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

4.1 What is a Particle System?

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."

Particle Systems—a Technique for Modeling a Class of Fuzzy Objects, author: William Reeves, ACM Transactions on Graphics, Vol. 2, No. 2, April 1983.

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

This chapter will be dedicated to looking at implementation strategies for coding a particle system. How do we organize our code? Where do we store information related to individual particles vs. information related to the system as a whole? The examples we'll look at focus on managing the data associated with a particle system. The examples will use simple shapes for the particles and apply only the most basic behaviors (gravity, etc.). However, by using this framework and building in more interesting ways to render the particles and compute behaviors, you can achieve a variety of effects.

4.2 Why Particle Systems for us?

We'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 we listed (explosions!) is attractive and potentially useful. But really, there's an even better reason for us to concern ourselves with particle systems. If we want to get anywhere in this nature of code life, we're going to need to work with systems of ***many*** things. We're going

to want to look at balls bouncing, birds flocking, ecosystems evolving, all sorts of things in plural.

Just about every chapter after this one is going to need to deal with a list of objects. Yes, we've done this with an array in some of our first vector and forces examples. But we need to go where no array has gone before.

First, we're going to want to deal with flexible quantities of elements. Sometimes we'll have zero things, sometimes one thing, sometimes ten things, and sometimes ten thousand things. Second, we're going to want to take a more sophisticated object-oriented approach. Instead of simply writing a class to describe a single Particle, we're 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 main program that looks like the following:

```
ParticleSystem ps;                    $$ Ah, isn't this main program so simple and lovely?

void setup() {
  size(200,200);
  ps = new ParticleSystem();
}

void draw() {
  background(255);
  ps.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 Processing sketches with multiple classes and classes that keep lists of instances of other classes will prove very useful as we get to more advanced chapters in this book.

Finally, working with Particle Systems is also a good excuse for us to tackle two other advanced object-oriented programming techniques: inheritance and polymorphism. With the examples we've seen up until now, we've always had an array of a single type of object: "Movers" or "Oscillators." With inheritance (and polymorphism), we'll see a convenient way that we can store a single list that contains 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 there are typical implementations of particle systems, and that's where we 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 meant that's how you should make yours.

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

4.3 A Single Particle

Before we can get rolling on the system itself, we've got to work on writing the class to describe a single Particle. The good news: we've done this already. Our "Mover" class from Chapter 2 serves as the perfect template. For us, a particle is an independent body that moves about the screen. It has ***location***, ***velocity***, and ***acceleration***, a constructor to initialize those variables, and functions to ***display()*** itself and ***update()*** its location.

```
class Particle {
  PVector location;                $$ A "Particle" object is just another name for our "Mover"
                                      It has location, velocity, and acceleration
  PVector velocity;
  PVector acceleration;

  Particle(PVector l) {
    location = l.get();
    acceleration = new PVector();
    velocity = new PVector();
  }

  void update() {
    velocity.add(acceleration);
    location.add(velocity);
  }

  void display() {
    stroke(0);
    fill(175);
    ellipse(location.x,location.y,8,8);
  }
}
```

This is about as simple as a particle can get. From here, we could take our particle in several directions. We could add an ***applyForce()*** function to affect the particle's behavior (we'll do precisely this in a future example). We could add variables to describe color and shape, or reference a ***PImage*** to draw the particle. For now, however, let's focus on adding just one additional detail: *lifespan*.

Typical particle systems involve something called an ***emitter***. The emitter is the source of the particles and controls the initial settings for the particles, location, velocity, etc. An emitter might emit a single burst of particles, or a continuous stream of particles, or both. The point is that for a typical implementation such as this, a particle is born at the emitter but does not live forever. If it were to live forever, our Processing sketch would eventually grind to a halt as the number of particles increases to an unwieldy number over time. As new particles are born, we need old particles to die. This creates the illusion of an infinite stream of particles, and the performance of our program does not suffer. There are many different ways we could decide when a particle dies. For example, it could come into contact with another object, or it could

simply leave the screen. For our first Particle class, however, we're simply going to add a "lifespan" variable. The timer will start at 255 and count down to 0, when the particle will be considered "dead." And so we expand the Particle class as follows:

```
class Particle {
  PVector location;
  PVector velocity;
  PVector acceleration;
  float lifespan;            $$ A new variable to keep track of how long the particle
                                has been "alive"

  Particle(PVector l) {
    location = l.get();
    acceleration = new PVector();
    velocity = new PVector();
    lifespan = 255;          $$ We start at 255 and count down for convenience
  }

  void update() {
    velocity.add(acceleration);
    location.add(velocity);
    lifespan -= 2.0;         $$ Lifespan decreases
  }

  void display() {
    stroke(0,lifespan);      $$ Since our life ranges from 255 to 0 we can use it for alpha
    fill(175,lifespan);
    ellipse(location.x,location.y,8,8);
  }
}
```

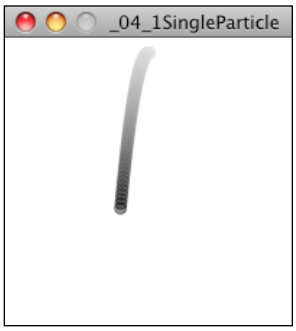
The reason we chose to start the lifespan at 255 and count down to 0 is for convenience. With those values, we can assign lifespan to act as the alpha transparency for the ellipse as well. When the particle is "dead" it will also have faded away onscreen.

With the addition of the lifespan variable, we'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 we are writing the ParticleSystem class whose task will be to manage the list of particles themselves. Writing this function is pretty easy; we just need to check and see if the value of lifespan is less than zero. If it is we *return true*, if not *return false*.

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

Before we get to the next step of making many particles, it's worth taking a moment to make sure our Particle works correctly and create a sketch with one single Particle object. Here is the full code below, with two small additions. We add a convenience function called ***run()*** that simply calls both ***update()*** and ***display()*** for us. In addition, we give the Particle a random initial velocity as well as an downward acceleration (to simulate gravity).

Example 4.x: A Single Particle




```
Particle p;

void setup() {
  size(200,200);
  p = new Particle(new PVector(width/2,10));
  smooth();
}

void draw() {
  background(255);
  p.run();                    $$ Operating the single Particle
  if (p.isDead()) {
    println("Particle dead!");
  }
}


class Particle {
  PVector location;
  PVector velocity;
  PVector acceleration;
  float lifespan;

  Particle(PVector l) {
    acceleration = new PVector(0,0.05);                  $$ For demonstration purposes we assign
    velocity = new PVector(random(-1,1),random(-2,0));      the Particle an initial velocity and
    location = l.get();                                     constant acceleration
    lifespan = 255.0;
  }

  void run() {                      $$ Sometimes it's convenient to have a "run" function that
    update();                           calls all the other functions we need
    display();
  }

  void update() {
    velocity.add(acceleration);
    location.add(velocity);
    lifespan -= 2.0;
  }

  void display() {
    stroke(0,lifespan);
    fill(0,lifespan);
    ellipse(location.x,location.y,8,8);
  }

  boolean isDead() {                $$ Is the Particle alive or dead?
    if (lifespan < 0.0) {
      return true;
    } else {
      return false;
    }
  }
}
```

Exercise: Rewrite the example so that the Particle can respond to force vectors via an ***applyForce()*** *function.*

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

Now that we have a class to describe a single Particle, we're ready for the next big step. How do we keep track of many particles, when we can't ensure exactly how many particles we might have at any given time?

4.4 The ArrayList

In truth, we could use a simple array to manage our Particle objects. Some particle systems might have a fixed number of particles, and arrays are magnificently efficient in those instances. *Processing* also offers ***expand()***, ***contract()***, ***subset()***, ***splice()*** and other methods for resizing arrays. However, for these examples, we're going to take a more sophisticated approach and use the Java class ArrayList (found in the java.util package: http://download.oracle.com/javase/6/docs/api/java/util/ArrayList.html).

Using an ArrayList is conceptually similar to a standard array, but the syntax is different. Here is some code (that assumes the existence of a generic Particle class) demonstrating identical results: first with an array, and second with an ArrayList.

