A New Method of Simulating Swarms and Herds Using PSO and Other Additions: A Social Behavior Approach

Ammar Alnahhas[1], Aghyad Al-Kabbani[2], Eyad Alshami[3], Ahmad Alkhous[4]
Damascus University - Faculty of Information Technology Engineering
Syrian Arab Republic
eng.a.alnahhas@gmail.com
aghkab@gmail.com
eyad.dedo@gmail.com
iteahmad@rocketmail.com

Journal of Digital Information Management

ABSTRACT: *We present a new method for simulating swarms and herds of animals using a modified version of particle swarm optimization algorithm [1], a robust collision detection and resolution system and a mechanism for restricting velocity and angular acceleration to make the movement natural as possible.*

Our model is designed to simulate the search for food butitcan be used for locating safe spots and predator-prey interaction by changing the fitness function, in which each swarm member interacts with the environment and the other swarm members to produce a movement that leads the member and the swarm change their position to a more fitting one.

Categories and Subject Descriptors
[D.4.6 Security and Protection]: [K.6.5 Security and Protection]; **[I.2.11 Distributed Artificial Intelligence]:** Intelligent agents

General Terms: Swarm Optimization, Intelligent Agents

Keywords: Swarm Simulation, Particle Swarm Optimization

Received: 12 May 2016, Reviesd 19 June 2016, Accepted 25 June 2016

DOI: 10.6025/jdim/2016/14/5/331-337

1. Overview

A swarm is a large number of homogenous, simple agents interacting locally among themselves, and their environment, with no central control to allow a global interesting behavior to emerge. This global behavior helps the swarm achieving a global goal such as searching for food or escaping a predator. As each member in the swarm acts independently (i.e.: there is no central control), with a separate memory and processing power , and the judgment of each member is based on its information about the world and the information provided by other members, we can say that the behavior of the swarm is the natural version of peer to peer distributed systems, every member benefits from the resulting collective intelligence and tries to find a solution to the search/escape problem, leading to better adaption with the environment, as adapting with the environment is one of the definitions of intelligence, this behavior is considered as an intelligent one.

2. What makes a swarm?

In his paper[2], Reynolds states that a simulated swarm must satisfy the following behaviors:

a. Collision Avoidance: The swarm members always maintain prudent separation from their neighbors.

b. Velocity Matching: The swarm members head in approximately the same direction at approximately the same speed.

c. Swarm centering: The swarm members stay close to each other.

We have implemented the first behavior as a robust collision detection and resolution system as discussed in detail (sections 6 and 7).

The second and third behaviors were guaranteed by the use of the PSO algorithm which also works as a food search routine (section 3).

In addition, to ensure a natural movement as possible, the restriction of both velocity and angular acceleration is necessary (sections 4 and 5).

3. Particle Swarm Optimization algorithm

Particle Swarm Optimization (PSO) algorithm was originally based on swarm behavior [1, 3], this was our reason to implement such an algorithm as the main motive for swarms to group and search for an optimal location with a fitness criterion such as food supply, lack of predators or presence of other swarms.

However, plain PSO[1] was not realistic to simulate a natural swarm; the resulting movement was much like jumps, because plain PSO updates the velocity of a swarm member directly without affecting the acceleration first, as shown in the following algorithm:

```
PSO()
Begin
        While (fitness(SwarmBestPos)< value)
        Begin
                For each  member x
                        If(fitness(x.currentPosition)>fitness(x.bestPosition))
                                x.bestPostion=x.currentPosition;
        SwarmBestPos=Max(fitness(all current positions));
        For each member x
        Begin
        x.velocity=
                K*x.velocity + c1* r1* (x.bestPosition – x.currentPosition) +
                c2 *r2 *( SwarmBestPos – x.currentPosition);
                x.currentPostion=x.velocity +x.currentPostion;
                End
        End
End.
```

Where:

x. velocity is the velocity vector of a swarm member x.

x. current Position: is the position vector of a swarm member x.

x. Best Position: is the position a swarm member x has passed through.

Swarm Best Pos: is the swarm member that has the best position so far, with respect to the whole swarm.

K is a real factor representing a swarm member's inertia, in other word its " trust " in its current position.

c_1 is a real factor representing the " trust " of a swarm member in its best position so far.

c_2 is a real factor representing the " trust " of a swarm member in its best position the swarm has ever passed through

r_1, r_2 are random real values.

