CMPSC 383: Multi-Agent and Robotic Systems

Final Project Progress Report

1 Introduction

Flocking, in the context of multi-agent systems, involves the collective behavior of a group of interacting, mobile agents. Flocking patterns are exhibited by such a group when the individual agents attempt to move together with their neighbors. This behavior in multi-agent systems mimics the similar flocking behaviors of groups of animals such as birds or fish. Applications of this type of behavior are diverse: multi-robot teams can use flocking to coordinate their movement towards a goal while maintaining a consistent formation or level of proximity [5, 7], finding optimized solutions to mathematical problems [1], and simulating the behavior and appearance of flocks or swarms of animals.

Boids particular simulation model for multi-agent flocking behaviour [2]. It was first described by Craig Reynolds in 1987 [6]. Boids uses a set of three rules for the manipulation: separation, cohesion, and alignment [2, 6]. More detail on this model can be found in Section 2.2. This model has been used in a number of computer graphics applications, including motion pictures and video games, as well as in robotics applications [5, 7].

Our project is an implementation of this Boids simulation. To do so, we used the Haskell programming language, a purely functional language with lazy evaluation. This document describes some particulars of Haskell and how they relate to the task of agent programming, the details of the mechanism of our implementation, and the results we acquired from our implementation.

2 Method

2.1 Language Choice

We chose Haskell as the implementation language for our Boids simulation. Haskell is a purely functional language with lazy evaluation [**jones2003haskell**, 3]. It has been observed that Haskell supports the rapid prototyping of software systems, allowing working systems to be implemented quickly and with minimal complexity [3]. The functional programming paradigm in general has been noted to support modularity and allow problems to be decomposed easily [4]. These observations, along with our previous experience in the language, influenced our choice of Haskell as a platform for our project.

As our class experience with implementing multi-agent simulations has been solely through the object-oriented programming (OOP) paradigm, our language choice presented a challenge, but also a valuable learning experience. Our implementation of the Boid agents involves the definition of an abstract data type, which contains the position, velocity, and neighborhood radius of an individual boid. In Haskell, this is defined as follows:

```
data Boid = Boid { position :: Point
                 , velocity :: Vector
                 , radius   :: Float
                 }
```

Note that Haskell, as a purely functional programming language, does not allow mutation of existing values. Therefore, once a `Boid` is created, it cannot be mutated, and updating the `Boid`'s state requires the creation of an entirely new `Boid` instance.

Also, in contrast to a corresponding OOP implementation, which might define some `Boid`behaviour (i.e. methods) to accompany this basic data structure, this `Boid` data type is kept distinct from the functions which act upon it.. Instead, we define a type called `Update`:

```
type Update = Boid -> Boid
```

Thus, a function of type `Update` is a function that takes a `Boid` and returns a new `Boid`. We use `Update` to define `Behaviour`:

```
type Perception = [Boid]
type Behaviour = Perception -> Update
```

This defines a `Behaviour` as a function that maps an agent's perception of its environment (in this case, a `Boid`'s visible neighborhood) to a function for updating its environment. These examples demonstrate how implementing a multi-agent simulation in a purely-functional language requires a different conception of what it means programmatically for an Agent to behave.

2.2 Boids

Boids models the behaviour of a flock in as being effected by three primary steering forces: *cohesion*, the tendency of an individual to stay close to the centre of the flock; *separation*, the tendency of an individual to avoid collision with other individuals, and *alignment*, the tendency of an individual to match velocities with its neighbors [2, 6]. Each of these steering forces is modeled as a vector, which are then summed to compute the position of a given boid at each time interval.

The separation steering vector $\vec{s_i}$ for a given boid b_i may be calculated as the negative sum of the position vector of b_i and each visible boid b_j, using the following formula:

$$\vec{s_i} = -\sum_{\forall b_j \in V_i} (p_i - p_j)$$

where V_i is the set of boids visible by b_i (i.e. the neighborhood) [2]. In our implementation, this formula corresponds to the following Haskell source code:

```
separation :: Boid -> Perception -> Vector
        -- :: Boid -> [Boid] -> V3 Float
```

```
separation self neighbors =
    let p = position self
    in sumV . map (^-^ p) $ positions neighbors
```

The cohesion steering vector $\vec{k}_i$ for a given boid b_i may be calculated by finding the centre of density c_i of the visible boids V_i using the formula

$$c_i = \sum_{\forall b_j \in V_i} \frac{p_j}{m}$$

where m is the cardinality of V_i. The steering vector may then be calculated by subtracting b_i's position from c_i [2]:

$$\vec{k}_i = c_i - p_i$$

In our implementation, these formulae corresponds to the following Haskell source code:

```
centre :: Perception -> Vector
    -- :: [Boid] -> V3 Float
centre boids =
    let m = fromIntegral $ length boids :: Float
    in sumV $ map (^/ m) $ positions boids

cohesion :: Boid -> Perception -> Vector
      -- :: Boid -> [Boid] -> V3 Float
cohesion self neighbors =
    let p = position self
    in centre neighbors - p
```