```
int total = 10;                                  $$ THE STANDARD ARRAY WAY
Particle[] parray = new Particle[total];

void setup() {
  for (int i = 0; i < parray.length; i++) {      $$ This is what we're used to, accessing elements on
    parray[i] = new Particle();                     the array via an index and brackets -- []
  }
}

void draw() {
  for (int i = 0; i < parray.length; i++) {
    Particle p = parray[i];
    p.run();
  }
}


                                                          $$ THE NEW ARRAYLIST WAY (Using Generics!)
int total = 10;
ArrayList<Particle> plist = new ArrayList<Particle>(); $$ Have you ever seen this syntax before?
                                                          $$ ArrayList<Particle>
                              This is a new feature in Java 1.6 which Processing now supports,
                              which allows us to in advance specify what type of object we intend
                              to put in the ArrayList.

void setup() {
  for (int i = 0; i < total; i++) {
    plist.add(new Particle());          $$ An object is added to an ArrayList with add()
  }
}

void draw() {
  for (int i = 0; i < plist.size(); i++) {       $$ The size of the ArrayList is returned by size().
    Particle p = plist.get(i);                   $$ An An object is accessed from the ArrayList with
                                                 get(). Because we are using "Generics" we do not
                                                 need to specify a type when we pull objects out of
```

```
                                                    the ArrayList.
    p.run();
  }
}
```

This last for loop looks pretty similar to our code that looped through a "regular" array. We initialize a variable called "i" to zero and count up by one accessing each element of the ArrayList until we get to the end. However, if you use generics, i.e.

```
ArrayList<Particle> plist = new ArrayList<Particle>();
```

you can write something called an "enhanced for loop." It looks like this:

```
for (Particle p: particles) {
  p.run();
}
```

Let's translate that. Say "for each" instead of "for" and say "in" instead of ":". Now you have:

"For each Particle p in particles, run that Particle p!"

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

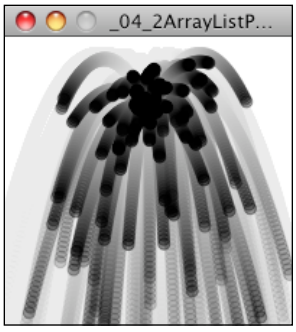
```
for (Particle p: particles) {        $$ This enhanced loop also works for regular arrays!
  p.run();
}
```

Simple, elegant, concise, lovely. Take a moment. Breathe. I have some bad news. Yes, we love that enhanced loop and we will get to use it. But not right now. Our Particle System examples will require a feature that makes using that loop impossible. Let's continue.

The code we've written above doesn't take advantage of the ArrayList's resizability, and it uses a fixed size of 10. We need to design an example that fits with our Particle System scenario, where we emit a continuous stream of Particle objects, adding one new particle with each cycle through ***draw()***. We'll skip rehashing the Particle class code here, as it doesn't need to change.

Example 4.x: ArrayList of particles

```
ArrayList particles;

void setup() {
  size(200,200);
  particles = new ArrayList();
}

void draw() {
  background(255);

  $$  A new Particle object is added to the ArrayList
      every cycle through draw().
  particles.add(new Particle(new PVector(width/2,50)));
```



```
  for (int i = 0; i < particles.size(); i++) {
    Particle p = (Particle) particles.get(i);
    p.run();
  }
}
```

Run the above code for a few minutes and you'll start to see the frame rate slow down and down and down until the program grinds to a halt (my tests yielded horrific performance after 15 minutes.) The issue of course is that we are creating more and more and more particles without removing any.

Fortunately, the ***ArrayList*** class has a convenient ***remove()*** function that allows us to delete a Particle (by referencing its index). This is why we cannot use the new enhanced for loop we just learned; the enhanced loop provides no means for deleting elements while iterating. Here, we want to call ***remove()*** when the Particle's ***isDead()*** function returns true.

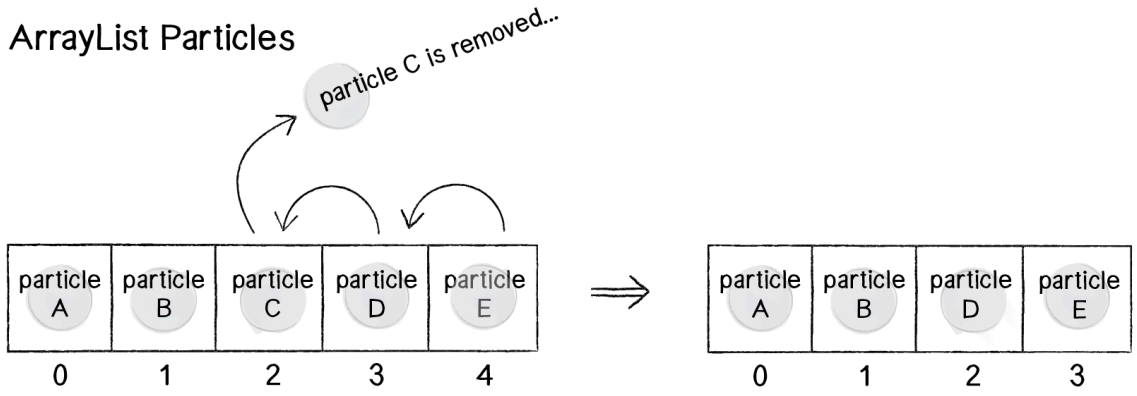
```
  for (int i = 0; i < particles.size(); i++) {
    Particle p = particles.get(i);
    p.run();
    if (p.isDead()) {          $$ If the Particle is "dead" we can go ahead and delete it from
      particles.remove(i);        the list.
    }
  }
```

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

