Game AI Movement Algorithm Simulations

Wei Zeng
University of Utah
380 S 400 E APT 414
Salt Lake City, UT, USA
(858)6665965
u1330357@utah.edu

ABSTRACT
In this report I would like to discuss various kinds of movement algorithms and how they perform with different input parameters and methods of implementations. The evaluation of performance was based on observation on the simulation using openframework.

Keywords
Game AI, Kinematic Movement Algorithm, Dynamic Movement Algorithm

INTRODUCTION
Movement algorithms form the most fundamental behaviors in a game AI system therefore making it extra important for ai/game engineers to have detailed understanding of them. In this simulation, I will explore some of the findings during experiments and discuss the various differences in behaviors. Also I will try to identify the advantages and disadvantages of each movement algorithm and mention the solution.

KINEMATIC MOTION
Kinematic movement is the simplest of all steering algorithms in this paper since it does not take acceleration into account. The example of a kinematic movement would be a kinematic seek algorithm which moves toward its target at the given max speed. There exist many disadvantages for an algorithm like kinematic seek. For example, since the character always moves at full speed, it lacks the ability to generate a physically immersive movement.

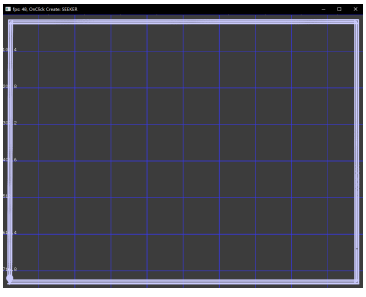

Figure 1: This is an example of Kinematic Seek seeking the corners of the window (trails in purple).

SEEK STEERING BEHAVIOR

Seeking targets could be one of the most ubiquitous behaviors in video games. From the above section we discussed a bit about a kinematic way of implementing seeking and also mentioned its flaw in presenting physics simulations. In this section, I will discuss a few dynamic seeking algorithms that help solve the issue.

Dynamic Seek

The biggest difference between a dynamic and kinematic seek is that dynamic seek outputs acceleration. Instead of traveling at full speed, a dynamic seek will accelerate as fast as it could to match the position of its target. This effectively generates a smooth transition from stationary to dynamic (full speed). Though such an algorithm handles the change of position well, it does not output anything to fix the orientation of the character. To resolve this issue, the simulation calls LookWhereYouAreGoing to perform a smooth orientation matching to match its velocity's direction. Though everything looks smooth both positional and orientational wise, one biggest flaw of Seek is that it never stops, meaning that after arriving at the target position, with the velocity it has on itself, it will keep going beyond the target for a bit, which could be problematic in scenarios where we want the stop at the location of the target in game. One interesting finding through the simulation is that after running seek for a few minutes the seeker eventually orbits around the target (see Figure 2, 3).

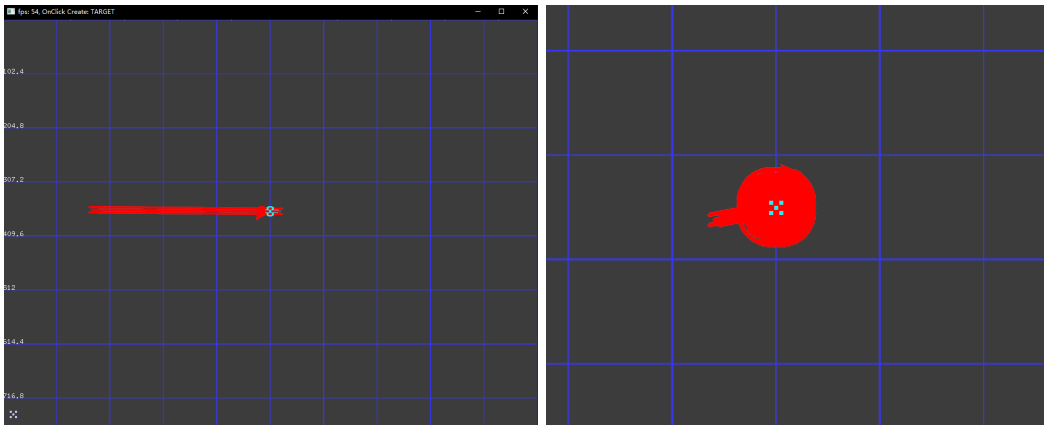

Figure 2, 3: This is the captions for Seek(red) seeking the target in light blue, as we could see from the trail, it goes back and forth at the position of the target.

