DOT PRODUCT: ***A* • *B = ax*bx + ay* by***

For example, if we have the following two vectors:

A = (-3,5)
B = (10,1)

A • **B** = -3*10 + 5*1 = -30 + 5 = -25

Notice that the result of the dot product is a scalar value (a single number) and not a vector.

In Processing, this would translate to:

```
PVector a = new PVector(-3,5);
PVector b = new PVector(10,1);

float n = a.dot(b);    $$ The PVector class includes a function to calculate the dot product
```

And if we were to look in the guts of the PVector source, we'd find a pretty simple implementation of this function:

```
public float dot(PVector v) {
  return x*v.x + y*v.y + z*v.z;
}
```

This is simple enough, but why do we need the dot product, and when is it going to be useful for us in code?

One of the more common uses of the dot product is to find the angle between two vectors. Another way the dot product can be expressed is:

THE DOT PRODUCT: $\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}|\,|\mathbf{b}| \cos\theta$

In other words, A dot B is equal to the magnitude of A times magnitude of B times cosine of theta (with theta defined as **the angle between the two vectors A and B**.)

The two formulas for dot product can be derived from one another with trigonometry (see: http://mathworld.wolfram.com/DotProduct.html), but for our purposes we can be happy with operating on the assumption that:

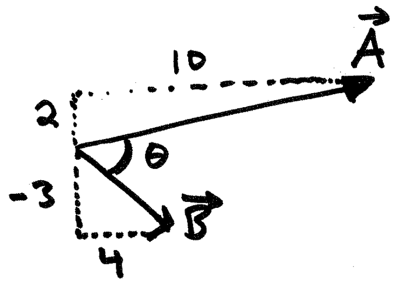
***A* • *B* = |*A*| * |*B*| * cos(theta)**
***A* • *B* = ax*bx + ay* by**

both hold true and therefore:

ax*bx + ay* by = |A| * |B| * cos(theta)

Now, let's start with the following problem. We have the vectors A and B:

A = (10,2)
B = (4,-3)



We now have a situation where we know everything except for theta. We know the components of the vector (ax,ay,bx,by) and we can calculate the magnitude of each vector as we did in Chapter 1 with the Pythagorean theorem. We can therefore solve for cos(theta):

cos(theta) = *A* • *B* / |*A*| * |*B*|

Once we've solved for cosine of theta, we can take the ***inverse*** cosine (often expressed as arccosine) to solve for theta.

theta = arccos (*A* • *B* / |*A*| * |*B*|)

Let's now do the math with actual numbers:

|*A*| = 10.2
|*B*| = 5

Therefore:

theta = arccos (10*4 + 2*-3 / 10.2 * 5)
theta = arccos (34 / 51)
theta = ~ 48 degrees

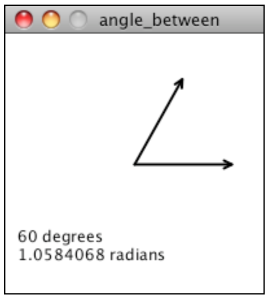
The Processing version of this would be:

```
PVector a = new PVector(10,2);
PVector b = new PVector(4,-3);
float theta = acos(a.dot(b) / (a.mag() * b.mag()));
```

And, again, if we were to dig into the guts of the Processing source code, we would see a function that implements this exact algorithm.

```
  static public float angleBetween(PVector v1, PVector v2) {
    float dot = v1.dot(v2);
    float theta = (float) Math.acos(dot / (v1.mag() * v2.mag()));
    return theta;
```

```
}
```



```
60 degrees
1.0584068 radians
```

Exercise: Create a sketch that displays the angle between two PVector objects.

A couple things to note here:

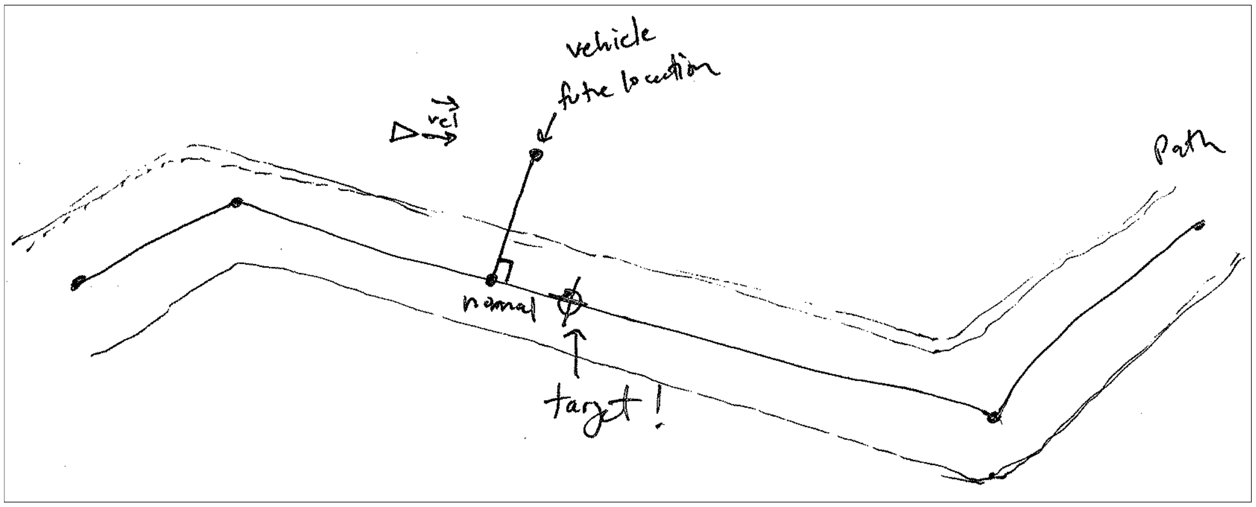
1) If two vectors (**A** and **B**) are orthogonal (i.e. perpendicular), the dot product ($A \bullet B$) is equal to zero.

2) If two vectors are unit vectors then the dot product is simply equal to cosine of the angle between, i.e. $A \bullet B = cos(theta)$ if A and B are of length 1.

6.8 Path Following

Now that we've got a basic understanding of the dot product under our belt, we can return to a discussion of Craig Reynolds's path-following algorithm. Let's quickly clarify something. We are talking about path ***following***, not path ***finding***. Pathfinding refers to a research topic (commonly studied in artificial intelligence) related to solving for the shortest distance between two points, often in a maze. With path following, the path already exists and we're asking a vehicle to follow that path.

Before we work out the individual pieces, let's take a look at the overall algorithm for path following, as defined by Reynolds.



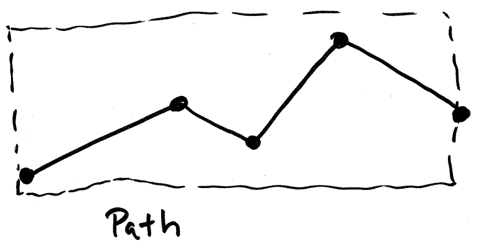
Step 1. Predict the future. Compute the vehicle's theoretical location N frames in the future. This is yet another example of how our vehicles have an intelligent ability to perceive their environment. Instead of knowing only its current location, a vehicle can extrapolate its future location according to its velocity.

Step 2. How far away from the path are we? Calculate the distance between the vehicle's future location and the path. If it is within the path, do nothing. Otherwise, continue:

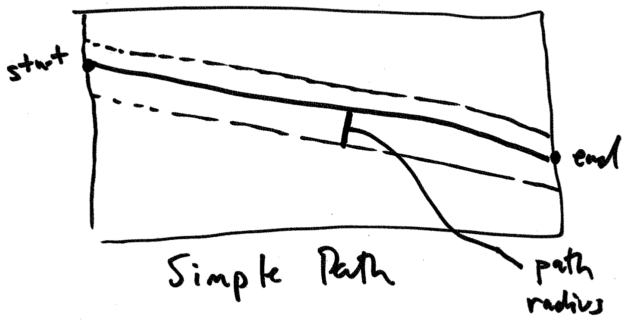
Step 3. Find a target point on the path. Take the point on the path that is "normal" (more on this in a moment) to the vehicle's future location. Then look ahead on the path and set a target location.

Step 4. Steer. Set the vehicle's steering force to seek that target.