```
  for (int i = 0; i < particles.size(); i++) {
    Particle p = particles.get(i);
    p.run();
    particles.add(new Particle(new PVector(width/2,50)));  $$ Adding a new Particle to the list
  }                                                           while iterating?
```

This is a somewhat extreme example (with flawed logic), but it proves the point. In the above case, for each Particle in the list, we add a new Particle to the list (manipulating the ***size()*** of the ArrayList). This will result in an infinite loop as *i* can never increment past the size of the ArrayList.

While removing elements from the ArrayList 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 problem we must first establish an important fact. When an object is removed from the ArrayList, 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 respectively to 2 and 3.



Let's pretend we are *i* looping through the ArrayList.

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 3,4 to 2,3
when i = 3 --> Check Particle E --> Do not delete

Notice the problem? We never checked Particle D! When C was deleted from slot #2, D moved into slot #2, but *i* already moved on to equal 3. This is not a disaster given that the next time around, Particle D will get checked. Still, the expectation is that we are writing code to iterate through every single element of the ArrayList. Skipping an element is unacceptable.

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

```
for (int i = particles.size()-1; i >= 0; i--) {     $$ Looping through the list backwards
   Particle p = (Particle) particles.get(i);
   p.run();
   if (p.isDead()) {
     particles.remove(i);
   }
 }
```

This is a perfectly fine solution in 99 cases out of 100. But sometimes, the order that the elements are drawn could be important and you may not want to iterate backwards. Java provides a special class—***Iterator***—that takes care of all of the details of iteration for you. You get to say:

Hey, I'd like to iterate through this ArrayList. Could you continue to give me the next element in the list one at a time until we get to the end? And if I remove elements or move them around in the list while we're iterating, will you make sure I don't look at any elements twice or skip any by accident?

An ArrayList can produce an ***Iterator*** object for you.

```
Iterator<Particle> it = particles.iterator();  $$ Note that with the Iterator object, we can also
                                                  use the new <ClassName> generics syntax and
                                                  specify the type that the Iterator will
                                                  reference
```

Once you've got the ***Iterator***, the ***hasNext()*** function will tell us whether there is a Particle for us to run and the ***next()*** function will grab that Particle object itself.

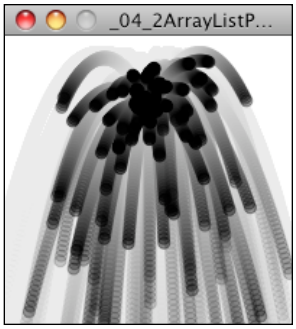
```
  while (it.hasNext()) {           $$ An Iterator object doing the iterating for you
    Particle p = it.next();
    p.run();
  }
```

And if you call the ***remove()*** function on the ***Iterator*** object during the loop, it will delete the current Particle object (and not skip ahead past the next one as we saw with counting forward through the ***ArrayList***).

```
   if (p.isDead()) {
     it.remove();            $$ An Iterator object doing the deleting for you
   }
```

Putting it all together, we have:

Example 4.x: ArrayList of Particles with Iterator




```
ArrayList particles;

void setup() {
  size(200,200);
  particles = new ArrayList();
}

void draw() {
  background(255);

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

  Iterator it = particles.iterator();
  while (it.hasNext()) {                $$ Using an Iterator object
    Particle p = it.next();                instead of counting with int i
    p.run();
    if (p.isDead()) {
       it.remove();
    }
  }
}
```

4.5 The Particle System class

OK, let's review where we are. We've done two things. We've written a class to describe an individual ***Particle*** object. We've conquered the ***ArrayList*** and used it to manage a list of many ***Particle*** objects (with the ability to add and delete at will).

We could stop here. However, one additional step we can and should take is to write a class to describe the list of ***Particle*** objects itself—the ***ParticleSystem*** class. This will allow us to remove the bulky logic of looping through all particles from the main tab, as well as open up the possibility of having more than one particle system. A system of systems!

If you recall the goal we set at the beginning of this chapter, we wanted our main tab to look like:

Example 4.x: A single simple Particle System

```
ParticleSystem ps;                                   $$ Just one wee ParticleSystem!

void setup() {
  size(200,200);
  ps = new ParticleSystem();
}

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

Let's take the code from Example 4.x and review a bit of object-oriented programming, looking at how each piece from the main tab can fit into the ***ParticleSystem*** class.

ArrayList in the main tab	ArrayList in the ParticleSystem class

`ArrayList<Particle> particles;`
 	
`void setup() {`
`  size(200,200);`
`  particles = new ArrayList<Particle>();`
`}`
 	
`void draw() {`
`  background(255);`
 	
`  particles.add(new Particle());`
 	
`  Iterator<Particle> it = particles.iterator();`
`  while (it.hasNext()) {`
`  Particle p = it.next();`
`  p.run();`
`  if (p.isDead()) {`
`  it.remove();`
`  }`
`  }`
`}`
	`class ParticleSystem {`
	`  ArrayList<Particle> particles;`
	
	`  ParticleSystem() {`
	`  particles = new ArrayList<Particle>();`
	`  }`
	
	`  void addParticle() {`
	`  particles.add(new Particle());`
	`  }`
	
	`  void run() {`
	`  Iterator<Particle> it = particles.iterator();`
	`  while (it.hasNext()) {`
	`  Particle p = it.next();`
	`  p.run();`
	`  if (p.isDead()) {`
	`  it.remove();`
	`  }`
	`  }`
	`  }`
	`}`

[DIAGRAM HOW THE PARTS FEED FROM ONE TO THE OTHER, MENTION CONSTRUCTOR IS LIKE OBJECT'S "SETUP"]

We could also add some 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 in 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 should be initialized in the constructor:

```
class ParticleSystem {
  ArrayList particles;
  PVector origin;                          $$ This particular ParticleSystem implementation
                                              includes an origin point where each Particle
  ParticleSystem(PVector location) {          begins.
    origin = location.get();
    particles = new ArrayList();
  }

  void addParticle() {
    particles.add(new Particle(origin));   $$ The origin is passed to each Particle when it
  }                                            is added.
```

Exercise: Make the origin point move dynamically. Have the particles emit from the mouse location or use the concepts of velocity and acceleration to make the system move autonomously.

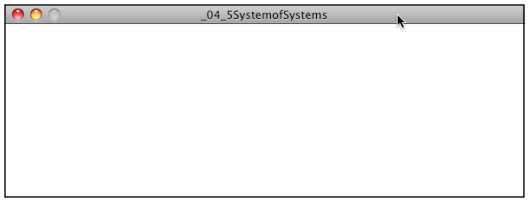
Exercise: Building off Chapter 3's "Asteroids" example, use a Particle system to emit particles from the ship's "thrusters" whenever a thrust force is applied. The particles' initial velocity should be related to the ship's current direction.

4.6 A System of Systems

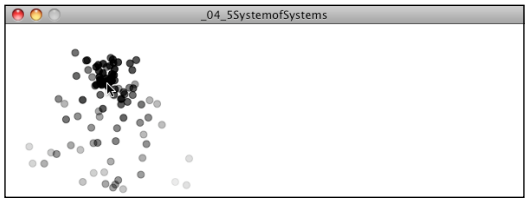
Let's review for a moment where we are. We know how to talk about an individual Particle object. We also know how to talk about a system of Particle objects, and this we call a "Particle System." And we've defined a Particle System as a collection of independent objects. But isn't a Particle System itself an object? If that's the case (which it is), there's no reason why we couldn't also have a collection of many Particle Systems, i.e. a system of systems.

