# EAE6610 Assignment 1 Report

**Jonah Brooks**
Entertainment Arts & Engineering (EAE)
University of Utah
Jonah.Brooks@utah.edu

## ABSTRACT

This is the report for assignment 1 of EAE6610, AI for Games. In this report I detail how I approached the assignment of creating and using movement algorithms for simple AI in an openFrameworks C++ environment and the explorations I did to better understand the material. I detail how I implemented Kinematic Seek steering, Dynamic Seek steering, Dynamic Wander steering, and Flocking behavior.

## Keywords

AI for Games, EAE6610, Movement Algorithms

## INTRODUCTION

This assignment focused on creating and using AI movement algorithms in a C++ openFrameworks program. I was tasked with implementing various movement algorithms including Kinematic Seek, Dynamic Seek, and Dynamic Wander. I was also asked to blend Dynamic Separation, Velocity Match, and Dynamic Arrive to get a flock of AI agents to follow a center of mass guided by an AI agent moving around with Dynamic Wander.

## MOVEMENT ALGORITHMS

Details on each of the movement algorithms I implemented are detailed below.

### Kinematic Motion

For this section of the assignment, I was asked to implement a basic Kinematic Motion algorithm to move a Boid (a small representation of an AI agent) around the border of the screen using direct modification of the Boid's velocity. This section also served as a starting point for the other movement algorithms.

#### *Rigidbody*

Part of this section was to implement a basic 2D Rigidbody data structure to store each Boid's position, velocity, orientation, and rotational velocity. This same structure was used throughout the assignment for Boid's using kinematic and dynamic movement.

#### *Breadcrumbs*

Another part of this section was to make sure Boid's can leave a trail of breadcrumbs to indicate the path they have been traveling. For this, I chose to implement a class called AiAgent that stored, among other things, a deque of breadcrumbs. I also included in that class variables to dictate how many breadcrumbs that instance of a Boid should leave and how frequently a new breadcrumb should be generated.

### *Movement*
Finally, this section asked to have a Boid move in a clockwise circle around the edge of the screen (See Figure 1 for the path the Boid takes). To accomplish this, I chose to store a list of points distributed to the corners of the screen. Then I implemented and used Kinematic Seek to drive the Boid from point to point, with a simple distance check to determine if a point has been reached. Instead of implementing a formal Kinematic Align algorithm, I chose to just write the code to force the Boid to snap orientation to the direction of travel as part of a function in AiAgent called UpdateKinematic.

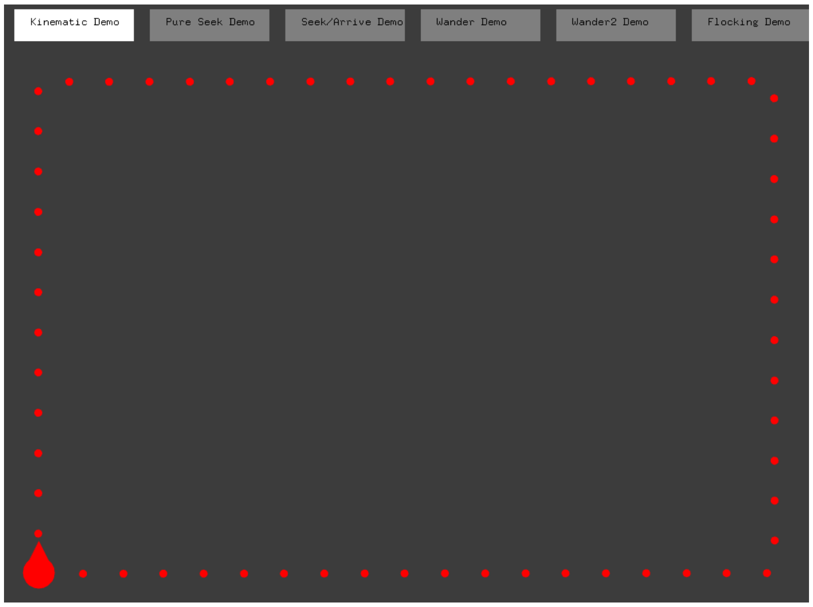

**Figure 1:** The Kinematic Seek demo after the Boid has traveled one full circuit. The small red dots are the breadcrumbs left behind showing the path the Boid took.

## Dynamic Seek
For this section of the assignment, I was asked to implement and use a Dynamic Seek algorithm to get a Boid to seek to the position of a mouse click. I implemented two demos for Dynamic Seek, one simply using drag to stop the Boid when it arrives and one that uses Dynamic Arrive. I also implemented the LookWhereYouAreGoing algorithm and placed a call to it at the bottom of AiAgent's Update function to ensure that the Boids' always try to face the direction they are going.