However, this strategy of directly updating the velocity makes the swarm movement inconsistent and much closer to pulses than to natural movement.

We have implemented PSO (as shown in**[4]**) to give a force to each swarm member instead of directly modifying the velocity, on each iteration the vector sum of the forces affecting each swarm member is calculated, and then the acceleration, velocity and position are computed, as in the following equations:

To simplify the equations we assumed that the force of the PSO motive is the only present force, time frame and masses are set to unity.

Update the acceleration, velocity and then the position of each particle according to the following equations:

1. $a_{(t+1)}[x] = K*a_{(t)}[x] + c_1*r_1*\ (BestP[x] - p_{(t)}[x]) + c_2*r_2*$ 1. (BestP[GBest] – $p_{(t)}[x]$)

2. $v_{(t+1)}[x] = v_{(t)}[x] + a_{(t+1)}[x]$

3. $p_{(t+1)}[x] = p_{(t)}[x] + v_{(t+1)}[x]$

Another modification was to lower the inertia weight (*K* in the PSO formula) of the swarm members to a value close but not equal to zero (in the range of [0.01, 0.1]), our experiments have shown that this ensures smoothness and freedom of movement, the values of the other two coefficients (c_1 and c_2) were set to 1.49445 (as mentioned in**[5]**) which yields a realistic movement while maintaining the effectiveness of the search.

4. Velocity restriction

As in the real world swarms, all animals have a maximum velocity, to achieve that in our model the velocity was restricted to be in a range of the virtual-world bounds [-m*a,m*a] inclusive for each of the world's coordinate axis, where m is parameter that represent the degree of freedom, we have found out that 1/50 is a good value for m in practice.

5. Angular velocity restrictions

All animals have a limit on their ability to change their movement direction, e.g. a real swarm member needs to turn back little by little, to illustrate this property we have implemented a simple technique; which is as follows:

• If the angle between the current velocity and the new velocity θ is less than the maximum allowed change of movement direction (MAXANG) then the current velocity is left as it is.

• Otherwise, the current velocity is updated toward the old velocity with angle of (θ -MAXANG) according to the perpendicular vector to the velocities (current and old) plane, using a quaternion initialized with the previous parameters (the angle and the perpendicular vector) to generate the rotation matrix as follows:

α is the rotation angle

$$\alpha = \theta - MAXANG$$

A quaternion (where v is the perpendicular vector to the velocities plane)

$$q = \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \cos\frac{\alpha}{2} \\ v_x \sin\frac{\alpha}{2} \\ v_y \sin\frac{\alpha}{2} \\ v_z \sin\frac{\alpha}{2} \end{bmatrix}$$

The rotation matrix is

$$R = \begin{bmatrix} 1-(2y^2+2z^2) & 2xy+2zw & 2xz-2yw \\ 2xy-2zw & 1-(2x^2+2z^2) & 2yz+2xw \\ 2xz+2yw & 2yz-2xw & 1-(2x^2+2y^2) \end{bmatrix}$$

The rotation is applied by multiplying the rotation matrix by the new velocity vector to produce a new realistic velocity vector [6].

6. Collision detection

Many interactive animation systems, such as games or simulations, require large numbers of virtual entities which are moving and interacting with each other. In these applications, we cannot predict in advance how the user or the entities will behave, so we must create the animation as we watch it, i.e. in real-time.

This means that the image must be redrawn at least 10 times per second, although for true real-time performance 60 frames per second (f.p.s) may be required [7].

In our application, we had to make a virtual reality system, which can simulate thousands of moving entities; Crowd simulations; Flocks of birds or other organisms. Anything, in fact, where you have large numbers of entities moving around a virtual world in real-time.

There are many bottlenecks in such systems. Depending on the level of realism required, rendering and motion synthesis algorithms require a large amount of processing power.

Increasing the computational power, add hardware accelerators, and develop parallel algorithms that may be implemented on multiple processors will postpone the problem rather than eliminate it, and such computational power will not be available to a wide range of users.

An additional challenge is maintaining a constant frame rate. The time taken to render a given scene is dependent on the current level of complexity. Some frames may require only one object to be rendered, whereas a sudden change of view may cause many more interacting objects to be visible in subsequent frames.

In a computer world, there is nothing to stop geometrically modelled objects from simply floating through each other. A Collision Handling system is necessary to enforce solidness, and ensure that entities behave as expected when they come into contact.