This line of thinking could of course take us 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, this is how the world works. 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 we need to be right now. It is, however, quite useful to know how to write a Processing sketch that keeps track of many Particle Systems, each of which keep track of many Particles. Let's take the following scenario.

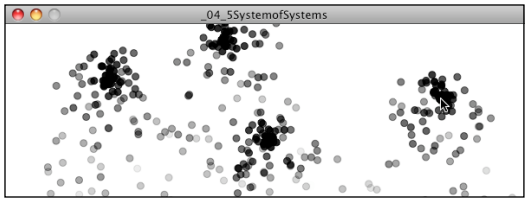
You start with a blank screen.



You click the mouse and generate a Particle System at the mouse's location.



Each time you click the mouse a new Particle System is created at the mouse's location.



In Example 4.x, we stored a single reference to a Particle System object in the variable "ps."

```
ParticleSystem ps;

void setup() {
  size(200,200);
  ps = new ParticleSystem(1,new PVector(width/2,50));
}

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

For this new example, what we want to do instead is create an ***ArrayList*** to keep track of multiple instances of Particle Systems. When the program starts (i.e. in ***setup()***), the ArrayList is empty.

```
ArrayList<ParticleSystem> systems;             $$ This time the type of thing we are putting
                                                  in the ArrayList is a ParticleSystem itself!
void setup() {
  size(600,200);
  systems = new ArrayList<ParticleSystem>();
}
```

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

```
void mousePressed() {
  systems.add(new ParticleSystem(1,new PVector(mouseX,mouseY)));
}
```

And in ***draw()***, instead of referencing a single ParticleSystem, we now look through all the systems in the ArrayList and call ***run()*** on each of them.

```
void draw() {
  background(255);
  for (ParticleSystem ps: systems) {     $$ Since we aren't deleting elements we
    ps.run();                               can use our enhanced loop!
    ps.addParticle();
  }
}
```

Exercise: Rewrite example 4.x so that each ParticleSystem doesn't live forever. When a ParticleSystem is empty (i.e. has no Particles left in its ArrayList) remove it from the ArrayList systems.

Exercise: Create a simulation of an object shattering into many pieces. How can you turn one large shape into many small particles? What if there are several large shapes on the screen and they shatter when you click on them?

4.7 Particle Systems: why we need inheritance and polymorphism

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 Processing or Java 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, we'll also see how understanding these topics will help us to 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 in Processing. 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 we've got here is clearly a Particle System—a collection of individual pieces of confetti (i.e. particles). We might be able to cleverly design our Particle class to have variables that store its color, shape, behavior, etc. And perhaps we initialize the values of these variables randomly. 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 {
  // etc.
}

class FunConfetti {
  // etc.
}

class WackyConfetti {
  // etc.
}
```

This is a nice solution: we have three different classes to describe the different kinds of pieces of confetti that could be part of our Particle System. The ParticleSystem constructor could then have some code to pick randomly from the three classes when filling the ArrayList. Note this probabilistic method is the same one we employed in our random walk examples in the introduction (see ***[REF]***).

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

OK, we now need to pause for a moment. We've done nothing wrong. All we wanted to do was wish our friend a happy birthday and enjoy writing some code. But while the reasoning behind the above approach is quite sound, we've opened up two major problems.

#1: Aren't we going to be copying/pasting a lot of code between the different "confetti" classes?

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

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

#2: How will the ArrayList know which objects are which type?

This is a pretty serious problem. Remember, we were using generics to tell the *ArrayList* what type of objects we're going to put inside it. Are we suddenly going to need three different ArrayLists?

```
ArrayList<HappyConfetti> a1 = new ArrayList<HappyConfetti>();
ArrayList<FunConfetti>   a2 = new ArrayList<FunConfetti>();
ArrayList<WackyConfetti> a3 = new ArrayList<WackyConfetti>();
```

This seems awfully inconvenient, given that we really just want one list to keep track of all the stuff in the ParticleSystem. This is not necessary because of *polymorphism*.

Polymorphism will allow us to consider objects of different types as the same type and store them in a single ArrayList..

Now that we understand the problem, let's look at these two concepts with a bit more detail and then create a Particle System example that implements both inheritance and polymorphism.

4.8 Inheritance basics

Inheritance allows us to create new classes that are based on existing classes.

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

```
class Dog {
  int age;

  Dog() {
    age = 0;
  }

  void eat() {
    // eating code goes here
  }

  void sleep() {
    // sleeping code goes here
  }

  void bark() {
    println("WOOF!");
  }
}
```

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

They also have a unique function for barking or meowing.

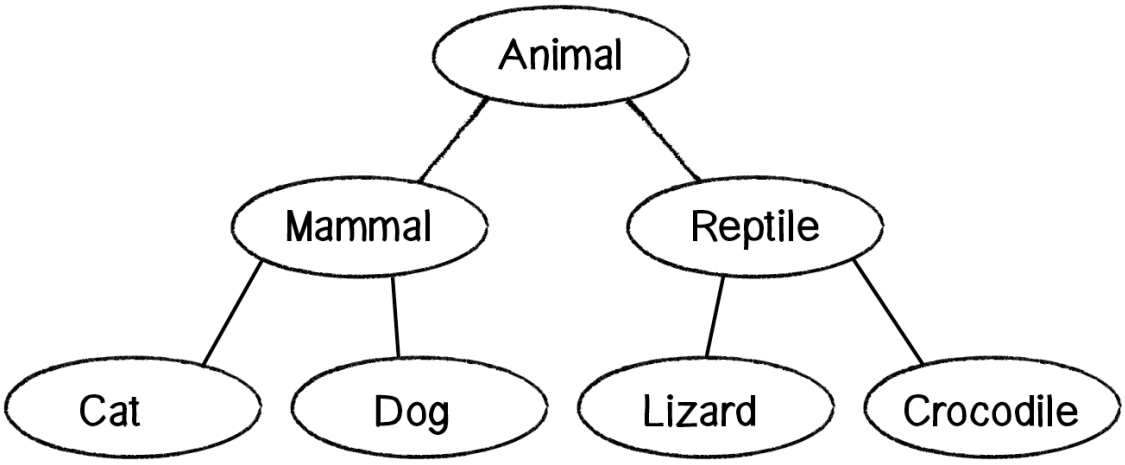
Finishing with dogs, we can now move on to cats.

```
class Cat {
  int age;

  Cat() {
    age = 0;
  }

  void eat() {
    // eating code goes here
  }

  void sleep() {
    // sleeping code goes here
  }

  void meow() {
    println("MEOW!");
  }
}
```

As we move onto fish, horses, koala bears, and lemurs, this process will become rather tedious as we rewrite the same code over and over again. What if, instead, we could develop a generic Animal class to describe any type of animal? After all, all animals eat and sleep. We could then say the following:

- 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 allows us to program just this. With ***inheritance***, classes can inherit properties (variables) and functionality (methods) from other classes. A Dog class is a child (aka ***subclass***) of an Animal class. Children inherit all variables and functions automatically from their parent (aka ***superclass***). Children can also include additional variables and functions not found in the parent. Inheritance follows a tree-structure (much like a phylogenetic "tree of life".) Dogs can inherit from Canines which inherit from Mammals which inherit from Animals, etc.



