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 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***.

```
1   class Vehicle {
2
3   PVector location;
4   PVector velocity;
5   PVector acceleration;
6           $$ What else do we need to add?
```

NOT FOUND: [breakout box]

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 uses the word "Vehicle" to describe his autonomous agents, so we will follow suit.[end breakout box]

In his 1999 paper "Steering Behaviors for Autonomous Characters", 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.

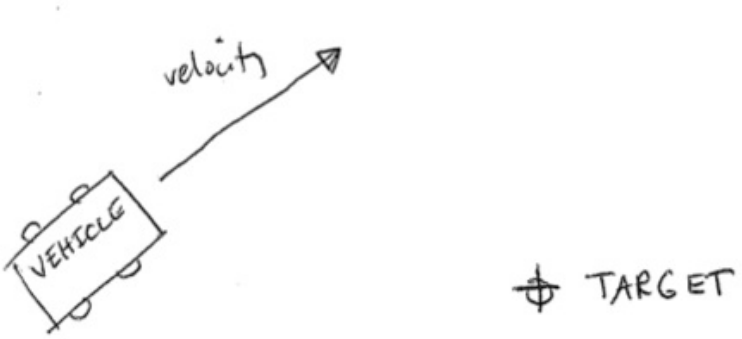
- **Action Selection**. A Vehicle has a goal (or goals) and can select an action (or a combination of actions) based on that goal. This is essentially where we left off with autonomous agents. The vehicle takes a look at its environment and calculates an action based on a desire: "I see a zombie marching 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.
- **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.
- **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 becauseyou 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, as they say in American Idol, 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.

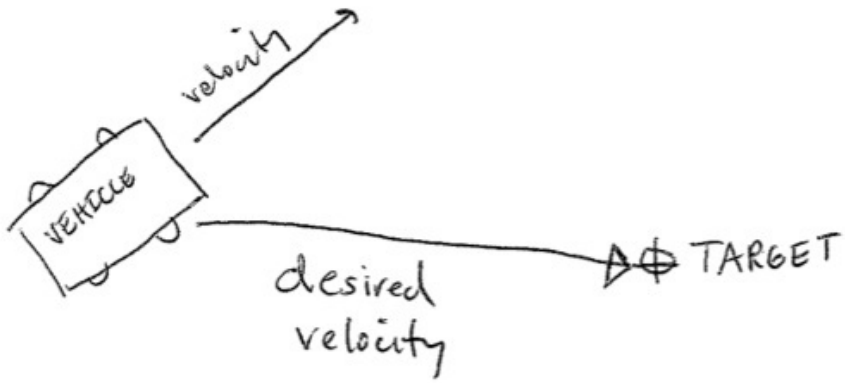
STEERING FORCE = DESIRED VELOCITY - CURRENT VELOCITY

Or as we might write in Processing:

```
1 | 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:

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



NOT FOUND:

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:

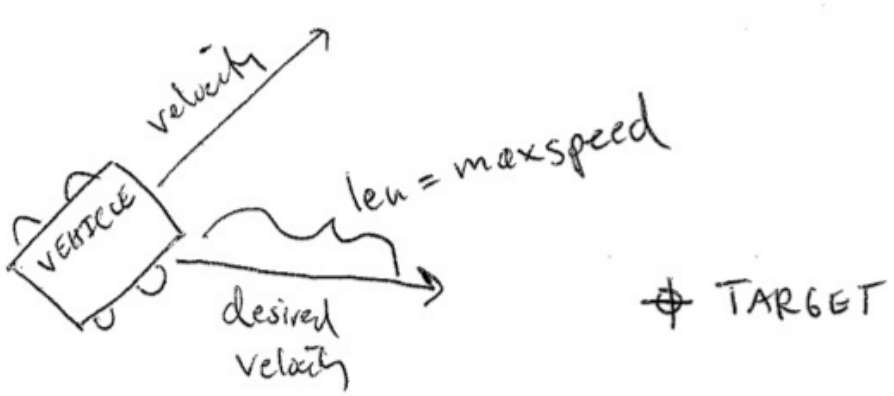
The vehicle desires to move towards the target at maximum speed.

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);
```



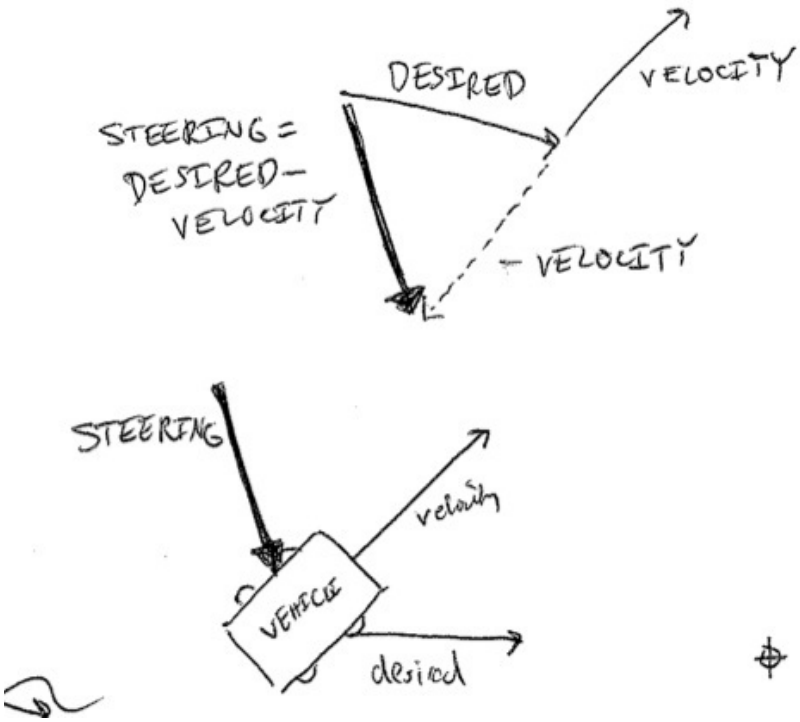
Putting this all together, we can write a function called seek() that receives a PVector target and calculates a steering force towards that target.

