Chapter 4. Particle Systems

"That is wise. Were I to invoke logic, however, logic clearly dictates that the needs of the many outweigh the needs of the few."

— Spock

In 1982, William T. Reeves, a researcher at Lucasfilm Ltd., was working on the film *Star Trek II: The Wrath of Khan*. Much of the movie revolves around the Genesis Device, a torpedo that when shot at a barren, lifeless planet has the ability to reorganize matter and create a habitable world for colonization. During the sequence, a wall of fire ripples over the planet while it is being "terraformed." The term ***particle system***, an incredibly common and useful technique in computer graphics, was coined in the creation of this particular effect.

> *"A particle system is a collection of many many minute particles that together represent a fuzzy object. Over a period of time, particles are generated into a system, move and change from within the system, and die from the system."*
>
> —William Reeves, "Particle Systems—A Technique for Modeling a Class of Fuzzy Objects," ACM Transactions on Graphics 2:2 (April 1983), 92.

Since the early 1980s, particle systems have been used in countless video games, animations, digital art pieces, and installations to model various irregular types of natural phenomena, such as fire, smoke, waterfalls, fog, grass, bubbles, and so on.

This chapter is dedicated to looking at implementation strategies for coding a particle system. How do you organize your code? Where do you store information related to individual particles versus information related to the system as a whole? The examples I'll cover will focus on managing the data associated with a particle system. They'll use simple shapes for the particles and apply only the most basic behaviors (such as gravity). However,

by building on this framework and adding more interesting ways to render the particles and compute behaviors, you can achieve a variety of effects.

4.1 Why You Need Particle Systems

I've defined a particle system to be a collection of independent objects, often represented by a simple shape or dot. Why does this matter? Certainly, the prospect of modeling some of the phenomena listed (explosions!) is attractive and potentially useful. But really, there's an even better reason to explore particle systems. If you want to get anywhere in this nature of code life, you're likely to find yourself developing systems of *many* things–balls bouncing, birds flocking, ecosystems evolving, all sorts of things in plural.

Just about every chapter after this one is going to deal with a list of objects. Yes, I've dipped my toe in the array waters in some of the first vector and forces examples. But now it's time to go where no array has gone before.

First, I'm going to want to deal with flexible quantities of elements. Some examples will have zero things, sometimes one thing, sometimes ten things, and sometimes ten thousand things. Second, I'm going to want to take a more sophisticated object-oriented approach. Instead of writing a class to describe a single particle, I'm also going to want to write a class that describes the collection of particles—the particle system itself. The goal here is to be able to write a sketch that looks like the following:

Ah, isn't this main program so simple and lovely?

```
let system;

function setup() {
  createCanvas(640, 360);
  system = new ParticleSystem();
}

function draw() {
  background(255);
  system.run();
}
```

No single particle is ever referenced in the above code, yet the result will be full of particles flying all over the screen. Getting used to writing p5.js sketches with multiple classes, and classes that keep lists of instances of other classes, will prove very useful as you get to later chapters in this book.

Finally, working with particle systems is also a good excuse to tackle two other object-oriented programming techniques: inheritance and polymorphism. With the examples you've seen up until now, I've always used an array of a single type of object, like "movers" or "oscillators." With inheritance (and polymorphism), I'll demonstrate a convenient way to store a

single list containing objects of different types. This way, a particle system need not only be a system of a single type of particle.

Though it may seem obvious to you, I'd also like to point out that my examples are modeled after conventional implementations of particle systems, and that's where I will begin in this chapter. However, the fact that the particles in this chapter look or behave a certain way should not limit your imagination. Just because particle systems tend to look sparkly, fly forward, and fall with gravity doesn't mean that those are the characteristics yours should have.

The focus here is on how to keep track of a system of many elements. What those elements do and how those elements look is up to you.

4.2 A Single Particle

Before I can get rolling on coding the system itself, I need to write the class to describe a single particle. The good news: I've done this already! The `Mover` class from Chapter 2 serves as the perfect template. A particle is an independent body that moves about the canvas. It has `position`, `velocity`, and `acceleration`, a constructor to initialize those variables, and functions to `display()` itself and `update()` its position.

```
class Particle {

  Particle(x, y) {
    this.position = createVector(x, y);
    this.acceleration = createVector();
    this.velocity = createVector();
  }

  update() {
    this.velocity.add(this.acceleration);
    this.position.add(this.velocity);
  }

  display() {
    stroke(0);
    fill(175);
    circle(this.position.x, this.position.y, 8);
  }
}
```

A "Particle" object is just another name for our "Mover." It has position, velocity, and acceleration.

This is about as simple as a particle can get. From here, I could take the particle in several directions. I could add the `applyForce()` function to affect the particle's behavior (I'll do precisely this in a future example). I could also add variables to describe color and shape, or load a `p5.Image` to draw the particle. For now, however, I'll focus on adding just one

additional detail: ***lifespan***.

Some particle systems involve something called an ***emitter***. The emitter is the source of the particles and controls the initial settings for the particles: position, velocity, and more. An emitter might emit a single burst of particles, or a continuous stream of particles, or both. The new feature here is that a particle born at the emitter does not live forever. If it were to live forever, the p5.js sketch would eventually grind to a halt as the amount of particles increases to an unwieldy number over time. As new particles are born, old particles need to be removed. This creates the illusion of an infinite stream of particles, and the performance of the sketch does not suffer. There are many different ways to decide when a particle is ready to be removed. For example, it could come into contact with another object, or it could leave the canvas. For this first `Particle` class, I'll choose to add a `lifespan` variable that acts like a countdown timer. The timer will start at 255 and count down to 0, when the particle will be considered "dead." The code for this in the `Particle` class as:

```
class Particle {

  constructor(x, y) {
    this.position = createVector(x, y);
    this.acceleration = createVector();
    this.velocity = createVector();
    this.lifespan = 255;                    // A new variable to keep track of how long the particle has been "alive". We start at 255 and count down for convenience
  }

  update() {
    this.velocity.add(this.acceleration);
    this.position.add(this.velocity);
    this.lifespan -= 2.0;                   // Lifespan decreases
  }

  display() {
    stroke(0, this.lifespan);               // Since the life ranges from 255 to 0 it can be used also for alpha
    fill(175, this.lifespan);
    circle(this.position.x, this.position.y, 8);
  }
}
```

The reason I chose to start the lifespan at 255 and count down to 0 is for convenience. With those values, I can assign `lifespan` to act as the alpha transparency for the circle as well. When the particle is "dead" it will also have faded away.

With the addition of the `lifespan` property, I'll also need one additional function—a function that can be queried (for a true or false answer) as to whether the particle is alive or dead. This will come in handy when writing the `ParticleSystem` class, whose task will be to manage the

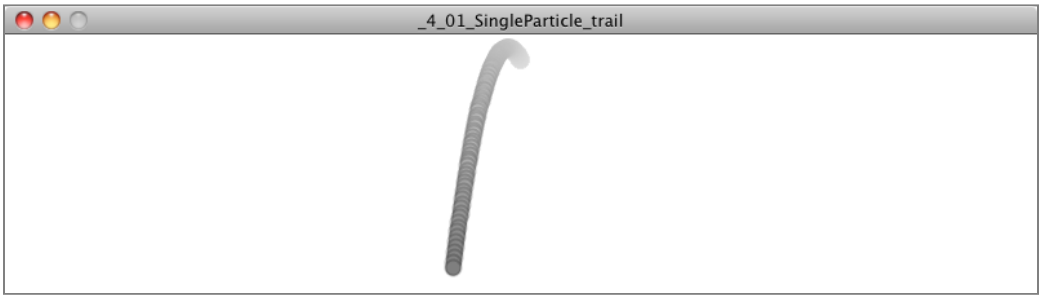
list of particles themselves. Writing this function is pretty easy; I just need to check and see if the value of `lifespan` is less than 0. If it is `return true`, if not `return false`.

```
isDead() {
  if (this.lifespan < 0.0) {                    Is the particle still alive?
    return true;
  } else {
    return false;
  }
}
```

Or more simply, I can return the result of the boolean expression itself!

```
isDead() {
  return (this.lifespan < 0.0);                 Is the particle still alive?
}
```

Before I get to the next step of making many particles, it's worth taking a moment to make sure the particle works correctly and create a sketch with one single `Particle` object. Here is the full code below, with one small addition–giving the particle a random initial velocity as well as adding `applyForce()` (to simulate gravity).