Here is how the syntax works with inheritance.

```
class Animal {                  $$ The Animal class is the parent (or super) class.
  int age;

  Animal() {                    $$ The variable age and the functions eat() and sleep() are
    age = 0;                       inherited by Dog and Cat.
  }

  void eat() {
    // eating code goes here
  }

  void sleep() {
    // sleeping code goes here
  }
}


class Dog extends Animal {      $$ The Dog class is the child (or sub) class.
                                   This is indicated with the code "extends Animal"

   Dog() {
     super();                   $$ super() means execute code found in the parent class.
   }
   void bark() {                $$ Since bark() is not part of the parent class,
    println("WOOF!");               we have to define it in the child class.
  }
}

class Cat extends Animal {
   Cat() {
     super();
   }
   void meow() {
    println("MEOW!");
  }
}
```

The following new terms have been introduced:

- ***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()*** *– super 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 hair color variable in addition to age, which is set randomly in the constructor. The class would now look like this:

```
class Dog extends Animal {
   color haircolor;                  $$ A child class can introduce new variables not
                                        included in the parent.
   Dog() {
     super();
     haircolor = color(random(255));
   }

   void bark() {
    println("WOOF!");
  }
}
```

Note how the parent constructor is called via ***super()***, setting the age to 0, but the hair color is set inside the Dog constructor itself. Suppose a Dog object eats differently than how the generic Animal does. Parent functions can be ***overridden*** by rewriting the function inside the sub class.

```
class Dog extends Animal {
   color haircolor;

   Dog() {
     super();
     haircolor = color(random(255));
   }

   void eat() {                      $$ A child can override a parent function if necessary.
     // How a dog specifically eats
   }

   void bark() {
    println("WOOF!");
  }
}
```

But what if a Dog should eat the same way an Animal does, but with some additional functionality? A subclass can both run the code from a parent class and incorporate some custom code.

```
class Dog extends Animal {
   color haircolor;

   Dog() {
     super();
     haircolor = color(random(255));
   }

   void eat() {
     // Call eat() from Animal                $$ A child can execute a function from the parent
     super.eat();                                while adding its own code as well.
     // Add some additional code
     // for how a dog specifically eats
     println("Yum!!!");
   }

   void bark() {
    println("WOOF!");
  }
}
```

4.9 Particle example with inheritance

Now that we've had an introduction to the theory of inheritance and its syntax, we can develop a working example in Processing based on our Particle class.

Let's review a simple Particle implementation (further simplified from example 4.x):

```
class Particle {
  PVector location;
  PVector velocity;
  PVector acceleration;

  Particle(PVector l) {
    acceleration = new PVector(0,0.05);
    velocity = new PVector(random(-1,1),random(-2,0));
    location = l.get();
  }

  void run() {
    update();
    display();
  }

  void update() {
    velocity.add(acceleration);
    location.add(velocity);
  }

  void display() {
    fill(0);
    ellipse(location.x,location.y,8,8);
  }
}
```

Next, we create a subclass from Particle (let's call it "Confetti"). It will inherit all the instance variables and methods from Particle. We write a new constructor with the name "Confetti" and execute the code from the parent class by calling ***super()***.

```
class Confetti extends Particle {

  $$ We could add variables for only Confetti here

  Confetti(PVector l) {
    super(l);
  }

                                          $$ Inherits update() from parent

  void display() {                        $$ Override the display method
    rectMode(CENTER);
    fill(175);
    stroke(0);
    rect(location.x,location.y,8,8);
  }
}
```

Let's make this a bit more sophisticated. Let's say we want to have the "Confetti" particle rotate as it flies through the air. We could, of course, model angular velocity and acceleration as we did in Chapter 3. Instead, we'll try a quick and dirty solution.

We know a particle has an x location somewhere between zero and the width of the window. What if we said: when the particle's x location is zero, its rotation should be zero and when its x location is equal to the width, its rotation should be equal to two PI? Does this ring a bell? Whenever we have a value with one range that we want to map to another range, we can use Processing's ***map()*** function! (See, intro ***[REF]***).

```
float angle = map(location.x,0,width,0,TWO_PI);
```

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

```
  void display() {
    rectMode(CENTER);
    fill(0,lifespan);
    stroke(0,lifespan);
    pushMatrix();                         $$ If we rotate() a shape in Processing we need to
    translate(location.x,location.y);        familiarize ourselves with transformations.  For
                                             more, visit:
                                             http://processing.org/learning/transform2d/
    float theta = map(location.x,0,width,0,TWO_PI*2);
    rotate(theta);
    rect(0,0,8,8);
    popMatrix();
  }
```

Exercise: Instead of using map() to calculate theta, how would you model angular velocity and acceleration?

Now that we have a "Confetti" particle that ***extends*** our "base" Particle class, we need to figure out how our Particle System class can manage particles of different types within the same system. To accomplish this goal, let's return to the animal kingdom inheritance example and see how the concept extends into the world of polymorphism.

4.10 Polymorphism basics

Now that we have the concept of inheritance down, we can imagine how we would program a diverse animal kingdom using ArrayLists—an array of dogs, an array of cats, array of turtles, of kiwis, etc. frolicking about.

```
ArrayList<Dog> dogs = new ArrayList<Dog>();          $$ Separate ArrayLists for each animal
ArrayList<Cat> cats = new ArrayList<Cat>();
ArrayList<Turtle> turtles = new ArrayList<Turtle>();
ArrayList<Kiwi> kiwis = new ArrayList<Kiwi>();

for (int i = 0; i < 10; i++) {
  dogs.add(new Dog());
}
for (int i = 0; i < 15; i++) {
  cats.add(new Cat());
}
for (int i = 0; i < 6; i++) {
  turtles.add(new Turtle());
}
for (int i = 0; i < 98; i++) {
  kiwis.add(new Kiwi());
}
```

As the day begins, the animals are all pretty hungry and are looking to eat. So it's off to looping time (enhanced looping time!). . .. . .

```
for (Dog d: dogs) {                                  $$ Separate loops for each animal
  d.eat();
}
for (Cat c: cats) {
  c.eat();
}
for (Turtle t: turtles) {
  t.eat();
}
for (Kiwi k: kiwis) {
  k.eat();
}
```

This works great, but as our world expands to include many more animal species, we're going to get stuck writing a lot of individual loops. Isn't this unnecessary? After all, the creatures are all animals, and they all like to eat. Why not just have one ArrayList of "Animal" objects and fill it with all different *kinds* of Animals?

```
ArrayList<Animal> kingdom = new ArrayList<Animal>(); $$ Just one ArrayList for all the animals!

for (int i = 0; i < 1000; i++) {
  if (i < 100) kingdom.add(new Dog());
  else if (i < 400) kingdom.add(new Cat());
  else if (i < 900) kingdom.add(new Turtle());
  else kingdom.add(new Kiwi());
}

for (Animal a: kingdom) {
  a.eat();
}
```

The ability to treat a Dog object as either a member of the Dog class or the Animal class (its parent) is known as ***polymorphism***, the third tenet of object-oriented programming.

