Flocking behaviour with Maya and the Maya API

Liviu-George Bitiușcă | MSc Computer Animation and Visual Effects
Bournemouth University

1 The Bigger Picture

More often than not we consider animation in the world of computers as the direct result of specific instructions that we lay upon innocent, yet dumb objects. A character animator, for instance, would likely define motion as a series of interpolated angles and positions that a mesh goes through with the help of its underlying rig. Likewise, an FX artist whose job is to blow things up might describe animation as the result of often fine-tuned physical forces being applied to a collection of objects that are just as careless in regard to what's happening to them as the character mentioned earlier. The difference between the two types of animation lies in its governing authority: value interpolation and physics respectively. However, there is another type of motion that is not exclusively described by neither value interpolation nor physics and that is ***behavioural animation***.

Behavioural animation deals with describing cognitive processes as *"any behaviour that can be attributed to mental activity, voluntary or involuntary, that is reacting to the environment"* (Parent, 2012). Examples of cognitive processes could be as simple as running towards a finish line in a running contest or as complex as deciding what shirt to wear on your first date. The more complex processes are usually within the domain of artificial intelligence (AI) and would be relevant for instance in defining "thinking" non-playing characters (NPCs) that respond in a believable way to player interactions. That area of study, however, falls outside the scope of the current discussion.

The other ("simple") types of cognitive processes are of much more interest when dealing with ***aggregate behaviour***, that is how a collection of objects can be perceived as a single entity by the way they interact with their surrounding environment. Good examples of such behaviour are particle systems, crowds and flocking systems. There are a suite of conceptual differences between the three, but perhaps the most important one is the level of intelligence of group members, ranging from none in the case of particles, to potentially a lot in the case of crowds, with flocking falling inbetween.

2 Introducing the boids model

Flocking is governed by relatively simple behavioural rules that can generally be described as reactions of flock members to outside stimuli. Internal properties can also be present, but usually they are there only to enhance outside response. Intelligence thus takes the form of avoiding obstacles and keeping formation, in accordance to the rules laid out by the animator. Physics still govern the motion of the system, but with a reasoning layer added on top of it.

Among the first to describe a set of rules for modelling this sort of coordinated animal motion (bird flocks, fish schools) inside a computer and certainly the most recognizable name in the field is Craig W. Reynolds, who in 1986 created the ***boids*** model - with "boids" being chosen as an umbrella term for generic "bird-oid" creatures. It is Reynolds' primary research, published in his paper *"Flocks, Herds, and Schools - A Distributed Behavioral Model"* (Reynolds, 1987) that forms the body of this discussion, as well as the more encompassing *"Steering Behaviors For Autonomous Characters"* (Reynolds, 1999), which has some clearer examples of the author's concepts.

It is a good time to mention at this point that that the aim of the rest of the discussion is to describe both the most important underlying concepts of the boids model, as well as how they were implemented as a plugin for Autodesk Maya using the Maya API. It is not a complete overview of all the techniques and concepts of flocking behaviour, as many things were intentionally left out for the purposes of fitting the project into the allocated time frame, but it should nevertheless form a good starting point for developing a more complex and complete flocking system.

3 How we can help Maya and how Maya can help us

Before dissecting the boids model, we have to decide how our development platform can sustain what we want to do and whether what we want to do is not already implemented. Maya does have a complex particle system of its own, but it does not feature standard flocking behaviours out of the box. One would have to use particle expressions that would likely be uncomfortable to control via the interface, or perhaps make some clever use of per-particle force fields which, while more accessible, wouldn't take away completely the need of using particle expressions. The best solution would therefore be a single custom node that would quickly transform any particle system into a flocking system. In return, connecting to Maya's inner dynamics system always comes with multiple advantages, like not having to worry about keeping track of all the particles or doing motion integration over time.

