Assignment 1 Report: Movement Algorithms

Nate Strohmyer
University of Utah
Salt Lake City, Utah
United States of America
719-304-4780
nate.stroh@live.com

ABSTRACT
In this paper I will be exploring different movement algorithms that can be used in games for AI movement. Movement algorithms are algorithms that specifically effect the position or orientation of an object. I will specifically be exploring kinematic motion, seek steering behaviors, wander steering behaviors and flocking behavior and blending/arbitration. For each movement algorithm I will discuss my implementation, steps to get desired behavior and my exploration of how parameters affect that movement algorithm.

Keywords
AI, Artificial Intelligence, Movement Algorithms, AI for Games

INTRODUCTION
As described in our course work and by Millington (2019), AI for games can be split into four major algorithm types being movement, pathfinding, decision-making, and strategy algorithms. These four algorithms are all related to each other and delegate to other algorithms. Movement algorithms are the lowest level of these four algorithms and is one of the fundamental bases for any game. For example, a decision-making algorithm might delegate responsibility to a pathfinding algorithm to get a path to a valid target and in turn the pathfinding algorithm might delegate the movement of that object to a movement algorithm. This makes movement algorithms imperative to any game that has movement. Given that most games have some form of movement, starting creation of an AI system with robust and functional movement algorithms allows for an excellent start and base for more complex systems/algorithms.

As mentioned before, I will be covering four movement algorithms in this paper: kinematic motion, seek steering behaviors, wander steering behaviors and flocking behavior and blending/arbitration. Kinematic motion is focused on getting an object moving and gets an object moving with the use of velocity rather than with acceleration. Seek steering behaviors are behaviors meant to "guide" or "seek" out specific positions. Essentially, these behaviors are meant to get your object to a specific location. Wander steering behaviors focus on getting your object to mindlessly meander or wander throughout the screen. However, it's important that this algorithm adjusts it's orientation gradually and smoothly to avoid unnatural and unwanted jittering as it changes directions to wander. Lastly, I'll be exploring flocking behavior and blending/arbitration. Flocking is essentially keeping several objects grouped up and going towards a common goal/center position. Flocking

Proceedings of DiGRA 2011 Conference: Think Design Play.

largely depends on the blending of different movement algorithms so I will be exploring blending in addition to the three main movement algorithms used to implement flocking.

KINEMATIC MOTION

Much of the work that went into getting kinematic motion working correctly involved project and Boid (my representation of the game object/shape I am moving) set up. Before I could even get a basic position matching algorithm done, I would need to get a working game object that could hold that data and do something with it. This consisted of four main parts being a rigid body, steering output, an update function, and a draw function. The rigid body is represented by two vector2s for velocity and position and two floats for orientation and rotation. The steering output was also a relatively simple struct consisting of a vector2 for linear acceleration and a float for rotational acceleration. My update function is where a lot of the data manipulation and physics calculations get completed. This function essentially added the acceleration from the steering output to their respective values (linear acceleration to velocity and rotation acceleration to rotation). Then update adds its velocity and rotation to position and orientation, successfully updating the boid's rigid body. While updating velocity the update function also factors in drag. Update also oversaw timing and deciding when and where to spawn breadcrumbs. Lastly, this function also handled toroidal world boundaries. Using OpenFrameworks also allowed me to get the screen size, so everything that depends on the edge of the screen (like toroidal edges or the dot placement in Figure 1) is variable and scales to the screen size. The last function I had to get running was the draw call for boid. This essentially needed to handle drawing a circle and triangle in the correct position and orientation. While this sounds easy, doing this effectively was a challenge. My first implementation involved getting three separate points on the triangle using sign and cosign formulas. While this worked well enough, I figured implementing the version where you centralize the origin around 0,0 would work best and implemented that. Lastly, this draw call was also in charge of drawing all of the crumbs for that boid.