Before we deal with the vehicle, let's define what we mean by a path. There are many ways we could implement a path, but for us, the simplest will be to define a path as a series of connected points:



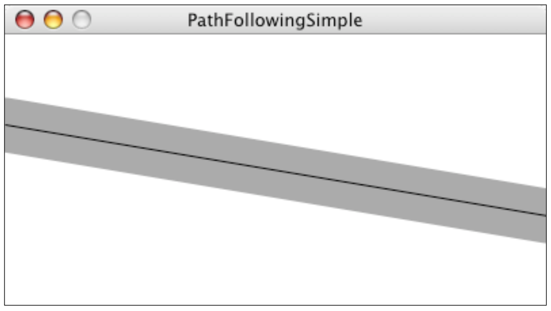
To start, let's think of our path in an even simpler way, as a line between two points.



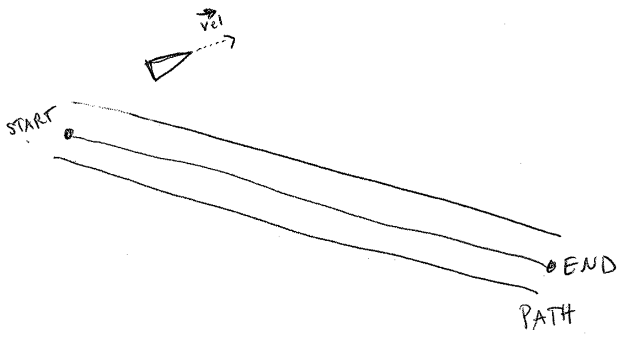
We're also going to consider a path to have a radius. If we think of the path as a road, the radius determines the road's width. With a smaller radius, our vehicles will have to follow the path more closely; a wider radius will allow them to stray a bit more.

Putting this into a class, we have:

```
class Path {

  PVector start;       $$ A Path is only two points, start and end
  PVector end;
  float radius;         $$ A path has a radius, i.e how wide is it

  Path() {
    radius = 20;        $$ Picking some arbitrary values to initialize path
    start = new PVector(0,height/3);
    end = new PVector(width,2*height/3);
  }

  void display() {      $$ Display the path
    strokeWeight(radius*2);
    stroke(0,100);
    line(start.x,start.y,end.x,end.y);
    strokeWeight(1);
    stroke(0);
    line(start.x,start.y,end.x,end.y);
  }
}
```



Now, let's assume we have a Vehicle (as depicted below) outside of the path's radius, moving with a velocity.

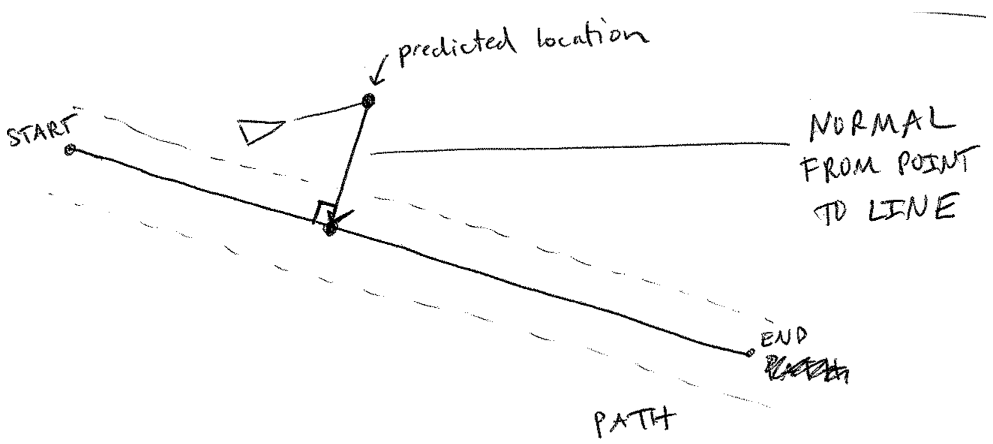


The first thing we want to do is predict, assuming a constant velocity, where that vehicle will be in the future:

```
PVector predict = vel.get();  $$ Start by making a copy of the velocity

predict.normalize();          $$ Normalize it and look 25 pixels ahead by scaling vector up
predict.mult(25);

PVector predictLoc = PVector.add(loc, predict);
                              $$ Add vector to location to find the predicted location
```

Once we have that location, it's now our job to find out its distance from the path that predicted location. If it's very far away, well, then, we've strayed from the path and need to steer back towards it. If it's close, then we're doing OK and are following the path nicely.

So, how do we find the distance between a point and a line? This concept is key. The distance between a point and a line is defined as the length of the "normal" between that point and line. The normal is a vector that extends from that point and is perpendicular to the line.



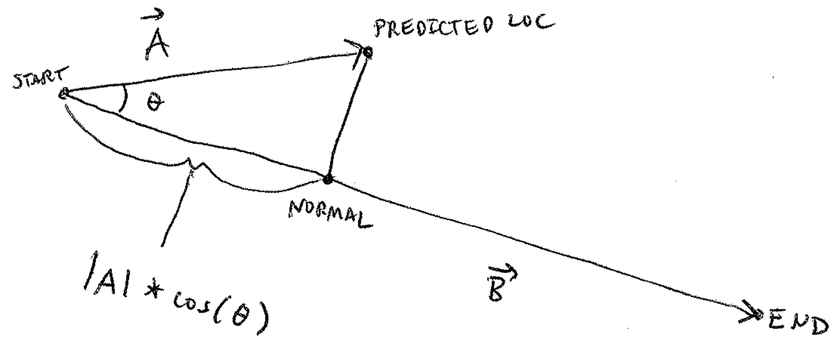
Let’s figure out what we do know. We know we have a vector (call it **A**) that extends from the path’s starting point to the vehicle’s predicted location:

```
PVector a = PVector.sub(predictLoc,path.start);
```

We also know that we can define a vector (call it **B**) that points from the start of the path to the end.

```
PVector b = PVector.sub(path.end,path.start);
```

Now, with basic trigonometry, we know that the distance from the path’s start to the normal point is **|A| * cos(theta)**.



If we knew theta, we could easily define that normal point as follows:

```
float d = a.mag()*cos(theta); $$ The distance from START to NORMAL
b.normalize();
b.mult(d);                    $$ Scale PVector b to that distance
PVector normalPoint = PVector.add(path.start,b);
                              $$ The normal point can be found by adding the scaled version of b
                              to the path’s starting point
```

And if the dot product has taught us anything, it's that given two vectors, we can get theta, the angle between.

```
float theta = PVector.angleBetween(a,b);    $$ What is theta? The angle between A and B
b.normalize();
b.mult(a.mag()*cos(theta));
PVector normalPoint = PVector.add(path.start,b);
```

While the above code will work, there's one more simplification we can make. If you notice that the desired magnitude for vector **B** is:

```
a.mag()*cos(theta)
```

or

|A|*cos(theta)

And if you recall:

$$A \bullet B = |A|*|B|*cos(theta)$$

Now, what if vector **B** is a unit vector, i.e. length 1? Then:

$$A \bullet B = |A|*1*cos(theta)$$

or

$$A \bullet B = |A|*cos(theta)$$

And what are we doing in our code? Normalizing b!

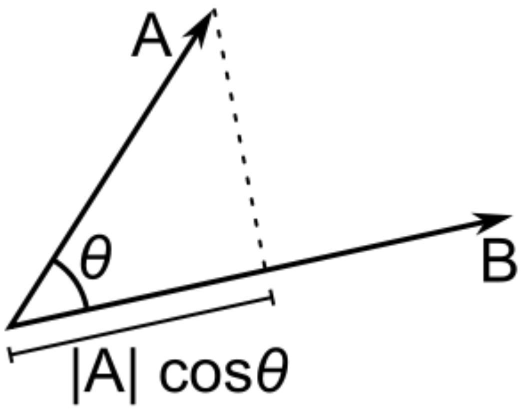
```
b.normalize();
```

Because of this fact, we can simplify our code as:

~~`float theta = PVector.angleBetween(a,b);`~~

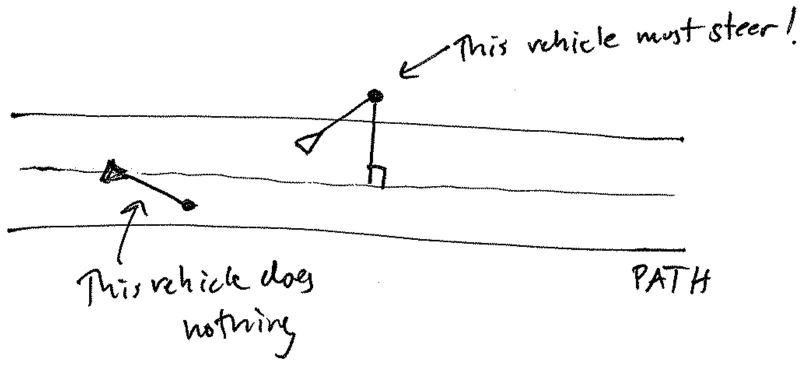
```
b.normalize();
b.mult(a.dot(b));    $$ We can use the dot product to scale B's length

PVector normalPoint = PVector.add(path.start,b);
```

This process is commonly known as "scalar projection."



|A| cos(θ) is the scalar projection of A onto B. And if we normalize B before computing the dot product, the scalar projection of A onto B is equal to A • B.