The node that deals with manipulating particle objects is the force field node. Specifically in the Maya API, it is the **MPxFieldNode** class that all force fields inherit from. To keep it simple, nodes of this type take in particles' **positions**, **velocities** and **masses** through an existing plug **InputData** (*Fig. 1*) and through **OutputForce** output an array of forces to be used by Maya's internal Nucleus solver to update the particle system. Our job is then to compute the "flocking forces" inside this node.

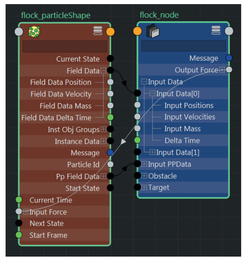


Figure 1: Connections between the particle object and the custom force field node (flock_node)

4 Boids model basics

Now that we established where our calculations will take place, let's start analyzing the boids model. First and foremost, it is important to determine what a boids "sees". If we think about a flock or a herd, any single animal only sees and cares about its immediate neighbours. Even if we think it might see all the way to the other side of the flock, it still wouldn't matter, since only its neighbours would actually affect its movement. A good analogy is thinking about being in a crowded place. Even if you see the people at the end of the hall, it's people in your immediate vicinity that directly affect your movement. And also you might not care about people behind you if you're moving away from them and you might not care about more than let's say 3 people at a time (*Fig. 2*).

In spite of that, searching through a particle system still has a time complexity of $O(n^2)$, because each boid has to consider all other boids even when only looking for its neighbours. There are methods to decrease search time by relying on space partitioning, but they will not be covered here.

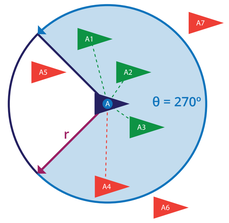


Figure 2: Neighbour search pattern

A looks for neighbours within radius r and FOV θ:

-> A1, A2, A3 meet all the neighbour criteria
-> A4 meets all criteria, but is excluded because A already has 3 neighbours and A4 is the farthest
-> A5 is excluded for being outside the FOV
-> A6, A7 are excluded for being too far

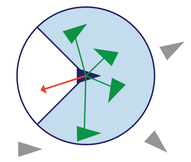


Figure 3: Separation force

*vec_sum = Σ (neighbour_pos - boid_pos).normalized) * falloff*
*separationF = -vec_sum * separation_scalar*

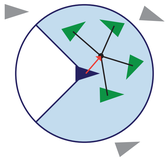


Figure 4: Cohesion force

*average_pos = average of neighbour's positions * falloff*
*cohesionF = (average_pos-boid_pos).normalized) * cohesion_scalar*

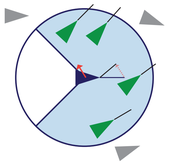


Figure 5: Alignment force

average_vel = average vector of neighbours' velocities
alignF = (average_vel - boid_vel) .normalized) align_scalar*

5 Separation Force

Separation describes the vector a boid should follow in order to avoid collision with its neighbours. The force is computed by first adding up the vectors going from the particle's position to each of the neighbours' positions. Each vector is normalized and scaled by a falloff (usually an inverse square function) depending on how far the neighbour is from the boid, with the closer ones having much greater influence than ones farther away. The result is averaged and negated in order to point in the direction away from the neighbours. Finally it is scaled by a user-controlled coefficient that determines how strong the separation force as a whole is. *(Fig. 3)*

Note: even though the drawing to the left doesn't show it, the radius for the separation force is usually smaller than on other forces, but the repelling force applied is stronger, to make up for the shorter distance.

Cohesion Force

Cohesion keeps the boids together, essentially being the opposite of Separation, but with a bigger radius to prevent boids as much as possible from going astray from their group. It is calculated as a vector from the boid's position to the average position of its neighbours. Like in the case of Separation, the result is multiplied by a distance-based falloff, normalized to get a unit direction vector and finally multiplied by a user-controlled coefficient. *(Fig. 4)*

Alignment Force

If for Separation and Cohesion forces we cared about the position of the neighbours, in the case of Alignment we are only concerned about the velocity of nearby particles. What Alignment force does is trying to keep the boid moving in the same direction as its neighbours. Not only does this help with cohesion, it also helps with separation and more precisely with collisions, because objects moving in the same direction are less likely to collide.