Example 4.1: A single particle

```
let particle;

function setup() {
  createCanvas(640, 360);
  particle = new Particle(width / 2, 10);
}

function draw() {
  background(255);
```

```
  particle.update();
  particle.display();
```

Operating the single Particle

```
  let gravity = createVector(0, 0.1);
  particle.applyForce(gravity);
```

Applying a gravity force

```
  if (p.isDead()) {
    print("Particle dead!");
  }
}

class Particle {

  constructor(x,y) {
    this.position =  createVector(x, y);
    this.velocity = createVector(random(-1, 1), random(-2, 0));
    this.acceleration = createVector(0, 0);
    this.lifespan = 255.0;
  }

  update() {
    this.velocity.add(this.acceleration);
    this.position.add(this.velocity);
    this.lifespan -= 2.0;
  }

  display() {
    stroke(0, this.lifespan);
    fill(0, this.lifespan);
    circle(this.position.x, this.position.y, 8);
  }

  applyForce(force) {
    this.acceleration.add(force);
  }

  isDead() {
    return (this.lifespan < 0.0);
  }
}
```

For demonstration purposes the Particle a random velocity.

Keeping the same physics model as with previous chapters

Is the Particle alive or dead?

Exercise 4.1

Create a `run()` function in the `Particle` class that handles `update()`, `display`, and `applyForce`. What are the pros and cons of this approach?

Exercise 4.2

Add angular velocity (rotation) to the particle. Create your own non-circle particle design.

Armed with a class to describe a single particle, I'm ready for the next big step. How do you keep track of many particles, not knowing exactly how many particles you might have at any given time?

4.3 The Array

Thankfully, the wonderful JavaScript `Array` has all the functionality we need for managing a list of `Particle` objects. The the built-in JavaScript functions available in the JavaScript class `Array` will allow me to add and remove particles and manipulate the arrays in all sorts of powerful ways. Although there are some cons to this approach, in order to keep the subsequent code examples more concise, I'm use a solution to Exercise 4.1 and assume a `run()` method that handles all of the particle's functionality. JavaScript Array Documentation (see page 0).

```
let total = 10;
let particles = [];                                    Starting with an empty array

function setup() {
  for (let i = 0; i < total; i++) {                    This is what you're probably used to, accessing
    particles[i] = new Particle(width/2, height/       elements on the array via an index and
2);                                                    brackets—[].
  }
}

function draw() {
  for (let i = 0; i < particles.length; i++) {
    let particle = particles[i];
    particle.run();
  }
}
```

This last `for` loop demonstrates how to call functions on every element of an array by accessing each index. I initialize a variable `i` with value 0 and count up by 1, accessing each element of the array until I reach the end. However, this is a good time to mention the JavaScript "`for of` loop", which is a bit more concise. The "for of" loop works with arrays as follows:

```
function draw() {
  for (let particle of particles){
    particle.run();
  }
}
```

Let's translate that. Say "each" instead of "let" and "in" instead of "of":

"For each particle in particles, update and display that particle!"

I know. You cannot contain your excitement. I can't. I know it's not necessary, but I just have to type that again.

```
for (let particle of particles){
  particle.run();
}
```

Simple, elegant, concise, lovely. Take a moment. Breathe. I have some bad news. Yes, I may love that "for of" loop and I will get to use it in examples. But not just yet.

The code I've written above doesn't take advantage of the JavaScript's ability to remove elements from an array. I need to design an example that fits with the particle system scenario, where a continuous stream of particles are emitted, adding one new particle with each cycle through `draw()`. I'll skip rehashing the `Particle` class code here, as it doesn't need to change. What we have so far is:

```
let particles = [];

function setup() {
  createCanvas(640, 360);
}

function draw() {
  background(255);
  particles.push(new Particle(width/2, 50));
  for (let particle of particles) {
    particle.run();
  }
}
```

A new Particle object is added to the array every cycle through draw().

Run the above code for a few minutes and you'll start to see the frame rate slow down further and further until the program grinds to a halt (my tests yielded horrific performance after fifteen minutes). The issue of course is that I am adding more and more particles without removing any.

Fortunately, particles can be removed from the array referencing the index position of the particle to be removed. This is why I cannot use the enhanced `for of` loop; this loop provides

no means for deleting elements while iterating. Instead, I can use the Array `splice()` method. (Yes, an array in JavaScript is actually an object created from the class `Array` with many methods!). The `splice()` method removes one or more elements from an array starting from a given index.

```
for (let i = 0; i < particles.length; i++) {
  let particle = particles[i];
  particle.run();
  if (particle.isDead()) {
    particles.splice(i, 1);                    Remove one particle at index i
  }
}
```

Although the above code will run just fine (and the program will never grind to a halt), I have opened up a medium-sized can of worms. Whenever you manipulate the contents of an array while iterating through that very array, you can get into trouble. Take, for example, the following code.

```
for (let i = 0; i < particles.length; i++) {
  let particle = particles[i];
  particle.run();
                                               Adding a new Particle to the list while iterating?
  particles.push(new Particle(width / 2, 50));
}
```

This is a somewhat extreme example (with flawed logic), but it proves the point. In the above case, for each particle in the array, I add a new particle to that array (and so the `length` of the array increases). This results in an infinite loop, as `i` will never increment past the size of the array.

While removing elements from the array during a loop doesn't cause the program to crash (as it does with adding), the problem is almost more insidious in that it leaves no evidence. To discover the flaw I must first establish an important fact. When an object is removed from the array with`splice()`, all elements are shifted one spot to the left. Note the diagram below where particle C (index 2) is removed. particles A and B keep the same index, while particles D and E shift from 3 and 4 to 2 and 3, respectively.

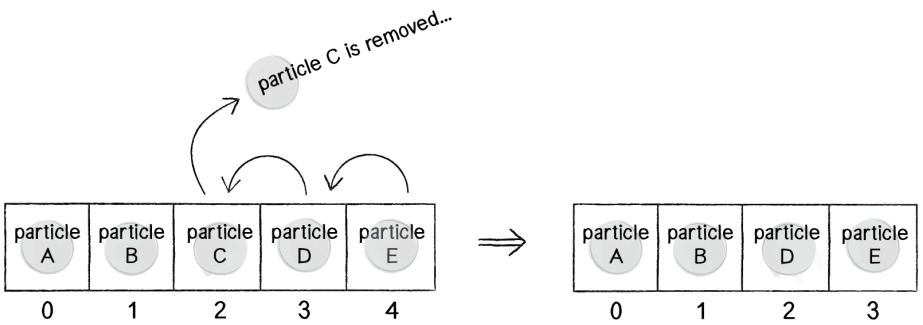


Figure 4.1

Let's pretend we are `i` looping through the `array`.

```
when i = 0 → Check particle A → Do not delete
when i = 1 → Check particle B → Do not delete
when i = 2 → Check particle C → Delete!
          Slide particles D and E back from slots 3 and 4 to 2 and 3
when i = 3 → Check particle E → Do not delete
```

Notice the problem? Particle D was never checked! When C was deleted from slot #2, D moved into slot #2, but `i` has already moved on to slot #3. This is not a total disaster, since particle D will get checked the next time around. Still, the expectation is that the code should iterate through every single element of the array. Skipping one is unacceptable!