Finally, the alignment steering vector $\vec{m}_i$ for a boid b_i may be calculated by averaging the velocities of the set of visible boids V_i using the following formula

$$\vec{m}_i = \sum_{\forall b_j \in V_i} \frac{\vec{v}_j}{m}$$

where $\vec{v}_j$ is the velocity of b_j. If the cardinality of V_i is zero, then $\vec{v}_i = 0$ [2]. In our implementation, this formula corresponds to the following Haskell source code:

```
alignment :: Boid -> Perception -> Vector
       -- :: Boid -> [Boid] -> V3 Float
alignment _ []        = V3 0 0 0
alignment _ neighbors =
    let m = fromIntegral $ length neighbors :: Float
    in (sumV $ map velocity neighbors) ^/ m
```

Once all three steering vectors have been calculated, they are combined to find the velocity $\vec{v_i\prime}$ of a boid

$$\vec{v_i\prime} = \vec{v_i} + S.\vec{s_i} + K.\vec{k_i} + M.\vec{m_i}$$

where S, K, and M are coefficients which control the weight of each steering force and are typically global parameters to the simulation.

The position of that boid at time $t + \Delta t$ maythen be updated using $\vec{v_i\prime}$

$$p\prime_i = p_i + \Delta t \vec{v_i}$$

In our implementation, these formulae corresponds to the following Haskell source code:

```haskell
steer :: Weights -> Behaviour
   -- :: Weights -> [Boid] -> Boid -> Boid
steer (s, c, m) neighbors self =
    let s_i  = s *^ separation self neighbors
        c_i  = c *^ cohesion self neighbors
        m_i  = m *^ alignment self neighbors
        v'   = velocity self ^+^ s_i ^+^ c_i ^+^ m_i
        p    = position self
        p'   = p ^+^ v'
    in self { position = p', velocity = v'}
```

2.3 Program Usage

Our implementation supports a number of command-line arguments to set simulation parameters and control the resulting visualization. The following options for controlling the simulation parameters are supported:

Height of the space can be set using the options `-x HEIGHT` or `--height=HEIGHT`.

Width of the space can be set using the options `-y WIDTH` or `--width=WIDTH`.

Number of Boids can be set using the options `-n BOIDS` or `--num=BOIDS`.

Boid visibility radius can be set using the options `-v RADIUS` or `--visibility=RADIUS`. The default is 50.

Simulation speed can be set using the options `-p SPEED` or `--speed=SPEED`. This value sets a coefficient by which the movement vector of each boid is divided every simulation tick. The default is 1000 at 30 frames per second.

Additionally, our implementation allows the user to select one of a set of preset behavior types. These control the values of the weight constants for the three boid steering forces, as discussed in Section 2.2.

Equal-Weight Behavior selected with the command-line flag `-e` or `--equal`, sets all weight coefficients to 1.

Cohesive Behavior selected with the flag `-c` or `--cohesive`, sets weights as follows: $S = 1, K = 0.5, M = 0.5$.

Swarming Behavior selected with the flag `-s` or `--swarm`, sets the alignment coefficient (M) equal to zero, creating true swarming behavior.

Our program may also be run in 'debug mode,' using the `-d` or `--debug` option. In debug mode, the program will run the visibility radius and velocity vector of each boid, so that the user can gain a greater understanding of how the simulation functions. Finally, the `-h` or `--help` flag will print out a summary of the available command-line options.

3 Results

The results of this simulation are visible in the following figures:

- Figure 1 demonstrates our simulation without our "debug-mode". Of course, as a static image without any representation of boid vectors, it is difficult to tell how the swarm is behaving.
- Figure 2 demonstrates our "debug mode" with equal weights for cohesion, separation, and alignment. A large degree of agent clustering is visible.
- However, with smaller radii as in Figure 3, we see a significantly decreased level of clustering of the agents.
- On the other hand, increasing the radii as in Figure 4, gives an even greater degree of clustering than before.
- With our "cohesive" mode, as in Figure 4, we see that the boids form a much more even distribution throughout the space. This is due the fact that the separation impulse is still present, but does not dominate the vectors of the boids, preventing them from separating a great distance apart.

