Boids Simulation

Mathematical Modeling

Brajmohan Meena (2016CS10376)
Sai K Reddy (2016CS10330)

Introduction:

Boids is an artificial life program, developed by *Craig Reynolds* in 1986, which simulates the flocking behavior of birds. His paper on this topic was published in 1987 in the proceedings of the ACM SIGGRAPH conference. The name "boid" corresponds to a shortened version of "bird-oid object", which refers to a bird-like object.

As with most artificial life simulations, Boids is an example of emergent behavior; that is, the complexity of Boids arises from the interaction of individual agents (the boids, in this case) adhering to a set of simple rules.

The rules applied in the simplest Boids world are as follows:

- **Separation**: steer to avoid crowding local flockmates
- **Alignment**: steer towards the average heading of local flockmates
- **Cohesion**: steer to move toward the average position (center of mass) of local flockmate

Boids:

The boid is the bird representation in Reynolds flocking simulation model. Each boid object should at least have the following attributes to describe the state it is in.

Location The *x* and *y* coordinates of the current position of the boid.
Course The angle of the current course of the boid.
Velocity The speed of which the boid is traveling.

The course and speed could of course be represented by a equivalent velocity vector instead. But often more attributes is needed to make the simulation more convincing. Putting an upper limit on how fast the boids can move and turn is a common improvement.

Rules:

A boid keeps on moving by watching his visible neighbors and deciding what direction to take next. Each neighbor influences this direction in different conflicting manners, depending on its type and distance from the simulated boid. These three rules are applied in the simplest Boids.

1. **Separation:**
   - Pushes boids apart to keep them from crashing into each other by maintaining distance from nearby flock mates.
   - Each boid considers its distance to other flock mates in its neighborhood and applies a repulsive force in the opposite direction, scaled by the inverse of the distance

   The implementation of separation is very similar to that of alignment and cohesion, so I'll only point out what is different. When a neighboring agent is found, the distance from the agent to the neighbor is added to the computation vector.

   ```
   v.x += agent.x - myAgent.x;
   v.y += agent.y - myAgent.y
   ```

   The computation vector is divided by the corresponding neighbor count, but before normalizing, there is one more crucial step involved. The computed vector needs to be negated in order for the agent to steer away from its neighbors properly.

   ```
   v.x *= -1;
   v.y *= -1;
   ```

2. **Cohesion:**
   - Cohesion is the rule that keeps the flock together, without it there would not be any flocking at all.
   - Each boid moves in the direction of the average position of its neighbors.
   - Compute the direction to the average position of local flock mates and steer in that direction.

   In this the *position* of boid is added.

   ```
   v.x += agent.x;
   v.y += agent.y;
   ```

   The computation vector is divided by the neighbor count, resulting in the position that corresponds to the center of mass. However, we don't want the center of mass itself, we want the direction *towards* the center of mass, so we recompute the vector as the distance from the agent to the center of mass. Finally, this value is normalized and returned.

```
v.x /= neighborCount;
v.y /= neighborCount;
v = new Point(v.x - myAgent.x, v.y - myAgent.y);
v.normalize(1);
return v;
```

3. Alignment:

- Drives boids to head in the same direction with similar velocities (velocity matching).
- Calculate average velocity of flock mates in neighborhood and steer towards that velocity.

We'll need two variables: one for storing the vector we'll compute, and another for keeping track of the number of neighbors of the agent.

```
var v:Point = new Point();
var neighborCount = 0;
```

With our variables initialized, we now iterate through all of the agents and find the ones within the *neighbor radius* - that is, those close enough to be considered neighbors of the specified agent. If an agent is found within the radius, its velocity is added to the computation vector, and the neighbor count is incremented.

If no neighbors were found, we simply return the zero vector.

Finally, we divide the computation vector by the neighbor count and normalize it (divide it by its length to get a vector of length), obtaining the final resultant vector.

```
v.x /= neighborCount;
v.y /= neighborCount;
v.normalize(1);
return v;
```