Simulation of Duck Behaviour using Swarm Intelligence and Finite-state Machines

Emma Broman

November 2018
TNM095 - AI for Interactive Media
Linköping University
Norrköping, Sweden
emmbr740@student.liu.se

***Abstract*—This article covers a project that was made in the course *TNM095 - AI for Interactive Media* at Linköping University, Sweden. The aim was to explore the uses of flocking algorithms and behaviour simulation methods by mimicking the behaviour of a bird flock. To do this, an implementation of Reynold's *boids* model, together with a finite-state machine, was implemented using the game engine Unity. The result was an interactive application where the user controls a player (a ball) object to interact with the flock of ducks and the result is sufficiently realistic for the scope of this project. Since the aim of the project was to explore and learn about the methods rather that producing a realistic result, the outcome of the project was considered successful.**

I. Introduction

Mimicking the behaviour of real world entities, like humans or animals, has proven useful for a wide range of applications. *Behaviour simulation* is a field in AI that has been widely used in the game industry to create a feeling of life and intelligence in enemies and non-playable characters, but also for robotics and logistic problems. Often it is useful to simulate the behaviour of groups of entities, to study how they act together in an applied situations.

Swarm intelligence is a term used for the simulation of the behaviour where groups of simple entities, like bird, ants or fish, work together to solve more complex problems. For example, the behaviour of ants can be simulated to solve path finding problems, and flocking algorithms can be used to study the behaviour of how a crowd of humans would react to an event. This could in turn be used for other applications, like perhaps planning emergency exits in a building.

A different application of swarm intelligence was mentioned in the article ”How 'artificial swarm intelligence' uses people to make better predictions than experts”, that was published on the site TechRepublic in January 2016 [1]. In the article, the author explains the result of a study where swarm intelligence was applied to a predicting problem. The performance of the swarm was compared to the performance of humans trying to make the same predictions. In the study, the swarm actually performed better, using the ”wisdom of the crowd”. The result is interesting, and might signify that there are more to learn from these kind of behaviour simulation methods, despite the fact that the main focus of the AI field today seems to lie mostly in the area of for example machine learning.

A. Aim

The aim of this project was to try to mimic the behaviour of a flock of birds using flocking simulation and a finite-state machine. The main purpose was to explore the area of behaviour simulation and learn about the uses and limitations of the chosen methods.

B. Delimitations

To simplify the implementation of the simulation, only the behaviour of birds walking on land was taken into consideration, i.e. the birds in this project cannot fly. The behaviour is also reduced to merely a few states to reduce the complexity of the state machine. This is because the aim of the project was to implement and try to understand the chosen methods, rather than achieving a realistic result. For the same reason, no explicit methods for path finding or collision detection have been implemented.

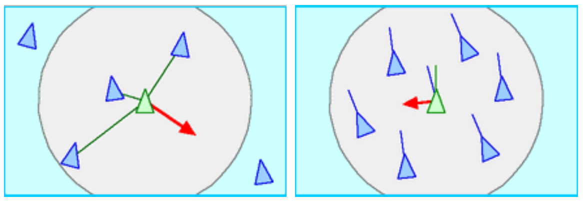
II. Theory

This section will cover the theoretical background relevant for the project, which can be split into three different parts: the *flocking behaviour* in general, the *duck behaviour study* and *finite-state machines* (FSM).

A. Flocking

Flocking is a phenomena that can be seen in nature for a wide range of animals; like bird flocks, fish schools or even humans. In many cases, it is simply beneficial for the individuals to stay in a group, e.g. to avoid predators or to find food. Several models for representing this kind of coordinated motion exists, and the one used in this project was Craig Reynolds' *boids* model [2]. This model uses three different steering rules that affect the motion of each agent (boid):

- **Separation:** if too close to a neighbour, steer away to avoid collision.
- **Alignment:** try to steer towards the average moving direction of the flock mates.
- **Cohesion:** strive towards the center (average) position of the flock, i.e. try to stick together.


