Implementation of Flocking Behavior

Without seperation rule

Hiran Harilal
Department of Computer Science
University of Hertfordshire
Hatfield, United Kingdom
hh16aca@herts.ac.uk

Abstract— **The Flocking behavior of multiple agents is a behavior that can be observed and related to certain phenomena in nature such as flocking birds and schooling fish. Recent literature has shown us the possibility of flocking even when a small percentage of these agents are updated of the desired position and velocity of the object or information. However, the problem still arises when it is to be decided as to which agents should be updated to have the ability to detect the information. This Flocking behavior can be emulated in computers by generating and combining the three simple rules of alignment, cohesion and separation. In this paper, we try to replicate the flocking behavior by taking the alignment and cohesion rule into consideration. It is mandatory for the agents to maintain a suitable distance to avoid collision. Therefore, the separation rule is of major importance, but here we try to simulate the flocking behavior without the separation rule. Experiments are conducted to test the accuracy of the flocking behavior.**

Keywords—Flocking; Multi-agent; Emergent-behavior; Simulation

I. INTRODUCTION

In nature, certain organisms exhibit collective behavior, which may be on a microscopic or macroscopic level. Some of these behaviors include flocking birds and schooling fish. The patterns exhibited by these organisms can be simulated by creating and combining certain rules. This is known as the Flocking Behavior of multiple agents and this concept is used in simulation games to create lifelike group movement. Recent literature has shown us the possibility of flocking even when a small percentage of these agents are updated of the desired position and velocity of the object or information. However, the problem still arises when it is to be decided as to which agents should be updated to have the ability to detect the information. This Flocking Behavior can be emulated in computers by generating and combining the three simple rules of alignment, cohesion and separation. In this paper, we try to replicate the flocking behavior by taking the alignment and cohesion rule into consideration hence omitting the separation rule. However, the separation rule is of utmost importance to determine the suitable distance the agents need to maintain to avoid the collision. But in this paper, we try to simulate the flocking behavior without the separation rule. As part of this research, experiments are conducted to test the accuracy of the flocking behavior. With the advent of powerful modern computers, the production of complex and quality animations has become a lot simpler. However, the management of such complex large-scale character animations has major potential in terms of research. Simple scenes with a few characters can still use traditional key framed sequences. However, there is further complexity to this problem when the film industry demands scenes comprising of thousands of characters. Over the years, the commercial animation packages have been developed to permit simulation of flocking behaviors of large groups. Some of these are Maya [Ali03] and Softimage [Sof03]. There are other standalone packages such as Massive [Mas03] and Behavior [Sof03]. In the field of Robotics, flocking can be contemplated as a particular case of coordination of multiple robots that related coverage, sensing, exploration and rescue purposes.

II. THE THREE RULES

In 1986, Reynolds introduced three heuristic rules that led to creation of the first computer animation of flocking [1]. Here are three quotes from [1] that describe these rules:

1) "Flock Centering: attempt to stay close to nearby flockmates,"
2) "Obstacle Avoidance: avoid collisions with nearby flockmates,"
3) "Velocity Matching: attempt to match velocity with nearby flockmates."

In the natural world, organisms exhibit certain behaviours when traveling in groups. This phenomenon, also known as *flocking*, occurs at both microscopic scales (bacteria) and macroscopic scales (fish). Using computers, these patterns can be simulated by creating simple rules and combining them. This is known as *emergent behaviour*, and can be used in games to simulate chaotic or life-like group movement. Before we begin, here's some terminology I'll be using:

Alignment

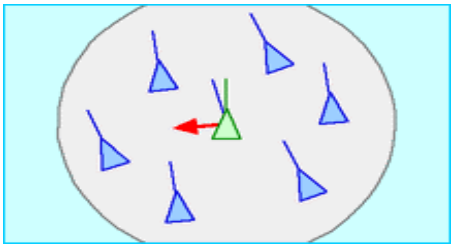


figure 1

- Pushes boids apart to keep them from crashing into each other by maintaining distance from nearby flock mates.
- Each boid considers its distance to other flock mates in its neighborhood and applies a repulsive force in the opposite direction, scaled by the inverse of the distance