To calculate Alignment, we take the average of the neighbours' velocities and find the difference between it and the boid's velocity. The result is normalized and, as always, multiplied by an user-controlled coefficient. (*Fig. 5*)

6 Beyond the bare minimum

With Separation, Cohesion and Alignment we already have a dynamic system displaying flocking behaviour. Tuning search radius, vision angle (FOV), the maximum number of neighbours and force strength coefficients will have an immediate effect on the shape of the flocking system, as its members will adapt to the new settings. And while it's nothing to write home about, this is nevertheless a crude representaion of flocking behaviour.

In order to make it more believable, we need to add things to it. And what is immediately noticeable at this point is that the flock is stationary. Fortunately, we have a couple of options to fix this problem: to add **Seek, Flee** and/or **Pursuit** forces. But before be can do that, we need an object to seek, pursuit or to flee from. The only valid way to get any kind of information into a Maya DG node is through an input plug. We would call the plug **Target** and we would need to know a few extra things about it beside its position: its **weight** (a value from 0 to 1 describing how much it attracts the boids), its **arrival distance** (how far from it do the boids start to slow down in order to intercept it) and its **radius** (how far does the attraction of the target extend from its center). Knowing these, we can create a compound plug on the field node with all the required inputs. (*Fig. 6*)

Figure 6: Target inputs on the flock node.
Note: Target is a plug array, taking in as many targets as necessary.

7 Seek/Flee Forces

Seek is the pursuit of a static target, whereas **Flee** is its inverse. Then all a boid needs to do is to speed towards the target's **position** (obtained by extracting the translation vector from the target's transformation **matrix**) or flee from it if it's within the target's **radius** and if the target's **weight** parameter is not 0. *(Fig. 7)*

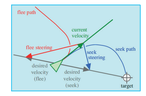


Figure 7: Seek and Flee (Reynolds, 1999)

Pursuit/Evasion Forces

These are really similar to Seek/Flee, however they consider the target's position *in the future*. That position cannot be known precisely, but can be assumed based on the target's velocity at the current frame. A simple forward integration is performed to approximate a position at some time in the future (shorter time if target is close, longer if target is far), then that value is used to calculate the **Pursuit/Evasion** forces. *(Fig. 8)*

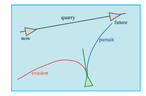


Figure 8: Pursuit and Evasion (Reynolds, 1999)

Arrival Force

Arrival force is a negative force to velocity and only gets applied when the boid is within the target's **arrival distance**. It effectively shrinks down velocity to zero as the boid gets closer to the target.

8 Obstacle avoidance

Obstacle avoidance gives boids the ability to navigate through an environment without colliding with set obstacles. It is important to make a distinction between collision detection and collision avoidance, the former detecting the very thing that the latter tries to prevent. Reynolds' proposed method, while simple in concept, does make sense in the real world as well, especially when thinking about large, convex obstacles, which a flying object would likely avoid by a large margin. His method relies thus on considering both the boid and the obstacle as their respective bounding spheres.

The obstacle avoidance behaviour tries to maintain a cylindrical volume of arbitrary length in front of the boid (therefore in the direction of velocity) free of intersections with any obstacle bounding sphere. The radius of the bounding sphere of the obstacle is calculated from the input **bounding box size** parameter, while the position of the obstacle is retrieved prom the passed in **matrix**. *(Fig. 9)*

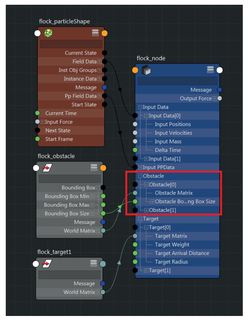


Figure 9: Obstacle inputs on the flock node.
Note: Obstacle is a plug array, taking in as many obstacles as necessary.