### *Pure Seek*
The first of the two Dynamic Seek demos I implemented is one that constantly calls the Dynamic Seek algorithm. This causes the Boid to overshoot the target point and oscillate back and forth after arriving. However, once I implemented drag into AiAgent's Update function, the Boid in this demo began stopping on the point after a few oscillations. Since this behavior functioned as a means of arriving at the point, I decided to leave it as one of the two Dynamic Seek demos instead of using a dedicated algorithm for arrival. (See Figure 2 for the path the Boid takes upon a single mouse click).

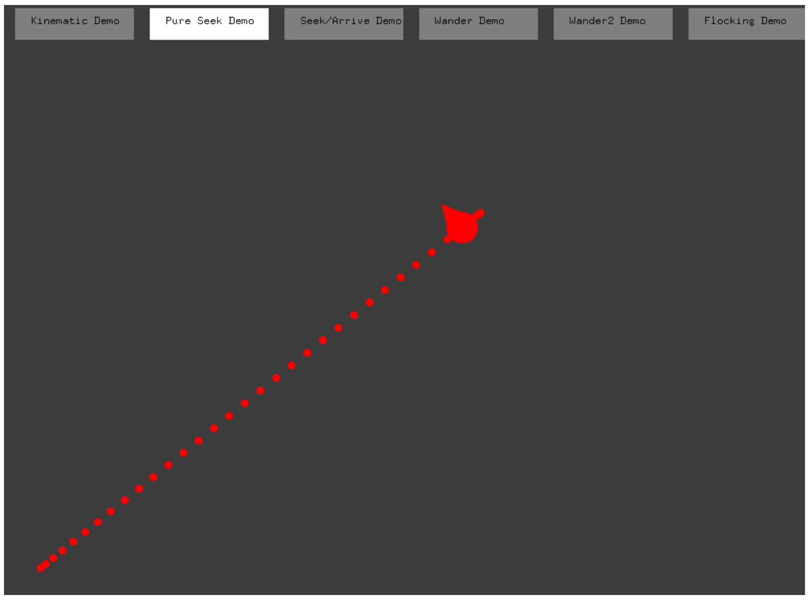

**Figure 2:** The path taken by the Boid to get to a mouse click. Note the red dots just to the upper right of the Boid indicating that the Boid overshot the target.

### *Seek/Arrive*

The second of the two Dynamic Seek demos I implemented is one that calls the Dynamic Seek algorithm until it reaches a certain distance to the target, then switches to using Dynamic Arrive. This has a much smoother look to it, as the Boid will slow to a stop as it reaches the target point rather than overshooting and oscillating (See figure 3). Before I implemented drag, however, this algorithm still overshot the target, but once drag was implemented this demo behaved exactly as I had hoped it would.

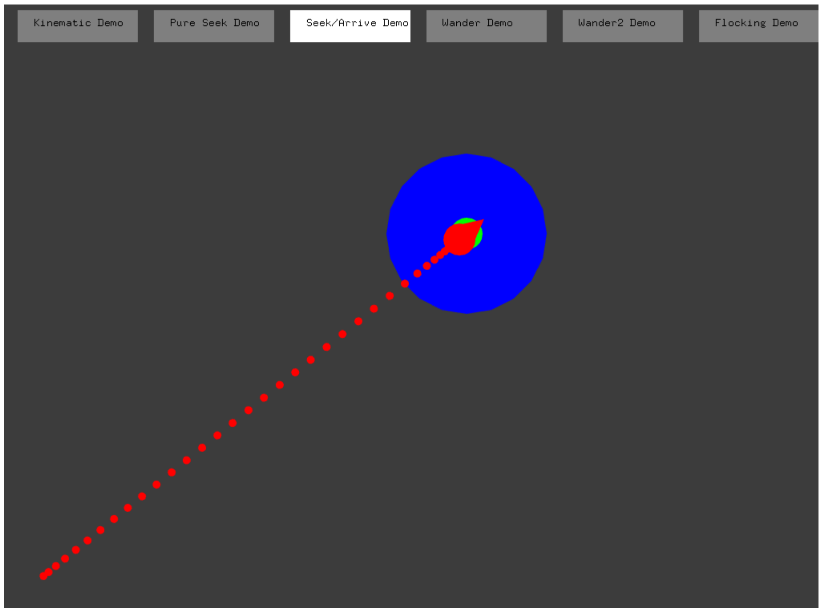

**Figure 3:** The path taken using Seek and Arrive.

### *Seek Conclusions*

In all, the second demo functions better, as it arrives more smoothly to the target point. However, it is noticeably more complicated, relying not only on a more complicated algorithm in Dynamic Arrive. I tried to mitigate this difference by only using Dynamic Arrive when close to the target point, but I felt the simplicity of the first demo warranted inclusion in my final submission for this assignment.