Cohesion

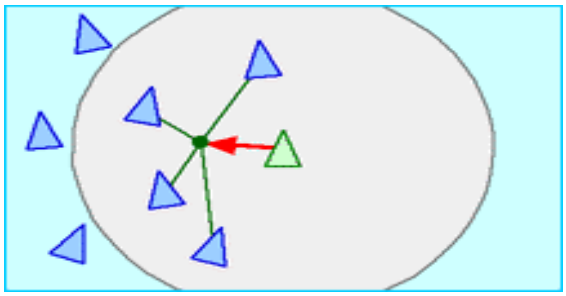


figure 2

- Keeps boids together as a group.
- Each boid moves in the direction of the average position of its neighbours.
- Compute the direction to the average position of local flock mates and steer in that direction

Separation

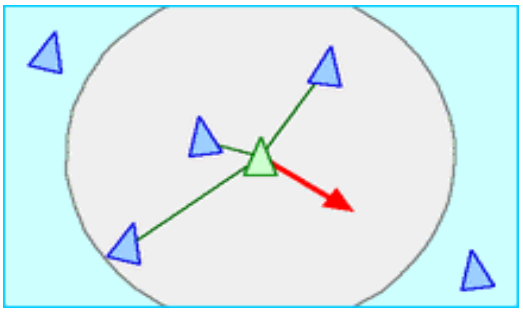


figure 3

- Drives boids to head in the same direction with similar velocities (velocity matching).
- Calculate average velocity of flock mates in neighbourhood and steer towards that velocity

III. OBJECTIVE

The first modelling of the flocking behaviour was done by Craig Reynolds in 1987 by following the three simple rules. It is evident that any mistake in any one of the behaviour can result in a crash. This main objective of this paper is to omit the separation rule logic from the whole algorithm and observe the working of the boids. Experiments are conducted to prove the objective. The main advantage of this procedure is that it reduces the code length and show more performance. This type of modelling was possible by using some of the pre-defined features of the Unity application. Unity is an application used for game development. The usage of different pre-defined physics properties we can replicate many of these AI behaviours without spending more time on programming it.

IV. IMPLEMENTATION

Unity is a cross-platform game engine developed by Unity Technologies, which is primarily used to develop video games and simulations for computers, consoles and mobile devices. First announced only for OS X, at Apple's Worldwide Developers Conference in 2005, it has since been extended to target 27 platforms. Unity is an all-purpose game engine that supports 2D and 3D graphics, drag and drops functionality and scripting through C#.

figure 4: Unity logo

A. Environment

Graphic objects in 2D are known as Sprites. Sprites are essentially just standard textures but there are special techniques for combining and managing sprite textures for efficiency and convenience during development. Unity provides a built-in Sprite Editor to let you extract sprite graphics from a larger image. This allows you to edit a number of component images within a single texture in your image editor. These sprites are used here to generate the boids. To have convincing physical behaviour, an object in a game must accelerate correctly and be affected by collisions, gravity and other forces. Unity's built-in physics engines provide components that handle the physical simulation for you. With just a few parameter settings, you can create objects that

behave passively in a realistic way (ie, they will be moved by collisions and falls but will not start moving by themselves). By controlling the physics from scripts, you can give an object the dynamics of the boids.

B. Boids

The boids/agents used in the computer models are created using the Sprite feature. Sprites are 2D game objects that improve efficiency and increase convenience by combining and managing texture during the process of development. For this project, I have developed a sprite in the shape of a triangle. This is done using the sprite creator. The boids that are created (also called prefabs) are dropped in the manager (game changer) where they can be controlled. The prefabs can be defined in such a way that we can give the number of clones required for simulation. After this process, we can affix the physics properties to the boids. In order for them to have an invisible colliding shield around them, we attach the 2D physics property known as the circle collider.