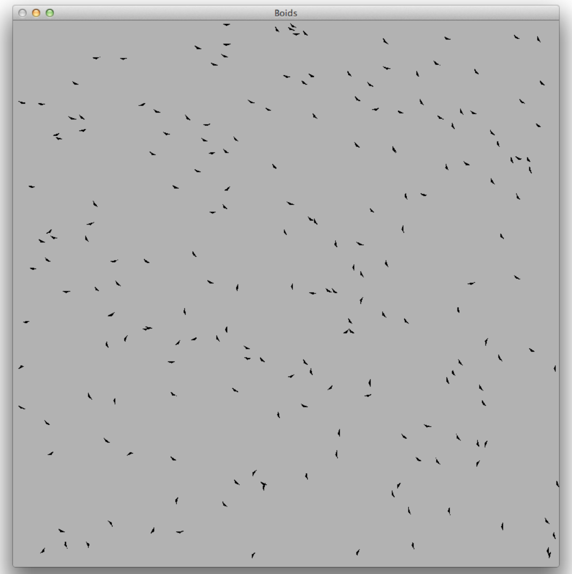

Figure 1: Simulation run in swarm mode, with debugging off.

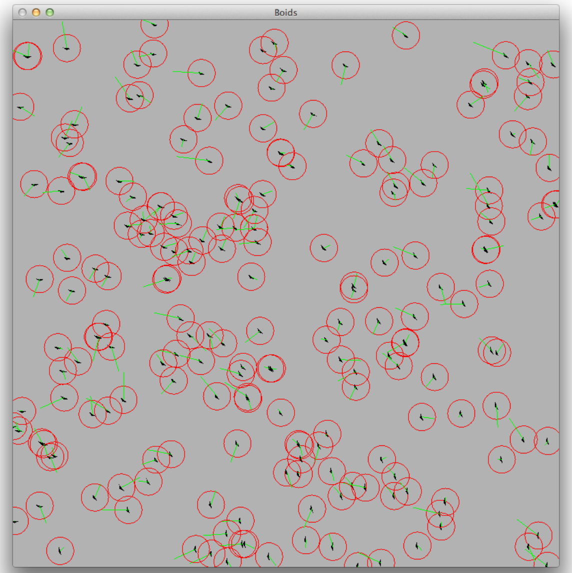

Figure 2: Simulation run in equal-weight mode, with debugging and radii of size 20.

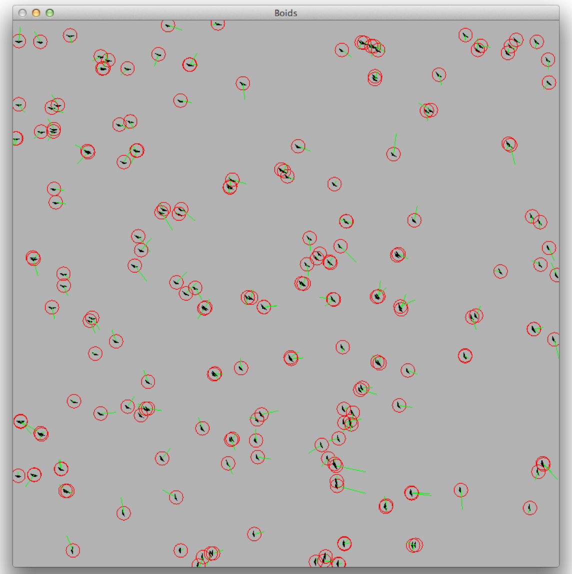

Figure 3: Simulation run in equal-weight mode, with debugging and radii of size 10.

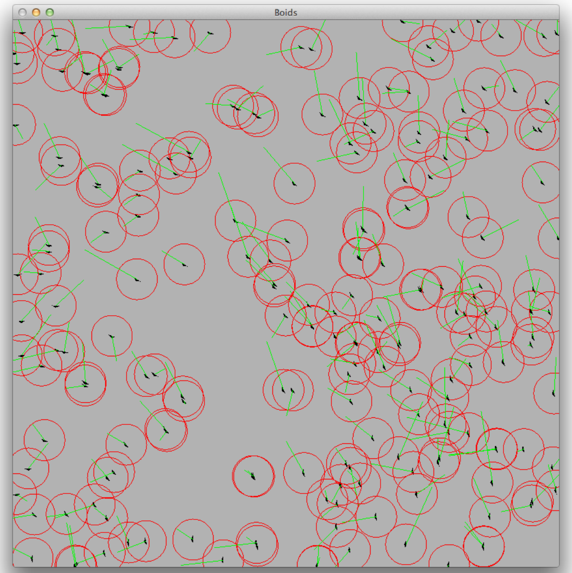

Figure 4: Simulation run in equal-weight mode, with debugging and radii of size 30.