The collision avoidance process is composed of the following steps (*illustrated in Fig. 10*):

1. toObstacle = $position_{obstacle}$ - $position_{boid}$

Vector going from the boid's bounding box centre tothe obstacle's bounding box centre.

2. vector up = v x toObstacle

Vector perpendicular to both v and toObstacle.

3. side = v x up

Vector in the same plane as v and toObstacle.

4. $Proj_{side.normal()}$toObstacle

When we project vector toObstacle onto the unit vector side, we get the length of the vector toObstacle projected on the plane defined by up and side, which is the plane perpendicular to the boid's velocity. This is the length we need for the obstacle avoidance.

5. $Proj_{v.normal()}$toObstacle

When we project the vector toObstacle onto the unit vector v, we get the length of the vector toObstacle projected along vector v. In other words, we get the distance to the collision object in v, which helps us determine which object is the first to collide if there are multiple objects colliding.

As soon as this data is gathered, we can check whether the *length of the side vector is greater than the radius of the obstacle bounding sphere plus the the radius of the boid sphere*, in order to determine if collision is prone to happen. And if so, we simply apply acceleration as the direction and magnitude that would takes us out of the collision course the fastest:

*avoidanceForce = -(side.normal()) * maxAccel*

where *maxAccel* is a scalar that caps acceleration. Even though we've introduced it only now, it operates as a global limiter as well, preventing boids from accelerating in an unrealistically fast way. It is often coupled with *maxSpeed*.

9 Summing up forces

We now have enough forces acting on our boids to get some interesting results. Only one thing left to do and that is to sum all forces, make sure to clamp very high values and send it off to Maya to integrate everything over time.

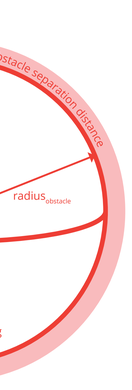


Figure 10: Obstacle avoidance calculation vectors

Figure 11: Possible trajectory of a boid en-route to its target, avoiding collision with an object's bounding sphere.

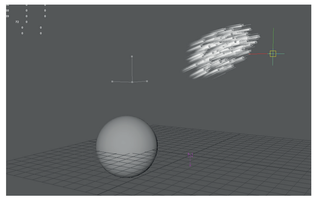
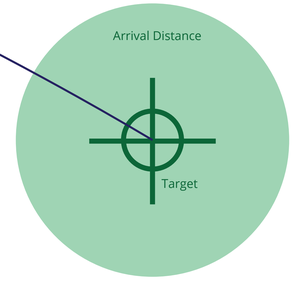


Figure 12: Interactive Target Following



10 Other implementation details

At the time of writing, all the discussed behaviours with the exception of pursuit and evasion have been implemented. While beneficial for the flocking accuracy, they are actually variations of Seek/Flee, which have been implemented. The node was written in Python and tested with Maya 2016. Time constraints made it impossible to add an user interface, but all the parameters can be controlled via the Channel Box / Attribute Editor. Future development would not only see a more complete set of behaviours implemented, but also a translation to C++, which would make for a dramatic performance increase when dealing with a higher number of boids.

Further research

The possibilities with flocking systems are far beyond what has been discussed and implemented here. Reynolds himself lists many other types of steering and cognitive behaviours that would have been extremely interesting to study and implement: path following, orientation, leader following, wander, flow following etc. And of course, over the years many developers implemented and extended these behaviours, a simple Google search providing too many examples to mention. Although in the context closely relevant to this poster, a prime example would be **techtoast.co.uk**, a company who offers a commercial flocking system plugin for Maya.

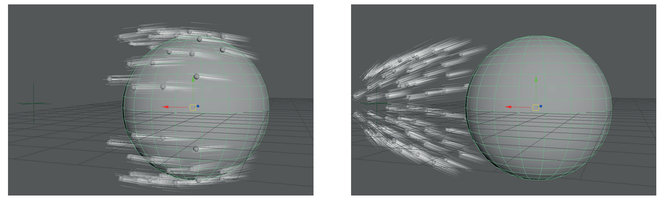


Figure 13: Obstacle avoidance tests