Once we have the normal point along the path, we have to decide whether the vehicle should steer towards the path and how. Reynolds's algorithm states that the vehicle should only steer towards the path if it strays beyond the path (i.e., if the distance between the normal point and the predicted future location is greater than the path radius).



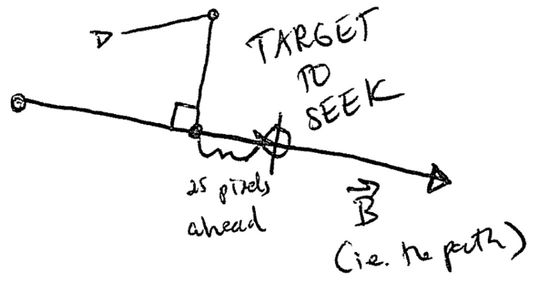
```
float distance = PVector.dist(predictLoc, normalPoint);

if (distance > path.radius) {          $$ If the vehicle is outside the path, seek the target
  seek(target);                        $$ We don't have to work out the desired velocity and
}                                         steering force; all that is taken care of by seek(),
                                          which we already wrote in Example 6.x
```

But what is the target?

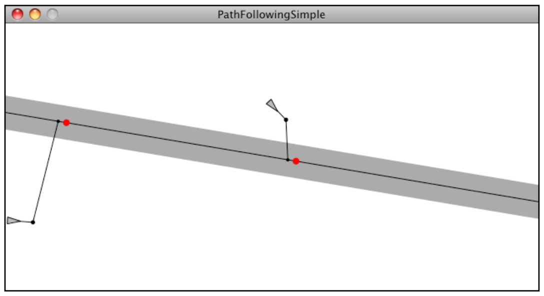
Reynolds's algorithm involves picking a point ahead of the normal on the path (see step #3 above). But for simplicity, we could just say that the target is the normal itself. This will work fairly well:

```
float distance = PVector.dist(predictLoc, normalPoint);
if (distance > path.radius) {
  seek(normalPoint);          $$ Seek the normal point on the path
}
```

Since we know the vector that defines the path (we're calling it "b"), we can implement Reynolds's "point ahead on the path" without too much trouble.



```
float distance = PVector.dist(predictLoc, normalPoint);
if (distance > path.radius) {
  b.normalize();       $$ Normalize and scale b (pick 25 pixels arbitrarily)
  b.mult(25);
  PVector target = PVector.add(normalPoint,b);
                       $$ By adding b to normalPoint, we now move 25 pixels ahead on the path
  seek(target);
}
```

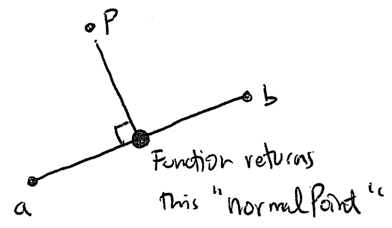


Putting it all together, we have the following steering function in our Vehicle class.

Example 6.x: Simple Path Following

```
  void follow(Path p) {

      $$ Step 1. Predict vehicle's future location
    PVector predict = vel.get();
    predict.normalize();
    predict.mult(25);
    PVector predictLoc = PVector.add(loc, predict);

    PVector a = p.start;                 $$ Step 2. Find normal point along path
    PVector b = p.end;
    PVector normalPoint = getNormalPoint(predictLoc, a, b);

    PVector dir = PVector.sub(b, a);     $$ Step 3. Move a little further along path
    dir.normalize();                        and set a target
    dir.mult(10);
    PVector target = PVector.add(normalPoint, dir);

    float distance = PVector.dist(normalPoint, predictLoc);
    if (distance > p.radius) {           $$ Step 4. If we are off the path, seek that target
      seek(target);                         in order to stay on the path.
    }
  }
```

Now, you may notice above that instead of using all that dot product/scalar projection code to find the normal point, we instead call a function: ***getNormalPoint()***. In cases like this, it's useful to break out the code that performs a specific task (finding a normal point) into a function that it can be used generically in any case where it is required. The function takes three PVectors: the first defines a point in Cartesian space and the second and third arguments define a line segment.

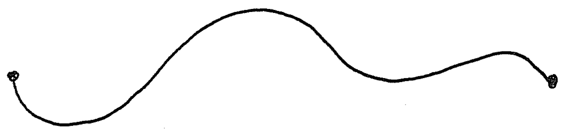


```
PVector getNormalPoint(PVector p, PVector a, PVector b) {
  PVector ap = PVector.sub(p, a);   $$ PVector that points from a to p
  PVector ab = PVector.sub(b, a);   $$ PVector that points from a to b

  ab.normalize();                   $$ Using the dot product for scalar projection
  ab.mult(ap.dot(ab));
  PVector normalPoint = PVector.add(a, ab);
                                    $$ Finding the normal point along the line segment
  return normalPoint;
}
```

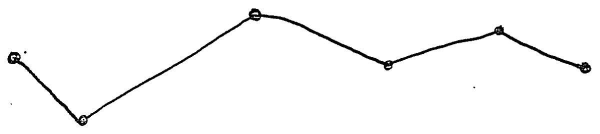
What do we have so far? We have a Path class that defines a path as a line between two points. We have a Vehicle class that defines a vehicle that can follow the path (using a steering behavior to seek a target along the path). What is missing?

Take a deep breath. We're almost there.

We've built a great example so far, yes, but it's pretty darn limiting. After all, what if we want our path to be something that looks more like:



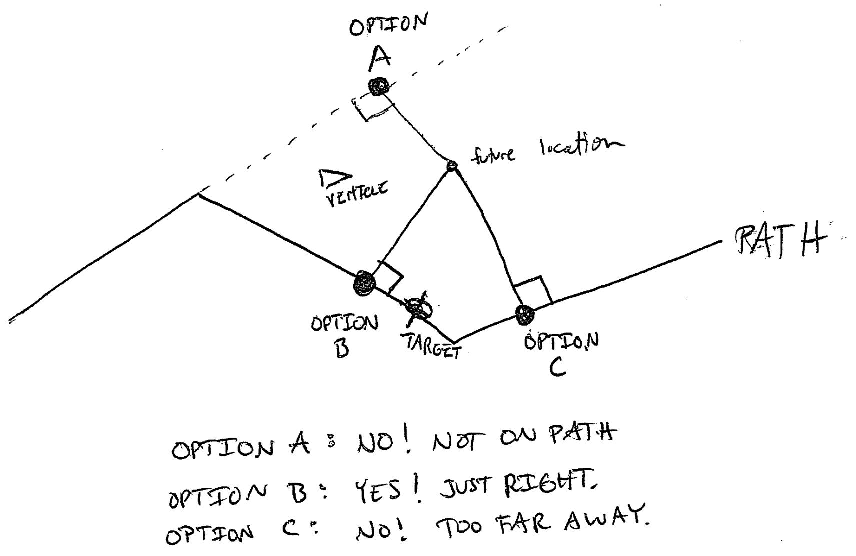
While it's true that we could make this example work for a curved path, we're much less likely to end up needing a cool compress on our forehead if we stick with line segments. In the end, we can always employ the same technique we discovered with Box2D—we can draw whatever fancy curved path we want and approximate it behind the scenes with simple geometric forms.



So, what's the problem? If we made path following work with one line segment, how do we make it work with a series of connected line segments? Let's take a look again at our vehicle driving along the screen. Say we arrive at Step 3.