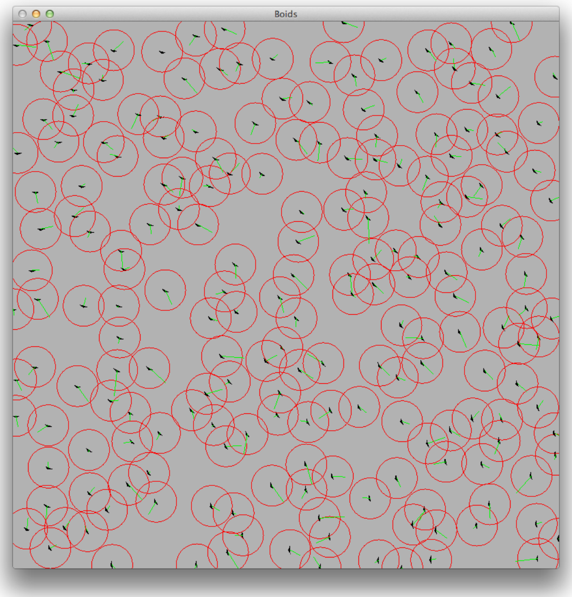

Figure 5: Simulation run in cohesion mode.

4 Conclusion

4.1 Challenges

One of our major challenges was the existence of a bug in the “alignment” behavior of our code. This bug caused all boids in the scene to speed up erratically, eventually causing overflow in the vector components of the boids’ velocities, and causing the program to crash. This was fixed after we realized that the neighborhood selection code for each boid included the boid itself, and thus added on the boid’s own velocity to itself in the alignment function.

Another challenge we faced early on was our early ambitions of creating a 3D simulation using the Haskell OpenGL bindings. These bindings caused us a number of problems, and abandoning them in favor of the 2D `gloss` library allowed us to focus on the details of our implementation with greater detail. Neither of us is particularly experienced with computer graphics, and we found that the higher level of abstraction provided by `gloss` made our task much easier. Furthermore, using OpenGL directly requires writing code that could be considered fairly unidiomatic in Haskell.

4.2 Takeaways

Both group members enjoyed the experience of implementing a multi-agent system using a purely functional programming language. As Hawk had significantly less experience programming in Haskell than Will, he found this assignment to be a particularly good learning experience in that language, but both members of the team enjoyed the opportunity to try something different than the previous course assignments. Furthermore, we found experimenting with Haskell’s libraries for vector mathematics, as necessitated by the boids algorithm, to be interesting and fun.

We also enjoyed being able to implement swarming and flocking behavior in a multi-agent system. Our previous experiences with swarming in class had only involved using preexisting programs, so we were interested in how such programs were written. Although Boids is not a particularly complex algorithm, the simulations it produces provide a fairly convincing approximation of real-life bird behavior. Both team members are very interested in 'artificial life' simulations in general — for his first final project in a computer science course at Allegheny, Will wrote an implementation of Conway's Game of Life in Processing. As this project is his last final project for a computer science course here at Allegheny, artificial life provides 'book ends,' in a sense, on his Allegheny career.

References

[1] Zhihua Cui and Zhongzhi Shi. "Boid particle swarm optimisation". In: *International Journal of Innovative Computing and Applications* 2.2 (2009), pp. 77–85 (cit. on p. 1).

[2] Christopher Hartman and Bedrich Benes. "Autonomous boids". In: *Computer Animation and Virtual Worlds* 17.3-4 (2006), pp. 199–206 (cit. on pp. 1–3).

[3] Paul Hudak and Mark P Jones. "Haskell vs. Ada vs. C++ vs. awk vs.... an experiment in software prototyping productivity". In: *Contract* 14.92-C (1994), p. 0153 (cit. on p. 1).

[4] John Hughes. "Why functional programming matters". In: *The Computer Journal* 32.2 (1989), pp. 98–107 (cit. on p. 1).

[5] Hongkyu Min and Zhidong Wang. "Design and analysis of Group Escape Behavior for distributed autonomous mobile robots". In: *Robotics and Automation (ICRA), 2011 IEEE International Conference on*. IEEE. 2011, pp. 6128–6135 (cit. on p. 1).

[6] Craig W Reynolds. "Flocks, herds and schools: A distributed behavioral model". In: *ACM SIGGRAPH Computer Graphics* 21.4 (1987), pp. 25–34 (cit. on pp. 1, 2).

[7] Martin Saska, Jan Vakula, and Libor Preucil. "Swarms of micro aerial vehicles stabilized under a visual relative localization". In: *Robotics and Automation (ICRA), 2014 IEEE International Conference on*. IEEE. 2014, pp. 3570–3575 (cit. on p. 1).