Math 3710W 1–15. Created In Connecticut
2020 University of Connecticut

From Flocks to Fish: How the Boids Algorithm Simulates Flocking Behavior

Written By: Ethan Rathbun

(*Submitted 18 December 2020*)

Fig. 1. The natural beauty of flocking birds

1 Introduction: The Birth of the Boid

How do you simulate the gracious flocking motions of dozens of birds all at once? Each bird makes their decision based on the movements of all the other birds creating a complex system of nested co-dependence, so how could one simulate this using the limited computing power of an early computer? This is a question that Craig Reynolds sought to answer in his 1987 paper "Flocks, Herds, and Schools: A Distributed Behavioral Model", which sparked the deep dive into the study of Swarm Intelligence. In particular, Reynolds noticed that the beautiful flocking motions of birds seen in the natural world was missing from the computer animator's toolkit. While a skilled animator could spend countless hours animating each individual bird in a flock to act in a realistic manner, there was no efficient way to simulate the process. In the aforementioned paper Reynolds devised a system of objects he referred to as "boids" (bird-oid objects) that would act as individuals but make decisions based on the behavior of other, surrounding boids. His algorithm, commonly referred to as the "Boids Algorithm" efficiently created convincing flocking behavior.

1.1 Introduction: Boids Algorithm in Computer Animation

Since the Boids Algorithm was developed multiple films, TV shows, and video games have adapted the algorithm to efficiently animate various creatures in a convincing way. The first example of this is shown in in the short film "Stanley and Stella in: Breaking the Ice". It was created by Craig Reynolds and others as a way to showcase the animation capabilities of the newly developed Boids Algorithm. In this iteration of the Boids Algorithm the boids were programmed to seek a goal object and avoid obstacles. The goal could be moved around dynamically in 3D space as to give the perception that the boids were making deliberate decisions about where the flock should head. The algorithm also saw use in the groundbreaking Video game "Half Life". There the programmers use the Boids Algorithm to allow bird-like, alien creatures to fly around in a way that dynamically adapted to the environment and the player, creating a convincing effect that would further immerse the player in the game's world.

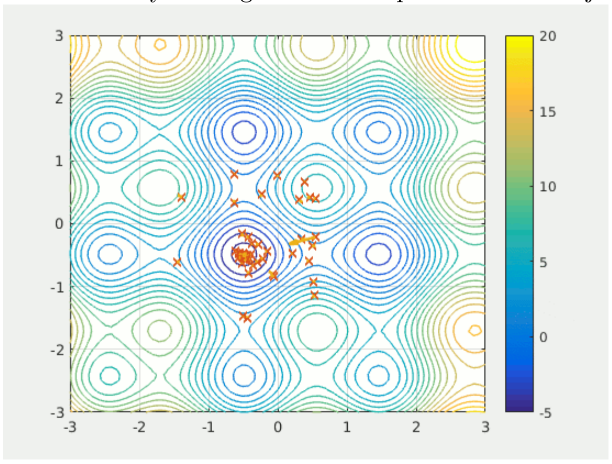
1.2 Introduction: Swarm Intelligence

While the Boids Algorithm was originally created with computer animation in mind, the algorithm was found to be more versatile than expected since then. This is because the model leads exhibits "emergent properties", meaning complex behavior can result from the algorithm's use of relatively simple rules. This lead to the creation of a field of research called "Swarm Intelligence". Using swarm intelligence techniques a programmer can solve practical and complex problems that have no obvious solutions. One instance of this is found in the "Particle Swarm Optimization Algorithm" (PSO), a modified form of the boids algorithm that was found to have optimization properties. This paper is not about optimization, however a brief summary of optimization problems will be provided.

In general there is some function the user is trying to optimize, called the objective function, meaning they are attempting to find either a global minimum or a global maximum (in general: global optimum). Many objective functions have multiple local optimums, meaning they are non-convex. This makes finding the global optimum extremely difficult as there is often no way to verify a given local optimum is global.

The idea behind PSO is to perform multiple simultaneous searches of the objective function all starting from different positions of the objective function's domain. These searches, called particles, each have a random starting velocity and position, and then share with each other when a more optimal point is found. These new optimal values then influence the movement of all the particles in the system, causing them to eventually swarm and converge on a local optimum. The particles use no information on the gradient of the objective function so the algorithm can be used on non-differentiable objective functions. While the algorithm is likely to find a local optimum, it is not guaranteed to find a global optimum, leading it to be classified as a meta-heuristic. Swarm intelligence has also been used to find approximations to the solution of NP-Complete problems such as the Traveling Salesmen Problem.

