Nature of Code

Patrick Dwyer
Fall 2005
Week 8 - October 25th

Individual Autonomous Agent Behavior

In the late 90s, Craig Reynolds developed algorithmic steering behaviors for animated characters. These behaviors allowed individual elements to navigate their digital environments in a "life-like" manner with strategies for seeking, fleeing, wandering, arriving, pursuing, evading, path following, obstacle avoiding, and responding to their environment. Used in the case of a single autonomous agent, these behaviors are fairly simple to understand and implement. By building a system of multiple characters, each steering according to simple locally-based rules, surprising levels of complexity emerge, the most famous example being Reynolds' ***boids*** model for *flocking* and *swarming* behavior.

We can't get anywhere with our simulation without first understanding the concept of a steering vector. In the above examples, we have a `Boid` class which extends our earlier `Particle` class. This new object has additional variables, such as a maximum speed and maximum steering force. It also has a method to compute a steering vector towards a given target location.

```
Steering Vector = Desired Vector minus Velocity
```

Where *desired vector* is defined as the vector pointing from the object's location directly towards the target.

Our `steer` method receives a location vector (`target`) and returns a force vector (`steer`):

```
Vector3D steer(Vector3D target) {
      Vector3D steer;
      Vector3D desired = Vector3D.sub(target,loc);
      float d = desired.magnitude();

      if (d > 0) {
            desired.normalize();
            desired.mult(maxspeed);
            steer = Vector3D.sub(desired,vel);
            steer.limit(maxforce);
      } else {
            steer = new Vector3D(0,0);
      }
      return steer;
}
```

© 2005, Patrick Dwyer - Creative Commons Attribution-NonCommercial-ShareAlike

Seeking

With the `steer` method figured out, we can now have a Boid object seek a given location. We calculate the steering vector and store it in the object's acceleration.

```
void seek(Vector3D target) {
      accel = steer(target,false);
}
```

Reynolds' seeking example: http://www.red3d.com/cwr/steer/SeekFlee.html.

In our examples, however, we accumulate various steering forces (instead of simply setting acceleration equal to one given steering vector.) Remember, when we do this, we must reset acceleration to a zero vector at the end of each cycle, otherwise our acceleration spins out of control.

```
void update() {
      vel = vel.add(accel);
      vel.limit(maxspeed);
      loc = loc.add(vel);
      accel.setXYZ(0,0,0);
}


void seek(Vector3D target) {
      accel = accel.add(steer(target,false));
}
```

Arrival

Arrival can be achieved in a nearly identical fashion as seeking, only instead of pursuing a target at maximum velocity, the object slows down as it approaches the destination. One solution for implementing this behavior is to modify the steering vector calculation as follows (note in the above examples for this week, the steering method receives a true or false flag to indicate whether it should apply the distance based damping or not). Here, the magnitude of the *desired* vector shrinks as the object approaches the destination. The full method with comments is available in the `Boid.pde` source file.

For a full explanation of "arrival", visit: http://www.red3d.com/cwr/steer/Arrival.html.

```
Vector3D steer(Vector3D target, boolean slowdown) {
      Vector3D steer;
      Vector3D desired = Vector3D.sub(target,loc);
      float d = desired.magnitude();
      if (d > 0) {
            desired.normalize();
            if ( (slowdown) && (d < 100.0f) ) {
                  desired = desired.mult(maxspeed*(d/100.0f));
            } else {
                  desired = desired.mult(maxspeed);
            }
            steer = Vector3D.sub(desired,vel);
            steer.limit(maxforce);  //limit to maximum steering force
```

© 2005, Patrick Dwyer - Creative Commons Attribution-NonCommercial-ShareAlike

```
      } else {
            steer = new Vector3D(0,0);
      }
      return steer;
}
```

Wandering

Reynolds' method for wandering is a bit more complex. It involves steering towards a random point on a circle projected at a given length in front of the the object. Normally, we think of wandering as applying a random steering vector each frame of animation. Reynolds solution is more sophisticated as it implements an ordered wandering where the steering at one moment is related to the previous one (note the conceptual similarity here to what Perlin noise achieves for us).

For a full explanation, visit: http://www.red3d.com/cwr/steer/Wander.html.