Polymorphism (from the Greek *polymorphos*, meaning many forms) refers to the treatment of a single object instance in multiple forms. A Dog is certainly a Dog, but since Dog ***extends*** Animal, it can also be considered an Animal. In code, we can refer to it both ways.

```
Dog rover = new Dog();
Animal spot = new Dog();
```

Although the second line of code might initially seem to violate syntax rules, both ways of declaring a Dog object are legal. Even though we declare spot as a Animal, we're really making a Dog object and storing it in the spot variable. And we can safely call all of the Animal methods on spot because the rules of inheritance dictate that a Dog can do anything an Animal can.

What if the Dog class, however, overrides the ***eat()*** function in the Animal class? Even if spot is declared as an Animal, Java will determine that its true identity is that of a Dog and run the appropriate version of the ***eat()*** function.

This is particularly useful when we have an array or ArrayList.

4.11 Particle System with polymorphism

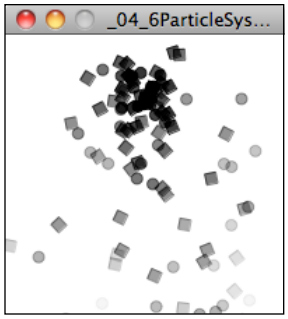
Let's pretend for a moment that polymorphism doesn't exist and rewrite a Particle System class to include many Particle objects and many Confetti objects.

```
class ParticleSystem {
  ArrayList<Particle> particles;      $$ We're stuck doing everything twice with two lists!
  ArrayList<Confetti> confetti;
  PVector origin;

  ParticleSystem(PVector location) {
    origin = location.get();
    particles = new ArrayList<Particle>();
    confetti = new ArrayList<Confetti>();
  }

  void addParticle() {
    particles.add(new Particle(origin));
    particles.add(new Confetti(origin));
  }

  void run() {
    Iterator it = particles.iterator();
    while (it.hasNext()) {
      Particle p = it.next();
      p.run();
      if (p.isDead()) {
        it.remove();
      }
    }
    it = confetti.iterator();
    while (it.hasNext()) {
      Confetti c = it.next();
      c.run();
```

```
      if (c.isDead()) {
        it.remove();
      }
    }

  }
}
```

Notice how we have two separate lists, one for Particle objects and one for Confetti objects. Every action we want to perform we have to do twice! Polymorphism allows us to simplify the above by just making one ArrayList of Particle objects that contains both standard Particle objects as well as Confetti objects. We don't have to worry about which are which; this will all be taken care of for us! (Also, note that the code for the main program and the classes has not changed so we aren't including it here. See website for full example.)

Example 4.x: Polymorphism




```
class ParticleSystem {
  ArrayList<Particle> particles;      $$ One list, for anything that is
                                         a Particle or extends Particle
  PVector origin;

  ParticleSystem(PVector location) {
    origin = location.get();
    particles = new ArrayList<Particle>();
  }

  void addParticle() {
    float r = random(1);
    if (r < 0.5) {                          $$ We have a 50% chance of
                                               adding each kind of Particle
      particles.add(new Particle(origin));
    } else {
      particles.add(new Confetti(origin));
    }
  }

  void run() {
    Iterator it = particles.iterator();
    while (it.hasNext()) {
      Particle p = it.next();         $$ Polymorphism allows us to treat everything as a Particle
                                         whether it is a Particle or Confetti
      p.run();
      if (p.isDead()) {
        it.remove();
      }
    }
  }
}
```

Exercise: Create a Particle System with different "kinds" of particles in the same system. Try varying more than just the look of the particles. How do you deal with different behaviors using inheritance?

4.8 Particle System with forces

So far this chapter, we've been focusing on structuring our code in an object-oriented way to manage a collection of Particle objects. Maybe you noticed, or maybe you didn't, but during

this process we unwittingly took a couple steps backward from where we were in previous chapters. Let's examine the constructor of our simple Particle class.

```
  Particle(PVector l) {
    acceleration = new PVector(0,0.05);  $$ We're setting acceleration to a constant value!
    velocity = new PVector(random(-1,1),random(-2,0));
    location = l.get();
    lifespan = 255.0;
  }
```

And now let's look at the ***update()*** function.

```
  void update() {
    velocity.add(acceleration);
    location.add(velocity);
                                  $$ Where is the line of code to clear acceleration?
    lifespan -= 2.0;
  }
```

Our Particle class is structured to have a constant acceleration, one that never changes. A much better framework would be to follow Newton's second law (F = M* A) and incorporate the force accumulation algorithm we worked so hard on in Chapter 2 (see p. XXX).

Step 1 would be to add in the ***applyForce()*** function (remember, we need to make a copy of the PVector before we divide it by mass).

```
  void applyForce(PVector force) {
    PVector f = force.get();
    f.div(mass);
    acceleration.add(f);
  }
```

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

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

And our Particle class is complete!

```
class Particle {
  PVector location;
  PVector velocity;
  PVector acceleration;
  float lifespan;

  float mass = 1;            $$ We could vary mass for more interesting results

  Particle(PVector l) {
    acceleration = new PVector(0,0);          $$ We now start with acceleration of 0
```

```
    velocity = new PVector(random(-1,1),random(-2,0));
    location = l.get();
    lifespan = 255.0;
  }

  void run() {
    update();
    display();
  }

  void applyForce(PVector force) {            $$ Newton's second law & force accumulation
    PVector f = force.get();
    f.div(mass);
    acceleration.add(f);
  }

  void update() {                             $$ Standard update
    velocity.add(acceleration);
    location.add(velocity);
    acceleration.mult(0);
    lifespan -= 2.0;
  }

  void display() {                            $$ Our Particle is a circle
    stroke(255,lifespan);
    fill(255,lifespan);
    ellipse(location.x,location.y,8,8);
  }

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

Now that the Particle class is completed, we have a very important question to ask. Where do we call the ***applyForce()*** function? Where in the code is it appropriate to apply a force to a particle? The truth of the matter is that there's no right or wrong answer; it really depends on the exact functionality and goals of a particle Processing sketch. Still, we can create a generic situation that would likely apply to most cases and create a model for applying forces to individual particles in a system.

Let's consider the following goal:

Apply a force globally every time through draw() to all particles.

Let's just pick an easy one: a force pointing down, like gravity.

