Nature of Code

Patrick Dwyer
Fall 2005
Week 5 - October 4th

Particle Systems

A particle system is a collection of independent objects, often represented by a simple shape or dot. It can be used to model many irregular types of natural phenomena, such as explosions, fire, smoke, sparks, waterfalls, clouds, fog, petals, grass, bubbles, and so on. In a system, each particle will have its own set of properties related to its behavior (for example, velocity, acceleration, etc.) as well as its look (for example, color, shape, image, etc.). Particle Systems make it very easy to define the behavior of an object once, and manage many instances of that object to create complex motion and simulations.

We are going to look at strategies for coding a particle system. How do we organize our code? Where do we store information related to individual particles versus 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 lots of different effects.

The first thing we need to examine is the particle object. We need to create a class that defines the basic elements of our particle and the properties it needs to simulate motion.

```
class Particle {

    /*
        The way each of our particles work is
        familiar from the basic forces;
        location, velocity and acceleration.
    */
    Vector3D loc;
    Vector3D vel;
    Vector3D accel;

    // Keep track of the size of our particle, and how long it's been
    // active
    float r;
    float timer;
      ....
```

This is nothing really new, just some variables to keep track of location, velocity, acceleration, and size. This should remind you of our work so far with basic forces and motion. However, we are adding an important element, a timer to keep track of the particle's life. Our particles, sadly, will not live forever.

```
//function to update location
void simulate() {
```

```
        vel.add(acc);
        loc.add(vel);
        timer -= 1.0;
    }
```

We've modified our `simulate` method so that the timer counts down each cycle, as well as handling our motion. The amount the `timer` counts at each step is hard coded here, and would probably be better as a property of our particle, so we could better control how long each individual particle lives.

We also need to create a method for testing our particle to know if it is dead or alive. This method will check the timer property, and return TRUE for a dead particle if the `timer` has gone below zero.

```
    boolean isDead() {

        if (timer <= 0.0) {
            return true
        } else {
            return false;
        }
    }
```

The remainder of the Particle class is similar to the other classes we've written; constructors, render method, etc. We could easily take this Particle class as is, and create an array of Particle objects, using a for loop to work with each one in our draw method. This has many limitations that we can address by going a step further, and writing a ParticleSystem class to control our particles for us; this will remove the need for us to manipulate the individual particles, freeing us to create more interesting behavior with the system as a whole.

Before we write our ParticleSystem class we need to investigate an alternative to our basic Arrays; a variable length collection called ArrayList.

Resizable Arrays with ArrayList

Most of our examples up until now (with the exception of a few) have used standard java arrays to keep track of ordered lists of information. We might have an array of 10 objects, looping through them each cycle to update locations, render them, etc. However, in the case of a standard array we are limited to the number of objects we initially created the array to hold; if we instantiated a 10 item array, it can only ever hold 10 items. There are certainly alternatives (using a very large array and having a separate variable to keep track of how much of the array we should use at any given time), but it would be much more useful if we could dynamically size the array at run-time. In the case of a particle system, this will really help.

To accomplish our goal of having a resizable array, we will use the java class ArrayList, which can be found in the java.util package.

The reference page is here: http://java.sun.com/j2se/1.4.2/docs/api/java/util/ArrayList.html.

Remember, this class comes from our Java libraries and is not part of the processing reference. The ArrayList is immediately available for us to use in Processing, but in order to know how to use an ArrayList object, we must consult the java API.

Using an ArrayList is conceptually similar to a standard array, but the syntax will be quite different. Here is some code (that assumes the existance of a class "Particle") demonstrating the same functionality, first with an array, and second with an ArrayList.

```
int MAX = 10;

//declaring the array
Particle[] parray = new Particle[MAX];

//declaring the arraylist
ArrayList plist = new ArrayList();

//the following code you would usually find in setup
for (int i = 0; i < parray.length; i++) {
  parray[i] = new Particle();
}

for (int i = 0; i < MAX; i++) {
  plist.add(new Particle());
}

//the following code you would usually find in draw
for (int i = 0; i < parray.length; i++) {
  Particle p = parray[i];
  p.run();
  p.render();
}

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

Note that in this last for loop, we have to make sure to cast the object we pull out of the ArrayList. The ArrayList doesn't keep track of the type for things stored inside -- it's our job to remind it!

Particle System Class

The main component of our ParticleSystem class will be an ArrayList to contain all of the particles in the system. We will still need some additional properties to track the progress and state of the system, such as the origin of the particle system, an image texture to draw for each particle, etc.

```
class ParticleSystem {

    // The list of particles
```

```
    ArrayList particles;

    // The source of the particle system
    Vector3D origin;

    // The image we're using to draw our system
    PImage img;

    ParticleSystem(int num, Vector3D origin_, PImage img_) {

        // Set the particles array
        particles = new ArrayList();

        // Save the center of the simulation
        origin = origin_.copy();

        // save the rendering image
        img = img_;

        // Create all of the starting particles
        for (int i = 0; i < num; i++) {
            particles.add(new Particle(origin, img));
        }
    }
```

The next step is to write a method that calls methods on all the particles in the system. As we've seen from how an ArrayList works, this is fairly simple:

```
void simulate() {
        for (int i = 0; i < particles.size(); i++) {

            // we need to cast the particle into its type
            Particle p = (Particle)particles.get(i);

            // Run the simulation
            p.simulate();
        }
}
```

However, while we cycle through each particle, we want to check and make sure the particle is still alive; if it is not, we should remove it from the ArrayList. There is a problem here -- when an element is removed at a specified position in this list, any subsequent elements are shifted to the left (i.e. one is subtracted from their indices). This will result in skipping elements as they are deleted (if item N is deleted, item N+1 becomes item N and is not checked since the loop has already checked item N!) This is easily, solved, however, by going through the ArrayList backwards.