(a) Separation (b) Alignment

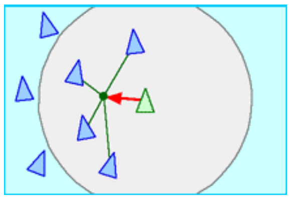

(c) Cohesion

Fig. 1. The three rules of boid flocking

Each of these rules result in a vector that will contribute to the final motion of the boid. This is illustrated in figure 1. One possible implementation, which uses a velocity vector for the boid to achieve the flocking behaviour, is the following:

1) Find the neighbouring agents, e.g. those within a certain distance.
2) For each neighbour, compute and accumulate each of the contributing vectors according to the rules above (will result in three vectors, one for each rule: separation, alignment, cohesion).
3) Divide each vector with the number of neighbours.
4) Multiply each vector by a weight for the corresponding steering behaviour and add the result to the velocity of the agent.

A simple pseudo code example of the algorithm can be found in the book ”...So How Can We Make Them Scream?” by Ingemar Ragnemalm [4].

Additional rules can be added to add some variation to the flock, and thus make the behaviour more interesting. For example, a ”leader” entity can be used to add a sense of purpose to the flock, with all agents aiming for a common goal. There can also be other kind of attractors, detractors or other outer forces that affect the flocks’ movement, or some heuristic determining what move is the most beneficial at the time.

The method described above is suitable for systems with a limited number of agents/boids and the greatest difficulty in that case is to find weights that give a desirable result. However, for a large system some optimisation might be necessary to reduce the number of tests needed to search for nearby agents. The method currently implies a double loop with the complexity $O(n^2)$, where n is the total number of boids. By using some kind of subdivision method for finding the neighbours, this can be reduced to $O(n * logn)$.

B. Duck behaviour

Since it proved to be more difficult than imagined to find scientific articles describing how ducks behave on land, a short field study on the matter was done at the start of the project. This study included watching online videos and interacting with the local ducks that reside in the parks of Norrköping to examine their behaviour. By combining the observations with some assumptions and simplification, the behaviour to be implemented was summarised by the following:

- If nothing in the environment affects the duck, walk around in a flock and sometimes lay down to rest.
- If there is food in sight, go and eat.
- If there is an enemy, walk away from it. However, for safety reasons, try to stick to the group.

C. Finite-state machines

A finite-state machine, FSM for short, is a common model for implementing AI logic in games. The behaviour of an agent is described using a number of states, each representing some desired behaviour, and transitions between these states. The current state determines the current behaviour of the agent, and the agent can exist in exactly one state at a time. In games the transitions can be triggered based on events in the game world, for example if an enemy is within a certain distance.

A FSM can be described using a graph representation, where the nodes correspond to the state and the edges to the transitions. Figure 2 shows a graph representing the simple FSM that was implemented in this project. This figure will be further explained in section III.

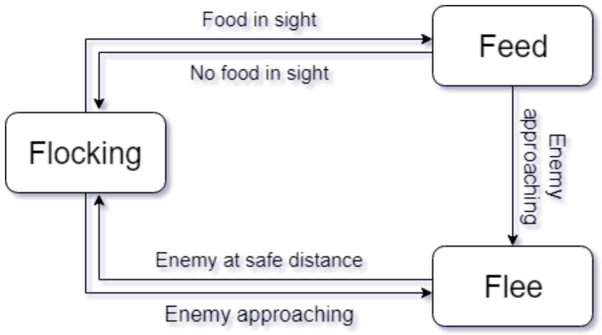


Fig. 2. Simple FSM describing the behaviour of a single duck. The ”Flocking” state means that the duck just acts according to the rules of boid flocking [2].

A drawback of FSMs are that the complexity of the machine increases exponentially with every added state. In the worst case each new state introduces 2^n extra transitions, where n is the number of existing states. As one might realise, this quickly becomes unmanageable and slow. The problem can be somewhat reduced by using a stack-based FSM [3] or in some cases a hierarchical solution, but it is still important to be aware of this limitation.

III. Method

A 3D game engine was used to handle the graphics and visual parts of the project, as well as some simple collision handling. This made it easier to focus on implementing the AI methods.

Firstly, the flocking behaviour was implemented and experimented with until suitable weights and distances were determined. Then, a simple FSM was implemented and a couple of possible states were added to the definition of the ducks' behaviour. This also included adding a player object that the user could control to interact with the agents. Lastly, a 3D duck model with some simple animation was added to the boid objects, as well as some vegetation in the environment. The 3D duck model used for this project was bought for a small sum in the Unity Asset store, and the vegetation models for the environment were downloaded for free.

A. *Game Engine - Unity3D*

The game engine used in the project was *Unity3D* for Windows, version 5.5. Unity is widely used by professionals, students and hobbyists and has a large community that provide different tutorials, in-game assets like 3D models and materials, answers to questions, etc.