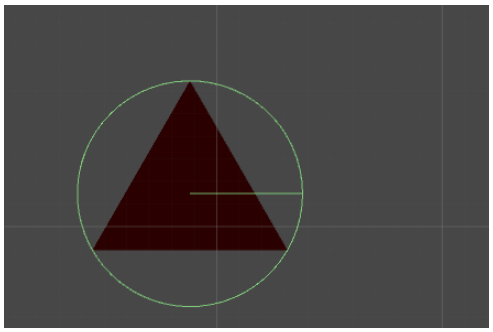


figure 5 Generated Boid

C. Clones

Only one boid has been generated for the simulation. This has been done, as mentioned earlier, by using sprite. After this has been done, we construct the required number of clones from the boid. The number of boids created is taken as an input from the user. To make the simulation more authentic, it is important that each of the boids have their own will and behavior. Randomness is proportionated to reality here. That is, the more random the motions, the higher the realistic factor is. Therefore, to increase this randomness, a 2D vector has been defined, and this takes random values from a range, and declares it for the boids. When the input for the number of boids is entered, the generated clones start motions from random positions.

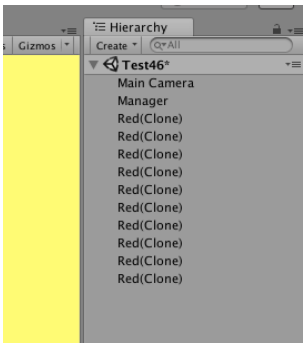


Figure 6 Generated Clones

The above figures depict the generated clones of the red(Prefab). red is the name of the boid. The main advantage of creating clones is that we can track the position of each boid. All of the properties of the boids can be acquired by simply clicking on the clone.
The below figure shows the properties of the clones

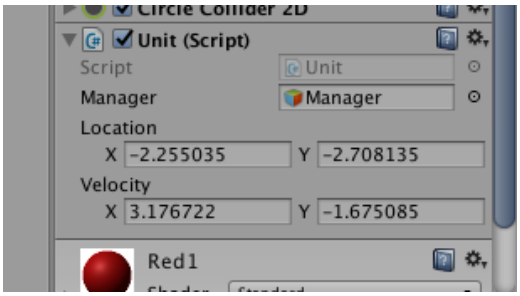


figure 7 Varying properties of each boid

D. Working Principle

Before getting into the working principle of the code, it is necessary to have some prerequisite knowledge on the features of the tool that has enabled us to implement this project. Supporting multiple languages has been facilitated thanks to the combination of the brilliance of Unity's architecture and Mono's implementation of .NET's IL. However, this is not free to test, support and record the usage of multiple languages in game development on the Unity engine. The main features include:

- 2D environment of Unity
- Ability to add physics components.

- **Physics Collider**

For a game object to have any physical presence on the scene, it should have a defined physical shape and must interact with the relevant physics engine. For this, it must use one of the collider components. Unity has 2 physics engines, one for 3D and the other for 2D. When using the 2D physics engine, a collider 2D component must be used. Each of the 2D colliders is optimized for a particular shape. The circle collider 2D works with round circular objects. With normal 2D

physics collisions, the 2D physics engine will prevent 2D colliders from passing through each other. The material property is a reference to the physics material 2D being used by the collider 2D and this can be none. One way of attaching a collider 2D component is to use the Add Component button and select Physics 2D and select a Collider 2D Component. When a collider 2D component is attached Unity will try to size the collider 2D to the sprite. If the size and shape of the collider 2D are undesirable then the shape can be edited by either changing the values in the inspector or changing the collider 2D itself in the scene view. To edit a collider 2D in the scene view hold down the shift key. This will display handles on the 2D collider's gizmo in the scene view. These handles are drag-able. A game object does not need a rigid-bodied 2D to use a collider 2D. A common pattern for best performance would be to attach collider 2D components but not rigid body components to all of the static, or non-moving game objects that need to interact with 2D physics. And attach a collider 2D and a rigid body 2D to the dynamic, or moving, game object that needs to interact with the scene.