Fig. 2. Particles eventually converge on a local optimum of the objective function



2 Background: A Summary of What You Need to Know

What makes the Boids Algorithm so beautiful and effective is its simplicity, this means there are no overly complex concepts that one needs to understand in order to understand the algorithm. In general one should be comfortable with the following: vectors and vector operations, difference equations, and computational complexity. These topics will be explored in more detail in the following subsections.

2.1 Background: Vectors and Vector Operations

Vectors are versatile mathematical objects that will be used to represent the velocity of the boids in this model. The intuitive representation of vectors comes in the form of a line with an arrow at the end (Fig. 3). Vectors have two key properties, a magnitude and a direction within the space they exist in. Intuitively the magnitude of a vector can be thought of as the length of the line that represents it, while the direction can be thought of as the angle at which the arrow is pointing. Mathematically vectors are represented as follows:

$$\overrightarrow{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

This vector has n components represented as x_i, meaning it exists within a n-dimensional space. Each component represents the magnitude of the vector within a given direction of the space it occupies. To simplify things this paper will focus on

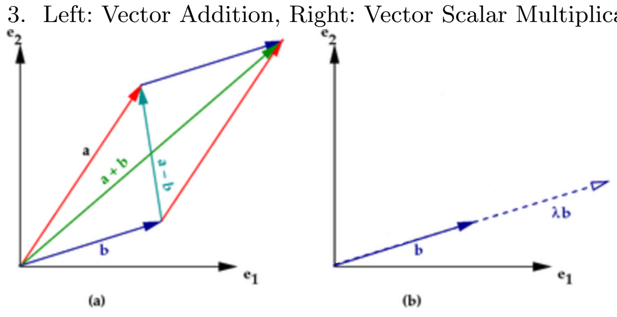
2-dimensional space with all components representing real numbers, however the concepts can be applied to any n-dimensional space. In 2 dimensions you can think of vectors as having an x and a y component in a Cartesian plane. Bellow it will be shown how to perform addition between vectors as well as how to multiply a vector by a number value called a “scalar”.

$$\vec{a} + \vec{b} = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} a_1 + b_1 \\ a_2 + b_2 \end{bmatrix}$$

$$c \cdot \vec{a} = c \cdot \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} c \cdot a_1 \\ c \cdot a_2 \end{bmatrix}$$

A visual representation of vector addition and scalar multiplication can be seen in Fig. 3.

Fig. 3. Left: Vector Addition, Right: Vector Scalar Multiplication



2.2 Background: Difference Equations

Difference equations are a way to represent discrete changes in a variable over discrete steps in time. Here discrete means the changes in time are distinct and occur in steps rather than in a continuous fashion. If one were to go for a run and keep track of their progress once every minute, they would create a discrete graph of their progress with the discrete time step being 1 minute. Generally difference equations are represented in the following way:

$$x_{n+1} = f(x_n, x_{n-1}, \cdots, x_{n-j})$$

Here n represents a discrete time step, so x_3 will represent the value of x after 3 steps or iterations of the function. To get the value of x at a given n one has to first choose a starting value of x, often represented as x_0, and then iterate through the given function n times. Below a simple example will be given:

$$x_{n+1} = 3x_n\,, \quad x_0 = 2$$

$$\begin{array}{c|c} n = 0 & x_n = x_0 = 2 \\ n = 1 & x_n = 3(2) = 6 \\ n = 2 & x_n = 3(6) = 18 \\ \vdots & \vdots \end{array}$$

The study of difference equations is a vast field of applied and theoretical mathematics, however, in this paper, understanding how they are represented and how they are used is sufficient.