There are two solutions to this problem. The first solution is to iterate through the array backwards. If you are sliding elements from right to left as elements are removed, it's impossible to skip an element. Here's how the code looks:

```
for (let i = particles.length - 1; i >= 0; i--) {    Looping through the list backwards
  let particle = particles[i];
  particle.run();
  if (particle.isDead()) {
    particles.splice(i, 1);
  }
}
```

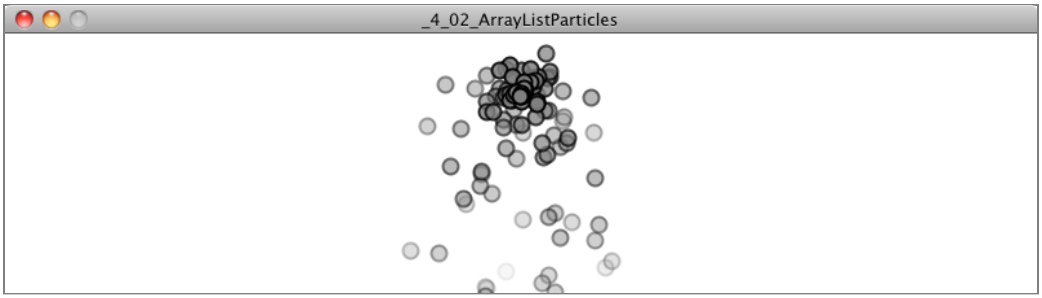
A second solution is to use something known as a "higher-order" function. A higher-order function is one that receives another function as an argument (or returns a function). In the case of JavaScript arrays there are many higher-order functions. A common one is `sort()` which takes as its argument a function that defines how to compare two elements of the array (therefore sorting the array according to that comparison.) Here, I can make use of the higher order function `filter()`. `filter()` checks each item in the specified array and keeps only the item(s) where the given condition is true (removing those that return false).

```
particles = particles.filter(function(particle) {
  return !particle.isDead();                          Keep particles that are not dead!
});
```

This is more commonly written using JavaScript's arrow notation. (To learn more, you can watch this higher-order functions and arrow notation video tutorial (see page 0).)

```
particles = particles.filter(particle => !particle.isDead());
```

For the purposes of this book, I am going to stick with the `splice()` method, but I encourage you to explore writing your code with higher-order functions and arrow notation.



Example 4.2: Array of particles

```
let particles = [];

function setup() {
  createCanvas(640, 360);
}

function draw() {
  background(255);

  particles.push(new Particle(width / 2, 50);

  for (let i = particles.length - 1; i >= 0; i--
) {
    let particle = particles[i];
    particle.run();
    if (particle.isDead()) {
      particles.splice(i, 1);
    }
  }
}
```

Looping through the array backwards for deletion

4.4 The Particle System Class

OK. Now I've done two things. I've written a class to describe an individual `Particle` object. I've conquered the array and used it to manage a list of many particles (with the ability to add and delete at will).

I could stop here. However, one additional step I can and should take is to write a class to describe the list of `Particle` objects itself—the `ParticleSystem` class. This allows me to remove the bulky logic of looping through all particles from `draw()`, as well as open up the

possibility of having more than one particle system.

If you recall the goal I set at the beginning of this chapter was to write code like:

```
let system;                                    Just one wee ParticleSystem!

function setup() {
  createCanvas(640, 360);
  system = new ParticleSystem();
}

function draw() {
  background(255);
  system.run();
}
```

Let's take the code from Example 4.2 and review a bit of object-oriented programming, looking at how each piece `setup()` and `draw()` can fit into the `ParticleSystem` class.

Array in `setup()` and `draw()`	Array in the ParticleSystem class

`let particles = [];`
 	
`function setup() {`
`  createVector(640, 360);`
`}`
 	
`function draw() {`
`  background(255);`
 	
`  particles.push(new Particle());`
 	
`  for (let i = particles.length - 1; i >= 0; i--) {`
`    let particles = particles[i];`
`    particle.run();`
`    if (particle.isDead()) {`
`      particles.splice(i, 1);`
`    }`
`  }`
`}`
	`class ParticleSystem {`
	
	`  constructor() {`
	`    this.particles = [];`
	`  }`
	
	`  addParticle() {`
	`    this.particles.push(new Particle());`
	`  }`
	
	`  run() {`
	`    for (let i = this.particles.length - 1; i >= 0; i--) {`
	`      let particle = this.particles[i];`
	`      particle.run();`
	`      if (particle.isDead()) {`
	`        this.particles.splice(i, 1);`
	`      }`
	`    }`
	`  }`
	`}`

I could also add new features to the particle system itself. For example, it might be useful for the `ParticleSystem` class to keep track of an origin point where particles are made. This fits

with the idea of a particle system being an “emitter,” a place where particles are born and sent out into the world. The origin point could be initialized in the constructor.

Example 4.3: Simple Single Particle System

```
class ParticleSystem {

  ParticleSystem(x, y) {
    this.origin = createVector(x, y);
    this.particles = [];
  }

  addParticle() {
    this.particles.add(new Particle(origin.x, origin.y));
  }
```

This particular ParticleSystem implementation includes an origin point where each Particle begins.

The origin is passed to each Particle when it is added.

Exercise 4.3

Make the origin point move dynamically. Emit particles from the mouse position or use the concepts of velocity and acceleration to make the system move autonomously.

Exercise 4.4

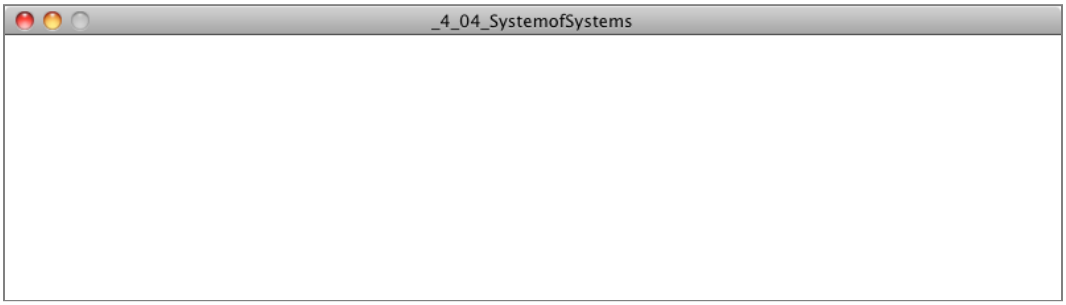
Building off Chapter 3’s “Asteroids” example, use a particle system to emit particles from the ship’s “thrusters” whenever a thrust force is applied. The particles’ initial velocity should be related to the ship’s current direction.

4.5 A System of Systems

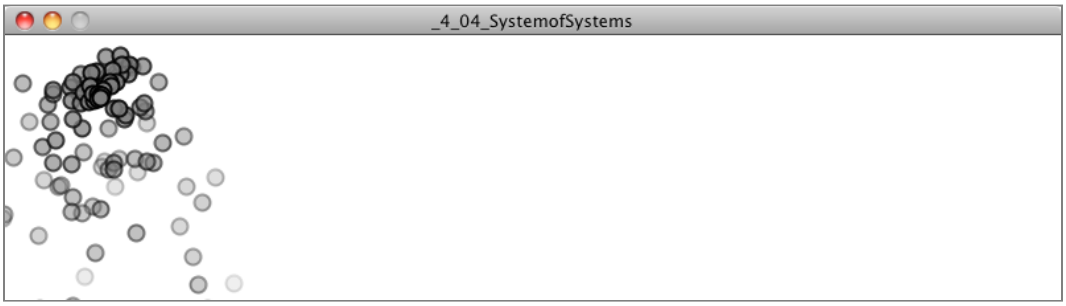
Let’s take a moment to recap what I’ve covered so far. I described an individual `Particle` object. I also described a system of `Particle` objects, and this we call a “particle system.” And I’ve defined a particle system as a collection of independent objects. But isn’t a particle system itself an object? If that’s the case (which it is), there’s no reason why I couldn’t also build a collection of many particle systems, i.e. a system of systems.

