flock.rb

Six Silberman

August 25, 2012

Abstract

`flock.rb` is a very simple flocking simulation written in the Ruby programming language. This document explains how it works.

This document has two parts. The first explains flocking. The second explains the code in `flock.rb` in the order it appears.

Obtaining

`flock.rb` should be available at `http://wtf.tw/etc/flock.rb`.

1 Flocking

Flocking describes the behavior of a group of birds, fish, or insects moving together. The simplest simulation of flocking involves any number of agents (the birds, fish, or insects) that begin in random positions but are otherwise identical. Three instructions govern their movement:

- **Separate**. Move away from other agents that move too close.
- **Align**. Move toward the average heading of nearby agents.
- **Cohere**. Move toward the average position of nearby agents.

One way to do this is to keep a list of agents. Suppose there are n agents. We can call the list F, for "flock," and denote a single agent by F_i, where i is an integer between 1 and n.

At any time t each agent has a position $\mathbf{p}$ and a velocity $\mathbf{v}$. These are written in boldface because they are vectors, i.e., they have an x and a y component.

As this is a simulation in a computer, we represent time discretely rather than continuously. So at some "time step" t, the position $\mathbf{p}_t$ of any agent is:

$$\mathbf{p}_t = \mathbf{p}_{t-1} + \mathbf{v}_{t-1}$$

Equivalently, the position of agent i at time t is:

$$\mathbf{p}_{i,t} = \mathbf{p}_{i,t-1} + \mathbf{v}_{i,t-1} \tag{1}$$

where we have assumed that the unit of $\mathbf{v}$ is distance per time step.

At each time step we also have

$$\mathbf{v}_{i,t} = \mathbf{v}_{i,i-1} + \mathbf{dv}_{i,t} \tag{2}$$

where $\mathbf{dv}$ denotes the change in velocity of the agent.

The flocking behavior appears through the computation of the value of $\mathbf{dv}$. Specifically, suppose we have three functions, separate, align, and cohere, that return vectors. Then

$$\mathbf{dv}_{i,t} = \mathsf{separate}_{i,t} + \mathsf{align}_{i,t} + \mathsf{cohere}_{i,t} \tag{3}$$

1.1 separate

The instruction for separate is: move away from other agents that move too close.

We must specify how close is too close. Suppose "too close" is the same for all agents, so we require only a single variable to specify it. We can call this distance r. Then

$$\mathsf{separate}_i = \sum_{\substack{i \neq j \\ ||\mathbf{p}_i - \mathbf{p}_j|| \leq r}} \mathbf{p}_i - \mathbf{p}_j \tag{4}$$

That is, for some agent i, separate is the sum of vectors pointing away from all other agents j that are inside the distance r.

1.2 align

The instruction for align is: move toward the average heading of nearby agents.

We must specify what counts as nearby. Suppose "nearby" is the same for all agents and that we denote this distance R. Then

$$\mathsf{align}_i = \sum_{\substack{i \neq j \\ ||\mathbf{p}_i - \mathbf{p}_j|| \leq R}} \mathbf{v}_j \tag{5}$$

That is, for some agent i, align is the sum of the velocity vectors of all other agents j inside the distance R.

1.3 cohere

The instruction for cohere is: move toward the average position of nearby agents.

We should use the same interpretation of "nearby" as align. So:

$$\mathsf{cohere}_i = \sum_{\substack{i \neq j \\ ||\mathbf{p}_i - \mathbf{p}_j|| \leq R}} \mathbf{p}_j \tag{6}$$

That is, for some agent i, cohere is the sum of the position vectors of all other agents j inside the distance R.

1.4 The flocking time step

Combining Eqs. 1 - 6, we have

$$\mathbf{p}_{i,t} = \mathbf{p}_{i,t-1} + \mathbf{v}_{i,t} \tag{7}$$

$$\mathbf{v}_{i,t} = \mathbf{v}_{i,t-1} + \sum_{\substack{i \neq j \\ ||\mathbf{p}_{i,t-1} - \mathbf{p}_{j,t-1}|| \leq r}} \mathbf{p}_{i,t-1} - \mathbf{p}_{j,t-1} + \sum_{\substack{i \neq j \\ ||\mathbf{p}_{i,t-1} - \mathbf{p}_{j,t-1}|| \leq R}} \mathbf{p}_{j,t-1} + \mathbf{v}_{j,t-1} \tag{8}$$

2 flock.rb

flock.rb includes additions to the built-in Ruby class Array and two new classes, Vector and VectorArray. The flocking time step is computed in the Sketch class, which extends Processing::App. Processing::App is provided by the ruby-processing gem, which flock.rb uses to visualize the flocking behavior.