2.3 Background: Computational Complexity

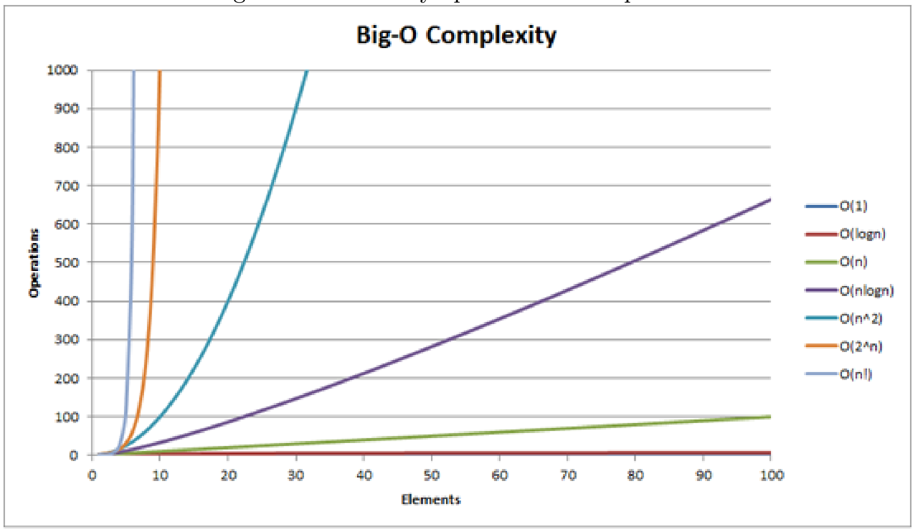
Much of the focus of computer science research goes into finding algorithms that can generate solutions to a given problem, however not all algorithms are equal. One key trait of an algorithm is efficiency. More rigorously, efficiency is measured by the number of operations the algorithm needs to perform before it arrives at the solution. This is often called the running time or time complexity. This is measured as a function of the size of the problem given to the algorithm, often represented as n. For example, you should expect it to take more operations to find the largest number out of 100 numbers than to find the largest number out of 10 numbers. Yet the size of the problem is not the only factor, say you had an unorganized list of students in your math course next semester and you want to see if your friend, Alan T., is in the class. Now look at the following two instances of this problem:

Instance 1	*Instance* 2
Alan T.	*Shafi G.*
Andrew N.	*John C.*
Katherine J.	*Benoit M.*
⋮	⋮
Kurt G.	*Alan T.*

If your strategy is to look down the list of names until you find your friend's name or until you reach the end, then clearly you will finish significantly faster in instance 1 than in instance 2. In fact, instance 1 represents the best case scenario as you only need to check 1 name, while instance 2 represents the worst case scenario since you need to check every name. Thus computer scientists devised 3 different ways to measure the computational complexity of an algorithm: upper bound on time complexity (worst case), lower bound on time complexity (best case), and a tight bound on time complexity (bounds both worst and best case). For simplicity this paper will focus on the upper bound for running time as it is the most commonly used. This upper bound is referred to as "Big-O" notation, meaning the computational complexity of an algorithm is represented as $\mathcal{O}(f(n))$ where n is the size of the problem and f is some function of n. Now return to the math class problem from earlier, in the worst case scenario the algorithm will check all n names in the list of students, so the time complexity in Big-O notation is $\mathcal{O}(n)$, known as linear time. If the algorithm had to check an array of size $n \times n$ then the time complexity would be $\mathcal{O}(n^2)$ which is parabolic. In general, algorithms of the form $\mathcal{O}(n^c)$ for some constant c are called polynomial-time algorithms. Any algorithm that can be bounded by a polynomial time function of n is generally considered efficient, although efficiency is often measured relative to algorithms that already exist for the given problem. If an algorithm needs to check every subset in a set of size n it will have $\mathcal{O}(2^n)$ time complexity, this cannot be bounded by any polynomial function of n so the algorithm is not considered efficient. Big-O notation, along with the other two bounds, is an asymptotic bound where overall behavior as n grows large is considered most important. This means constants and slower growing terms are discarded when the final Big-O time complexity is given. See the following examples.

$f(n)$	$Big - O\ Complexity$	$Efficient?$
$3n^2 + n + 5$	$\mathcal{O}(n^2)$	*yes*
$2^n + n^{10}$	$\mathcal{O}(2^n)$	*no*
$n! + 2^n$	$\mathcal{O}(n!)$	*no*
$n \cdot log(n) + n$	$\mathcal{O}(n \cdot log(n))$	*yes*

Fig. 4. Different asymptotic time complexities



3 Mathematical Formulation: Overview and Assumptions

In the following sections a detailed, mathematical formulation of the Boids Algorithm will be presented, but before then the following assumptions must be made:

1. The boids exist within some finite 2-dimensional space within which they can freely roam.
2. There are no external factors that impact the movement of the boids.
3. Each boid is aware of all the boids within a certain radius and is unaware of those outside the radius.
4. Boids can not physically collide with each other, but rather pass through each other.