This line of thinking could of course take you even further, and you might lock yourself in a basement for days sketching out a diagram of a system of systems of systems of systems of systems of systems. Of systems. After all, I could create a description of the world in this way. An organ is a system of cells, a human body is a system of organs, a neighborhood is a system of human bodies, a city is a system of neighborhoods, and so on and so forth. While this is an interesting road to travel down, it's a bit beyond where I'd like to be right now. It is, however, quite useful to know how to write a p5.js sketch that keeps track of many particle systems, each of which keep track of many particles. Take the following scenario.

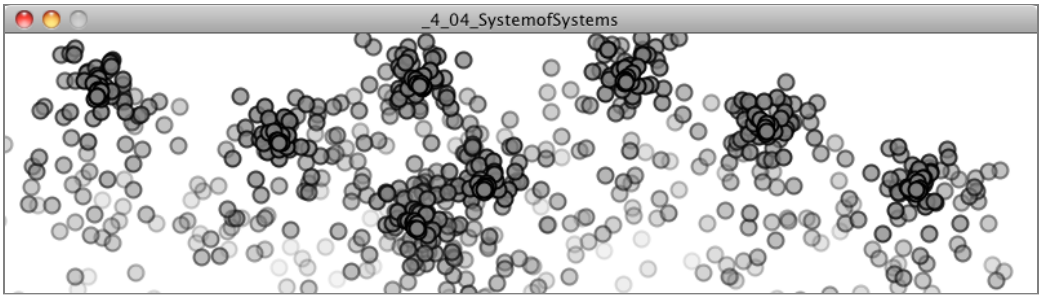
You start with a blank screen.



You click the mouse and generate a particle system at the mouse's position.



Each time you click the mouse, a new particle system is created at the mouse's position.



In Example 4.3 (see page 0), I stored a single reference to a `ParticleSystem` object in the variable `system`.

```
let ps;

function setup() {
  createCanvas(640, 360);
  system = new ParticleSystem(1, createVector(width/2, 50));
}

function draw() {
  background(255);
  system.run();
  system.addParticle();
}
```

For this new example, what I want to do instead is create an `Array` to keep track of multiple instances of particle systems themselves. When the sketch begins, i.e. in `setup()`, the `Array` is empty.

Example 4.4: System of systems

```
let systems = [];                    This time, the type of thing we are putting in the
                                     ArrayList is a ParticleSystem itself!

function setup() {
  createCanvas(600, 200);
}
```

Whenever the mouse is pressed, a new `ParticleSystem` object is created and placed into the `Array`.

```
function mousePressed() {
  systems.push(new ParticleSystem(createVector(mouseX, mouseY)));
}
```

And in `draw()`, instead of referencing a single `ParticleSystem` object, I can now iterate over all the systems in the `Array` and call `run()` on each of them.

```
function draw() {
  background(255);

  for (let system of systems) {
    system.run();
    system.addParticle();
  }
}
```

Since we aren't deleting elements, we can use the for of loop!

Exercise 4.5

Rewrite Example 4.4 so that each particle system doesn't live forever. When a particle system is empty (i.e. has no particles left), remove it from the array `systems`.

Exercise 4.6

Create a simulation of an object shattering into many pieces. How can you turn one large shape into many small particles? Can you create several large shapes on the screen that each shatter clicked on?

4.6 Inheritance and Polymorphism: An Introduction

You may have encountered the terms *inheritance* and *polymorphism* in your programming life before this book. After all, they are two of the three fundamental principles behind the theory of object-oriented programming (the other being *encapsulation*). If you've read other programming books, chances are it's been covered. My beginner text, *Learning Processing*, has close to an entire chapter (#22) dedicated to these two topics.

Still, perhaps you've only learned about it in the abstract sense and never had a reason to really use inheritance and polymorphism. If this is true, you've come to the right place. Without these two topics, your ability to program a variety of particles and particle systems is extremely limited. (In the next chapter, I'll also demonstrate how understanding these topics will help you use physics libraries.)

Imagine the following. It's a Saturday morning, you've just gone out for a lovely jog, had a delicious bowl of cereal, and are sitting quietly at your computer with a cup of warm chamomile tea. It's your old friend So and So's birthday and you've decided you'd like to make

a greeting card with p5. How about some confetti for a birthday? Purple confetti, pink confetti, star-shaped confetti, square confetti, fast confetti, fluttery confetti, etc. All of these pieces of confetti with different appearances and different behaviors explode onto the screen at once.

What you've got here is clearly a particle system—a collection of individual pieces of confetti (i.e. particles). You might be able to cleverly design a `Particle` class to have variables that store color, shape, behavior, etc. For a variety, you initalize those variables with random values. But what if your particles are drastically different? This could become very messy, having all sorts of code for different ways of being a particle in the same class. Well, you might consider doing the following:

```
class HappyConfetti {

}

class FunConfetti {

}

class WackyConfetti {

}
```

This is a nice solution: create three different classes to describe the different kinds of pieces of confetti that are part of your particle system. The `ParticleSystem` constructor could then have some code to pick randomly from the three classes when filling the array. Note that this probabilistic method is the same one I employed in the random walk examples in the Introduction (see page 0).

```
class ParticleSystem {
  constructor(num) {
    this.particles = [];

    for (let i = 0; i < num; i++) {
      let r = random(1);
      if (r < 0.33) {                                  Randomly picking a "kind" of particle
        this.particles.add(new HappyConfetti());
      } else if (r < 0.67) {
        this.particles.add(new FunConfetti());
      } else {
        this.particles.add(new WackyConfetti());
      }
    }
  }
```

OK, I need to pause for a moment. You've done nothing wrong. All you wanted to do was wish your friend a happy birthday and enjoy writing some code. But while the reasoning

behind the above approach is quite sound, there's a problem.

> ***Aren't you going to be copying/pasting a lot of code between the different "confetti" classes?***

Yes. Even though the kinds of particles are different enough to merit our breaking them out into separate classes, there is still a ton of code that they will likely share. They'll all have vectors to keep track of position, velocity, and acceleration; an `update()` function that implements the motion algorithm; etc.

This is where ***inheritance*** comes in. Inheritance allows you to write a class that *inherits* variables and functions from another class, all the while implementing its own custom features.

Now that we understand the problem, let's look at this concept in a bit more detail and then create a particle system example that implements inheritance.

4.7 Inheritance Basics

Let's take a different example, the world of animals: dogs, cats, monkeys, pandas, wombats, and sea nettles. I'll start by coding a `Dog` class. A `Dog` object will have an age variable (an integer), as well as `eat()`, `sleep()`, and `bark()` functions.

```
class Dog {

  constructor() {
    this.age = 0;
  }

  eat() {
    print("Yum!");
  }

  sleep() {
    print("Zzzzzz");
  }

  bark() {
    print("WOOF");
  }
}
```

Now, let's move on to cats.

```
class Cat {

  constructor() {
    this.age = 0;
  }

  eat() {
    print("Yum!");
  }

  sleep() {
    print("Zzzzzz");
  }
```

Dogs and cats have the same variables (age) and functions (eat, sleep).

```
  meow() {
    print("MEOW!");
  }
}
```

No bark(), instead a unique function for meowing.