In this section, we address the problem of collision detection, and demonstrate how interactions between objects may be handled in real-time, reducing the variability of the frame-rate by interrupting processing when required.

Traditional collision detection algorithms require a large amount of geometrical intersection tests, checking if any of the polygons used to model the surface of one entity touch or penetrate any polygon on the other entity. To improve the efficiency of such algorithms, spatial representations of entities were generated, to localize the areas where the actual collision occurred, these include Sphere-Trees.

a. The implementation

We have used a data structure that is called sphere-tree that is well suited to geometric learning tasks.

• Sphere-Trees tune themselves to the structure of the represented data

• support dynamic insertions and deletions

• have good average-case efficiency

• deal well with high-dimensional entities

• Are easy to implement.

A Sphere-Tree is a complete binary tree in which

• Asphere is associated with each node in such a way that an interior node's sphere is the smallest that contains the spheres of its children.

• The leaves of the tree hold the information relevant to the application.

• The interior nodes are used only to guide efficient search through the leaf structures.

The benefits of using sphere-trees rather than MBB[8]areas follows:

• The ease of implementation

• The ease of calculation, overlapping can be detected using the distance between the two spheres and comparing it to the sum of their radius.

• The radius represent the amount of awareness given to the entity

Our implementation:

• First we have a sphere object which has a variables holding information on the sphere such as the center of the sphere(which is the center of the object bounded by the sphere) and the radius of the sphere, these objects represent the leaves in our bounding volume tree.

• Inner_Sphere object which have a radius and a center just like sphereobjects, and has two sphere pointers representing the two children of this inner sphere.

```
For every entity _i in the swarm do:
Begin for
        creatInitBoundingSphere (_i);
End for;
```

We have used a bottom-up approach for building our Sphere-Tree structure [9]**,** our implementation starts with an initial step with bounding every entity with a sphere, so we end up with spheres representing the entities under test and these spheres will be the leaves in our bounding volume tree:

The later steps are divided into two phases the first phase is building the tree:

• Calculating the distance between every two spheres

• Take the pair of spheres with the least distance

• Bound this pair with a bounding sphere and make this sphere a parent node for the two entities inside it.

• Consider this new bounding sphere as an a new sphere

• Repeat the first step.

```
For every sphere _s in the swarm do:
Begin for
        calculateDistances (-s);
        Sphere newBoundingSphere =
                creatBoundingSphere (-s);
        addBoundingSphereToSwarm (Sphere);

End for;
```

Figure one shows an example of this algorithm:

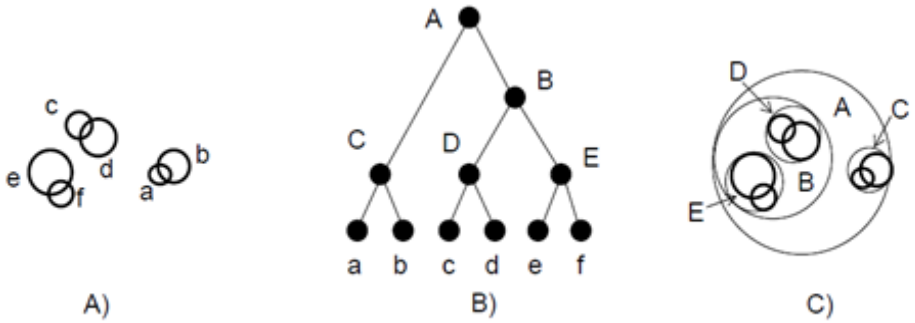


Figure1. Example of building Sphere-tree for a school of fish.

```
checkCollision (Sphere left, Sphere right)
Begin
        If (checkSpheresCollision(left, right)== true)
        begin if
                If (checkSpheresCollision(left.left, right) == true);
                        checkCollison (left.left, right);
        else
                        checkCollison (left.right, right);
        End if;
End;
```

second sphere, if true then

• Check which of the two children of the second sphere collide with that first child

• Else, check for the second child of the first sphere.

We repeat these steps until we get to the leaves that collide with each other, and then we can solve the collisions.

The main problem in this implementation is the time consumed in calculating the distances between the spheres, building the tree and solve the collisions, especially when having a large number of entities moving around, which can take a long time depending on the actual positions of the entities in each frame.