- **Cohesion**

goal ← (0,0);
neighbours ← getNeighbours(boid); **foreach** *nBoid in neighbours* **do**

goal ← goal + positionOf(nBoid); **end**

goal ← goal / neighbours.size(); steerThoward(goal, boid);

Steer to move toward the average position of local flockmates. Cohesion is the rule that keeps the flock together, without it there would not be any flocking at all. Cohesion rule shows how this algorithm could be implemented, it finds the average position of the neighbourhood boids and tries to move the boid towards it.

- **Alignment**

dCourse ← 0;
dVelocity ← 0;
neighbours ← getNeighbours(boid); **foreach** *nBoid in neighbours* **do**

dCourse ← dCourse + getCourse(nBoid) - getCourse(boid);

dVelocity ← dVelocity + getVelocity(nBoid) - getVelocity(boid); **end**

dCourse ← dCourse / neighbours.size();

dVelocity ← dVelocity / neighbours.size();

boid.addCourse(dCourse);

boid.addVelocity(dVelocity);

steer towards the average heading of local flockmates. This rule tries to make the boids mimic each other's course and speed. If this rule was not used the boids would bounce around a lot and not form the beautiful flocking patterns that can be seen in real flocks.

Take the note from above that there is no mention of the separation rule. Instead, in light of this project, we use physics colliders in place of the separation rule. Once the boid is created, we add a circular invisible shield around it which is the physics component and this shield is called the Physics Collider. They function as the boundary for the boids so even when there is the collision, it cannot be seen as it is the collider surface that has been colliding. Once the code has been run, we can observe that the boids maintain a consistent distance between each other. The Circle collider can be seen in figure 5.

V. EXPERIMENTAL SETUP

Experiment:
In this experiment, boids have been created and the simulation is done in Unity. The observed values are then taken for further calculation and analysis on Excel.
Aim:
To check the working of the Flocking Behavior of the created boids using just the Alignment and Cohesion barring the Separation rule.
Objective:
In this project, there are two main types of Flocking Behavior we focus on; hence we divide this experiment into two parts, which gives us two main objectives:

1. To check whether the boids maintain a consistent distance between them while flocking
2. To check whether the boids exhibit the same Flocking behavior when a goal position is defined.

Procedure:

1. Maintaining a consistent distance between them

Our main aim through this procedure is to find out if the distance between two boids remains the same throughout even if there is a change in the parameter. In this case we take the varying parameter as velocity. The location and velocity of a boid can be obtained by clicking the related clone.
Random values of velocity are taken and any values from 0 to 5 can be considered. To test our theory, we need to find out if the boids are equidistant from each other. We do this by taking any two boids and noting their position. The positions are given in the cartesian form with the coordinates x and y, (x, y). To find the distance between two coordinates we use the formula,

$$\sqrt{(x_1\text{-}x_2)^2 + (y_1\text{-}y_2)^2}$$

Where:
(x_1, y_1) is the position of boid 1
(x_2, y_2) is the position of boid 2
These values are noted alongside the random values of velocities and are taken down and further calculated on Excel.

2. Defining a Goal Position

The procedure under this objective remains the same as the above mentioned with the additional factor of setting a goal. In this experiment, the goal has been set to (10,2). The boids are observed to follow the goal and are expected to maintain an equidistance from each other. The values are noted down and we follow the same procedure by taking the random values for the velocities and noting down their position.

VI. Observations and Findings

As the experiment was divided into 2 parts, there are 2 sets of observations:

1. Velocity Vs. Distance – Equidistance:

In the first part of the experiment, we can observe that as the velocity increases, the boid starts becoming unstable. The force is kept constant at this stage.