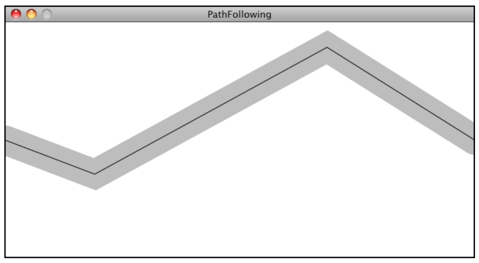
Step 3. Find a target point on the path.

To find the target, we need to find the normal to the line segment. But now that we have a series of line segments, we have a series of normal points (see above)! Which one do we choose? The solution we'll employ is to pick the normal point that is (a) closest and (b) on the path itself.


OPTION A
future location
VEHICLE
PATH
OPTION B
TARGET
OPTION C
OPTION A: NO! NOT ON PATH
OPTION B: YES! JUST RIGHT.
OPTION C: NO! TOO FAR AWAY.

If we have a point and an infinitely long line, we'll always have a normal. But, as in the path-following example, if we have a point and a line segment, we won't necessarily find a normal that is on the line segment itself. So if this happens for any of the segments, we can disqualify those normals. Once we are left with normals that are on the path itself (only two in the above diagram), we simply pick the one that is closest to our vehicle's location.

In order to write the code for this, we'll have to expand our Path class to have an ArrayList of points (rather than just two, a start and an end.)



```
class Path {

  ArrayList<PVector> points;
              $$ A Path is now an ArrayList of points
                (PVector objects)
  float radius;

  Path() {
    radius = 20;
    points = new ArrayList<PVector>();
  }

  void addPoint(float x, float y) {          $$ This function allows us to add points
```

```
    PVector point = new PVector(x,y);           to the path
    points.add(point);
  }

  void display() {                           $$ Display the path as a series of points
    stroke(0);
    noFill();
    beginShape();
    for (PVector v : points) {
      vertex(v.x,v.y);
    }
    endShape();
  }
}
```

Now that we have the Path defined, it's the vehicle's turn to deal with multiple line segments. All we did before was find the normal for one line segment. We can now find the normals for all the line segments in a loop.

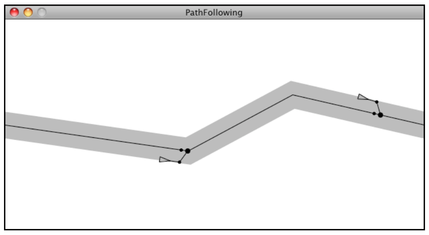
```
for (int i = 0; i < p.points.size()-1; i++) {
  PVector a = p.points.get(i);
  PVector b = p.points.get(i+1);
  PVector normalPoint = getNormalPoint(predictLoc, a, b);
                              $$ Finding the normals on each line segment
```

Then we should make sure the normalPoint is actually between points a and b. Since we know our path goes from left to right in this example, we can test if the x location of normalPoint is outside the x locations of a and b.

```
     if (normalPoint.x < a.x || normalPoint.x > b.x) {
        normalPoint = b.get();  $$ Use the end point of the segment as our normal point if we
     }                          can't find one.
```

As a little trick, we'll say that if it's not within the line segment, let's just pretend the end point of that line segment is the normal. This will ensure that our vehicle always stays on the path, even if it strays out of the bounds of our line segments.

Finally, we'll need to make sure we find the normal point that is closest to our vehicle. To accomplish this, we can start with a "world record" of some very high number and progressively save normal point that beats the record as we go through the loop in a variable called "target". When all is said and done, we'll have the closest normal point in that variable.



Example 6.x: Path Following

```
PVector target = null;
float worldRecord = 1000000;  $$ Start with a very high record
                              that can easily be beaten

for (int i = 0; i < p.points.size()-1; i++) {
  PVector a = p.points.get(i);
  PVector b = p.points.get(i+1);
  PVector normalPoint = getNormalPoint(predictLoc, a, b);
  if (normalPoint.x < a.x || normalPoint.x > b.x) {
    normalPoint = b.get();
  }
  float distance = PVector.dist(predictLoc, normalPoint);

  if (distance < worldRecord) {      $$ If we beat the record then this should be our target!
    worldRecord = distance;
    target = normalPoint.get();
  }
}
```

Exercise: Update the path-following example so that the path can go in any direction. (Hint, you'll need to use the min() and max() function when determining if the normal point is inside the line segment.)

```
if (normalPoint.x < ____(____,____) || normalPoint.x > ____(____,____)) {
  normalPoint = b.get();
}
```

Exercise: Create a path that changes over time. Can the points that define the path itself have their own steering behaviors?

6.8 Complex Systems

Remember our purpose? To breathe life into the things that move around our Processing windows? By learning to write the code for an autonomous agent and building a series of examples of individual behaviors, hopefully our souls feel a little more full. But this is no place to stop and rest on our laurels. We're just getting started. After all, there is a deeper purpose at work here. Yes, a vehicle is a sentient being making decisions about how to seek and flow and follow. But what is a life led alone, without the love and support of others? Our purpose here is not only to build individual behaviors for our vehicles, but to put our vehicles into systems of many vehicles and allow those vehicles to interact with each other.

Let's think about a tiny, crawling ant—one single ant. An ant is an autonomous agent; it can perceive its environment (using antennae to gather information about the direction and strength of chemical signals) and make decisions about how to move based on those signals. But can a single ant acting alone build a nest, gather food, defend its queen? An ant is a simple unit and can only perceive its immediate environment. A colony of ants, however, is a sophisticated complex system, a "superorganism" that collectively works together to accomplish difficult and complicated goals.

We want to take what we've learned during the process of building autonomous agents in Processing into simulations that involve many agents operating in parallel—agents that have an ability not only to perceive their physical environment but also the actions of their fellow agents, and then act accordingly. We want to create complex systems in Processing.

What is a complex system? A complex system is typically defined as a system that is "more than the sum of its parts." While the individual elements of the system may be incredibly simple and easily understood, the behavior of the system as a whole can be highly complex, intelligent, and difficult to predict. Here are three key principles of complex systems.

- ***Simple units with short-range relationships.*** This is what we've been building all along: vehicles that have a limited perception of their environment.

- ***Simple units operate in parallel.*** This is what we need to simulate in code. For every cycle through Processing's ***draw()*** loop, each unit will decide how to move (to create the appearance of them all working in parallel.)

- ***System as a whole exhibits emergent phenomena.*** Out of the interactions between these simple units ***emerges*** complex behavior, patterns, and intelligence. Here we're talking about the result we are hoping for in our sketches. Yes, we know this happens in nature (ant colonies, termites, migration patterns, earthquakes, snowflakes, etc.), but can we achieve the same result in our Processing sketches?

Following are two additional features of complex systems that will help frame the discussion as well as well as provide guidelines for features we will want to include in our software simulations.

- ***Non-linearity***. This aspect of complex systems is often casually referred to as "The Butterfly Effect," coined by mathematician and meteorologist Edward Norton Lorenz, a pioneer in the study of chaos theory. In 1961, Lorenz was running a computer weather simulation for the second time and, perhaps to save a little time, typed in a starting value of 0.506 instead of 0.506127.   The end result was completely different from the first result of the simulation.   In other words, the theory is that a single butterfly flapping its wings on the other side of the world could cause a massive weather shift and ruin our weekend at the beach. We call it "non-linear" because there isn't a linear relationship between a change in initial conditions and a change in outcome.   A small change in initial conditions can have a massive effect on the outcome.   In the next chapter, we'll see how even in a system of many zeros and ones, if we change just one bit, the result will be completely different.

- ***Competition and cooperation***. One of the things that often makes a complex system tick is the presence of both competition and cooperation between the elements. In our upcoming flocking system, we will have three rules—alignment, cohesion, and separation. Alignment and cohesion will ask the elements to "cooperate", i.e. work together to stay together and move together. Separation, however, will ask the elements to "compete" for space. As we get to the flocking system, try taking out the cooperation or the competition and you'll see how you are left without complexity.

Complexity will serve as a theme for the remaining content in this book. In this chapter, we'll begin by adding one more feature to our Vehicle class: an ability to look at neighboring vehicles.