As I move on to rewriting the same code for fish, horses, koalas, and lemurs, this process will become rather tedious. A better solution is to develop a generic `Animal` class that can describe any type of animal. All animals eat and sleep, after all. I could then say:

- A dog is an animal and has all the properties of animals and can do all the things animals do. Also, a dog can bark.
- A cat is an animal and has all the properties of animals and can do all the things animals do. Also, a cat can meow.

Inheritance makes this all possible. With inheritance, classes can inherit properties (variables) and functionality (methods) from other classes. A `Dog` class is a child (***subclass***) of an `Animal` class. Children will automatically inherit all variables and functions from the parent (***superclass***), but can also include functions and variables not found in the parent. Like a phylogenetic "tree of life," inheritance follows a tree structure. Dogs inherit from mammals, which inherit from animals, etc.

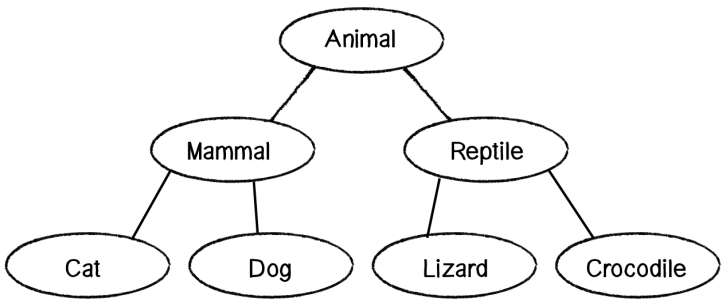


Figure 4.2

Here is how the syntax works with inheritance.

```
class Animal {

  constructor() {
    this.age = 0;
  }

  eat() {
    print("Yum!");
  }

  sleep() {
    print("Zzzzzz");
  }
}

class Dog extends Animal {
  constructor() {
    super();
  }
  bark() {
    print("WOOF!");
  }
}

class Cat extends Animal {
  constructor() {
    super();
  }
  meow() {
    print("MEOW!");
  }
}
```

The Animal class is the parent (or super) class.

Dog and Cat will inherit the variable age.

Dog and Cat will inherit the functions eat() and sleep().

The Dog class is the child (or sub) class, indicated by the code "extends Animal".

super() executes code found in the parent class.

bark() is defined in the child class, since it isn't part of the parent class.

This brings up two new terms:

- ***extends*** – This keyword is used to indicate a parent for the class being defined. Note that classes can only extend *one* class. However, classes can extend classes that extend other classes, i.e. `Dog extends Animal`, `Terrier extends Dog`. Everything is inherited all the way down the line.
- ***super()*** – This calls the constructor in the parent class. In other words, whatever you do in the parent constructor, do so in the child constructor as well. Other code can

be written into the constructor in addition to `super()`. `super()` can also receive arguments if there is a parent constructor defined with matching arguments.

A subclass can be expanded to include additional functions and properties beyond what is contained in the superclass. For example, let's assume that a `Dog` object has a haircolor variable in addition to age. The class would now look like this:

```
class Dog extends Animal {

  constructor() {
    super();
    this.haircolor = color(210, 105, 30);   // A child class can introduce new variables not included in the parent.
  }

  bark() {
    print("WOOF!");
  }
}
```

Note how the parent constructor is called via `super()`, which sets the age to 0, but the haircolor is set inside the `Dog` constructor itself. If a `Dog` object eats differently than a generic `Animal` object, parent functions can be *overridden* by rewriting the function inside the subclass.

```
class Dog extends Animal {

  constructor() {
     super();
     this.haircolor = color(random(255));
  }

  eat() {                                   // A child can override a parent function if necessary.
    print("Woof! Woof! Slurp.")             // A Dog's specific eating characteristics
  }

  bark() {
    print("WOOF!");
  }
}
```

But what if a dog eats the same way as a generic animal, just with some extra functionality? A subclass can both run the code from a parent class and incorporate custom code.

```
class Dog extends Animal {

   constructor() {
     super();
     this.haircolor = color(random(255));
   }

   eat() {
     super.eat();                          Call eat() from Animal. A child can execute a
                                           function from the parent while adding its own
                                           code.
     print("Woof!!!");                     Add some additional code for a Dog's specific
                                           eating characteristics.
   }

   bark() {
    print("WOOF!");
  }
}
```

4.8 Particles with Inheritance

Now that I've offered an introduction to the theory of inheritance and its syntax, it's time write a working example in p5.js based on the `Particle` class.

Let's review a simple `Particle` implementation, further simplified from Example 4.1 (see page 0):

```
class Particle {

  Particle(pos) {
    this.acceleration = createVector(0, 0.05);
    this.velocity = createVector(random(-1, 1), random(-2, 0));
    this.position = pos.copy();
  }

  run() {
    this.update();
    this.display();
  }

  update() {
    this.velocity.add(this.acceleration);
    this.position.add(this.velocity);
  }

  display() {
    fill(0);
    ellipse(this.position.x, this.position.y, 8, 8);
  }
}
```

Next, I'll create a subclass that extends `Particle` (I'll call it `Confetti`). It will inherit all the instance variables and methods from `Particle`. I'll also include a constructor and execute the code from the parent class with `super()`.

```
class Confetti extends Particle {

  constructor(pos) {
    super(pos);
  }                                          I could add variables for only Confetti here.

                                             There is no code here because update() is
                                             inherited from the parent.

  display() {                                Override the display method.
    rectMode(CENTER);
    fill(175);
    stroke(0);
    rect(this.position.x, this.position.y, 8,
8);
  }

}
```

Let's make this a bit more sophisticated. Let's say I want to have the `Confetti` particle rotate as it flies through the air. One option, of course, is to model angular velocity and acceleration as described in Chapter 3. For ease, however, I'll try a quick and dirty solution.

I know a particle has an *x* position somewhere between 0 and the width of the canvas. What if I said: when the particle's *x* position is 0, its rotation should be 0; when its *x* position is equal to the width, its rotation should be equal to `TWO_PI`? Does this ring a bell? Whenever a value has one range that you want to map to another range, you can use p5's `map()` function, which I discussed in the Introduction (see page 0)!

```
let angle = map(this.position.x, 0, width, 0, TWO_PI);
```

And just to give it a bit more spin, I can actually map the angle's range from 0 to `TWO_PI * 2`. Here's how this code fits into the `display()` function.

```
display() {
  let theta = map(this.position.x, 0, width, 0, TWO_PI * 2);

  rectMode(CENTER);
  fill(0, this.lifespan);
  stroke(0, this.lifespan);

  push();
  translate(this.position.x, this.position.y);
  rotate(theta);
  rectMode(CENTER);
  rect(0, 0, 8, 8);

  pop();
}
```

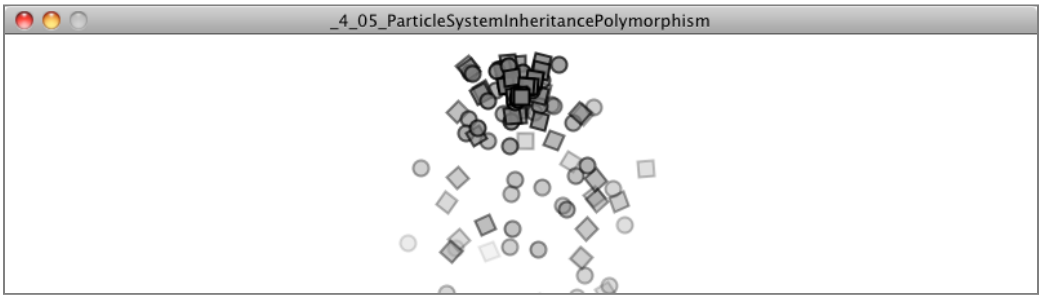
To rotate() a shape in p5, transformations are necessary. For more, visit: http://p5.org/learning/transform2d/

Exercise 4.7

Instead of using `map()` to calculate theta, try modeling angular velocity and acceleration?