Unity uses a script-based programming system, meaning that scripts can be attached to so called "game objects" to control their movements and behaviour. The code for this project was written in C# but it is also possible to write code for Unity using a version of Javascript.

Unity also provides some built-in basic functionality that is useful for creating games, for example simple physics and collision handling. This was useful for this project, since no explicit collision handling apart from the boid separation was implemented. Instead, the build in rigid body collision in unity was used for collision with the game world walls, as well as the player object.

B. *Flocking*

Firstly, a number of agents are generated at random positions within the game area, with the initial velocity zero. Since the flocking is implemented in 2D, all agents have the same y coordinate. Then, every frame, the movement of the agent in the xz-plane is updated. To do this, the agent must first find its current neighbours. To avoid comparing the distance from the agent to every other boid in the scene every frame, the built in collision system in Unity was taken advantage of to find all game objects of type "boid" within a certain distance. This was done by creating an invisible sphere and test all objects colliding with this sphere and save references to them in an array.

Once the neighbours are found, the vectors to be accumulated with the velocity of the boid are computed according to what is described in section II-A. Lastly, the position and rotation of the agent are updated based on the velocity.

The flocking behaviour was implemented on individual level, meaning that each agent has a script attached to it that controls its behaviour. A fixed maximum speed was set, to make the movement more realistic.

C. *The Player and Food*

The controllable player object is a simple sphere that can be controller using the arrow or WASD keys. In contrast to the boids, the player's movement is done using Unity's built in rigid body component, which allows us to add forces to objects to make them move. Additionally, the space key on the keyboard has been bound to instantiate a "food" object. The food has a certain capacity that is reduced when the ducks are close enough, to represent them eating it. Once the capacity is reduced to zero, the food disappears and no longer affect the agents.

D. *FSM*

The finite-state machine was implemented based on a tutorial that is provided on Unity's website [5]. It uses an approach that exploits the possibility to connect scripts through the *Inspector* interface in Unity to implement the Delegate design pattern. This leads to a pluggable solution that uses polymorphism and small concrete classes to achieve the behaviour, instead of for example a large switch statement.

The FSM implementation consists of four different types of classes:

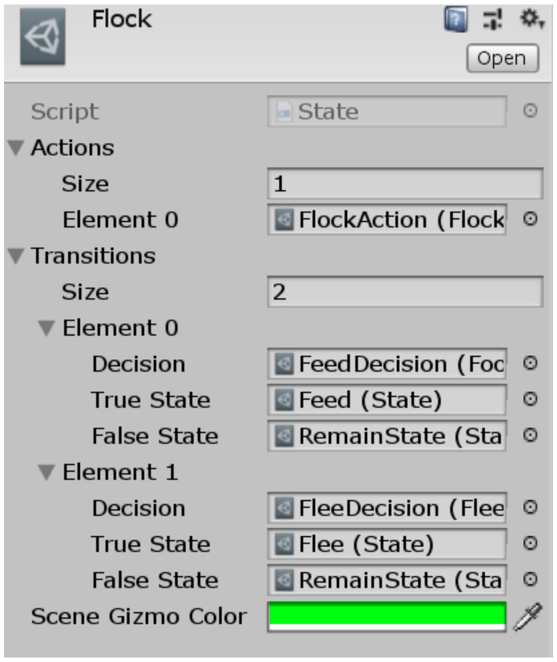
- **Decisions:** Performs a test that returns either true or false.
- **Transitions:** Consists of a decision and two states: one state that is transitioned to if the decision is true and another for if it is false.
- **Actions:** Represents an action to be performed while an agent is in a state. The actual behaviour that is executed.
- **States:** Keeps track of two lists: transitions and actions. Is responsible for checking for transitions and performing actions.

Decision and *Action* are simple abstract classes that contains only one method that needs to be overrided by concrete subclasses. *Transition* is a serializable class that merely connects two states to a Decision. The actual decision making is performed in the *State* class, which is responsible for checking for transitions (i.e. if any decision is true) and performing any actions associated with the state.

So, to add a new state to the FSM what is needed is to:

- create an instance of the State class and decide how many transitions that are needed,
- write one or more subclasses to Action that describes the behaviour of the agent in that state,
- write decisions that determines when this state should be transitioned to or from,
- connect the new state to the other states in the FSM, preferably by dragging and dropping the created scripts into the correct slot in the Unity Inspector interface for the relevant states (see figure 3).