## Dynamic Wander

For this section of the assignment, I was asked to implement and use a Dynamic Wander algorithm to get a Boid to randomly, but not erratically, wander around the screen. It also asked for boundaries of the screen to be handled by wrapping the Boid back to the other side. For this section, I chose to implement a slightly modified version of the first Dynamic Wander algorithm we learned in class, as detailed below. I also made a slightly more complicated version with my own modifications in order to get output more in line with what I was hoping to see.

### *Wander Demo*

The first demo, titled Wander Demo in the app, uses a slightly modified version of what was first presented in class. In class, we were taught an algorithm that projected a vector in the direction of orientation of the Boid, then randomly appended a vector to the end to make the Boid change direction. At first, I implemented it as described, but I wasn’t satisfied with how little the Boid changed direction, and I was concerned about it changing velocity based on the magnitude of the resulting target vector. As such, I chose to make a single vector with a magnitude equal to the offset passed in, then rotate that vector by a random amount (constrained by other input parameters). This led to behavior much closer to what I was hoping to see, except for it tending to stick to a linear path. (See Figure 4 for the path this algorithm takes)

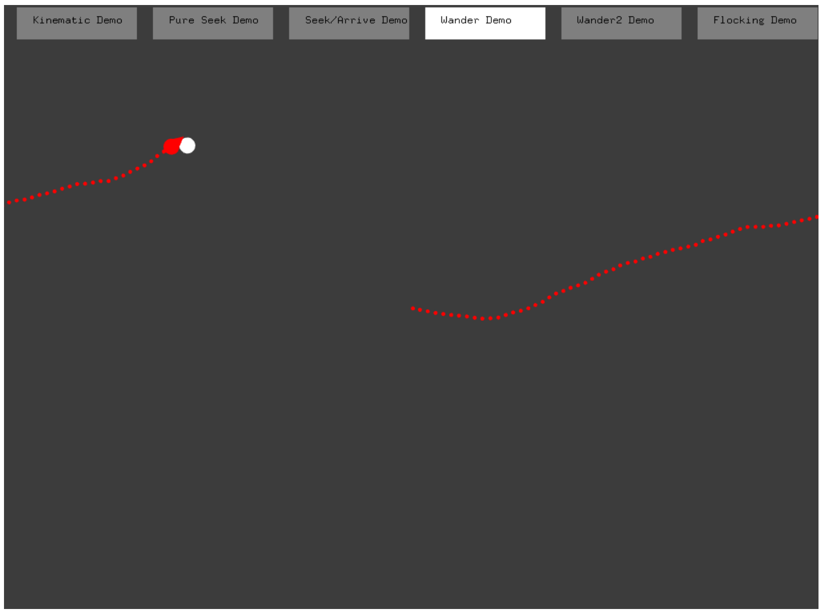

**Figure 4:** The path taken after a few seconds of wandering. Note the world wrapping on the edge and the mostly linear path. The white dot is the target position at the time of the screenshot.

### *Wander2 Demo*

The second demo, titled Wander2 Demo in the app, was my attempt at getting Dynamic Wander to behave more in line with what I wanted to see. In the first demo, even after my modifications, the Boid tended to travel in a straight line with slight deviations. I was hoping to generate behavior in which the Boid would travel around the screen in a path more like an ant wandering around, with some loops and U-turns and fewer straight-line paths. To get this behavior, I decided it was worth it to add a bit of complexity to the algorithm. I used the same basic algorithm structure as in the first Dynamic Wander demo but added a bit of state tracking that made the Boid tend to favor slight turns in the same direction as the last call. The effect of this change is that, instead of mostly tending to follow a straight line with some random deviations in either direction, the Boid tended to follow gradual arcs while occasionally turning back to a straight line or arc in the other direction. (See Figure 5 for an example of the path this algorithm tends to take).