Because of the unpredictable and free movement of the

The second phase is to detect the collision, if any, between the spheres in the tree, by calculating the difference between, the sum of their radius and the distance between them.
Starting from the root of the tree:

• Check for a collision between the two children of the sphere under consideration.

• If the two spheres collide then there is a chance that the children of these spheres will collide

• Check if a first child of the first sphere collide with the

entities in our applications, some frames will have a lot of entities moving and gathering in one certain position so we will have a lot of collisions, and in some frames each entity positioned apart from the others, So the number of collisions are small, this will affect the frame rate, therefore we have adapted an interrupted collision detection algorithm. So we give a constant amount of time for each frame, and if the collision detection phases take more time, the detection will be terminated, the result of the detection phases will be solved, and other undetected collisions will not be treated as collisions. This is a tradeoff between the speed and the accuracy, and we choose the speed.

7. Collision resolution

To simulate the natural tendency to avoid collision between swarm members, two avoidance scenarios were presented,

in both of them each one of the two particles is given a force composed of a magnitude and an orientation defined as follows:

• If the angle between the velocity victors of the two particles is greater than 90 degrees: The force of each of the particles has the orientation of its particle's velocity rotated MAXANG degrees according to the perpendicular line to the plane defined by the two particles' velocities, both vectors' orientations are rotated in the same direction with the addition of a small random rotation value to the rotating angle to illustrate a natural reaction [figure 2-a].

• Else (the angle between the velocities of the two particles is not greater than 90 degrees): In this scenario, the forces orientations are similar to result of an elastic collision between two spheres [figure 2-b].

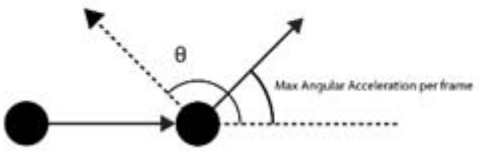


Figure 2-a

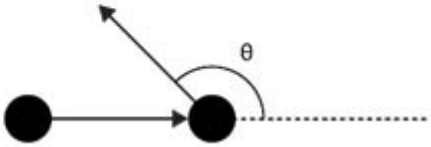


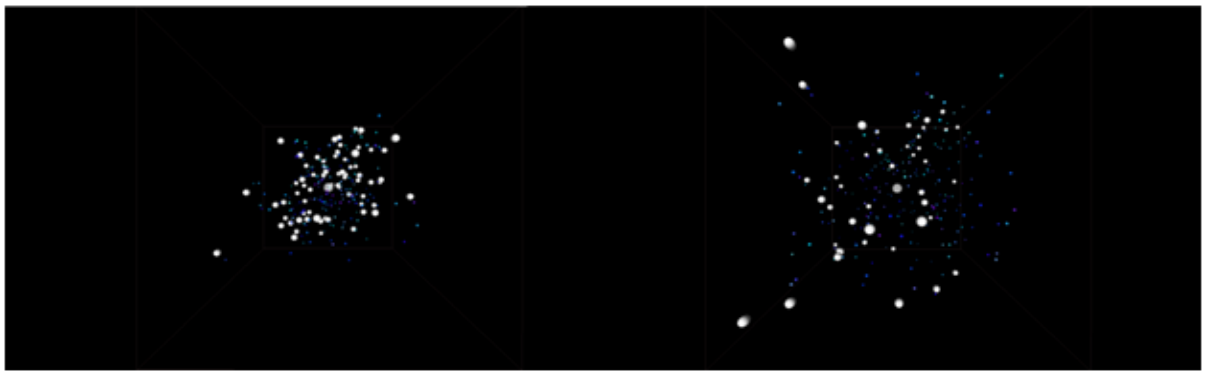
Figure 2-b

In both scenarios, the magnitude of the evasion forces are made large enough to override the effect of the main motive force resulting from the PSO algorithm, and the magnitude is reduced in a linear fashion, so that the effect of the evasion force is prolonged to cover multiple frames and to mimic the short-term memory of a swarm member.