Also, since the decisions are checked every frame there is a need for a "dummy state" that represents staying in the same state. This one has been called *RemainState* and its use is demonstrated in figure 3. The "Scene Gizmo Color" that can be seen in figure 3 is the colour of a wired sphere surrounding the agent, that changes colour depending on the current state. This was used for debugging wile implementing the FSM.



Flock

Open

Script	State
▼ Actions	
Size	1
Element 0	FlockAction (Flock
▼ Transitions	
Size	2
▼ Element 0	
Decision	FeedDecision (Foc
True State	Feed (State)
False State	RemainState (Sta
▼ Element 1	
Decision	FleeDecision (Flee
True State	Flee (State)
False State	RemainState (Sta
Scene Gizmo Color	🟩

Fig. 3. The view in the Unity Inspector for the *Flock* state, showing the two possible transitions to the states *Feed* or *Flee* (compare with figure 2). The state also has one action which is performed while the agent is is this state, the *FlockAction*.

IV. Results

The result is an interactive application where the user controls a player (a ball) object to interact with the flock of ducks. The behaviour that comes from the interaction is determined by a simple FSM (see figure 2) and the implementation performs well for a large number of agents. The player can attract the ducks by placing out food, and scare them away by rolling towards them. A print screen from the running application is shown in figure 4 and 5. However, it is difficult to observe the behaviour from still images since it is very dependent on the interaction from the player and the movement of the agents.

Since no leader was implemented, the agents just move aimlessly around the game area in whatever direction the flock is heading, unless the player places out a piece of food.

V. Discussion

The result is a minimal solution with a very simple FSM consisting of just a few states and limited flocking behaviour, but still gives a pretty satisfying impression at first glance, seeming more complex than it actually is. The duck flock simulation could possibly be used as a nice feature in a game, or perhaps be the base of a small game itself with some more work. However, when studying the agents closer, it is apparent that every agent behaves exactly the same way, which makes sense since no randomness in the behaviour was included in the implementation. The implemented behaviours could also use some fine tuning to be more realistic to those of actual ducks. For example, currently all agents immediately know when there is food in the game world, and the ”Feed” state

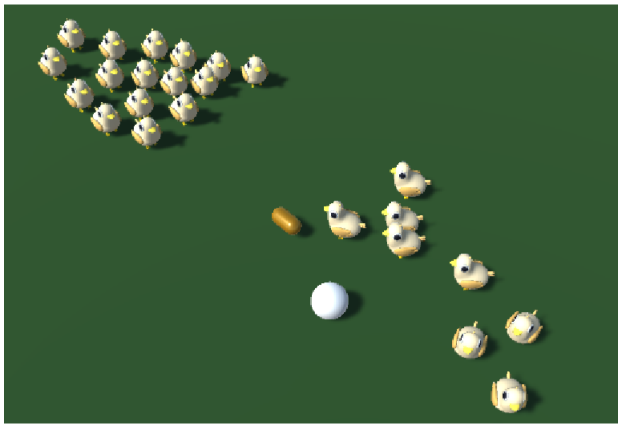


Fig. 4. A print screen from a running version of the result. The white ball is the player and the capsule in the center is the ”food”.

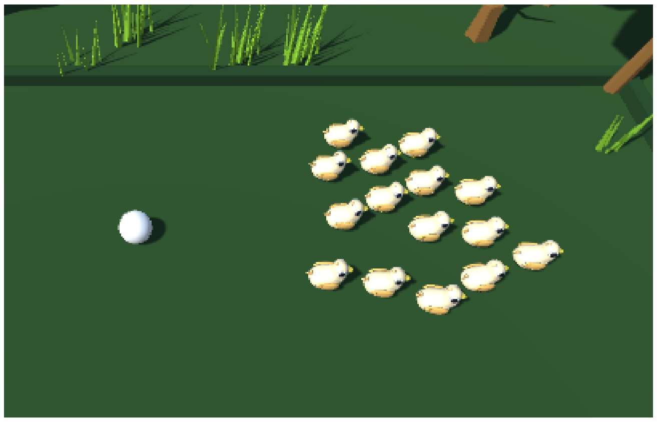


Fig. 5. A print screen from a running version of the result showing the flocking behaviour, where the birds are unaffected by the player.

is transitioned to. A better solution would perhaps be to use some sort of ray tracer or field of view implementation to simulate the perception of the duck. The same thing could have been done in the step for finding the neighbours, something Reynolds mentions when describing the boid model in his article [2]. Currently, the agents can detect other agents that are immediately behind them, which is not very realistic. However, the result of this is barely noticeable in the application. Furthermore, the flocking algorithm might be in need of some fine tuning. However, the most time consuming part of the project was already finding suitable values for the variables and distances needed for the flocking behaviour. These values could be fine tuned for an eternity and still only achieve marginally better results, so it is probably not worth the effort.

