Module 4 Lab – Agent-based models

Boids

In 1987 Craig Reynolds published "Flocks, herds and schools: A distributed behavioral model", which describes an agent-based model of herd behavior. Agents in this model are called "Boids", which is both a contraction of "birdoid" and an accented pronunciation of "bird" (although Boids are also used to model fish and herding land animals).

Each agent simulates three behaviors:

Flock centering: Move toward the center of the flock.

Collision avoidance: Avoid obstacles, including other Boids.

Velocity matching: Align velocity (speed and direction) with neighboring Boids.

Boids make decisions based on local information only; each Boid only sees (or pays attention to) other Boids in its field of vision.

The Boid algorithm

The algorithm defines two classes: `Boid`, which implements the `Boid` behaviors, and `World`, which contains a list of Boids and a "carrot" the Boids are attracted to.

The `Boid` class defines the following methods:

- `center`: Finds other Boids within range and computes a vector toward their centroid.
- `avoid`: Finds objects, including other Boids, within a given range, and computes a vector that points away from their centroid.
- `align`: Finds other Boids within range and computes the average of their headings.
- `love`: Computes a vector that points toward the carrot.

Here's the implementation of `center()`:

```
def center(self, boids, radius=1, angle=1):
      neighbors = self.get_neighbors(boids, radius, angle)
      vecs = [boid.pos for boid in neighbors]
      return self.vector_toward_center(vecs)
```

The parameters radius and angle are the radius and angle of the field of view, which determines which other Boids are taken into consideration. radius is in arbitrary units of length; angle is in radians.

`center` uses `get_neighbors()` to get a list of `Boid` objects that are in the field of view. `vecs` is a list of Vector objects that represent their positions.

Finally, `vector_toward_center()` computes a Vector that points from self to the centroid of neighbors.

Here's how `get_neighbors()` works:

```
def get_neighbors(self, boids, radius, angle):
      neighbors = []
      for boid in boids:
            if boid is self:
                continue

            # if not in range, skip it
            offset = boid.pos - self.pos
            if offset.mag > radius:
                continue

            # if not within viewing angle, skip it
            if self.vel.diff_angle(offset) > angle:
                continue

            # otherwise add it to the list
            neighbors.append(boid)

      return neighbors
```

For each other `Boid`, `get_neighbors()` uses vector subtraction to compute the vector from `self` to `boid`. The magnitude of this vector is the distance between them; if this magnitude exceeds `radius`, we ignore `boid`.

`diff_angle()` computes the angle between the velocity of `self`, which points in the direction the Boid is heading, and the position of boid. If this angle exceeds `angle`, we ignore `boid`. Otherwise `boid` is in view, so we add it to `neighbors`.

Now here's the implementation of `vector_toward_center()`, which computes a vector from `self` to the centroid of its `neighbors`.

```
def vector_toward_center(self, vecs):
      if vecs:
          center = np.mean(vecs)
          toward = vector(center - self.pos)
          return limit_vector(toward)
      else:
          return null_vector
```

`VPython` vectors work with `NumPy`, so `np.mean()` computes the mean of `vecs`, which is a sequence of vectors. `limit_vector` limits the magnitude of the result to 1; `null_vector` has magnitude 0.

We use the same helper methods to implement `avoid()`:

```
def avoid(self, boids, carrot, radius=0.3, angle=np.pi):
      objects = boids + [carrot]
      neighbors = self.get_neighbors(objects, radius, angle)
      vecs = [boid.pos for boid in neighbors]
      return -self.vector_toward_center(vecs)
```

`avoid()` is similar to `center()`, but it takes into account the carrot as well as the other Boids. Also, the parameters are dfferent: `radius` is smaller, so Boids only avoid objects that are too close, and angle is wider, so Boids avoid objects from all directions. Finally, the result from `vector_toward_center()` is negated, so it points away from the centroid of any objects that are too close.

Here's the implementation of align:

```
def align(self, boids, radius=0.5, angle=1):
      neighbors = self.get_neighbors(boids, radius, angle)
      vecs = [boid.vel for boid in neighbors]
      return self.vector_toward_center(vecs)
```

`align()` is also similar to `center()`; the big difference is that it computes the average of the neighbors' velocities, rather than their positions. If the neighbors point in a particular direction, the Boid tends to steer toward that direction.

Finally, `love()` computes a vector that points in the direction of the carrot.

```
def love(self, carrot):
      toward = carrot.pos - self.pos
      return limit_vector(toward)
```

The results from `center()`, `avoid()`, `align()`, and `love()` are what Reynolds calls "acceleration requests", where each request is intended to achieve a different goal.

Arbitration

To arbitrate among these possibly conflicting goals, we compute a weighted sum of the four requests:

```
def set_goal(self, boids, carrot):
      w_avoid = 10
      w_center = 3
      w_align = 1
      w_love = 10

      self.goal = (w_center * self.center(boids) + w_avoid *
      self.avoid(boids, carrot) +
      w_align * self.align(boids) + w_love * self.love(carrot))

      self.goal.mag = 1
```

`w_center()`, `w_avoid()`, and the other weights determine the importance of the acceleration requests. Usually `w_avoid()` is relatively high and `w_align()` is relatively low.

After computing a goal for each Boid, we update their velocity and position:

```
def move(self, mu=0.1, dt=0.1):
      self.vel = (1-mu) * self.vel + mu * self.goal
      self.vel.mag = 1
      self.pos += dt * self.vel
      self.axis = self.length * self.vel
```

The new velocity is the weighted sum of the old velocity and the goal. The parameter `mu` determines how quickly the birds can change speed and direction. Then we normalize velocity so its magnitude is 1, and orient the axis of the Boid to point in the direction it is moving.

To update position, we multiply velocity by the time step, `dt`, to get the change in position. Finally, we update `axis` so the orientation of the Boid when it is drawn is aligned with its velocity.

Many parameters influence flock behavior, including the radius, angle and weight for each behavior, as well as maneuverability, `mu`. These parameters determine the ability of the Boids to form and maintain a flock, and the patterns of motion and organization within the flock. For some settings, the Boids resemble a flock of birds; other settings resemble a school of fish or a cloud of flying insects.

Problem 1. To run Boids7.py, you will need VPython, a library for 3-D graphics and animation. Using Anaconda, run the following in a terminal or Command Window:

```
conda install -c vpython vpython
```

Then run Boids7.py. It should either launch a browser or create a window in a running browser, and create an animated display showing Boids, as white cones, circling a red sphere, which is the carrot. If you click and move the mouse, you can move the carrot and see how the Boids react.

Add interactivity by allowing the user to modify the parameters for the weighted sums. This is to allow customized simulation of different possible Boid behavior.

Problem 2. To generate more bird-like behavior, Flake suggests adding a behavior to maintain a clear line of sight; in other words, if there is another bird directly ahead, the Boid should move away laterally. What effect do you expect this rule to have on the behavior of the flock? Implement it and see.