Now that I have a `Confetti` subclass that extends the base `Particle` class, the next step is to also add `Confetti` objects to the array of particles defined in the `ParticleSystem` class.

4.10 Particle Systems with Inheritance



Example 4.5: Particle system inheritance

```
class ParticleSystem {

  constructor(position) {
    this.origin = this.position.copy();
    this.particles = [];                    // One list, for anything that is a Particle or extends Particle
  }

  addParticle() {
    const r = random(1);
    if (r < 0.5) {                          // A 50% chance of adding each kind of particle.
      this.particles.add(new
Particle(this.origin));
    } else {
      this.particles.add(new
Confetti(this.origin));
    }
  }

  run() {
    for (let i = this.particles.length - 1; i >= 0; i--) {
      let p = this.particles[i];
      p.run();
      if (p.isDead() {
        this.particles.splice(i, 1);
      }
    }
  }
}
```

Exercise 4.8

Create a particle system with more than two “kinds” of particles. Try varying the behavior of the particles in addition to the design.

4.11 Particle Systems with Forces

So far in this chapter, I’ve focused on structuring code in an object-oriented way to manage a collection of particles. Maybe you noticed, or maybe you didn’t, but during this process I unwittingly took a couple steps backward from the examples in previous chapters. Take a look at the constructor of the `Particle` class.

```
constructor(pos) {
  this.acceleration = createVector(0, 0.05);          Setting acceleration to a constant value!
  this.velocity = createVector(random(-1, 1), random(-2, 0));
  this.position = pos.copy();
  this.lifespan = 255.0;
}
```

And now the `update()` function:

```
update() {
  this.velocity.add(this.acceleration);
  this.position.add(this.velocity);

  [inline]// Where is the line of code to clear acceleration?

  this.lifespan -= 2.0;
}
```

The `Particle` class is structured to have a constant acceleration, one that never changes. A better framework would be to return to Newton’s second law (F = M * A) and incorporate the force accumulation algorithm from Chapter 2 (see page 0).

Step 1 is to add in the `applyForce()` function. (Remember, you need to make a copy of the `p5.Vector` before dividing it by mass.)

```
applyForce(force) {
  let f = force.copy();
  f.div(this.mass);
  this.acceleration.add(f);
}
```

Once I have this, I can add one more line of code to clear the acceleration at the end of `update()`.

```
update() {
  this.velocity.add(this.acceleration);
  this.position.add(this.velocity);
  this.acceleration.mult(0);                    There it is!
  this.lifespan -= 2.0;
 }
```

And the `Particle` class is complete!

```
class Particle {

  constructor(pos) {
    this.acceleration = createVector(0, 0);     Now start with acceleration of 0,0.
    this.velocity = createVector(random(-1, 1),random(-2, 0));
    this.position = pos.copy();
    this.lifespan = 255.0;
    this.mass = 1;                              We could vary mass for more interesting results.
  }

  run() {
    this.update();
    this.display();
  }

  applyForce(force) {                           Newton's second law and force accumulation
    const f = force.copy();
    f.div(this.mass);
    this.acceleration.add(f);
  }

  update() {                                    Standard update
    this.velocity.add(this.acceleration);
    this.position.add(this.velocity);
    this.acceleration.mult(0);
    this.lifespan -= 2.0;
  }

  display() {                                   The Particle is a circle.
    stroke(255, this.lifespan);
    fill(255, this.lifespan);
    ellipse(this.position.x, this.position.y,
8, 8);
  }
```

```
  isDead() {                                          Should the Particle be deleted?
    if (this.lifespan < 0.0) {
      return true;
    } else {
      return false;
    }
  }
}
```

Now that the `Particle` class is complete, I have a very important question to ask. Where do you call the `applyForce()` function? Where in the code is it appropriate to apply a force to a particle? In my view there's no right or wrong answer; it really depends on the exact functionality and goals of a particular p5 sketch. Still, let's proceed by considering a generic situation that would likely apply to most cases and craft a model for applying forces to individual particles in a system.

Consider the following goal: apply a force globally every time through `draw()` to all particles. I'll pick an easy one for now: a force pointing down, like gravity.

```
let gravity = createVector(0, 0.1);
```

I said it should always be applied, i.e. in `draw()`, so let's take a look at our `draw()` function as it stands.

```
function draw() {
  background(100);
  system.addParticle();
  system.run();
}
```

Well, it seems there's a small problem. `applyForce()` is a method written inside the `Particle` class, but there is no reference to the individual particles themselves, only the `ParticleSystem` object: the variable `system`.

Since I want all particles to receive the force, however, I can decide to apply the force to the particle system and let it manage applying the force to all the individual particles:

```
function draw() {
  background(100);

  let gravity = createVector(0, 0.1);
  system.applyForce(gravity);                         Applying a force to the system as a whole
  system.addParticle();
  system.run();
}
```

Of course, if I call a new function on the `ParticleSystem` object in `draw()`, well, I have to

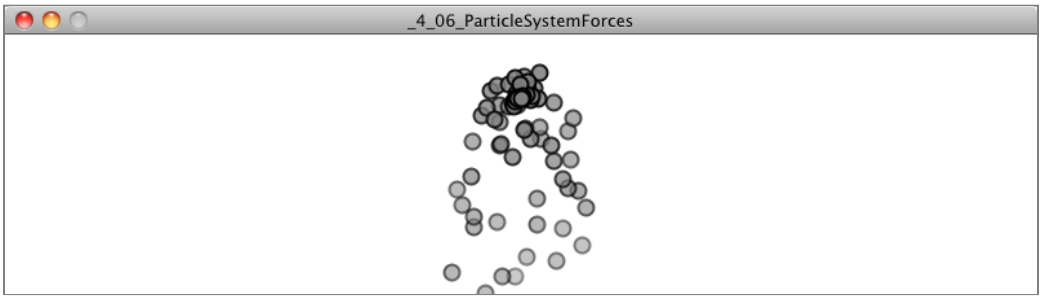
write that function in the `ParticleSystem` class. Let's describe the job that function needs to perform: receive a force as a `p5.Vector` and apply that force to all the particles.

Now in code:

```
applyForce(force) {
  for (let p of this.particles) {
    p.applyForce(force);
  }
}
```

It almost seems silly to write this function. What the code says is "apply a force to a particle system so that the system can apply that force to all of the individual particles." Nevertheless, it's actually quite reasonable. After all, the `ParticleSystem` object is in charge of managing the particles, so if you want to talk to the particles, you've got to talk to them through their manager. (Also, here's a chance for the enhanced loop since no particles are being deleted!)

Here is the full example (assuming the existence of the `Particle` class written above; no need to include it again since nothing has changed):



Example 4.6: Particle system with forces

```
let system;

function setup() {
  createCanvas(640, 360);
  system = new ParticleSystem(createVector(width/2, 50));
}

function draw() {
  background(100);

  const gravity = createVector0, 0.1);          Apply a force to all particles.
  ps.applyForce(gravity);
```

```
  ps.addParticle();
  ps.run();
}


class ParticleSystem {

  constructor(position) {
    this.origin = this.position.copy();
    this.particles = [];
  }

  addParticle() {
    this.particles.push(new Particle(this.origin));
  }

  applyForce(force) {
    for (let p of this.particles) {
      p.applyForce(force);
    }
  }

  run() {
    for (let i = this.particles.length - 1; i >=
0; i--) {
      const p = particles[i];
      p.run();
      if (p.isDead()) {
        this.particles.splice(i, 1);
      }
    }
  }
}
```

Using a for of loop to apply the force to all particles

Can't use the enhanced loop because checking for particles to delete.