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

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.



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 with a variable that limits the magnitude of the steering force. Let's call it maxforce. And so finally, we have:

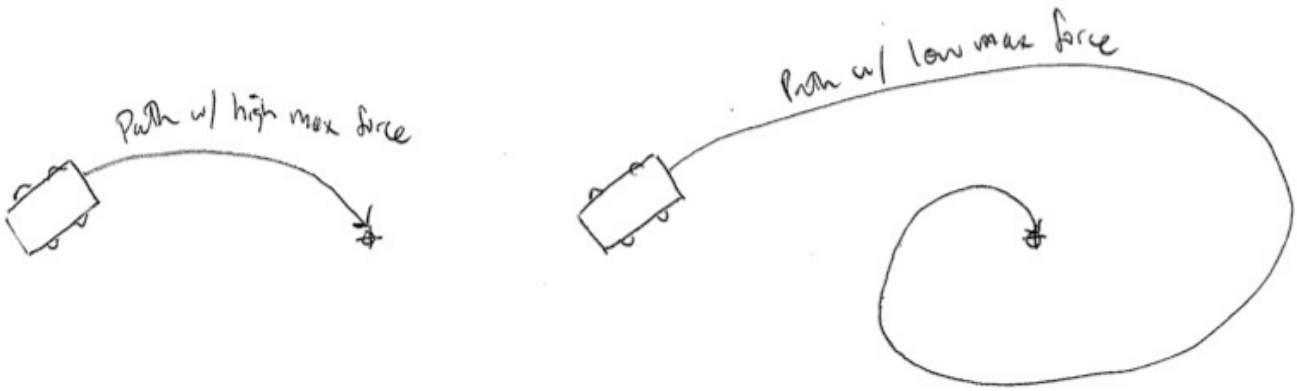
```
1   class Vehicle {
2  PVector location;
3  PVector velocity;
4  PVector acceleration;
5  float maxspeed;      // Maximum speed
6  float maxforce;      // Maximum force
```

followed by:

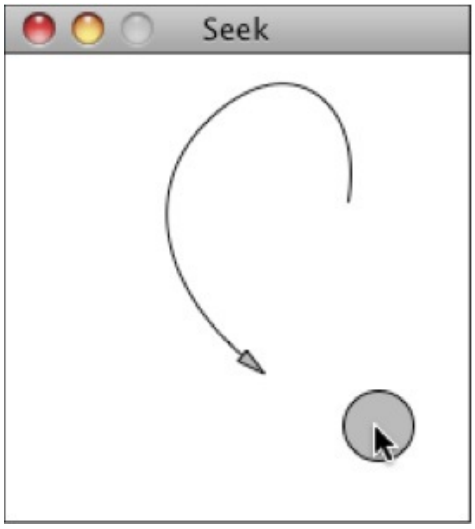
```
1    void seek(PVector target) {
2    PVector desired = PVector.sub(target,location);
3    desired.normalize();
4    desired.mult(maxspeed);
5    PVector steer = PVector.sub(desired,velocity);
6
7    steer.limit(maxforce);  $$ Limit the magnitude of the steering force
8
9    applyForce(steer);
10  }
```

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 achieve the result we want. 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 better its health, the better it can steer.)



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




```
1    *Example 6-1: Seeking a Target*
2  class Vehicle {
3
4    PVector location;
5    PVector velocity;
6    PVector acceleration;
7    float r;            $$ Additional variable for size
8    float maxforce;
9    float maxspeed;
10
11   Vehicle(float x, float y) {
12     acceleration = new PVector(0,0);
13     velocity = new PVector(0,0);
14     location = new PVector(x,y);
15     r = 3.0;
16     maxspeed = 4;          $$ Arbitrary values for maxspeed and force; try varying these
17     maxforce = 0.1;
18   }
19
20   void update() {          $$ Our standard “Euler integration” motion model
21     velocity.add(acceleration);
22     velocity.limit(maxspeed);
23     location.add(velocity);
24     acceleration.mult(0);
25   }
26
27   void applyForce(PVector force) {    $$ Newton’s second law; we could divide by ma
```

```
28      acceleration.add(force);
29    }
30
31    void seek(PVector target) {        $$ Our seek steering force algorithm
32      PVector desired = PVector.sub(target,location);
33      desired.normalize();
34      desired.mult(maxspeed);
35      PVector steer = PVector.sub(desired,velocity);
36      steer.limit(maxforce);
37      applyForce(steer);
38    }
39
40    void display() {                   $$ Vehicle is a triangle pointing in
41      float theta = velocity.heading2D() + PI/2;   the direction of velocity; since it is draw
42      fill(175);                       pointing up, we rotate it an additional 90
43      stroke(0);                         degrees
44      pushMatrix();
45      translate(location.x,location.y);
46      rotate(theta);
47      beginShape();
48      vertex(0, -r*2);
49      vertex(-r, r*2);
50      vertex(r, r*2);
51      endShape(CLOSE);
52      popMatrix();
53    }
```

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.