6.9 Group Behaviors Part I: Let's not run into each other.

A group is certainly not a new concept. We've done this before—in Chapter 4, where we developed a framework for managing collections of particles in a ParticleSystem class. There, we stored a list of particles in an ArrayList. We'll do the same thing here: store a bunch of Vehicle objects in an ArrayList.

```
ArrayList<Vehicle> vehicles;       $$ Declare an ArrayList of Vehicle objects

void setup() {
  vehicles = new ArrayList<Vehicle>;     $$ Initialize and fill the ArrayList with a bunch
  for (int i = 0; i < 100; i++) {           of Vehicles
    vehicles.add(new Vehicle(random(width),random(height)));
  }
}
```

Now when it comes time to deal with all the vehicles in ***draw()***, we simply loop through all of them and call the necessary functions.

```
void draw(){
  for (Vehicle v : vehicles) {
    v.update();
    v.display();
  }
}
```

OK, so maybe we want to add a behavior, a force to be applied to all the vehicles. This could be seeking the mouse.

```
    v.seek(mouseX,mouseY);
```

But that's an individual behavior. We've already spent thirty-odd pages worrying about individual behaviors. We're here because we want to apply a group behavior. Let's begin with "separation", a behavior that states, "Avoid colliding with your neighbors!"

```
    v.separate();
```

Is that right? It sounds good, but it's not. What's missing? In the case of seek, we said "Seek ***mouseX*** and ***mouseY***." In the case of separate, we're saying "separate from ***everyone else***." Who is everyone else? It's the list of all the other vehicles.

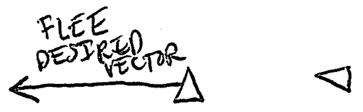
```
    v.separate(vehicles);
```

This is the big leap beyond what we did before with Particle Systems. Instead of having each element (particle or vehicle) operate on its own, we're now saying, "You, the vehicle, when it comes time for you to operate, you need to operate with an awareness of everyone else. So I'm going to go ahead and pass you the ArrayList of everyone else."

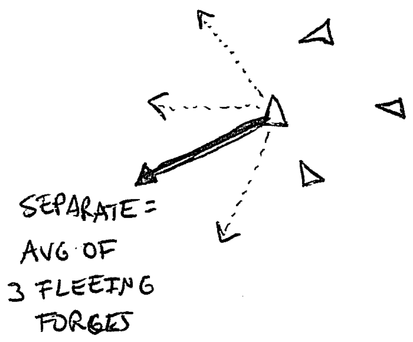
This is how we've mapped out ***setup()*** and ***draw()*** to deal with a group behavior.

```
ArrayList<Vehicle> vehicles;

void setup() {
  size(320,240);
  vehicles = new ArrayList<Vehicle>();
  for (int i = 0; i < 100; i++) {
    vehicles.add(new Vehicle(random(width),random(height)));
  }
}

void draw() {
  background(255);

  for (Vehicle v : vehicles) {
    v.separate(vehicles);          $$ This is really the only new thing we're doing in this
    v.update();                       section.  We're asking a Vehicle object to examine all the
    v.display();                      other vehicles in the process of calculating a separation
  }                                   force.
}
```

Of course, this is just the beginning. The real work happens inside the ***separate()*** function itself. Let's figure out how we want to define separation. Reynolds states: "Steer to avoid crowding." In other words, if a given vehicle is too close to you, steer away from that vehicle. Sound familiar? Remember the seek behavior where a vehicle steers towards a target? Reverse that force and we have the flee behavior.



But what if more than one vehicle is too close? In this case, we'll define separation as the average of all the vectors pointing away from any close vehicles.



Let's begin to write the code. As we just worked out, we're writing a function called ***separate()*** that receives an ArrayList of Vehicle objects as an argument.

```
void separate (ArrayList<Vehicle> vehicles) {

}
```

Inside this function, we're going to loop through all of the vehicles and see if any are too close.

```
float desiredseparation = 20;   $$ This variable specifies how close is too close.

for (Vehicle other : vehicles) {

  float d = PVector.dist(location, other.location);   $$ What is the distance between me
                                                         and another Vehicle?
  if ((d > 0) && (d < desiredseparation)) {

     $$ Here is the code that will be executed if the Vehicle
        is within 20 pixels.
  }
}
```

Notice how in the above code, we are not only checking if the distance is less than a desired separation (i.e. too close!), but also if the distance is greater than zero. This is a little trick that makes sure we don't ask a vehicle to separate from itself. Remember, all the vehicles are in the ArrayList, so if you aren't careful you'll be comparing each vehicle to itself!

Once we know that two vehicles are too close, we need to make a vector that points away from the offending vehicle.

```
  if ((d > 0) && (d < desiredseparation)) {
    PVector diff = PVector.sub(location, other.location);
    diff.normalize();                  $$ A PVector pointing away from the other's location.
  }
```

This is not enough. We have that vector now, but we need to make sure we calculate the average of all vectors pointing away from close vehicles. How do we compute average? We add up all the vectors and divide by the total.

```
PVector sum = new PVector();    $$ Start with an empty PVector
int count = 0;                  $$ We have to keep track of how many Vehicles are too close
for (Vehicle other : vehicles) {
  float d = PVector.dist(location, other.location);
  if ((d > 0) && (d < desiredseparation)) {
    PVector diff = PVector.sub(location, other.location);
    diff.normalize();
    sum.add(diff);              $$ All the vectors together and increment the count
    count++;
  }
}

if (count > 0) {                $$ We have to make sure we found at least one close
  sum.div(count);                  vehicle.  We don't want to bother doing anything
}                                  if nothing is too close (not to mention we can't
                                   divide by zero!)
```

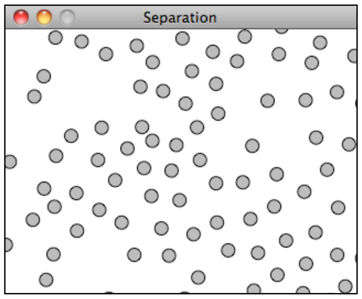
Once we have the average vector (stored in the PVector object "sum"), that PVector can be scaled to maximum speed and become our desired velocity—we ***desire*** to move in that direction at maximum speed! And once we have the desired velocity, it's the same old Reynolds story: steering equals desired minus velocity.

```
if (count > 0) {
    sum.div(count);

    sum.normalize();          $$ Scale average to maxspeed (this becomes desired)
    sum.mult(maxspeed);

    PVector steer = PVector.sub(sum,vel);      $$ Reynolds Steering formula
    steer.limit(maxforce);

    applyForce(steer);        $$ Apply the force to the Vehicle's acceleration
  }
```

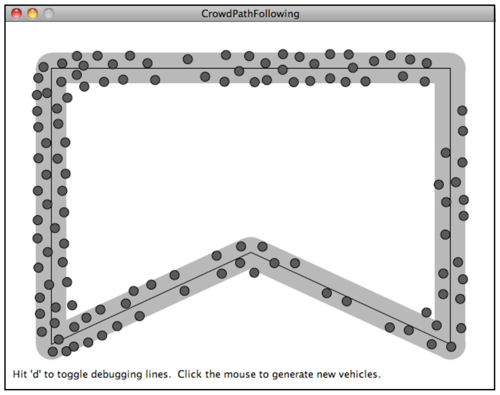
Let's see the function in its entirety. There are two additional improvements, noted in the code bubbles.

Example 6.x: Group Behavior: Separation

```
  void separate (ArrayList<Vehicle> vehicles) {
    float desiredseparation = r*2;         $$ Note how the desired separation is based
    PVector sum = new PVector();              on the Vehicle's size.
    int count = 0;
    for (Vehicle other : vehicles) {
      float d = PVector.dist(location, other.location);
      if ((d > 0) && (d < desiredseparation)) {
        PVector diff = PVector.sub(location, other.location);
        diff.normalize();
        diff.div(d);                       $$ What is the magnitude of the PVector pointing away
        sum.add(diff);                        from the other vehicle?  The closer it is, the more
        count++;                              we should flee.  The farther, the less. So we divide
                                              by the distance to weight it appropriately.
      }
    }
    if (count > 0) {
      sum.div(count);
      sum.normalize();
      sum.mult(maxspeed);
      PVector steer = PVector.sub(sum, vel);
      steer.limit(maxforce);
      applyForce(steer);
    }

  }
```



Exercise: Rewrite separate() to work in the opposite fashion ("cohesion"). If a vehicle is beyond a certain distance, steer towards that vehicle. This will keep the group together. (Note that in a moment, we're going to look at what happens when we have both cohesion and separation in the same simulation.)


CrowdPathFollowing
Hit 'd' to toggle debugging lines. Click the mouse to generate new vehicles.

