CMPSC 383: Multi-Agent and Robotic Systems

Final Project Progress Report

1 Language Choice

We chose Haskell as the implementation language for our Boids simulation. Haskell is a purely functional language with lazy evaluation. It has been observed that Haskell supports the rapid prototyping of software systems, allowing working systems to be implemented quickly and with minimal complexity [2]. This observation, 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 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 does not allow mutation of existing values. Therefore, once a Boid is created, it cannot be mutated, and instead updating the BoidâĂŹs state requires the creation of an entirely new Boid instance.

Also, in contrast to a corresponding OOP implementation, which might define some Boid behaviour to accompany this basic data structure, this Boid data type is kept distinct from its update method. 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 a `Boid`'s neighborhood to an update function. 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 Boids

Boids is a simple simulation of flocking behaviour which mimics the appearance of a flock of birds [1]. It was first described by Craig Reynolds in 1987 [3].

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 [1, 3]. 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) [1]. 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 [1]:

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

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 $\vec{v}_i = 0$ [1].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
```

3 References

[1] Christopher Hartman and Bedrich Benes. Autonomous boids. *Computer Animation and Virtual Worlds*, 17(3-4):199–206, 2006.

[2] Paul Hudak and Mark P Jones. Haskell vs. ada vs. c++ vs. awk vs.... an experiment in software prototyping productivity. *Contract*, 14(92-C):0153, 1994.

[3] Craig W Reynolds. Flocks, herds and schools: A distributed behavioral model. *ACM Siggraph Computer Graphics*, 21(4):25–34, 1987.