While these assumptions do prevent the model from being a perfect simulation of real life flocking behavior, the model will still create convincing visuals that can be adapted and added to for more accurate models.

3.1 Mathematical Formulation: The Boids and Their Representation

Each of the boids will have two properties: a position and a velocity, both of which will be represented with an x-component and a y-component in the 2-dimensional Cartesian plane. Mathematically, and in computer code, the velocity and position of a boid can be represented with two vectors. The following notation has been adopted:

$$i^{th}\ boid = b_i$$
$$velocity\ of\ i^{th}\ boid = vel_i = \begin{bmatrix} v_{xi} \\ v_{yi} \end{bmatrix}$$
$$position\ of\ i^{th}\ boid = pos_i = \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

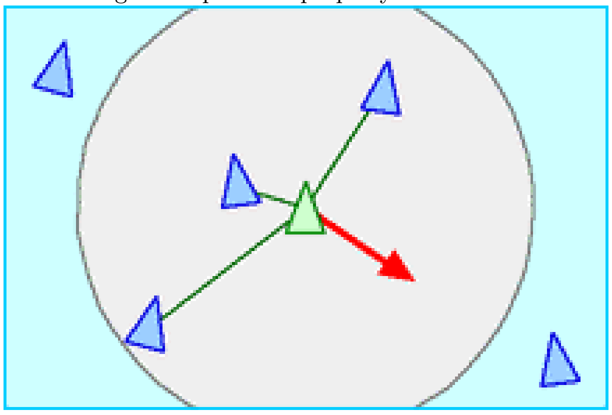
3.2 Mathematical Formulation: A Change in Velocity

The position of each boid will update after a discrete step in time. Thus the movement of each boid is represented by the following difference equation:

$$(pos_i)_{n+1} = (pos_i)_n + vel_i$$

This difference equation is fairly simple on the surface, however the complexity comes from how the velocity value of a boid is found. In particular, the velocity of a boid follows three rules:

Fig. 5. Separation property of the boids



Separation: boids will avoid coming in too close contact with eachother, this is designed to prevent all the boids from converging into a single point and give more realistic spacing to the flock. More explicitly boids will take note of other boids who are too close to them and adjust their velocity away from the other boids. Here, there is a predefined distance called the *alert* distance (alert). A naïve approach to this problem is the following:

$$separation_i = \sum pos_i - pos_j \quad \forall\ j \mid dist(pos_i,\ pos_j) \leq alert$$

Where the function "dist" calculates the distance between two vectors using the euclidean norm, as shown below:

$$dist(pos_i,\ pos_j) = \sqrt{(x_i - x_j)^2 + (x_i - y_j)^2}$$

The problem here is that the value of $pos_i - pos_j$ will linearly decrease as b_i and b_j get closer to each other. This will cause closer together boids to not push away from each other, while further apart birds will push away strongly. This leads to the development of the following summation:

$$pos_i - pos_j = \text{diff}$$

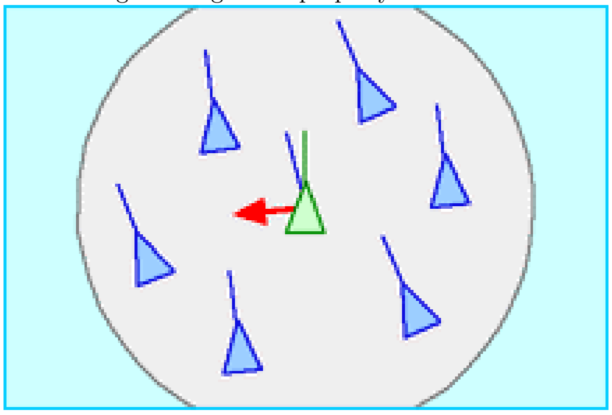
$$separation_i = \sum (alert - |\text{diff}|) \star sign(\text{diff}) \quad \forall\ j \mid dist(pos_i,\ pos_j) \leq alert$$