Exercise: Add the separation force to path following to create a simulation of Reynolds's "Crowd Path Following."

6.10 Combinations

The previous two exercises (6.x, 6.x) hint at what is perhaps the most important aspect of this chapter. After all, what is a Processing sketch with one steering force compared to one with many? How could we even begin to simulate emergence in our sketches with only one rule? The most exciting and intriguing behaviors will come from mixing and matching multiple steering forces, and we'll need a mechanism for doing so.

You may be thinking, Duh, this is nothing new. We do this all the time. You would be right. In fact, we did this as early as Chapter 2.

```
  PVector wind = new PVector(0.001,0);
  PVector gravity = new PVector(0,0.1);
  mover.applyForce(wind);
  mover.applyForce(gravity);
```

Here we have a Mover object that responds to two forces. This all works nicely because of the way we designed the Mover object to accumulate the force vectors into its acceleration vector. In this chapter, however, our forces stem from internal desires of the Mover objects (now called "Vehicles" themselves). And those desires can be weighted. Let's consider a sketch where all vehicles have two desires:

- ***Seek the mouse location.***
- ***Separate from any vehicles that are too close.***

We might begin by adding a function to the Vehicle class that manages all of the behaviors. Let's call it ***applyBehaviors()***.

```
void applyBehaviors(ArrayList<Vehicle> vehicles) {
  separate(vehicles);
  seek(new PVector(mouseX,mouseY));
}
```

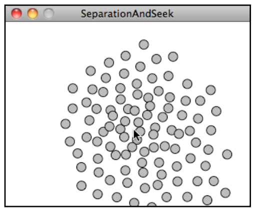
Here we see how a single function takes care of calling the other functions that apply the forces—***separate()*** and ***seek()***. We could start mucking around with those functions and see if we can adjust the strength of the forces they are calculating. But it would be easier for us to ask those functions to return the forces so that we can adjust their strength before applying them to the Vehicle's acceleration.

```
  void applyBehaviors(ArrayList<Vehicle> vehicles) {
    PVector separate = separate(vehicles);
    PVector seek = seek(new PVector(mouseX,mouseY));
    applyForce(separate);                    $$ We have to apply the force here since
    applyForce(seek);                           seek() and separate() no longer do so.
  }
```

Let's look at how the seek function changed.

```
  PVector seek(PVector target) {          $$ Seek now returns a PVector
    PVector desired = PVector.sub(target,loc);
    desired.normalize();
    desired.mult(maxspeed);
    PVector steer = PVector.sub(desired,vel);
    steer.limit(maxforce);
    ~~applyForce(steer);~~            $$ Instead of applying the force, we return the PVector
    return steer;
  }
```

This is a subtle change, but incredibly important for us: it allows us alter the strength of these forces in one place.



Example 6.x: Combining Steering Behaviors: Seek and Separate

```
void applyBehaviors(ArrayList<Vehicle> vehicles) {
  PVector separate = separate(vehicles);
  PVector seek = seek(new PVector(mouseX,mouseY));

  separate.mult(1.5);   $$ These values can be whatever you want them to be!
  seek.mult(0.5);          They can be variables that are customized for
                           each vehicle as well as change over time.
  applyForce(separate);
  applyForce(seek);
}
```

Exercise: Redo example 6.x so that the behavior weights are not constants. What happens if they change over time (according to a sine wave or Perlin noise)? Or if some vehicles are more concerned with seeking and others more concerned with separating? Can you introduce other steering behaviors as well?

6.11 Flocking

Flocking is an group animal behavior that is characteristic of many living creatures, such as birds, fish, and insects. In 1986, Craig Reynolds created a computer simulation of flocking behavior and documented the algorithm in his paper, "Flocks, Herds, and Schools: A Distributed

Behavioral Model."   Recreating this simulation in Processing will bring together all the concepts in this chapter.

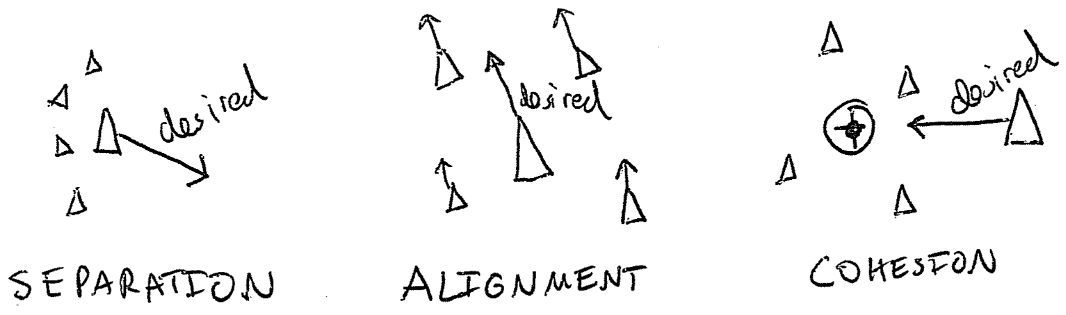
1. *We will use the steering force formula (steer = desired - velocity) to implement the rules of flocking.*
2. *These steering forces will be group behaviors and require each vehicle to look at all the other vehicles.*
3. *We will combine and weight multiple forces.*
4. *The result will be a complex system—intelligent group behavior will emerge from the simple rules of flocking without the presence of a centralized system or leader.*

The good news is, we've already done items 1 through 3 in this chapter, so this section will be about just putting it all together and seeing the result.

Before we begin, I should mention that we're going to change the name of our Vehicle class (yet again). Reynolds uses the term "boid" (a made-up word that refers to a bird-like object) to describe the elements of a flocking system and we will do the same.

Let's take an overview of the three rules of flocking.

1. Separation *(also known as "avoidance"): Steer to avoid colliding with your neighbors.*
2. Alignment *(also known as "copy"): Steer in the same direction as your neighbors.*
3. Cohesion *(also known as "center"): Steer towards the center of your neighbors (stay with the group).*



Just as we did with our separate and seek example, we'll want our Boid objects to have a single function that manages all the above behaviors. We'll call this function ***flock()***.

```
void flock(ArrayList<Boid> boids) {
  PVector sep = separate(boids);            $$ The three flocking rules
  PVector ali = align(boids);
  PVector coh = cohesion(boids);
```

```
    sep.mult(1.5);                          $$ Arbitrary weights for these forces
    ali.mult(1.0);                             (Try different ones!)
    coh.mult(1.0);

    applyForce(sep);                        $$ Applying all the forces
    applyForce(ali);
    applyForce(coh);
  }
```

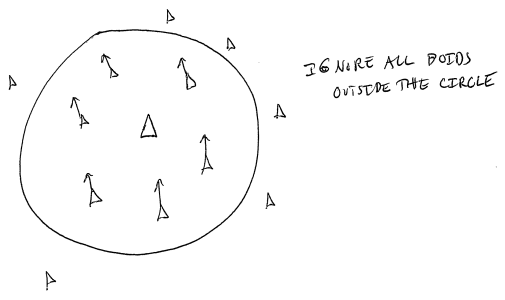
Now, it's just a matter of implementing the three rules. We did separation before; it's identical to our previous example. Let's take a look at alignment—steer in the same direction as your neighbors. As with all of our steering behaviors, we've got to boil down this concept into a desire: the boid's desired velocity is the average velocity of its neighbors.

So our algorithm is to calculate the average velocity of all the other boids and set that to desired.

```
  PVector align (ArrayList<Boid> boids) {
    PVector sum = new PVector(0,0);       $$ Add up all the velocities and divide by the total
    for (Boid other : boids) {               to calculate the average velocity.
      sum.add(other.velocity);
    }
    sum.div(boids.size());

    sum.normalize();                      $$ We desire to go in that direction at maximum speed
    sum.mult(maxspeed);

    PVector steer = PVector.sub(sum,velocity); $$ Reynolds steering force formula
    steer.limit(maxforce);
    return steer;
  }
```

The above is pretty good, but it's missing one rather crucial detail. One of the key principles behind complex systems like flocking is that the elements (in this case, boids) have short-range relationships. Thinking about ants again, it's pretty easy to imagine an ant being able to sense its immediate environment, but less so an ant having an awareness of what another ant is doing hundreds of feet away. The fact that the ants can perform such complex collective behavior from only these neighboring relationships is what makes them so exciting in the first place.