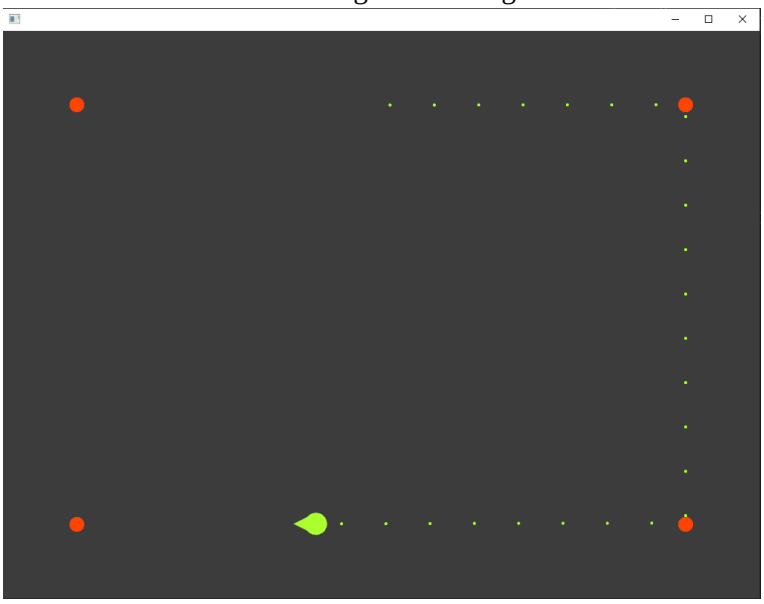

Figure 1: Boid kinematic seeking to the four corners of the screen. Note the breadcrumbs (trailing green dots) that show the path of the boid.

Now that the boid and breadcrumbing was set up, I could dive into kinematic seek. Kinematic seek is simple in that you're just getting the direction you need to go and going

your max speed in that direction. I also had to use my physics system in a weird way to simulate kinematic behavior. Rather than set the steering output, which was effectively my acceleration, I just set my velocity to the output of kinematic seek. This got me the desired effect and you can even start seeing the pitfalls of a kinematic system. For example, once kinematic seek starts seeking the same position you start getting a lot of jittering and snapping. The only parameter to really play with in seek is your max speed and the parameter itself doesn't leave much to explore. For this behavior I just adjusted the max speed until I thought it was fast enough.

SEEK STEERING BEHAVIORS

As described above seek steering behaviors are position matching movement algorithms so they are designed to match the position of the original boid with some target position. For this assignment I implemented three seek steering behaviors: dynamic seek, dynamic pursue, and position matching arrive.

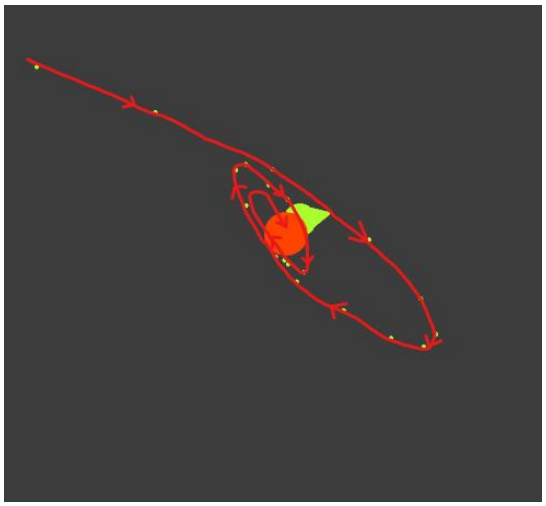

Figure 2: Boid dynamic seeking a position. Note the oscillating path.

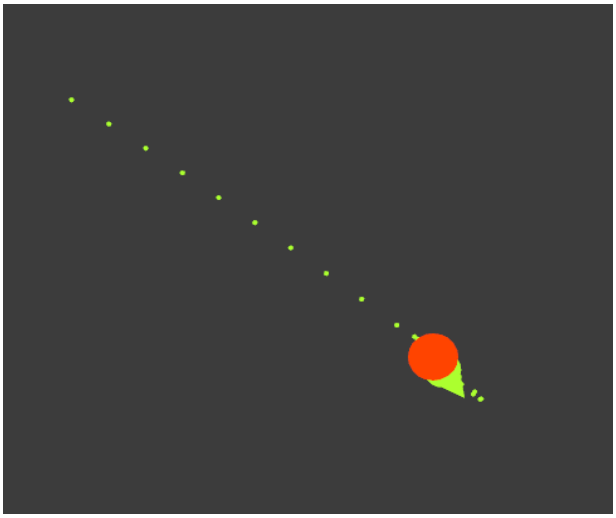

Figure 3: Boid dynamically pursuing clicked position. Note that it still oscillates.

Dynamic seek works very similarly to kinematic seek in that we're looking at the positions of the boid and target to get the direction we need to go. However, dynamic seek instead modifies acceleration rather than speed/velocity. This gets us much less oscillation since we have to build up to a max speed rather than instantly get there but it is still present as our velocity always pushes us further causing dynamic seek to adjust our position yet again. Like kinematic seek, dynamic seek doesn't have too many parameters to adjust. I pretty much just played with the max acceleration until I got it going a speed I liked that didn't grow too unmanageable or oscillated too much.