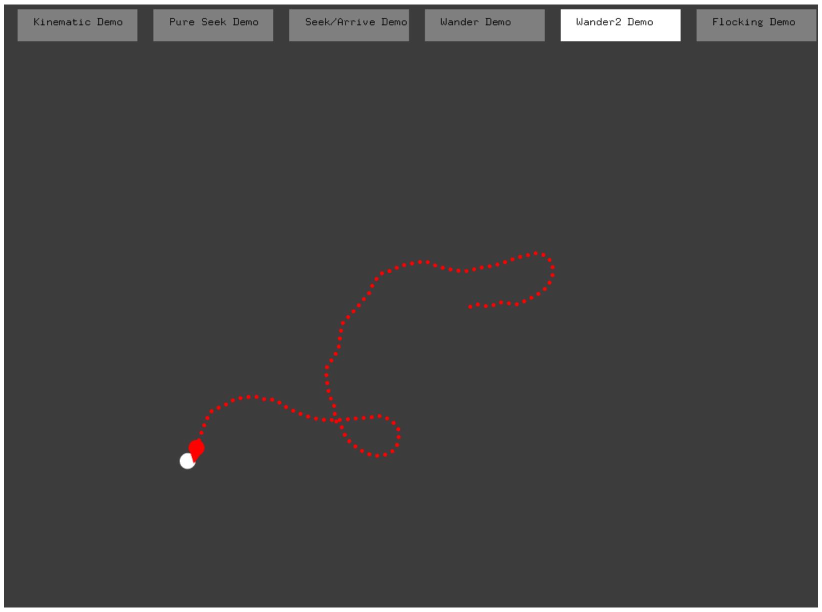

**Figure 5:** The path taken by my modified Wander algorithm. Note the U-turn and the loop, which is something that would be very rare in the original algorithm.

### *Wander Conclusions*

While the second algorithm is more complicated, it matches the “wandering ant” style I was hoping for better than the first. I can’t really say which is the better algorithm, as they each have their own tradeoffs. The first algorithm is simpler, and therefore faster, while still producing a decent wander effect. However, I am partial to the second algorithm due to it tending to wander a bit more fervently and rely less on straight lines and slight oscillations of heading.

## **Flocking**

The final section of this assignment was to implement and use the algorithms needed to create a Flocking behavior. For this, I mostly followed along with what was described in class. I implemented Separation and Velocity Match, and reused Dynamic Arrive. I then spawn in many small Boids and one larger one. The larger one follows the Wander2 algorithm I discussed above, while the smaller ones use a blending of the three algorithms mentioned in the last sentence. I toyed around with some of the inputs to the various functions to get a good feel for the flock, but I didn't have to change any of the algorithms to make it work. The biggest thing I found myself needing to tweak was the inputs to Separate. It took me a while to get the Boids to not just crash into each other constantly. I was able to get a decent flocking behavior in the end, but I can't say a great deal about what changes in the algorithms would do to the outcome.

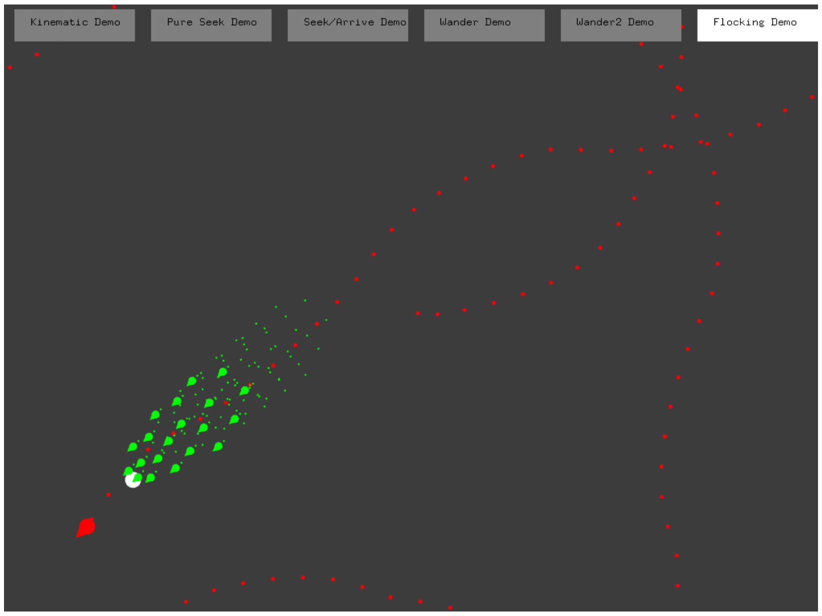

**Figure 6:** The red Boid is the leader. The green Boids are the followers. The white dot is the center of mass between the leader and the followers.

## CONCLUSION

In all, this was a fun assignment to work on. I enjoyed seeing the different behaviors and playing around with the inputs to the algorithms to see how the Boids would behave. I'm quite happy with the different demos I put together, even though I do wish I had more time to really sink my teeth into different implementations.