4.12 Particle Systems with Repellers

What if I wanted to take this example one step further and add a `Repeller` object—the inverse of the `Attractor` object covered in Chapter 2 (see page 0) that pushes any particles away that get close? This requires a bit more sophistication because, unlike the gravity force, each force an attractor or repeller exerts on a particle must be calculated for each particle.

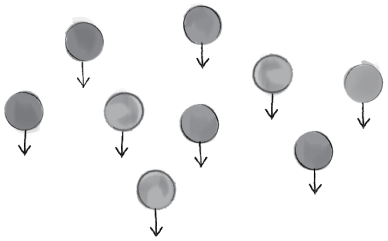


Figure 4.3: Gravity force—vectors are all identical

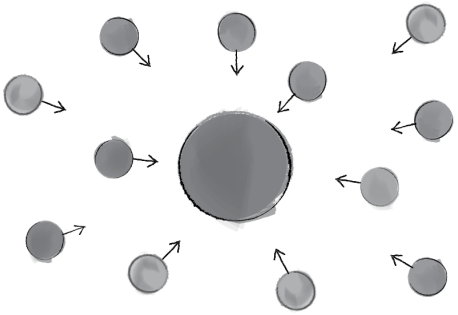


Figure 4.4: Attractor force—vectors are all different

Let's start solving this problem by examining how I might incorporate a new `Repeller` object into a simple particle system plus forces example. I'm going to need two major additions to the code:

1. A `Repeller` object (declared, initialized, and displayed).
2. A function that passes the `Repeller` object into the `ParticleSystem` so that it can apply a force to each particle object.

```
let system;
let repeller;                                    New thing: we declare a Repeller object.

function setup() {
  createCanvas(640,360);
  system = new ParticleSystem(createVector(width/2, 50));
  repeller = new Repeller(width / 2 - 20,        New thing: we initialize a Repeller object.
height / 2);
}

function draw() {
  background(100);
  system.addParticle();

  let gravity = createVector(0, 0.1);
  system.applyForce(gravity);

  system.applyRepeller(repeller);                New thing: a function to apply a force from a
                                                 repeller.
  system.run();
  repeller.display();                            New thing: display the Repeller object.
}
```

Making a `Repeller` object is quite easy; it's a duplicate of the `Attractor` class from Chapter 2, Example 2.6.

```
class Repeller {

  constructor(x, y)  {
    this.position = createVector(x, y);
    this.r = 10;
  }

  display() {
    stroke(255);
    fill(255);
    ellipse(this.position.x, this.position.y, this.r * 2, this.r * 2);
  }
}
```

A Repeller doesn't move, so you just need position.

The more difficult question is, how do I write the `applyRepeller()` function? Instead of passing a `p5.Vector` into a function like with `applyForce()`, I need to instead pass a `Repeller` object into `applyRepeller()` and ask that function to do the work of calculating the force between the repeller and all particles. Let's look at both of these functions side by side.

applyForce(force)

```
applyForce(force) {
  for (let p of this.particles) {
    p.applyForce(force);
  }
}
```

applyRepeller(repeller)

```
applyRepeller(repeller) {
  for (let p of particles) {
    let force = repeller.repel(p);
    p.applyForce(force);
  }
}
```

The functions are almost identical. There are only two differences. One I mentioned before—a `Repeller` object is the argument, not a `p5.Vector`. The second difference is the more important one. I must calculate a custom `p5.Vector` force for each and every particle and apply that force. How is that force calculated? In a function called `repel()`, the inverse of the `attract()` function from the `Attractor` class.

```
repel(particle) {
```

All the same steps we had to calculate an attractive force, only pointing in the opposite direction.

```
  let dir = p5.Vector.sub(this.position,
particle.position);
```

1) Get force direction.

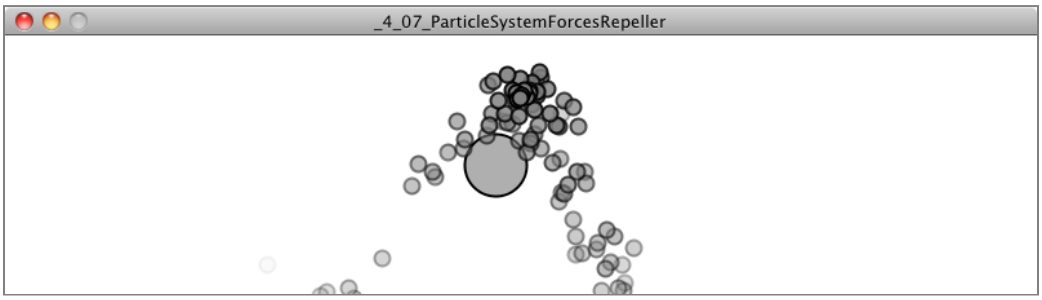
```
  let d = dir.mag();
  d = constrain(d, 5, 100);
```
2) Get distance (constrain distance).

```
  let strength = -1 * this.G / (d * d);
```
3) Calculate magnitude.

```
  dir.setMag(strength);
  return dir;
}
```
4) Make a vector out of direction and magnitude.

Notice how throughout this entire process of adding a repeller to the environment, I never once considered editing the `Particle` class itself. A particle doesn't actually have to know anything about the details of its environment; it simply needs to manage its position, velocity, and acceleration, as well as have the ability to receive an external force and act on it.

Now look at this example in its entirety, again leaving out the `Particle` class, which hasn't changed.



Example 4.7: ParticleSystem with repeller

```
let system;
```
One ParticleSystem

```
let repeller;
```
One repeller

```
function setup() {
  createCanvas(640, 360);
  system = new ParticleSystem(createVector(width/2, 50));
  repeller = new Repeller(width/2 - 20, height/2);
}

function draw() {
  background(100);
  system.addParticle();
  let gravity = createVector(0, 0.1);
  system.applyForce(gravity);
```
We're applying a universal gravity.

```
  system.applyRepeller(repeller);
```
Applying the repeller

```
  system.run();
  repeller.display();
}
```

```
class ParticleSystem {
```

The ParticleSystem manages all the Particles.

```
  constructor(position) {
    this.origin = this.position.copy();
    this.particles = [];
  }

  addParticle() {
    this.particles.add(new Particle(this.origin));
  }
```

```
  applyForce(force) {
    for (let p of this.particles) {
      p.applyForce(force);
    }
```

Applying a force as a p5.Vector

```
  }

  applyRepeller(repeller) {
```

```
    for (let p of this.particles) {
      let force = repeller.repel(p);
      p.applyForce(force);
    }
```

Calculating a force for each Particle based on a Repeller

```
  }

  run() {
    for (let i = this.particles.length - 1; i >= 0; i--) {
      const p = this.particles[i];
      p.run();
      if (p.isDead()) {
        this.particles.splice(i, 1);
      }
    }
  }
}

class Repeller {

  constructor(x, y)  {
    position = createVector(x,y);
```

```
    this.strength = 100;
```

How strong is the repeller?

```
    this.r = 10;
  }

  display() {
    stroke(255);
    fill(255);
    ellipse(this.position.x, this.position.y, this.r * 2, this.r * 2);
  }

  repel(particle) {
    let dir = p5.Vector.sub(this.position,
particle.position);
    let distance = dir.mag();
    distance = constrain(d, 5, 100);
    let force = -1 * this.strength / (distance *
distance);
    dir.mult(force);
    return dir;
  }
}
```

This is the same repel algorithm we used in Chapter 2: forces based on gravitational attraction.

Exercise 4.9

Expand the above example to include many repellers (using an array).