Dynamic pursue is also implemented very similarly to both seek algorithms with the only real difference being a prediction factor pursue does. Dynamic pursue considers not only the position of the target but also the velocity so pursue can kind of guess where the target will be. For this specific part of the assignment, pursue acted in the exact same way as dynamic seek as the target the boid was pursuing was a static point with no velocity. When it came to parameter adjustments to dynamic pursue, I found that the max acceleration had many of the same effects as it did on dynamic seek. The interesting parameter to adjust was the prediction time. Adjusting this parameter directly adjusted how far out pursue tried to predict their location. I ended up going with a much smaller number for prediction time as I felt this allowed pursue to be more accurate with its predictions, especially if the target was moving erratically. I also did a bunch of exploring related to how pursue works when chasing a target vs seek. The results were fairly interesting in that seek always ends up as more of a follower and has a curve in it's path to the target. Pursue on the other hand was much more of a straight line to the target, even when given a lower speed than seek.

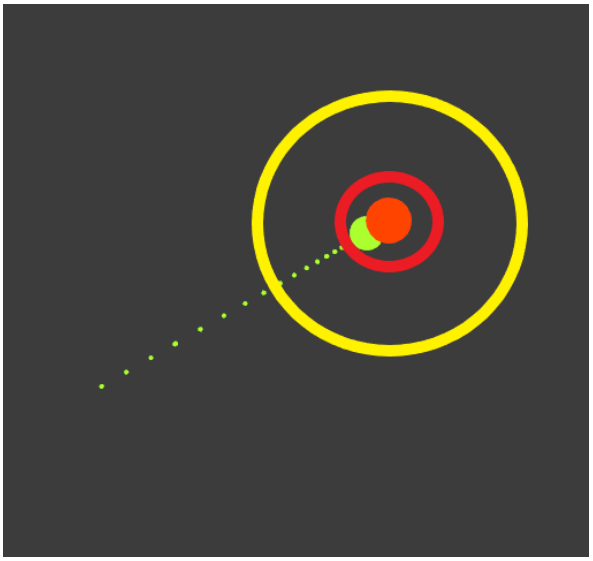

Figure 4: Boid position matching arrive to a clicked position. Note the breadcrumbs getting closer, meaning the boid is slowing down. Also note the two rings signifying the slow (yellow) and target (red) radius

Lastly, I implemented dynamic arrive to seek to a position. Dynamic arrive was much more in depth since it had a three step approach in order to stop appropriately and look good. In addition to position dynamic arrive considers the distance you are from the target position and adjusts accordingly. If you are outside of the slow radius (yellow ring in figure 4) you

accelerate as much as possible. If you are between the slow radius and target radius (red ring in figure 4) you slowly get slower and slower as your acceleration lerps to zero. If you are in the target radius you don't accelerate at all. Even with acceleration set to zero I ran into a problem where arrive would carry on through the target due to its left-over velocity. I thought of a couple solutions to this but ended up deciding on implementing drag into my physics update. I ended up deciding on implementing drag as this solved other problems with dynamic align and it produces a nice effect in general watching objects slowly come to a stop if they don't have velocity. Dynamic seeks parameters took much more time to get good results from. In addition to implementing drag to get the boid to completely stop, I had to do a lot of tweaking the slow radius and targeting radius. For example, if the slow radius was too large my object would spend too much time in the lerp phase making it agonizingly slow as it approached the target radius. If the slow radius was too small, I might not be able to slow down enough and go past the target. The target radius also had very similar problems in that if it was too large the boid might not even make it to the position it was supposed to. On the other hand, if the target radius was too small the carried over velocity would cause the boid to overshoot the target.

Overall, dynamic arrive gave me the best results by far. It consistently was able to make its way to the target and stop there. While the other two methods may have gotten there faster, they often overshot their destination resulting in weird oscillating patterns. The drag did help with this, and they would come to an eventual stop, but this was only after a couple of times bouncing back and forth. After further consideration, this result is not unexpected. Dynamic arrive is the only one of these position matching behaviors to look at the distance between it and its target and actually bother slowing down.