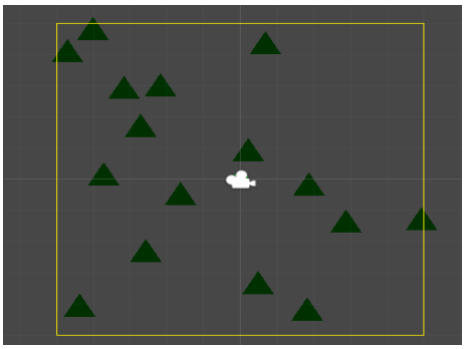


Figure 8 Unstable boids

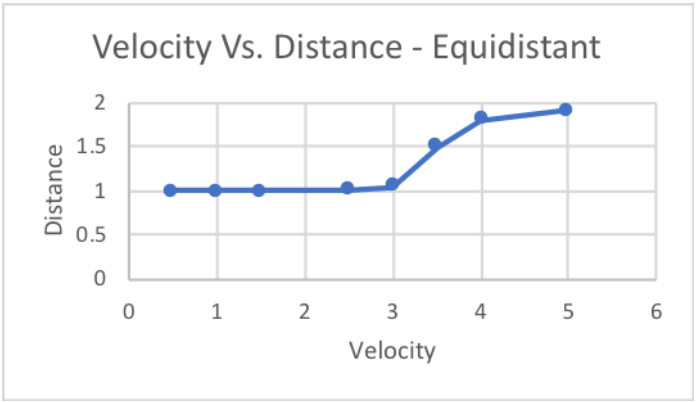
The location of the boids can be obtained by clicking in the clone. These values are noted and arranged in the form a table. The table below shows us how much of a variation is there in the distance between the boids when the velocity increases.

Position Of Boid 1		Position Of Boid 2		Distance	Velocity used
x	y	x	y		
-0.76	-2.013	-0.971	-0.09	1.9	5
-0.7592	-2.108	0.5818	-0.9647	1.8	4
-0.4148	-1.0436	0.497	-2.2446	1.5	3.5
1.2354	1.9677	0.2126	1.7681	1.04	3
1.5961	0.7485	0.5971	0.8465	1.004	2.5
1.0108	-0.0615	1.7141	-0.771	0.998	1.5
-0.3975	0.766	0.015	-0.1409	0.996	1
-1.3354	1.1036	-2.2569	0.7247	0.996	0.5
			AVG	1.279	

Table 1

We observe that as we increase the velocity the distance between the boids increases by a very small amount when the velocity increases. However, this increase in distance is not a very significant one. We further observe that the boids start becoming unstable when the velocity increases. The average distance between the boids has been calculated as 1.2.

During the simulation, we can see that the boids exhibit flocking.
The graph below gives us a clearer picture of the variation of distance with velocity. We see that there is a slight variation in the graph for the higher velocities. This is due to the instable boids as high velocity



Graph 1 Plotting the Equidistance

The graph shows almost a straight line showing that there is not much dependency on the velocity and the distance between the boids.

2. Velocity Vs. Distance – Goal seeking:

In the second part, we declare a goal position for the flock. In this experiment, we have taken the goal position as (10,2). This is done to check if they maintain an equidistance while moving towards the goal as well. The below given figure shows us how the goal is set and what the positions of the boids are:

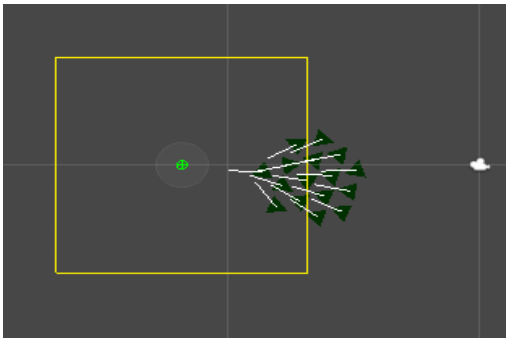


Figure 9 Moving towards Goal Position

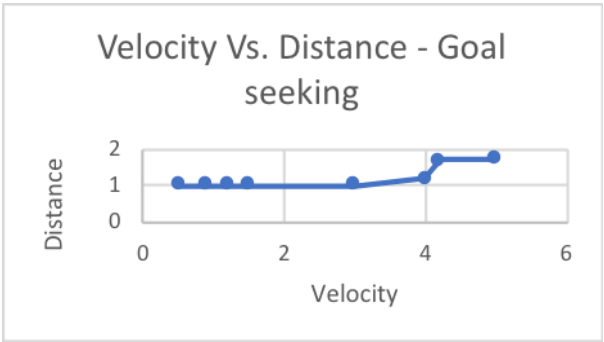
The table below compares the variation on distance as the velocity is increased.