Flocking can easily be simulated in the Ruby programming language with an object-oriented approach. flock.rb avoids object orientation to provide a clear mapping from the representation in Eqs. 7 - 8 to working code.

2.1 Additions to Array

flock.rb adds two methods to the built-in Ruby class Array: sum and to_va. sum computes the sum of an array of integers. to_va converts an array of Vector objects to a VectorArray object.

2.2 Vector

The Vector class provides methods for manipulating 2-dimensional vectors. Specifically, it includes methods for:

- adding and subtracting vectors
- multiplying a vector by a scalar
- getting the magnitude of a vector
- getting a unit vector in the same direction as a vector
- constraining a vector's components within a range
- constraining a vector's magnitude within a range
- computing the distance between vectors
- generating a random vector within a range

It also provides a convenience method for generating a zero vector.

2.3 `VectorArray`

The `VectorArray` class is a wrapper for the built-in Ruby `Array` class. It provides three methods for manipulating arrays of `Vector` objects: `x`, `y`, and `sum`. `x` returns an array of the x-components of the elements in the `VectorArray`; `y` does the same for the y-components. `sum` returns the vector sum of the elements.

2.4 `Sketch`

The `Sketch` class does the computation to model the flocking behavior and uses the methods provided by `ruby-processing`, via `Processing::Proxy`, to visualize it.

2.5 Constants

`Sketch` includes the following constants:

- `N`, the number of agents (n)
- `L`, the distance at which agents separate (r)
- `B`, the distance within which agents align and cohere (R)
- `W`, the width of the visualization in pixels
- `H`, the height of the visualization in pixels
- `Vmx`, the upper bound of the range from which to draw random values for velocity components

2.6 `setup`

The `setup` method creates the arrays of positions and velocities.

2.7 `step`

The `step` method performs the computation described in Eqns. 7 - 8.

First, it updates the positions. Eqn. 7 is implemented as:

```
@p = @p.map{|pi| (pi + @v[@p.index(pi)]).wrap(W, H)}
```

First, we consider

```
@p = @p.map{|pi| pi + @v[@p.index(pi)]}
```

`@p` is the array of position vectors. `map` iterates over `@p`, and for each element `pi` it returns

```
pi + @v[@p.index(pi)]
```

where if `pi` is element $\mathbf{p}_i$, `@v[@p.index(pi)]` is $\mathbf{v}_i$.

The call to `wrap` wraps the positions, so when agents leave the visualization off the bottom or right edge they are returned to the top or left.

Next, `step` updates the velocities. Eqn. 8 is implemented as nine lines of Ruby code.

First, we use `map` to iterate through `@v`, the array of velocity vectors. For each element `vi` we:

- set `i` to the element's index
- compute the change in velocity due to separation
- compute the change in velocity due to alignment and cohesion
- add the changes in velocity to the velocity vector and bound the result so it is no larger than `Vmx`

separate is computed as:

```
sep = @p.map{|pj| j = @p.index(pj)
                  d = @p[i] - pj
                  (i != j && d.mag <= L) ? d : Vector.zero
      }.to_va.sum
```

First we assign the index of the current position vector to `j`. Then we compute the vector leading away from agent j from the perspective of agent i and assign it to `d`. If i is not j and the magnitude of `d` is less than `L`—i.e., if agent j is "near" agent i—we return `d`; otherwise we return a zero vector.

That is,

```
(i != j && d.mag <= L) ? d : Vector.zero
```

is roughly equivalent to

```
if (i != j) and (d.mag <= L)
    return d
else
    return Vector.zero
end
```

We use `map` to collect these vectors, then convert the array returned by `map` into a `VectorArray` and use `VectorArray`'s `sum` method to obtain a vector sum.

align and cohere are computed (together) in the same way.

The final line of `step` adds the results of the two computations.

2.8 `draw`

The `draw` method visualizes the current state of the simulation. It is called at every time step. It redraws the backgroud, calls `step`, sets the fill color, and draws each agent.