WANDER STEERING BEHAVIORS

Wander steering behaviors are more movement algorithms that focus on moving a boid semi-randomly throughout the screen. Wander ended up taking much more time than I thought it would. While the algorithm was fairly easy to wrap your head around, I had a few hang ups with implementation. I started by getting the orientation of the wandering boid and getting a offset from that based off a parameter. Next, I turned this new orientation into a unit vector and multiplied it by an offset to get a new position. Lastly, I just added the position of the boid to this new position and seeked to it. Wander was a prime example of how parameters can make a huge difference in these algorithms. You can just get such varying results by just modifying the max acceleration of the algorithm. For example, modifying the max acceleration made huge differences moving faster pushed an object more drastically in this new direction making the boid seem more unpredictable. Overall,

all the parameters having such drastic effects on the behavior and the random nature of the algorithm made wander a very hard behavior to implement correctly.

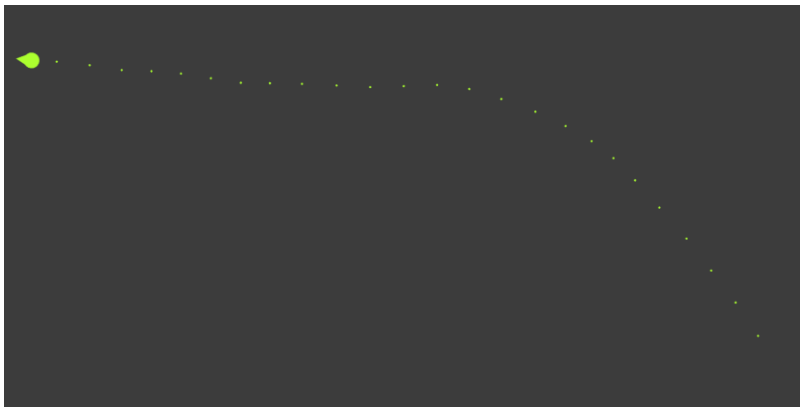

Figure 5: Boid wandering, this picture demonstrates the look where you are going orientation matching.

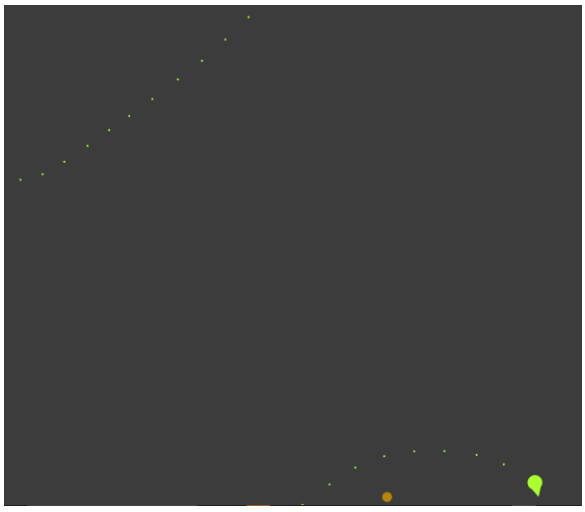

Figure 6: Boid wandering while calling Face on the position being seeked. Note the much more curved path it took. This picture also shows the debug circle (yellow gold circle) used to show where wander was going.

Another interesting thing about wander is how much different orientation algorithms affected the result of the wander algorithm. My default test case was the look where you are going (LWYAG) algorithm for orientation matching, and I thought this one looked pretty smooth. This implementation allowed for some drastic angle changing as LWYAG updates orientation over time which limits how far the boid is actually turning before the next call to wander. Another orientation matching algorithm I tested with wander was Kinematic algin. This was just essentially setting the boids orientation to its velocity and I expected it to be very erratic but it was more jittery than anything. For the most part, it's wander behavior was similar to LWYAG, just more jumpy and jittery. Lastly, I implemented a special version of wander that called face on the new seeked target. This got me the most interesting results as the orientation changed much more drastically. I

ended up getting a behavior that was almost drifting in a way. It had much more curving behavior and was almost always looking in a different direction than the way its velocity was taking it. I thought this was interesting as just by changing when and what orientation method we are calling we can have such a different behavior.

I would also like to point out a very interesting feature I made while trying to debug my wander code. I often found it super hard to try and figure out where wander was actually trying to take me, as it was very hard to try and piece together the target location and orientation. Sometimes the boid would end up just spinning or moving in a straight line. This led me to put an output parameter for a position that let me draw exactly where the position wander was seeking to is. This visual aid helped me pinpoint exactly what wander was doing and gave excellent feedback so I could adjust the wander parameters. I found myself even just messing with parameters to see what the arc of new positions would be.