Position of boid 1		Position of boid 2		Velocity	Distance
x	y	x	y		
7.5201	2.9687	7.3278	1.9865	0.5	1
10.1673	2.9519	10.3608	3.9329	0.9	1
10.9209	2.2927	11.114	3.2738	1.2	1
8.2735	2.3114	9.026	1.6537	1.5	1
9.2195	2.6348	9.4123	3.6159	3	1
8.9	0.77	8.452	1.8706	4	1.19
8.9118	0.6604	9.9748	1.9706	4.2	1.68
6.5751	2.6416	7.1368	1.0053	5	1.73
				AVG	1.2

Table 2

Again, in this case we observe that as the velocity increases there is very less change in the distance. There is very little difference in the values. The average distance in this case also is 1.2.

With the help of the graph we can determine that at lower velocities the distance is 1. This value increases due to the high instability of the boids at high velocities, hence explaining the slight variation towards the end.



Graph 2 Plotting the Distance – Goal position

VII. CONCLUSION

From the above observations, we can conclude that the boids exhibit flocking behavior even without the inclusion of the separation rule. While performing the experiments, we could take notice that the boids always maintained a constant distance between them. This was possible due to the physics features of Unity. The exclusion of the separation rule enhances the performance and has reduced the length of the code to a fair amount. Varying the other parameter have shown differences in the simulation of flocking. For instance, when the parameter of Max Velocity was changed, different behaviors were shown.

VIII. FUTURE WORKS

Implementing the flocking behaviour without separation was done and tested in Unity 2D. The same project can also be implemented in 3D where more parameters can be added. Addition of static and dynamic obstacles can also improve the behaviour of the flocks. Animation can be added to the flock to exhibit more realism.

In both the experiments the Max Force was kept constant. More scenarios and cases can be added to the experiment section to get more solid outputs. For example, varying the Max Force while keeping velocity constant. Qualitative tests like chi-square test can be conducted to prove the same.

REFERENCES

C. W. Reynolds. *Flocks, herds, and schools: a distributed behavioral model*. Computer Graphics (ACM SIGGRAPH '87 Conference Proceedings), 21(4):25–34, July 1987.

C. W. Reynolds. *Interaction with a group of autonomous characters*. In Proc. of Game Developers Conference, pages 449–460. CMP Game Media Group, San Francisco, CA, 2000.

J. Toner and Y. Tu. *Flocks, herds, and schools: A quantitative theory of flocking*. Physical Review E, 58(4):4828–4858, October 1998.

Vicsek, T., A. Cziro'ok, E. Ben-Jacob, and O. Cohen, I. Shochet, "*Novel type of phase transition in a system of self-deriven particles*," Physical Review Letters, Vol. 75, No. 6, pp. 1226–1229 (1995).

Su, H., X. Wang, and Z. Lin, "*Flocking of multi- agents with a virtual leader, part II: with a virtual leader of varying velocity*," Proc. 46th IEEE Conf. Decis. Contr., pp. 1429–1434 (2007).

Godsil, C., and G. Royle, "*Algebraic Graph Theory*," Springer, Berlin, (2001).

M.V. Abrahams and P.W. Colgan. *Fish schools and their hydrodynamic function: a reanalysis*. Environ. Biol. Fish., 20(1):79–80, 1987.

R.E. Baker, C.A. Yates, and R. Erban. *From microscopic to macroscopic descriptions of cell migration on growing domains*. Bull. Math. Biol., 72(3):719–762, 2010.

F. Cucker and E. Mordecki. *Flocking in noisy environments*. J. Math. Pure. Appl., 2007.

F. Cucker and S. Smale. *Emergent behavior in flocks.* IEEE. T. Automat. Contr., 52(5):852–862, 2007.