```
PVector gravity = new PVector(0,0.1);
```

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

```
void draw() {
  background(100);
  ps.addParticle();
  ps.run();
}
```

Well, it seems that we have a small problem. ***applyForce()*** is a method written inside the Particle class, but we don't have any reference to the individual particles themselves, only the ParticleSystem object: i.e. the variable "ps".

Since we want all particles to receive the force, however, we can decide to apply the force to the Particle System and let it manage applying the force to all the individual particles:

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

  PVector gravity = new PVector(0,0.1);  $$ Applying a force to the system as a whole
  ps.applyForce(gravity);

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

Of course, if we call a new function in the ParticleSystem from ***draw()***, well, we have to write that function in the ParticleSystem class. Let's describe the job that function needs to perform:

Receive a force as a PVector and apply that force to all the Particles.

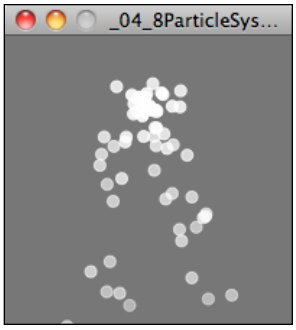
Now in code:

```
  void applyForce(PVector f) {
    for (Particle p: particles) {
      p.applyForce(f);
    }
  }
```

It almost seems silly to write this function. What we're saying 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 really quite reasonable. After all, the ParticleSystem object is in charge of managing the particles, so if we want to talk to the particles, we've got to talk to them through their manager. (Also, here's a chance for the enhanced loop since we aren't deleting particles!)

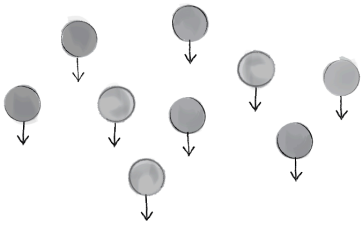
Here is the full example (assuming the existence of the Particle class written above; no need to include it again since nothing has changed):

Example 4.x: Particle System with Forces

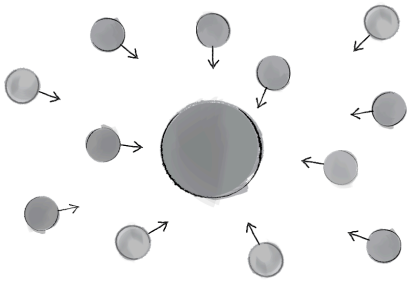



```
ParticleSystem ps;

void setup() {
  size(200,200);
  smooth();
  ps = new ParticleSystem(new PVector(width/2,50));
}

void draw() {
  background(100);

  PVector gravity = new PVector(0,0.1);        $$ Apply a force to all
  ps.applyForce(gravity);                         particles.

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


class ParticleSystem {
  ArrayList<Particle> particles;
  PVector origin;

  ParticleSystem(PVector location) {
    origin = location.get();
    particles = new ArrayList<Particle>();
  }

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

  void applyForce(PVector f) {                 $$ Using an enhanced loop to apply the
    for (Particle p: particles) {                 force to all particles
      p.applyForce(f);
    }
  }

  void run() {
    Iterator it = particles.iterator();        $$ Can't use the enhanced loop because
    while (it.hasNext()) {                        we want to check for particles to delete
      Particle p = (Particle) it.next();
      p.run();
      if (p.isDead()) {
        it.remove();
      }
    }
  }
}
```

What if we wanted to take this example one step further and add a "Repeller" object (see inverse of Attractor object, Chapter 2, p. XXX ***[REF]***) that pushes any particles away that get close? This requires a bit more sophistication because, unlike the gravity force, each force the Repeller exerts on each particle must be custom calculated for each Particle.



Universal Gravity Force
Vectors are all identical



Attractor Force
Vectors are all different!

Let's start solving this problem by examining how we would incorporate a new Repeller object into our simple particle system plus forces example. We're going to need two major additions to our 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.

```
ParticleSystem ps;
Repeller repeller;      $$ New thing #1, we need a Repeller class

void setup() {
  size(200,200);
  smooth();
  ps = new ParticleSystem(new PVector(width/2,50));
  repeller = new Repeller(width/2-20,height/2);   $$ New thing #1: we need a Repeller class
}

void draw() {
  background(100);
  ps.addParticle();

  PVector gravity = new PVector(0,0.1);
  ps.applyForce(gravity);

  ps.applyRepeller(repeller);      $$ New thing #2: we need a function to apply a force
                                      from a repeller

  ps.run();
  repeller.display();     $$ New thing #1: we need a Repeller class
}
```

Making a Repeller object is quite easy; it's a duplicate of the Attractor class from chapter 2 (example 2.x).

```
class Repeller {
  PVector location;          $$ A Repeller doesn't move so just location and size
  float r = 10;
```

```
  Repeller(float x, float y)  {
    location = new PVector(x,y);
  }

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

The more difficult question is, how do we write the ***applyRepeller()*** function? Instead of passing a PVector into a function like we do with ***applyForce()***, we're going 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()	applyRepeller()

`void applyForce(PVector f) {`
`  for (Particle p: particles) {`
`    p.applyForce(f);`
`  }`
`}`
	`void applyRepeller(Repeller r) {`
	`  for (Particle p: particles) {`
	`    PVector force = r.repel(p);`
	`    p.applyForce(force);`
	`  }`
	`}`

The functions are almost identical. There are only two differences. One we mentioned before—a Repeller object is the argument, not a PVector. Two is the important one. We must calculate a custom PVector force for each and every Particle and apply that force. How is that force calculated? In a function called ***repel()***, which is the inverse of the ***attract()*** function we wrote for the Attractor class.

```
PVector repel(Particle p) {
  PVector dir = PVector.sub(location,p.location);
  float d = dir.mag();
  dir.normalize();
  d = constrain(d,5,100);
  float force = -1 * G / (d * d);
  dir.mult(force);
  return dir;
}
```

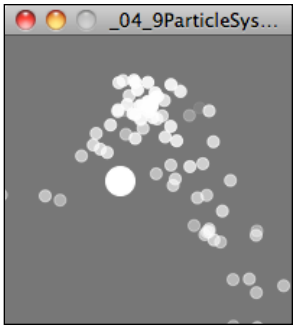
All the same steps we had to calculate an attractive force, only pointing the opposite direction.

- Get force direction
- Get distance (constrain distance)
- Calculate magnitude
- Make a vector out of direction and magnitude

Notice how throughout this entire process of adding a Repeller to the environment, we've 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 location, velocity, and acceleration, as well as have the ability to receive an external force and act on it.

So we can now look at this example in its entirety, again leaving out the Particle class, which hasn't changed.

Example 4.x: ParticleSystem with Repeller




```
ParticleSystem ps;      $$ One Particle System
Repeller repeller;      $$ One Repeller

void setup() {
  size(200,200);
  smooth();
  ps = new ParticleSystem(new PVector(width/2,50));
  repeller = new Repeller(width/2-20,height/2);
}

void draw() {
  background(100);
  ps.addParticle();

  PVector gravity = new PVector(0,0.1);        $$ We're applying a universal gravity
  ps.applyForce(gravity);

  ps.applyRepeller(repeller);                  $$ Applying the repeller

  ps.run();
  repeller.display();
}


class ParticleSystem {                         $$ The ParticleSystem manages all the Particles
  ArrayList<Particle> particles;
  PVector origin;

  ParticleSystem(PVector location) {
    origin = location.get();
    particles = new ArrayList<Particle>();
  }

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

  void applyForce(PVector f) {                 $$ Applying a force as a PVector
    for (Particle p: particles) {
      p.applyForce(f);
    }
  }

  void applyRepeller(Repeller r) {             $$ Calculating a force for each Particle
    for (Particle p: particles) {                 based on a Repeller
      PVector force = r.repel(p);
      p.applyForce(force);
    }
  }

  void run() {
    Iterator it = particles.iterator();
    while (it.hasNext()) {
      Particle p = (Particle) it.next();
      p.run();
      if (p.isDead()) {
        it.remove();
      }
    }
  }
}
```