FLOCKING BEHAVIOR AND BLENDING/ARBITRATION

Flocking is the grouping up of multiple objects as they move towards one goal/position. The trick here is to balance or blend different behaviors to get a group of boids that move together but don't stay on top of each other. Surprisingly flocking didn't actually give me too much trouble. Most of the complicated parts of the algorithm are delegated out to algorithms like arrive, separate and velocity match. The only parts I really had to handle specifically for flocking was making the centroid rigid body and arbitrating the strength of each algorithm. I was very surprised to see how simple such a complex behavior could be. I didn't even have to make any changes to the algorithms themselves. I was able to get the flocking behavior I wanted by passing different parameters and having different weighting for the different algorithms. I found that separation was by far the most important of the three algorithms as without a strong separation my flock quickly turns into a blue blob. The other two movement algorithms seemed to mainly just depend on behavior you want. For example, a higher weight on arrive got me a flock that was much more dense as the final steering output had much more pulling it towards the centroid of the flock. Making velocity matching weigh more got me a more fluid and spread-out flock. Rather than move towards each other the flock would just move in one general direction and loosely move towards each other.

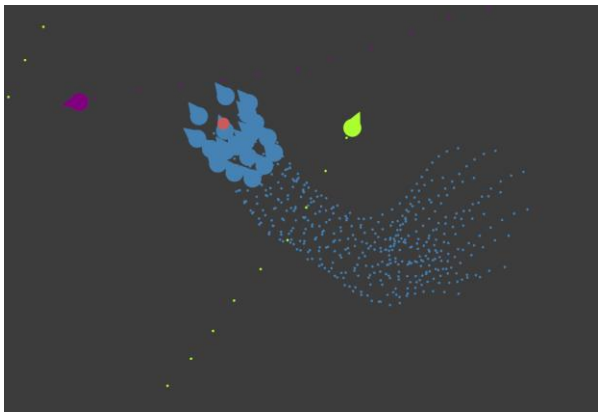

Figure 7: Flock following two wandering leaders

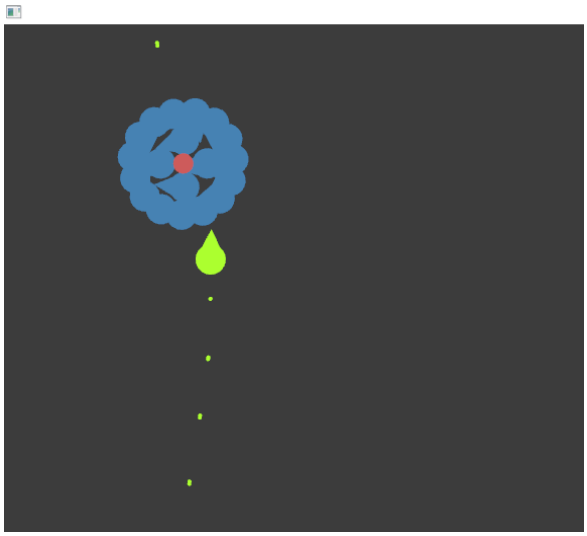

Figure 8: Flock with no leader (they're ignoring the grenn wander boid). Note how they clump up as tight as possible trying to minimize overlap while still be close to the centroid.

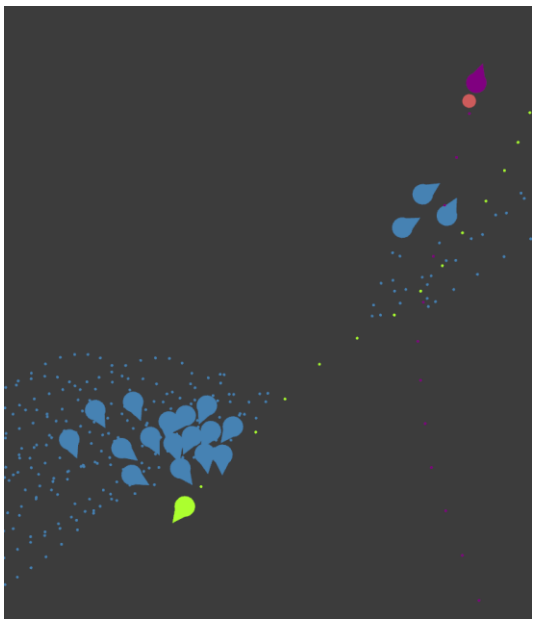

Figure 9: Two separate flocks where follower boids follow the closest leader.