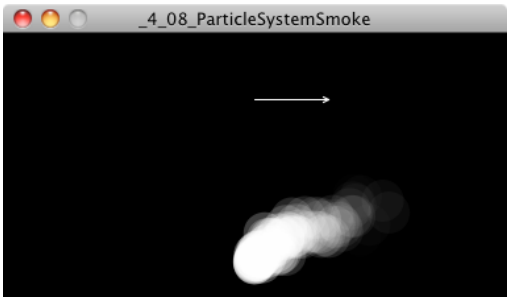
Exercise 4.10

Create a particle system in which each particle responds to every other particle. (Note that I'll be going through this in detail in Chapter 6.)

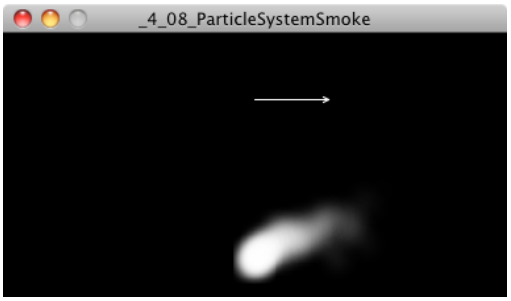
4.13 Image Textures and Additive Blending

Even though this book is almost exclusively focused on behaviors and algorithms rather than computer graphics and design, I don't think I would be able to live with myself if I finished a discussion of particle systems and never once looked at an example that involves texturing each particle with an image. The way you choose to draw a particle is the "juice," a key piece of the puzzle in terms of designing certain types of visual effects.

Consider a smoke simulation. Take a look at the following two images:



White circles



Fuzzy images with transparency

Both of these images were generated from identical algorithms. The only difference is that a white circle is drawn in image A for each particle and a "fuzzy" blob is drawn for each in B.

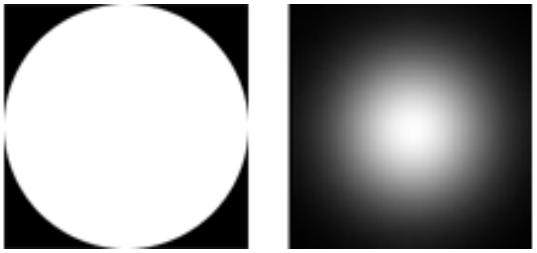


Figure 4.5

The good news here is that you get a lot of bang for very little buck. Before you write any code, however, you've got to make your image texture! I recommend using PNG format, as p5 will retain the alpha channel (i.e. transparency) when drawing the image, which is needed for blending the texture as particles layer on top of each other. Once you've made your PNG and deposited it in your sketch's "data" folder, you are on your way with just a few lines of code.

First, we'll need to declare a `p5.Image` object.

Example 4.8: Image texture particle system

```
let img;
```

Load the image in `preload()`.

```
function preload() {
  img = loadImage("texture.png");                    Loading the PNG
}
```

And when it comes time to draw the particle, use the image variable instead of drawing an ellipse or rectangle.

```
årender() {
  imageMode(CENTER);
  tint(255, this.lifespan);                    Note how tint() is the image equivalent of shape's fill().
  image(img, this.position.x, this.position.y);
}
```

Incidentally, this smoke example is a nice excuse to revisit the Gaussian distributions from the Introduction (see page 0). To make the smoke appear a bit more realistic, instead of launching allow the particles in a purely random direction initial velocity vectors mostly around a mean value (with a lower probability of outliers) can produce an effect that appears less fountain-like and more like smoke (or fire).

Using `randomGaussian()` velocities can be intialized as follows:

```
let vx = randomGaussian() * 0.3;
let vy = randomGaussian() * 0.3 - 1.0;
let vel = createVector(vx, vy);
```

Finally, in this example, a wind force is applied to the smoke mapped from the mouse's horizontal position.

```
function draw() {
  background(0);

  let dx = map(mouseX, 0, width, -0.2, 0.2);     Wind force direction based on mouseX.
  let wind = createVector(dx, 0);
  system.applyForce(wind);
  system.run();
  for (let i = 0; i < 2; i++) {                  Two particles are added each cycle through draw().
    ps.addParticle();
  }
}
```

Exercise 4.11

Try creating your own textures for different types of effects. Can you make it look like fire, instead of smoke?

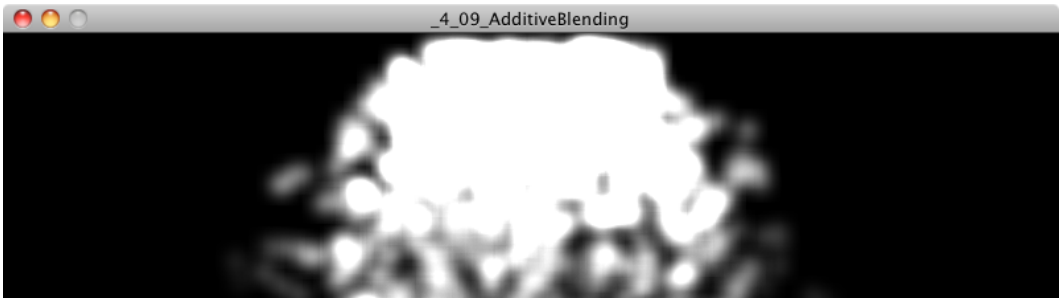
Exercise 4.12

Use an array of images and assign each `Particle` object a different image. Even though single images are drawn by multiple particles, make sure you don't call `loadImage()` any more than you need to, i.e. once for each image file.

Finally, it's worth noting that there are many different algorithms for blending colors in computer graphics. These are often referred to as "blend modes." By default, when you draw something on top of something else in p5, you only see the top layer—this is the default "blend" behavior. When pixels have alpha transparency (as they do in the smoke example), p5 uses an alpha compositing algorithm that combines a percentage of the background pixels with the new foreground pixels based on those alpha values themselves.

However, it's possible to draw using other blend modes, and a much loved blend mode for particle systems is "additive." Additive blending in Processing with particle systems was pioneered by Robert Hodgin (http://roberthodgin.com/) in his famous particle system and forces exploration, Magnetosphere, which later became the iTunes visualizer. For more see: Magnetosphere (http://roberthodgin.com/magnetosphere-part-2/).

Additive blending is one of the simpler blend algorithms and involves simply adding the pixel values of one layer to another (capping all values at 255 of course). This results in a space-age glow effect due to the colors getting brighter and brighter with more layers.



Example 4.9: Additive blending

```
function setup() {
  createCanvas(640, 360);
}
```

Then, before you go to draw anything, you set the blend mode using `blendMode()`:

```
function draw() {
  blendMode(ADD);                  Additive blending

  clear();                         Also need to clear() since the background is
                                   added and does not cover what was previously
                                   drawn.

  background(0);                   Also note that the "glowing" effect of additive
                                   blending will not work with a white (or very
                                   bright) background.

}                                  All other particle stuff goes here.
```

Exercise 4.13

Use `tint()` in combination with additive blending to create a rainbow effect.

Exercise 4.14

Try blending with other modes, such as `SUBTRACT`, `LIGHTEST`, `DARKEST`, `DIFFERENCE`, `EXCLUSION`,or `MULTIPLY`.

The Ecosystem Project

Step 4 Exercise:

Take your creature from Step 3 and build a system of creatures. How can they interact with each other? Can you use inheritance and polymorphism to create a variety of creatures, derived from the same code base? Develop a methodology for how they compete for resources (for example, food). Can you track a creature's "health" much like a particle's lifespan, removing creatures when appropriate? What rules can you incorporate to control how creatures are born into the system?

(Also, you might consider using a particle system itself in the design of a creature. What happens if an emitter is tied to the creature's position?)