It would also have been useful to implement some kind of collision handling for the edges of the game world. At the moment, only Unity's build in rigid body collision system is used to prevent the agents from walking through the walls. This sometimes results in weird behaviour, especially for small groups of agents, where they end up walking in a line along the wall or just straight towards it. This could have been prevented using a simple collision handling mechanism, e.g. *crash-and-*

turn [4].

The solution for the FSM is neat and simple to expand due to the Delegate pattern approach and the pluggabilty that comes with exploiting the Unity Inspector. A new state is easily added and can be done by creating transitions to already existing states by dragging and dropping scripts to the correct slot in the inspector. However, it is not trivial (but likely possible) to extend the model to allow hierarchical state machine, i.e. that a state can be a state machine itself, which would be desirable to include a more realistic behaviour in which the for example stops and rests from time to time. To do this would require some rethinking regarding the implementation of the State class.

The finite-state machine is a solution that works well in this situation, where the behaviour is dynamic (the agents react to external events) and consists of a relatively small number of states. However, for a more complex behaviour the number of transitions could quickly become unmanageable. In that case, there would be a need for a hierarchical or stack based implementation of the FSM.

VI. Conclusion

The flocking algorithm in combination with the finite-state machine produces sufficiently realistic behaviour for the agents using relatively simple code. The methods were both suitable for this project. Thanks to the FSM it is easy to expand the behaviour with additional states, if desired. However, if the desired behaviour becomes too complex it might be necessary to implement some sort of hierarchical or stack based FSM version.

In conclusion, the aim of the project, i.e. to learn more about the methods, was fulfilled and the project was successful from an educational point of view. However, the result could use some more work to be applicable to for example a game.

VII. Further development

As implied in the discussion, I would like to make some enhancements to the simulation to make it more realistic and interesting. Some examples of possible enhancements are:

- introduce some randomness in the behaviour of the agents,
- implement a field of view solution for simulating the perception of the agents so that they cannot detect agent behind them,
- implement some simple collision handling for the walls, e.g. *crash and turn*,
- introduce some game aspect to the interaction, for example that the player can score points in some way by interacting with the ducks.

It could also be nice to make the behaviour a bit more complex, for example by adding a rest and stop state in the FSM. Living ducks do not walk around aimlessly all the time; sometimes they stop and do not move at all and some times they lay down to rest or sleep. If these states would be added, it would be desirable to implement a hierarchical FSM where the flocking state in figure 2 is replaced by another state machine that alternates between the states *flocking* (walking), *resting* and *doing nothing*.

References

[1] Reese, H. (2016), 'How 'artificial swarm intelligence' uses people to make better than experts'. *TechRepublic*. Available: https://www.techrepublic.com/article/how-artificial-swarm-intelligence-uses-people-to-make-better-predictions-than-experts/ [accessed: Oct. 31, 2018]

[2] Reynolds, C. (2001), 'Boids Algorithm'. Available: http://www.red3d.com/cwr/boids/ [accessed: Oct. 31, 2018]

[3] Bevilaqua, F. (2013), 'Finite-State Machines: Theory and Implementation'. Available: https://gamedevelopment.tutsplus.com/tutorials/finite-state-machines-theory-and-implementation--gamedev-11867 [accessed: Oct. 31, 2018]

[4] Ragnemalm, I. (2017), '...So How Can We Make Them Scream?'. *TSBK03: Advanced Game Programming*. Available: http://www.computer-graphics.se/TSBK03/files/SHCWMTS-2017.pdf

[5] *Finite State AI with the Delegate Pattern* (n.d.), unity3d.com. Available: https://unity3d.com/learn/tutorials/topics/navigation/finite-state-ai-delegate-pattern [accessed: Oct. 24, 2018]