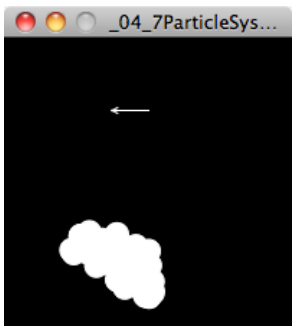
```
class Repeller {

  float strength = 100;            $$ How strong is the repeller?
  PVector location;
  float r = 10;

  Repeller(float x, float y)  {
    location = new PVector(x,y);
  }

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

  PVector repel(Particle p) {                        $$ This is the same repel algorithm we used
    PVector dir = PVector.sub(location,p.location);     in Chapter 2: forces based on
    float d = dir.mag();                                gravitational attraction.
    dir.normalize();
    d = constrain(d,5,100);
    float force = -1 * strength / (d * d);
    dir.mult(force);
    return dir;
  }
}
```

Exercise: Expand the above example to include many Repellers (using an array or ArrayList).

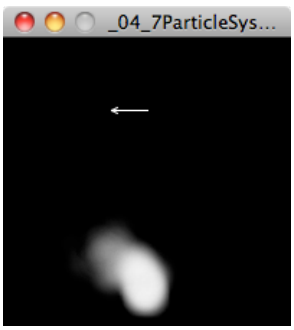
Exercise: Create a particle system in which each particle responds to every other particle. (Note we'll be doing this in detail in Chapter 6.)

4.9 Particle System with image textures / additive blending

Even though this book is really about behaviors and algorithms rather than computer graphics and design, I think we wouldn't be able to live with ourselves if we went through 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 a big part of the puzzle in terms of designing certain types of visual effects. Let's try to create a smoke simulation in Processing. Take a look at the following two images:

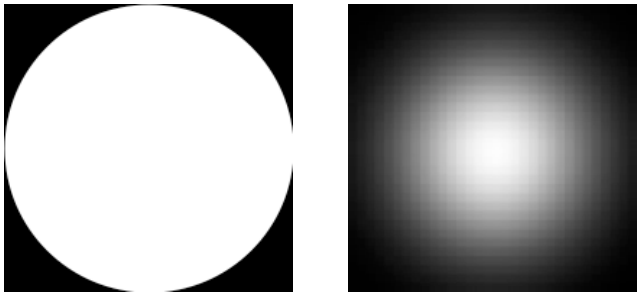


A: white circles



B: 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.



The nice 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 Processing 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 PImage object.

```
Example 4.x: Image Texture Particle System
PImage img;
```

Load the image in ***setup()***.

```
void setup() {
  img = loadImage("texture.png");        $$ Loading the PNG
}
```

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

```
  void render() {
    imageMode(CENTER);
    tint(255,lifespan);        $$ Note how tint() is the image equivalent of shape's fill()
    image(img,loc.x,loc.y);
  }
```

Incidentally, this smoke example is a nice excuse to revisit a Gaussian number distribution (see introduction, p.x ***[REF]***). To make the smoke appear a bit more realistic, we don't want to launch all the particles in a purely random direction. Instead, by creating initial velocity vectors mostly around a mean value (with a lower probability of outliers), we'll get an effect that appears less fountain-like and more like smoke (or fire).

Assuming a Random object called "generator", we could create initial velocities as follows:

```
    float vx = (float) generator.nextGaussian()*0.3;
```

```
    float vy = (float) generator.nextGaussian()*0.3 - 1.0;
    vel = new PVector(vx,vy);
```

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

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

  float dx = map(mouseX,0,width,-0.2,0.2);
  PVector wind = new PVector(dx,0);          $$ Wind force points towards mouseX
  ps.applyForce(wind);
  ps.run();
  for (int i = 0; i < 2; i++) {              $$ Two particles are added each cycle through draw
    ps.addParticle();
  }
}
```

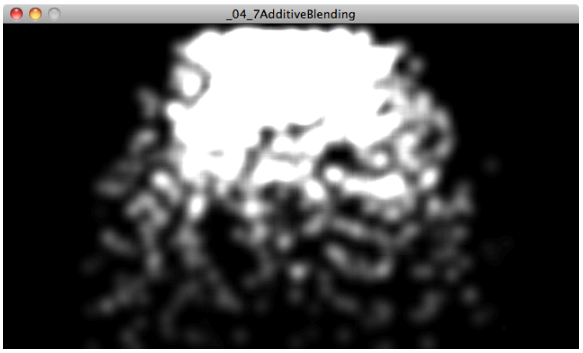
Exercise: Try creating your own textures for different types of effects. Can you make it look like fire, instead of smoke?

Exercise: 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 we draw something on top of something else in Processing, we only see the top layer—this is commonly referred to as a "normal" blend mode. When the pixels have alpha transparency (as they do in the smoke example), Processing uses an alpha compositing algorithm that combines a percentage of the background pixels with the new foreground pixels based on the alpha values.

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 was pioneered by Robert Hodgin (flight404.com) in his famous particle system and forces exploration, magnetosphere (which later became the iTunes visualizer). (For more see: http://roberthodgin.com/magnetosphere-part-2/)

Additive blending is in fact one of the simplest blend algorithms and involves adding the pixel values of one layer with 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.



To achieve additive blending in Processing, you'll need to work in OPENGL mode, i.e.

Example 4.x: Additive Blending

```
import processing.opengl.*;
import javax.media.opengl.*;       $$ You also need to import additional opengl libraries

void setup() {
  size(200,200,OPENGL);
}
```

You'll also need to set some OpenGL settings that are part of the JOGL API (the openGL engine that Processing uses). To do this, you need access to the "GL" object, i.e. the main renderer.

```
PGraphicsOpenGL pgl;
GL gl;

void setup() {
  size(200,200,OPENGL);
  pgl = (PGraphicsOpenGL) g;       $$ Accessing the Processing "GL" object
  gl = pgl.gl;
}
```

Then, before you go to draw anything, you can say:

```
void draw() {

  pgl.beginGL();
  gl.glDisable(GL.GL_DEPTH_TEST);        $$ only necessary if objects actually move in 3D space
  gl.glEnable(GL.GL_BLEND);                   $$ Enables blending
  gl.glBlendFunc(GL.GL_SRC_ALPHA,GL.GL_ONE);    $$ Sets the blend mode to additive


  pgl.endGL();

  background(0);             $$ Note the "glowing" effect of additive blending will not work
                                with a white (or very bright background)

  $$ All your other particle stuff would go here

  pgl.endGL();

}
```

(Thanks again to Robert Hodgin for writing a tutorial about this in March 2007: http://www.flight404.com/blog/?p=71)

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

The Eco-System 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 we tracked a particle's lifespan, removing them when appropriate? What rules can you incorporate to control how creatures are born?

(Also, you might consider using a Particle System itself in the design of a creature. What happens if your emitter is tied to the creature's location?)