In this scenario, diff is clearly a vector, so for the sake of convenience, and to correspond to their application in code, we define the following short hand, element-wise vector operations:

$$\begin{aligned}
\text{vec} &= \begin{bmatrix} x \\ y \end{bmatrix} \\
sign(\text{vec}) &= \begin{bmatrix} \frac{x}{|x|} \\ \frac{y}{|y|} \end{bmatrix} \\
a - \text{vec} &= \begin{bmatrix} a - x \\ a - y \end{bmatrix} \\
\text{vec}_i \star \text{vec}_j &= \begin{bmatrix} x_j \cdot x_i \\ y_j \cdot y_i \end{bmatrix}
\end{aligned}$$

The resulting "separation" vector will be a vector pointing away from nearby boids in the flock relative to the position of the current b_i. Thus if this vector is added to the velocity vector of b_i it will cause the boid to begin accelerating away from boids that are too close by.

Fig. 6. Alignment property of the boids



Alignment: boids will align their velocity in accordance to the other boids around them. This is designed to create the illusion that all boids are flying to the same end goal with the same direction. This is done by taking the average velocity of all the surrounding boids and then adjusting each boid to move in that direction. The radius within a which a boid considers the direction of the other boids is the *flock* distance (flock). This change in velocity is calculated with the following equation:

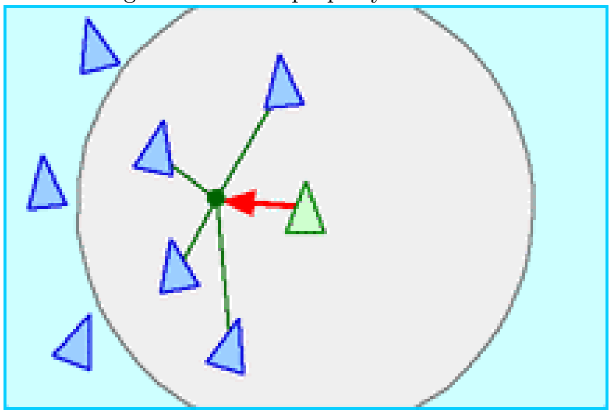
$$direction_i = \frac{1}{n}\sum vel_j \quad \forall\, j \mid dist(pos_i,\ pos_j) \leq flock$$

"direction" will then contain the average velocity of all boids within the radius of a given boid.

$$flocking_i = vel_i - direction_i$$

The resulting "flocking" vector will be a vector that points a given boid's velocity vector into the direction of the average velocity of all the boids around it.

Fig. 7. Cohesion property of the boids



Cohesion: lastly, boids will gravitate towards the center of all the boids within their sight radius, this is the main property that causes the boids to group into a flock. This property will use the same radius as the alignment property, that being "flock". This change in velocity is calculated with the following equations:

$$center_i = \frac{1}{n}\sum pos_j \quad \forall\, j \mid dist(pos_i,\, pos_j) \leq flock$$

"center" will then contain the position of the center of the flock within the radius of a given boid.

$$cohesion_i = pos_i - center_i$$

The resulting "cohesion" vector will be a vector that points a given boid's velocity vector towards the average position of all boids around it.

3.3 Mathematical Formulation: Putting it All Together

Once all the different rules have been applied to find the corresponding separation, alignment, and cohesion vectors, they are then added to the velocity of each boid using the following equation:

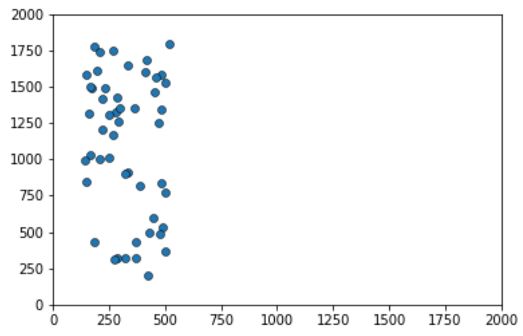
$$(vel_i)_{n+1} = (vel_i)_n + \alpha \cdot separation_i - \beta \cdot alignment_i - \delta \cdot cohesion_i$$

The parameters α, β, δ should all be within the range $[0, 1]$. As one changes these values different results will occur. Each of the parameters can be adjusted to give different results; this will be explored in the next section in more detail.