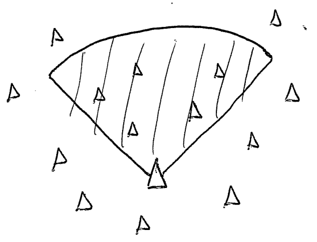
In our alignment function, we're taking the average velocity of all the boids, whereas we should really only be looking at the boids within a certain distance. That distance threshold is up to you, of course. You could design boids that can see only twenty pixels away or boids that can see a hundred pixels away.



Much like we did with separation (we only calculated a force for others within a certain distance), we'll want to do the same with alignment (and cohesion).

```
PVector align (ArrayList<Boid> boids) {
  float neighbordist = 50;                $$ This is an arbitrary value and could
  PVector sum = new PVector(0,0);            vary from boid to boid
  int count = 0;
  for (Boid other : boids) {
    float d = PVector.dist(location,other.location);
    if ((d > 0) && (d < neighbordist)) {
      sum.add(other.velocity);
      count++;                            $$ For average, we need to keep track of
    }                                        how many boids are within the distance
  }
  if (count > 0) {
    sum.div(count);
    sum.normalize();
    sum.mult(maxspeed);
    PVector steer = PVector.sub(sum,velocity);
    steer.limit(maxforce);
    return steer;
  } else {
    return new PVector(0,0);              $$ If we don't find any close boids the steering
  }                                          force is zero.
}
```

Exercise: Can you write the above code so that boids can only see other boids that are actually within their "peripheral" vision (as if they had eyes)?



Finally, we are ready for cohesion. Here our code is virtually identical to alignment—only instead of calculating the average velocity of the boid's neighbors, we want to calculate the average location of the boid's neighbors (and use that as a target to seek).

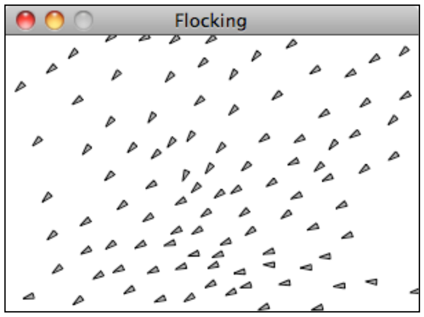
```
PVector cohesion (ArrayList<Boid> boids) {
  float neighbordist = 50;
  PVector sum = new PVector(0,0);
  int count = 0;
  for (Boid other : boids) {
    float d = PVector.dist(location,other.location);
    if ((d > 0) && (d < neighbordist)) {
      sum.add(other.location);                    $$ Adding up all the others' locations
      count++;
    }
  }
  if (count > 0) {
    sum.div(count);
    return seek(sum);              $$ Here we make use of the seek() function we wrote in
  } else {                          Example 6.x.  The target we seek is the average location
    return new PVector(0,0);        of our neighbors.
  }
}
```

It's worth taking the time to also write a class called Flock, which will be virtually identical to the ParticleSystem class we wrote in Chapter 4 with only one tiny change: When we call ***run()*** on each Boid object (as we did to each Particle object), we'll pass in a reference to the entire ArrayList of boids.

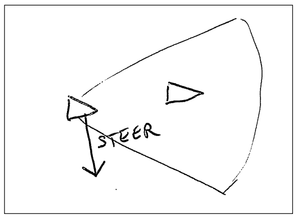
```
class Flock {
  ArrayList<Boid> boids;

  Flock() {
    boids = new ArrayList<Boid>();
  }

  void run() {
    for (Boid b : boids) {
      b.run(boids);            $$ Each Boid object must know about all the other Boids
    }
  }

  void addBoid(Boid b) {
    boids.add(b);
  }
}
```

And our main program will look like:

```
Flock flock;                $$ A Flock object manages the entire
                               group.
void setup() {
  size(300,200);
  flock = new Flock();
  for (int i = 0; i < 100; i++) {
    Boid b = new Boid(width/2,height/2);
    flock.addBoid(b);   $$ The Flock starts out with 100 Boids
  }
}

void draw() {
  background(255);
  flock.run();
}
```



Exercise: Combine Flocking with some other steering behaviors.



Exercise: In "The Computational Beauty of Nature" (Gary Flake, MIT Press, 2000), Gary Flake describes a fourth rule for flocking: "View: move laterally away from any boid that blocks the view." Implement this rule.

Exercise: Create a flocking simulation where all of the parameters (separation weight, cohesion weight, alignment weight, maximum force, maximum speed) change over time. They could be controlled by Perlin noise or by user interaction (for example, you could use a library such as controlp5 to tie the values to slider positions.)

Exercise: Visualize the flock in an entirely different way.

6.12 Algorithmic Efficiency (or Why does my $#@(*%#! run so slow?)

I would like to hide the dark truth behind we've just done, because I would like you to be happy and live a fulfilling and meaningful life. But I also would like to be able to sleep at night without worrying about you so much. So it is with a heavy heart that I must bring up this topic. Group behaviors are wonderful. But they can be slow, and the more elements in the group, the slower they can be. Usually, when we talk about Processing sketches running slowly, it's because drawing to the screen can be slow—the more you draw, the slower your sketch runs. This is actually a case, however, where the slowness derives from the algorithm itself. Let's discuss.

Computer scientists classify algorithms with something called "Big O notation", which describes the efficiency of an algorithm: how many computational cycles does it require to complete? Let's consider a simple analog search problem. You have a basket full of one hundred chocolate treats, only one of which is pure dark chocolate. That's the one you want to eat. To find it, you pick the chocolates out of the basket one by one. Sure, you might be lucky and find it on the first try, but in the worst-case scenario you have to check all one hundred before you find the dark chocolate. To find one thing in one hundred, you have to check one hundred things (or to find one thing in N things, you have to check N times.) Your Big O Notation is N. This incidentally is the Big O Notation that describes our simple particle system. If we have N particles, we have to run and display those particles N times.

Now, let's think about a group behavior (such as flocking). For every Boid object, we have to check every other Boid object (for its velocity and location). Let's say we have one hundred boids. For Boid #1, we need to check one hundred boids; for Boid #1, we need to check one hundred boids, and so on and so forth. For one hundred boids, we need to perform one hundred times one hundred checks, or ten thousand. No problem: computers are fast and can do things ten thousand times pretty easily. Let's try one thousand.

1,000 x 1,000 = 1,000,000 cycles.

OK, this is rather slow, but still somewhat manageable. Let's try 10,000 elements:

10,000 x 10,000 elements = 100,000,000 cycles.

Now, we're really getting slow. Really, really, really slow.

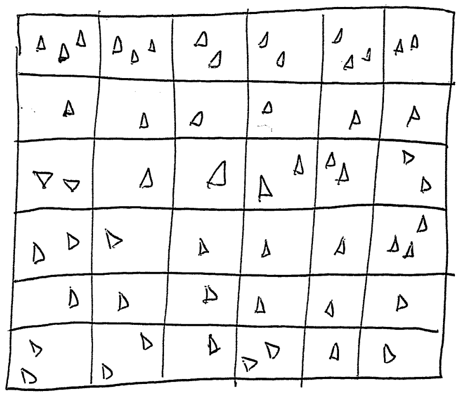
Notice something odd? As the number of elements increases by a factor of 10, the number of required cycles increases by a factor of 100. Or as the number of elements increases by a factor of N, the cycles increase by a factor of N times N. This is known as Big O Notation *N-Squared*.

I know what you are thinking. You are thinking: “No problem; with flocking, we only need to consider the boids that are close to other boids. So even if we have 1,000 boids, we can just look at, say, the five closest boids and then we only have 5,000 cycles.” You pause for a moment, and then start thinking: “So for each boid I just need to check all the boids and find the five closest ones and I’m good!” See the catch-22? Even if we only want to look at the close ones, the only way to know what the close ones are would be to check all of them.

Or is there another way?

Let’s take a number that we might actually want to use, but would still run too slow: 2,000 (4,000,000 cycles required.)

What if we could divide the screen into a grid? We would take all 2,000 boids and assign each boid to a cell within that grid. We would then be able to look at each boid and compare it to its neighbors within that cell at any given moment. Imagine a 10 x 10 grid. In a system of 2,000 elements, on average, approximately 20 elements would be found in each cell (20 x 10 x 10 = 2,000). Each cell would then require 20 x 20 = 400 cycles. With 100 cells, we’d have 100 x 400 = 40,000 cycles, a massive savings over 4,000,000.