Dynamic Arrive

Dynamic Arrive is perfect for solving the issue of seek due to its special design. The algorithm takes in extra parameters to create buffer zones to slow down and stop. A figure below shows the difference between seek and arrive with arrive stops at the position of its target whereas seek keeps going back and forth. Though the problem of stopping is solved, another issue popped up during simulation. What if the target travels

at the same speed of arrive/seek? Since both algorithms travel in the direction towards the target, it means they could only be following with no chance of catching up.

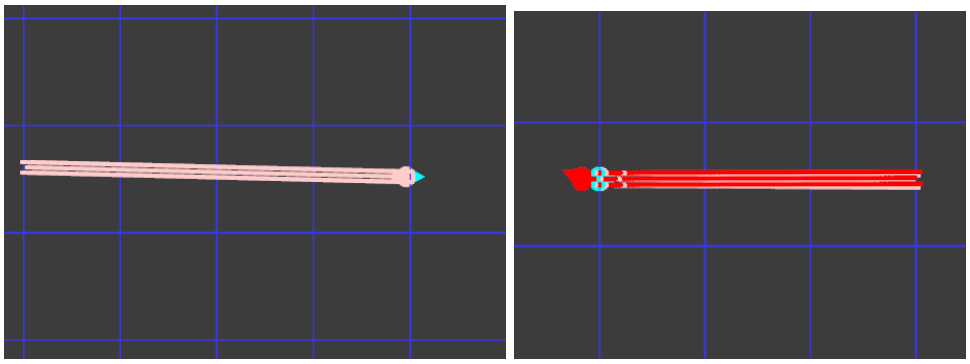

Figure 4, 5: This is the captions for Arrive(pink) seeking the target in light blue, as we could see from the trail, it stops at the position of its target whereas seek keeps going back and forth.

Dynamic Pursue

To solve this problem, yet another seek steering behavior needs to be introduced. Dynamic Pursue is a more advanced algorithm that would calculate a predicted target position according to the target's velocity. Such design gives potential for the character to catch up with targets traveling at the same or greater speed by taking a "shortcut". A figure below shows how Pursue was able to catch the target faster than everyone else.

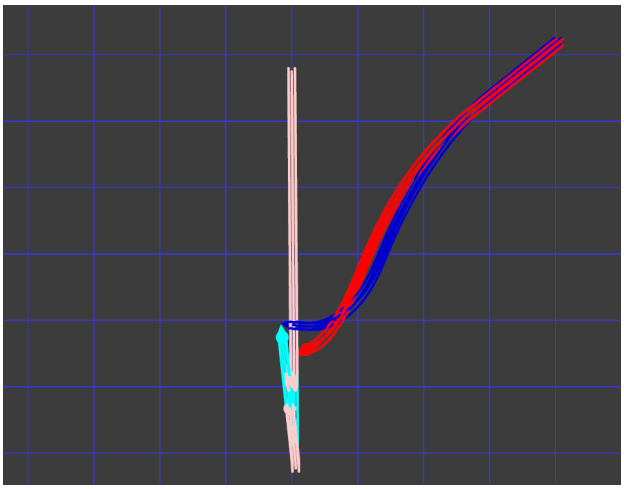

Figure 6: This is the caption for comparing Seek(red) and Pursue(blue) with pink trail as Arrive boids and light blue as target with evading Arrive boids.