3.4 Mathematical Formulation: Experimental Results and Model Performance

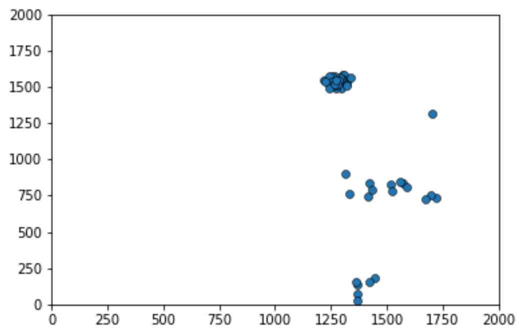
The following figures depict the resulting images created by the boids algorithm that has been described up until this point. The code written to generate the images closely follows the mathematical formulation for the boids model given in this paper, however some slight adjustments were made in order to make the code more efficient. For instance, instead of representing each individual boid as a pair of position and velocity vectors, they were instead all represented by two $2 \times n$ matrices. This was done using the python library "numpy" since coding matrix operations by hand in python would not only take a lot of time to code, but would also take significantly more time to run. Without getting into the details, this is because python is considered an "interpreted" language, while the code for numpy is written in a "compiled" language. In practice compiled languages run significantly faster than interpreted ones. It is also worth noting that, going along with the assumption made earlier, the boids are bounded within a certain space. If the boids touch the edge of that space they will "bounce" off the edge in the way one would expect a rubber ball to bounce off a wall.

Fig. 8. Boids initialized in a state of chaos within range $([100, 600], [100, 1900])$



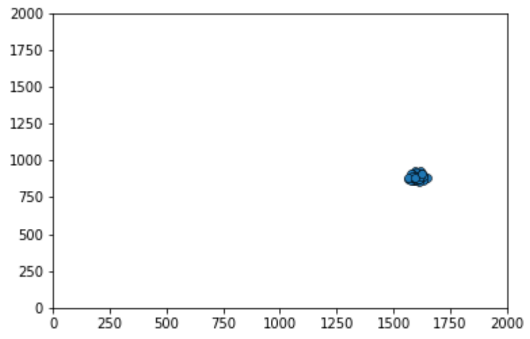
As previously explained, different results should be expected from different choices of parameters. Each of the experiments run will begin in a randomized state similar to the one seen in Fig. 8.

Fig. 9. After a short period in time the boids will cluster into a few flocks



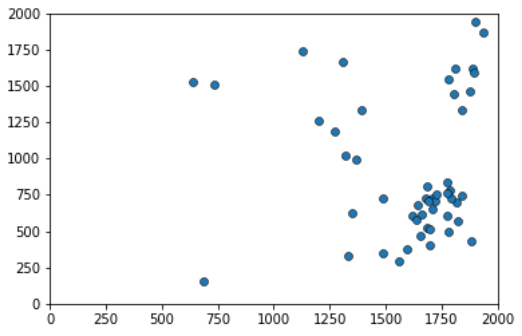
If all the parameters are balanced we should see the following progression: the boids start randomly distributed within some range, they then begin to group together into a few small groups (Fig. 9), then when the groups come near each other they join into a single, larger flock.

Fig. 10. With a large δ the boids quickly join into a tightly bound singularity



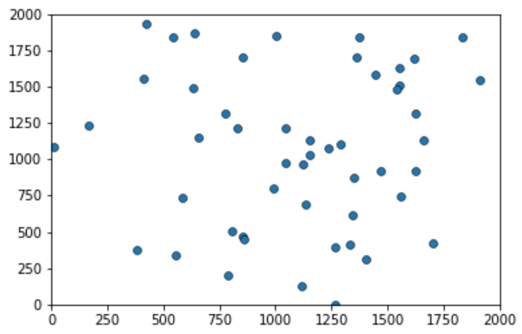
If some parameters are chosen to be significantly larger than other parameters one can expect certain, unique results. For instance, if $\delta \gg \alpha$ then one should expect boids that get within the given flocking radius to clump together into a singularity, this can be seen in Fig. 10.

Fig. 11. With a small δ the boids do not form coherent flocks



Similarly, if certain parameters are chosen to be very small then other unique results will occur. If one were to choose δ to be very small then the boids will almost never join together into flocks and will instead remain in a state of chaos. This can be observed in Fig. 11.

Fig. 12. Even with substantially large δ and β values, if the boids' starting velocity is too large they will not be able to form coherent flocks



The strength of δ, along with the other parameters, should be considered relative to the magnitude of the starting velocities of the boids. For instance, if the boids start with extremely high velocities then they may never join into groups, even with reasonably large δ and β values. This can be observed in Fig. 12.