Here is our implementation of the wander algorithm:

```
    public void wander() {
        // Radius of the circle
        float wanderRadius = 40.0f;

        // Distance the cirlce is offset from the boid
        float wanderDistance = 80.0f;

        // Range in which our wanderTheta can change each time
        float wanderChange = 0.5f;

        // Adjust our wanderTheta
        wanderTheta += random(-wanderChange, wanderChange);

        // start with a copy of the boid velocity
        Vector3D circleLoc = vel.copy();

        // normalize and scale to the wanderDistance offset
        circleLoc.normalize();
        circleLoc = circleLoc.multiply(wanderDistance);

        // position it relative to the boid location
        circleLoc.add(loc);

        // Figure out the wander circle heading
        float actualTheta = wanderTheta + vel.heading2D();

        // Determine the vector showing the offset of the point on the circle
        Vector3D circleOffset = new Vector3D(wanderRadius * cos(actualTheta),
            wanderRadius * sin(actualTheta), 0);
```

© 2005, Patrick Dwyer - Creative Commons Attribution-NonCommercial-ShareAlike

```
        // figure out the target position of the wander point
        Vector3D target = Vector3D.add(circleLoc, circleOffset);

        // get our acceleration towards our wander point
        accel = accel.add(steer(target, false));
    }
```

Group Behavior

Once we've mastered control over a single object navigating its environment, we can begin to experiment with a group of autonomous agents, each steering according to the relative positions and velocities of its neighbors. Our example will be Reynolds' rules for flocking Boids, implementing the following three rules:

- **Separation**: Determine if you are too close to any other boids. If you are then adjust your direction to avoid collision and clumping.
- **Alignment**:. Take the average of all the other boids' velocities and adjust your velocity to move in the general direction of the flock.
- **Cohesion**: Compute the center of the entire flock and steer towards the center.

To implement this we can use our ParticleSystem classes as a base, but restructure slightly. In this case, each `boid` not only needs to know about itself individually, but it needs to know about all the other `boid`s as well. We can accomplish this by passing the `ArrayList` of all boids through as an argument to an individual boids' `simulate` method:

```
class FlockSystem extends ParticleSystem {
      public FlockSystem() {
            super();
      }

      public void simulate() {
            for (int i = 0; i < particles.size(); i++) {
                  Boid b = (Boid)particles.get(i);
                  b.simulate(particles);
            }
      }
}
```

Once each individual boid (particle) knows about the whole list of boids it can perform calculations based on it, such as compute the average velocity of all particles, center of all particles, check for its neighbors, etc. For example, consider this method built into the `boid` class itself:

```
public Vector3D cohesion(ArrayList boids) {
      float neighborRange = 100.0f;
      Vector3D sum = new Vector3D(0, 0, 0);
```

© 2005, Patrick Dwyer - Creative Commons Attribution-NonCommercial-ShareAlike

```
        int count = 0;
        for (int i = 0; i < boids.size(); i++) {
              Boid other = (Boid)boids.get(i);
              float d = Vector3D.distance(loc, other.loc);
              if ( (d > 0) && (d < neighborRange) ) {
                    sum = sum.add(other.loc);
                    count++;
              }
        }

        if (count > 0) {
              sum = sum.divide((float)count);
              return steer(sum, false);
        }
        return sum;
      }
```

In the above we code, we perform the following algorithm:

- Compute the sum of all locations of all boids within 100 pixels
- Compute the average location (i.e. divide by total neighboring boids)
- Compute the steering vector towards the average location

Using the above method, along with two additional similar ones for separation and alignment, we now have all the elements for flocking, i.e. take the results of the three rules, weight them appropriately, and accumulate them together in the object's acceleration.

```
public void flock(ArrayList boids) {

      Vector3D sep = separate(boids);
      Vector3D ali = align(boids);
      Vector3D coh = cohesion(boids);

      sep = sep.multiply(10.0f);
      ali = ali.multiply(1.0f);
      coh = coh.multiply(2.0f);

      accel = accel.add(sep);
      accel = accel.add(ali);
      accel = accel.add(coh);
}
```

© 2005, Patrick Dwyer - Creative Commons Attribution-NonCommercial-ShareAlike

Resources

Craig Reynolds: http://www.red3d.com/cwr/

- Steering Behaviors For Autonomous Characters: http://www.red3d.com/cwr/steer/
- Boids Web Page: http://www.red3d.com/cwr/boids/
- Flocks, Herds, and Schools: A Distributed Behavioral Model, 1997 Siggraph Paper: http://www.red3d.com/cwr/papers/1987/boids.html

Assignment

At this point in the semester you should be actively pursuing the topics we cover that are of interest to you. Each week you should be working on either a small project, or creating prototypes of a project you would like to pursue as your final project.

If you want to work with the Flocking examples, here are a few ideas:

- What other rules could you apply to Boids to guide their behavior?
- What could the flocking affect besides location of an object?
- What kinds of groups could you simulate by changing the flocking parameters of each individual boid? Think about the following parameters that effect the boid:
  - Distance to neighbors for calculations of cohesion and alignment
  - Max Speed
  - Max Acceleration
  - Comfortability level (separation value)
  - Containment area

© 2005, Patrick Dwyer - Creative Commons Attribution-NonCommercial-ShareAlike

© 2005, Patrick Dwyer - Creative Commons Attribution-NonCommercial-ShareAlike