WANDER STEERING BEHAVIOR

The Dynamic Wander behavior was implemented the same way we discussed in class by putting up an explicit target. I found it a quite challenging yet interesting steering behavior to experiment with. Wander behavior takes in a lot of input parameters that could each affect the behavior tremendously. Out of all these parameters, I found wanderOffset to be the most interesting. With experiments, I found that with great wanderOffset, our boids would have a much smaller tendency of having an aggressive trail by which I mean a winding trail. From figures below, we could see that with a large wanderOffset (50), the trails of our boids are more linear (see Figure 7) than the trails left by boids using a smaller wanderOffset (5) (see Figure 9).

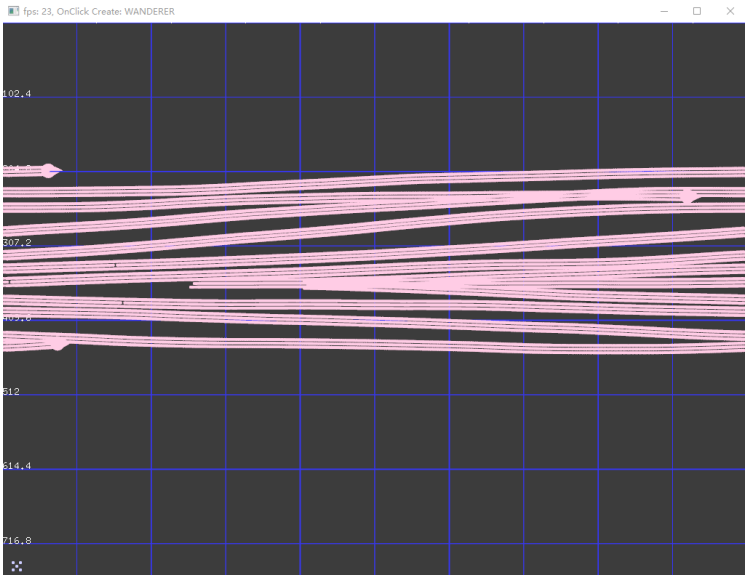

Figure 7: This is the caption for wandering behavior using a large wanderOffset with Face as the method of changing orientation.

In addition to input parameters, wandering behaviors could also vary with different delegation to the orientation changing behaviors. Such differences were also interesting findings which took a little bit of experimentation to figure out. In my simulation, I used LookWhereYouAreGoing (LWYAG) and Face as my two orientation changing options to delegate by wander. With the first one (see Figure 9), it is obvious to see that our wandering boids are not taking many big turns. This is because with “LWYAG”, the algorithm outputs an angular acceleration for our character boid to accelerate to the direction of current velocity. Since it is an acceleration, the velocity is not going to point in the direction of the explicit target direction yet, meaning that our boid will have a smaller turning each frame corresponding to the size of the acceleration. However, for Face behavior, because it tries to accelerate to the full angle difference between the explicit target’s orientation and character’s orientation (see Figure 8), the turns will become much larger than those using LWYAG (see Figure 9, 10).

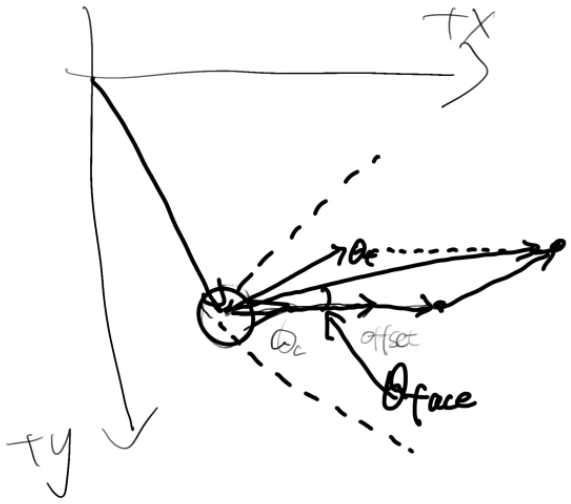

Figure 8: This is the caption for wandering behavior using small wanderOffset with Face as the method of changing orientation.