```
void simulate() {
    for (int i = particles.size() - 1; i >= 0; i--) {

        // we need to cast the particle into its type
        Particle p = (Particle)particles.get(i);

        // Run the simulation
        p.simulate();

        // Remove any dead particles
        if (p.isDead()) {
            particles.remove(i);
        }
    }
}
```

Finally, we can implement additional functionality to our system, such as methods that will birth new particles and a method that will test if the entire system itself is dead:

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

void addParticle(Particle p) {
      particles.add(p);
}

boolean isDead() {
      if (particles.isEmpty()) {
            return true;
      } else {
            return false;
      }
}
```

Once we have finished implementing the particle class and the particle collection class, our main program code is nice and elegant. We only have to declare a `ParticleSystem` as a global variable, call the constructor in `setup()` to instantiate it, and then call the run function in `draw()` (as well as choose to call `addParticle()` whenever new particles should be created.)

```
ParticleSystem ps;

void setup() {
    size(400, 400);
    colorMode(RGB, 255, 255, 255, 100);
```

```
    smooth();

    ps = new ParticleSystem(1, new Vector3D(width / 2, height / 2, 0));
}

void draw() {

    background(0);
    ps.simulate();
    ps.addParticle();
}
```

Inheritance

In the case of a particle system, we will often want to have systems containing different types of particles. In order to accomplish this, we would like to avoid writing a new class for every single particle. A better solution would be to create "subclasses" of our master particle class that could use the exiting data and functionality of a regular particle, adding other features as necessary.

Object oriented programming allows us define classes in terms of other classes. In other words, a class can be a subclass (aka "child") of a super class (aka "parent"). This concept is known as "inheritance."

Take this very typical example, where we have a class containing a few instance variables, a constructor to fill them, and a method that increments the x and y variables randomly.

```
class Shape {
      float x;
      float y;
      float s;

      Shape(float x_, float y_, float s_) {
            x = x_;
            y = y_;
            s = s_;
      }

      void shake() {
            x += random(-1,1);
            y += random(-1,1);
      }
}
```

Now what if we create a subclass from `Shape` (let's call it `Square`). It will inherit all the instance variables and methods from shape. We write a new constructor with the name `Square`, however, here we are executing the code from the parent class by calling `super`.

```
class Square extends Shape {
```

```
	//inherits all instance variables from parent
	//we could add variables for only Square here if we so

	Square(float x_, float y_, float s_) {
		super(x_,y_,s_);
	}

	//inherits shake method from parent

	//adds a new render method
	void render() {
		rectMode(CENTER);
		fill(255);
		noStroke();
		rect(x,y,s,s);
	}
}
```

Here is another subclass with some additional functionality. It adds an instance variable to keep track of color (this is just to show how this is possible, most likely we would want the super class to include color). It also calls the parent `shake` method (with `super`), but adds some additional code.

```
class Circle extends Shape {

	//inherits all instance variables from parent + adding one
	color c;

	Circle(float x_, float y_, float s_, color c_) {
		 // call the parent constructor
		super(x_,y_,s_);
		c = c_;
	}

	//call the parent jiggle, but do some more stuff too
	void shake() {
		super.shake();
		s += random(-1,1);
		s = constrain(r,0,100);
	}

	// adds a new render method
	void render() {
		ellipseMode(CENTER);
		fill(c);
```

```
            noStroke();
            ellipse(x,y,s,s);
      }
}
```

Polymorphism

Polymorphism (i.e. many forms) refers to the concept that we can treat an object instance in multiple ways. A Circle is a `Circle`, but it is also a `Shape` so we can refer to it as either.

```
Shape c1 = new Circle(100,100,20,color(255));
Circle c2 = new Circle(100,100,20,color(255));
```

Both of the above lines of code are legal. Even though we declare `c1` as a `Shape`, we're really making a `Circle` object and storing it in the `c1` reference. (We can safely call all the Shape methods on `c1` b/c the rules of inheritance dictate that a `Circle` can do anything a `Shape` can). At run-time, however, java will determine that this object really truly is a `Circle` and run the proper methods. This becomes particularly useful when we have an array. Here we can make an array of `Shapes`, put both `Circles` and `Squares` into the array, but not have to worry about which are which -- that will all be taken care of for us!!

```
Shape[] s = new Shape[25];
for (int i = 0; i < s.length; i++) {
      int r = int(random(2));

      //randomly put either circles or squares in our array
      if (r == 0) {
            s[i] = new Circle(100,100,10,color(255,0,0));
      } else {
            s[i] = new Square(100,100,10);
      }
}
```

Later, we can run through the array with a for loop. Again, even though some of the elements are circles and some are squares, we don't have to specify in our code since we can treat them all in the general form as "shapes".

```
for (int i = 0; i < s.length; i++) {
      Shape ashape = s[i];
      ashape.jiggle();
      ashape.render();
}
```

For a much better explanation of polymorphism, please visit:
http://www.javaranch.com/campfire/StoryPoly.jsp

Assignment

This week we continue the Two Week assignment from last week. Try and incorporate the techniques and methods of using particle systems into your work from last week; extend the work you've done using Polymorphism, Inheritance or Particle Systems.

Some ideas:

- Treat some part of your project as a particle and create a particle system around it. Particle Systems are an excellent way to create groups and collections of objects with individual behaviors.
- Use Inheritance and Polymorphism to alter some part of your code; create a few variations on an object, each with unique behaviors or properties, that can all be treated as their parent object. If you're drawing objects on screen use Polymorphism to create different shapes or images that are drawn to screen by different objects.
- How might you create a Particle System of Particle Systems? Note that the particles that make up a particle system don't have to behave only like we've created them here; they can do anything you want.