The number of leaders and followers also had a huge impact on the flock. A flock with no leader and no wander just ends up as a stand still blob that doesn't go anywhere, which makes sense given that arrive tells them to go to the center and separate keeps them from overlapping. However, velocity match is still zero so there's no pressure on the flock to move whatsoever. Adding another wandering leader also makes some interesting changes.

Adding another leader ends up making the flock followers matter less. essentially the centroid just becomes the midpoint between the leaders and is only slightly affected by the flock. This reflects how the flocks cohesion is affected by the number of boids in general. If you only have a couple boids, the centroid is primarily focused on the leader. This results in a flock that just looks like boids following the leader. Adding more boids on the other hand gives the flock more power making the centroid closer to the followers and making the flock more cohesive.

Having multiple leaders where followers follow the closet wanderer looks cool and watching the flocks separate and combine is very interesting. However, the overall behavior doesn't change all too much. For the most part the 2 flocks act in a similar way, they just lose and gain followers as time goes on.

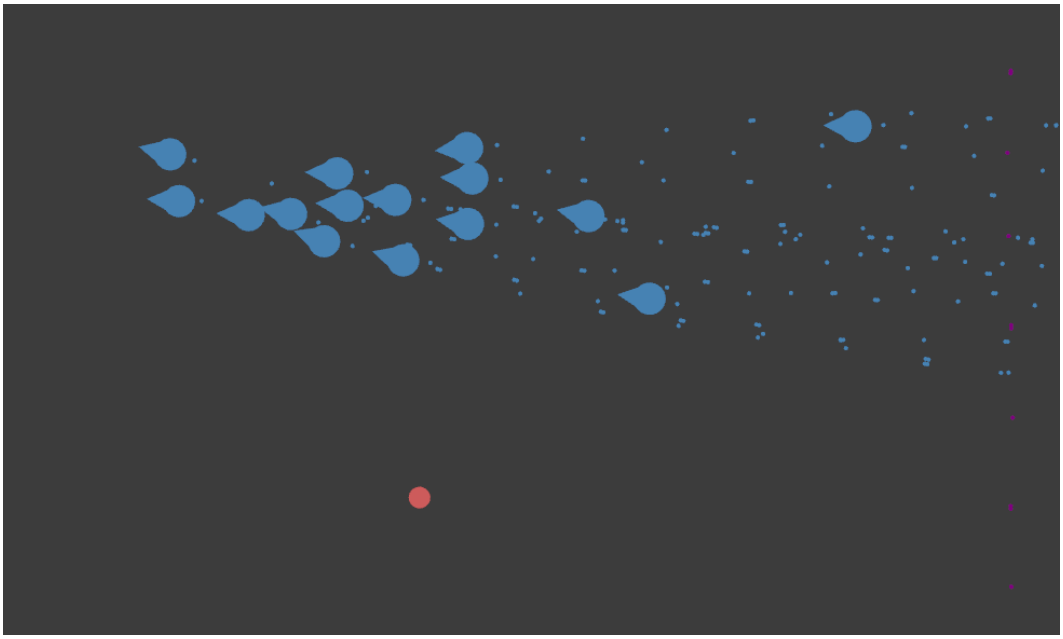

Figure 10: This flock is entirely driven by the flock. All boids are weighted the same and call wander. They then blend this wander into their steering output. Note the red circle showing the centroid of the flock.

Testing flocking with different sizes also got me thinking about flocks with more behaviors mixed in. Maybe there was even a way to move a flock without a leader. Essentially this ended up with me adding a wander behavior to the flocking blending. This essentially gave the flock some random directions to go, while at the same time trying to stay together. I had no clue what to expect at first, but the end product actually worked pretty well. Essentially the boids still managed to mostly stay together and at the same time kind of explore the screen randomly. They even went through the toroidal borders together!

CONCLUSION

Overall, this assignment gave me in depth experience and understanding into complex movement algorithms and how to effectively use them. Later features such as flocking showed how you can use seemingly basic algorithms to get amazing and interesting results. I also appreciated the amount of critical thinking required to complete the algorithms and assignment. Getting the algorithms to do exactly what you wanted required in depth understanding of the algorithm and constant manipulation of parameters to really understand how they effect the result. Inadvertently, building these movement algorithms has also honed my linear algebra skills to effectively implement them. Most importantly, this assignment gives me an excellent base block for my AI system to build on.

BIBLIOGRAPHY

Millington, I. *AI for Games Third Edition* CRC Press, 2019