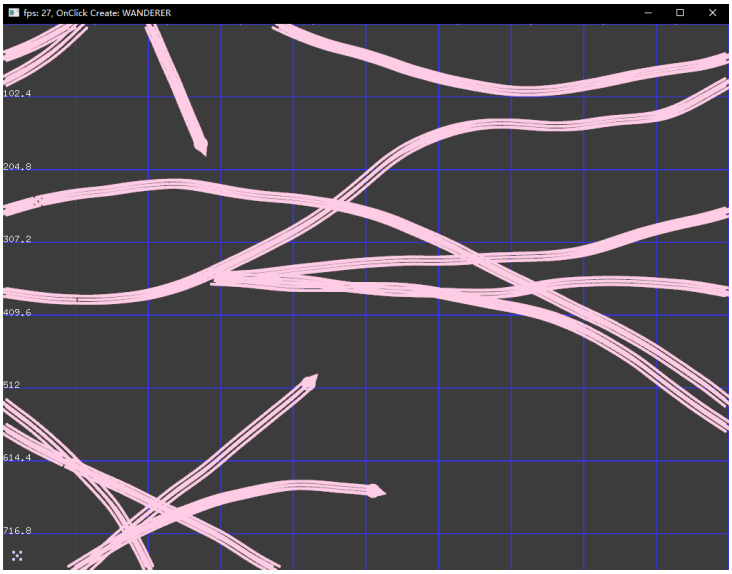

Figure 9: This is the caption for wandering behavior using small wanderOffset with Face as the method of changing orientation.

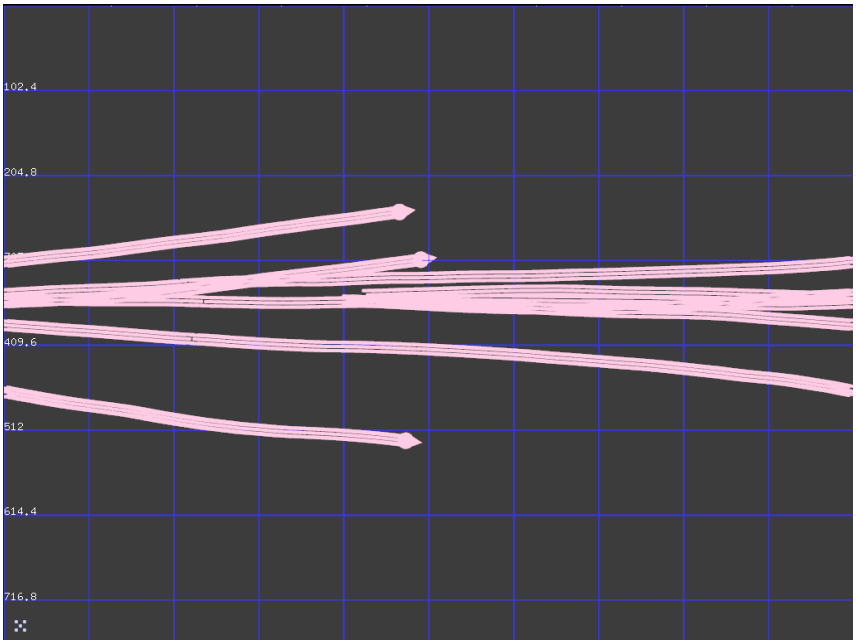

Figure 10: This is the caption for wandering behavior using small wanderOffset with LookWhereYouAreGoing as the method of changing orientation.

FLOCKING BEHAVIOR AND BLENDING/ARBITRATION

Without a doubt flocking is the most complicated behavior among everything we have discussed so far due to the fact that it requires multiple other behaviors working together to make it happen. In this simulation, flocking is more or less implemented the same way as what was discussed in class by calculating the centroid's position and velocity then delegate to separation, velocity match and arrive.

One interesting finding was when the leader went beyond the window and appeared from the opposite side of the screen, followers at the original side of the window would turn around to seek the leader coming out from the new side (see Figure 11). I was worried about such behavior but then realized that this could be caused by the shift of centroid's position since the leader carries the weight all over to the other side.