This technique is known as “bin-lattice spatial subdivision” and is outlined in more detail in (surprise, surprise) Reynolds’s 2000 paper: “Interaction with Groups of Autonomous Characters” (see: http://www.red3d.com/cwr/papers/2000/pip.pdf). How do we implement such an algorithm in Processing? One way is to keep multiple ArrayLists. One ArrayList would keep track of all the boids, just like in our Flocking example.

```
ArrayList<Boid> boids;
```

In addition to that ArrayList, we store an additional reference to each Boid object in a two-dimensional ArrayList. For each cell in the grid, there is an ArrayList that tracks the objects in that cell.

```
ArrayList<Boid>[][] grid;
```

In the main ***draw()*** loop, each Boid object then registers itself in the appropriate cell according to its location.

```
int column = int(boid.x) / resolution;
int row    = int(boid.y) /resolution;
grid[column][row].add(boid);
```

Then when it comes time to have the boids check for neighbors, they can look at only those in their particular cell (in truth, we also need to check neighboring cells to deal with border cases).

Example 6.x: Bin-Lattice Spatial Subdivision

```
int column = int(boid.x) / resolution;
int row    = int(boid.y) /resolution;
~~boid.flock(boids);~~
boid.flock(grid[column][row]);     $$ Instead of looking at all the boids, just this cell
```

We're only covering the basics here; for the full code, check the web site.

Now, there are certainly flaws with this system. What if all the boids congregate in the corner and live in the same cell? Then don't we have to check all 2,000 against all 2,000?

The good news is that this need for optimization is a common one and there are a wide variety of similar techniques out there. For us, it's likely that a basic approach will be good enough (in most cases, you won't need one at all) and we can stop here.

6.13 A few last notes: optimization tricks.

This is something of a momentous occasion. The end of Chapter 6 marks the end of our story of motion (in the context of this book, that is). We started with the concept of a vector, moved onto forces, designed systems of many elements, examined physics libraries, built entities with hopes and dreams and fears, and simulated emergence. The story doesn't end here, but it does take a bit of a turn. The next two chapters won't focus on moving bodies, but rather on systems of rules. Before we get there, I have a few quick items I'd like to mention that are important when working with the examples in Chapters 1-6. They also relate to optimizing your code, which fits in with the previous section.

Magnitude squared *(or sometimes distance squared)*

What is magnitude squared and when should you use it? Let's revisit how the magnitude of a vector is calculated.

```
float mag() {
  return sqrt(x*x + y*y);
}
```

Magnitude requires the square root operation. And it should. After all, if you want the magnitude of a vector then you've got to look up the Pythagorean theorem and compute it (we did this in Chapter 1). However, if you could somehow skip using the square root, your code would run faster. Let's consider a situation where you just want to know the relative magnitude of a vector. For example, is the magnitude greater than ten? (Assume a PVector v).

```
if (v.mag() > 10) {
  // Do Something!
}
```

Well, this is equivalent to saying:

```
if (v.magSq() > 100) {
  // Do Something!
}
```

And how is magSquared calculated?

```
float magSq() {
  return x*x + y*y;
}
```

Same as magnitude, but without the square root. In the case of a single PVector object, this will never make a significant difference on a Processing sketch. However, if you are computing the magnitude of thousands of PVector objects each time through ***draw()***, using ***magSq()*** instead of ***mag()*** could help your code run a wee bit faster. (Note magSq() is only available in Processing 2.0a1 or later.)

Sine and cosine lookup tables.

There's a pattern here. What kinds of functions are slow to compute? Square root. Sine. Cosine. Tangent. Again, if you just need a sine or cosine value here or there in your code, you are never going to run into a problem. But what if you had something like this?

```
void draw() {
  for (int i = 0; i < 10000; i++) {
     println(sin(PI));
  }
}
```

Sure, this is a totally ridiculous code snippet that you would never write. But it illustrates a certain point. If you are calculating the sine of pi ten thousand times, why not just calculate it once, save that value, and refer to it whenever necessary? This is the principle behind sine and cosine lookup tables. Instead of calling the sine and cosine functions in your code whenever you need them, you can build an array that stores the results of sine and cosine at angles between 0 to TWO_PI and just look up the values when you need them. For example, here are two arrays that store the sine and cosine values for every angle, 0 to 359 degrees.

```
float sinvalues[] = new float[360];
float cosvalues[] = new float[360];
for (int i = 0; i < 360; i++) {
  sinvalues[i] = sin(radians(i));
  cosvalues[i] = cos(radians(i));
}
```

Now, what if you need the value of of sine of pi?

```
int angle = int(degrees(PI));
float answer = sinvalues[angle];
```

A more sophisticated example of this technique is available on the Processing wiki:
http://wiki.processing.org/w/Sin/Cos_look-up_table

Making gajillions of unnecessary PVector objects

I have to admit, I am perhaps the biggest culprit of this last note. In fact, in the interest of writing clear and understandable examples, I often choose to make extra PVector objects when I absolutely do not need to. For the most part, this is not a problem at all. But sometimes, it can be. Let's take a look at an example.

```
void draw() {
  for (Vehicle v : vehicles) {
   PVector mouse = new PVector(mouseX,mouseY);
   v.seek(mouse);
  }
}
```

Let's say our ArrayList of vehicles has one thousand vehicles in it. We just made one thousand new PVector objects every single time through ***draw()***. Now, on any ol' laptop or desktop computer you've purchased in recent times, your sketch will likely not register a complaint, run slowly, or have any problems. After all, you've got tons of RAM, and Java will be able to handle making a thousand or so temporary objects and dispose of them without much of a problem.

If your numbers grow larger (and they easily could) or perhaps more likely, if you are working with Processing on Android, you will almost certainly run into a problem. In cases like this you want to look for ways to reduce the number of PVector objects you make. An obvious fix for the above code is:

```
void draw() {
  PVector mouse = new PVector(mouseX,mouseY);
  for (Vehicle v : vehicles) {
   v.seek(mouse);
  }
}
```

Now you've made just one PVector instead of one thousand. Even better, you could turn the PVector into a global variable and just assign the x and y value:

```
PVector mouse = new PVector();

void draw() {
  mouse.x = mouseX;
  mouse.y = mouseY;
  for (Vehicle v : vehicles) {
   v.seek(mouse);
  }
}
```

Now you never make a new PVector; you use just one over the length of your sketch!

In my examples, you'll find lots of opportunities to reduce the number of temporary objects. Let's look at one more. Here is a snippet from our ***seek()*** function.

```
    PVector desired = PVector.sub(target,location);
    desired.normalize();
    desired.mult(maxspeed);

    PVector steer = PVector.sub(desired,velocity);
    steer.limit(maxforce);         $$ Create a new PVector to store the steering force
    return steer;
```

See how we've made two PVector objects? First, we figure out the desired vector, then we calculate the steering force. Notice how we could rewrite this to create only one PVector.

```
    PVector desired = PVector.sub(target, location);
    desired.normalize();
    desired.mult(maxspeed);

    desired.sub(velocity);         $$ Calculate the steering force in the desired PVector
    desired.limit(maxforce);
    return desired;
```

We don't actually need a second PVector called steer. We could just use the desired PVector object and turn it into the steering force by subtracting velocity. I didn't do this in my example because it is more confusing to read. But in some cases, it may be greatly more efficient.

Exercise: Eliminate as many temporary PVector objects from the flocking example as possible. Also use ***magSquared()*** *where possible.*

Exercise: Use steering behaviors with Box2D or Toxiclibs.

The Eco-System Project:

Step 6 Exercise:

Use the concept of steering forces to drive the behavior of your creatures in your eco-system. Some possibilities:

- *Create "schools" or "flocks" of creatures.*
- *Use a seeking behavior for creatures to search for food (for chasing moving prey, consider "pursuit").*
- *Use a flow field for the eco-system environment. For example, how does your system behave if the creatures live in a flowing river.*
- *Build a creature with countless steering behaviors (as many as you can reasonably add). Think about ways to vary the weights of these behaviors so that you can dial those behaviors up and down, mixing and matching on the fly. How are creatures' initial weights set? What rules drive how the weights change over time?*
- *Complex systems can be nested. Can you design a single creature out of a flock of boids? And can you then make a flock of those creatures?*
- *Complex systems can have memory (and be adaptive). Can the history of your eco-system affect the behavior in its current state? (This could be the driving force behind how the creatures adjust their steering force weights.)*