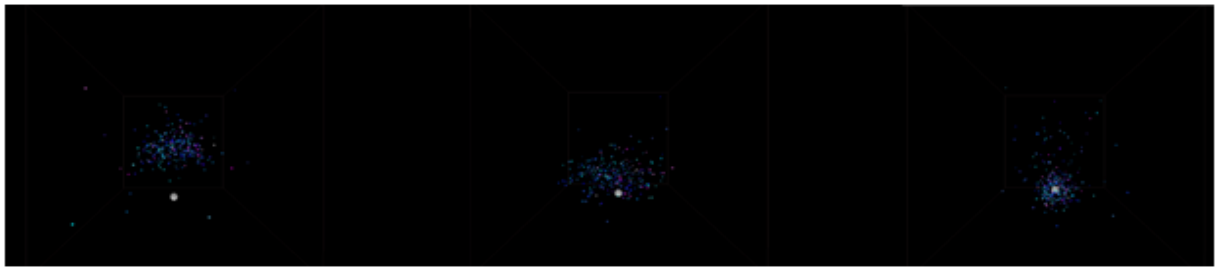
8. Conclusion

In this paper, we have assembled a new method to simulate a swarm of living creatures. The main motives in this behavioral model were a modified version of PSO algorithm, a collision avoidance system and restrictions on velocity (linear and angular), we have implemented and tested the resulting output to be realistic as possible with regards to behavior and motives (as shown in the screen shots), we hope to provide optimized working code for public use in the near future.

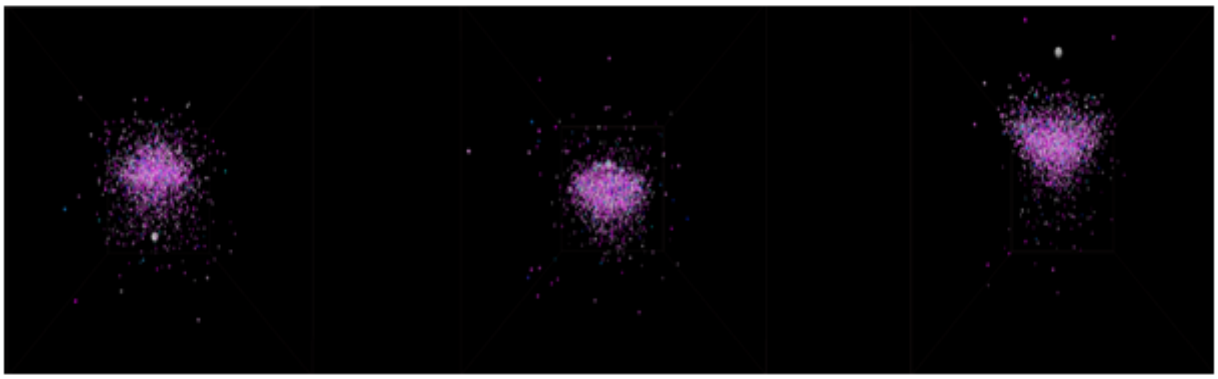
The following screenshots are samples of our implementation, each screenshot has a not bellow:



These two images represent the collision detection phase, where we are pausing the entities which are considered in a collision situation and drawing a white sphere around them.



In these images, we show how the swarm is chasing the target (the big white sphere) from left to right.



In these images, we have a swarm that consists of 3000 entities chasing the goal (big white sphere)

References:

[1] . Eberhart, R. C., Kennedy, j (1995). A new optimizer using particle swarm theory, *In:* Proceedings of the sixth international symposium on micro machine and human science, 1995, p. 39-43.

[2] Craig, W., Flocks, R (1987). Herds and Schools. A Distributed Behavioral Model, *In:* ACM SIGGRAPH 1987 Conference Proceedings, Anaheim, California (July 1987), 1987.

[3] Kennedy, J.Kennedy, J. F., Eberhart, R. C Shi, Y. (2001). *Swarm intelligence*: Morgan Kaufmann.

[4] Blackwell, T. (2007). Particle swarm optimization in dynamic environments,*In: Evolutionary computation in dynamic and uncertain environments*, ed: Springer, 2007, p. 29-49.

[5] Eberhart, R. C., Shi, Y. (2000). Comparing inertia weights and constriction factors in particle swarm optimization, *In:* Evolutionary Computation, 2000. Proceedings of the 2000 Congress on, 2000, p. 84-88.

[6] Millington, I. *(2007).Game physics engine development*: Morgan Kaufmann Publishers Amsterdam, 2007.

[7] Dingliana, C. (1999). real-time collision detection and response using sphere-trees, Technical report, Image Synthesis Group-Trinity College Dublin.

[8] Ericson, C. (2004). *Real-time collision detection*: CRC Press, 2004.

[9] Omohundro, S. M. (1989). *Five balltree construction algorithms*: International Computer Science Institute Berkeley, 1989.