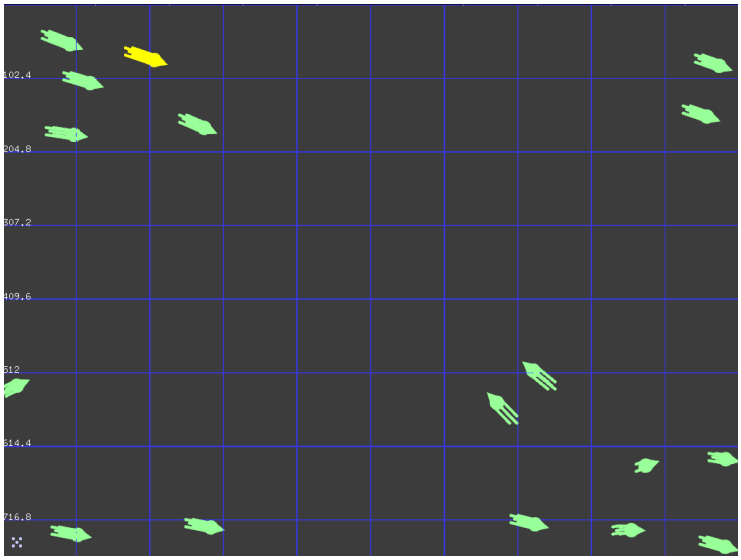

Figure 11: This is the caption for flocking behavior when the leader(yellow) comes out from the opposite side of the screen after going beyond the original side.

Originally, I thought adding more than one wandering leader to the flock would make the flock extremely unstable. However, after experimenting with the flock, I realized that this would not necessarily be the case as all followers, as time progressed, seemed to form a cohesion and travel to, basically, the centroid of the 3 leaders due to the great weights (see Figure 12).

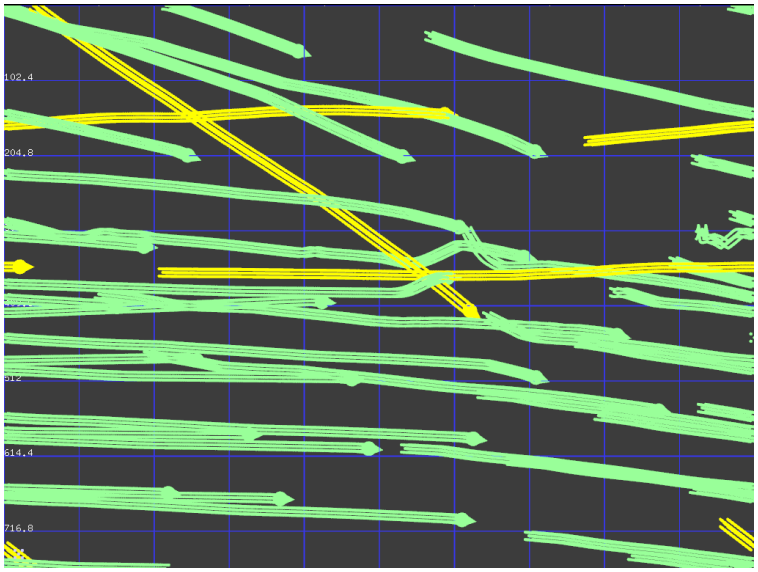

Figure 12: This is the caption for flocking behavior with 3 leaders in the flock.

Conclusion

After this project the most valuable takeaway is that I have gained a much better understanding of movement AI behaviors. More specifically, I realized that in game AI development, a good performing algorithm requires lots of effort in fine tuning and also in order to tune it in the right direction it also requires the developer to have a good understanding of the algorithm itself. This is the case with my experiment when I was developing my wander behavior because my first run of the simulation was looking more like traveling in a straight line. After reading more on the code and grasping a better understanding of the algorithm, I was able to figure out the effect of wanderOffset and orientation changing algorithm have on turning. At the end of everything, I came to the conclusion that there are just no strict correct answers to whatever we are putting into the algorithm and all it matters is that the behavior it generates matches the expectation of us as human beings.