3.5 Mathematical Formulation: Model Computational Complexity

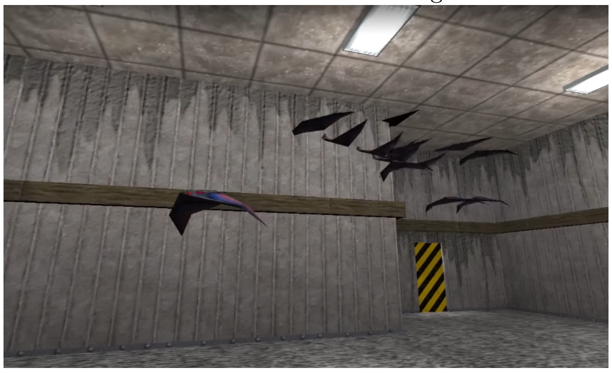
For this computational complexity analysis the complexity of each, discrete time step will be taken into consideration. This is because the overall complexity of the model will be represented by $\mathcal{O}(t \cdot f(n))$ where t is the number of discrete time steps taken, and $f(n)$ is the complexity of a single time step. During the calculations of the separation, alignment, and cohesion values each boid compares their position to the position of all other boids. Since there are n boids and each boid compares their position to $n-1$ other boids. This results in an $\mathcal{O}(n^2-n) = \mathcal{O}(n^2)$ time complexity. For the sake of efficiency, this distance calculation can be performed once and then used in each of the three velocity change calculations. After a given boid decides which boids are within its radius, a few element-wise matrix operations are performed. Sometimes these operations can be performed with a time complexity $\mathcal{O}(f(n)) < \mathcal{O}(n^2)$, however for simplicity all of these operations will be considered to have a $\mathcal{O}(n^2)$ time complexity. All of these operations are performed sequentially so the overall complexity of the model will be of the form:

$$\mathcal{O}(c \cdot n^2) = \mathcal{O}(n^2)$$

The practical performance of the model can be improved through the optimization of the c constant found on the left hand side of the previous equation. There are also more efficient, albeit more difficult to code, techniques for finding all the boids within a given range. These techniques were not used in the code created for this paper, however, they involve the usage of an optimized version of the k-nearest neighbors algorithm. Overall the model is still fairly efficient when put into the context of the initially proposed problem (simulating flocking behavior), but slow performance can be expected when using a very large number of boids.

3.6 Mathematical Formulation: Overall Conclusions

Fig. 13. Boids in the game "Half Life" dynamically choose and follow a leader to create a more consistent and convincing motion



While the iteration of the Boids Algorithm created for this paper is an accurate reformulation of the model, it follows the simplest form of the model that has a few key flaws. The primary flaw is the unreliability of the model; much of the resulting movement of the boids is heavily dependant on the initial randomization of the boids' positions and velocities. This means that it would not work well in computer animation unless unpredictable behavior is desired. One way this is resolved is by allowing the different flocks of boids to choose a flock leader. Using this method the boids will follow the movement of the leader who will then smoothly move around in a realistic way (Fig. 13).

Another way to mitigate this issue (as mentioned earlier in the paper) is by implementing a goal object which all the boids seek out. Even in its simplest form, the Boids Algorithm does a great job simulating the flocking motions of different creatures. It also allows one to change certain parameters for different desired results, or to experiment with the parameters to see what impact they have on the overall behavior of the model. Thus the Boids Algorithm has shown itself to be a useful and versatile tool for any flocking behavior that one may desire to recreate.

4 References

1. Reynolds, Craig (1987). Flocks, herds and schools: A distributed behavioral model. SIGGRAPH '87: Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques. Association for Computing Machinery. pp. 25–34. CiteSeerX 10.1.1.103.7187.
2. "The Boids." The Alan Turing Institute, alan-turing-institute.github.io/rsd-engineeringcourse/ch01data/084Boids.html.
3. Kennedy, J.; Eberhart, R. (1995). "Particle Swarm Optimization". Proceedings of IEEE International Conference on Neural Networks. IV. pp. 1942–1948. doi:10.1109/ICNN.1995.488968
4. Iztok Lebar Bajec, Frank H. Heppner (2009). "Organized flight in birds", Animal Behaviour, Volume 78, Issue 4,