Multi-agent movement behaviors in game AI

Jordan Lewis

June 10, 2010

Abstract

A common problem in some games with large numbers of AI agents is encoding individual behavior in such a way that a coherent group behavior is also formed. For example, a squad of enemy troops might look more realistic moving toward the player in a fixed (i.e. triangle) or fluid (i.e. follow a leader) formation than simply all trying to run at the player directly. We will explore various algorithms and methods for coordinated multi-agent movement in game AI. Some of these behaviors will not require inter-agent communication of more than just what's knowable from the physics environment, and some will - we will also attempt to note the differences in efficacy for these two sorts of algorithms.

1 Introduction

This paper attempts to explore a few different group AI behaviors. The paper is accompanied by a demo that showcases these behaviors - read the README in this directory for more information on how the demo works.

The behaviors we will examine hilight some different key decisions necessary when designing a group AI: the choices between centralized and decentralized information, and between uniform and nonuniform, behaviors across agents. I define a uniform group AI behavior to be running identically across all agents, as opposed to a non-uniform behavior which requires different algorithms to be run on different agents, for leader behavior for example. I define a centralized group AI algorithm as one that allows the agents to share state, as opposed to a decentralized algorithm that doesn't allow inter-agent communicaton.

2 Behaviors

2.1 Flocking

Flocking is one of the most well-known group AI algorithms there is. It was created by Craig Reynolds in 1987, for a demo called boids. It's a very simple uniform and decentralized algorithm that produces realistic and interesting results.

Flocking behavior is created by blending together three steering behaviors based on the average behavior of the other agents in neighborhoods of differing size around the current agent.

1. Separation ensures that agents don't get too close together, by steering away from the average position of the agents in the neighborhood.

2. Cohesion ensures that agents clump together to form flocks, by steering toward the average position of the agents in the neighborhood.

3. Matching ensures that agent clumps tend to move all together, by steering to try to match the average heading and velocity of the agents in the neighborhood.

I used the values 5, 10, and 15 for separation, cohesion, and matching respectively for my demo. This produced fairly good results, but typically the clumps would tend to lose speed and reach a static equilibrium. I solved this by adding a wander behavior with a low weight - this ensures that agents keep moving around within their flock, which in turn motivates the flock to move as a whole.

I implemented this algorithm with knowledge that the agents would not typically be allowed to have - the actual positions, velocities, and orientations of all the agents in the simulation. This is an optimization: perhaps the more genuine approach would be to have each agent scan its neighborhood with raycasts for other agents, and detect their velocity by sampling, but this is very computationally intensive and would produce results that were at best as interesting as the algorithm that uses global information.

2.2 Follow the Leader

Follow the leader is a very simple non-uniform, decentralized behavior. One leader agent wanders around the world, and the other agents attempt to seek to the leader's position.

Naively, this algorithm didn't produce very good results. The seek behavior resulted in oscillatory behavior where the seekers would continually overshoot the leader. This was because the wandering agent's velocity was never nearly as high as the seeking agents', due to the random acceleration changes in the wander behavior. Secondly, the agents would all ram into each other in a mad dash to get to the leader's position.

The overshooting oscillation behavior was solved by using the arrive behavior with a large slow-down radius around the leader. This produces smooth results that appear as though the agents are trying to follow behind the leader at some kind of respectful distance. The collision problem was solved by re-using the separation steering algorithm from the flocking behavior.

2.3 Static Formations

Static formations are another kind of non-uniform but decentralized group AI behavior. A static formation algorithm simply defines a relative position to the leader given an integer spot. This can be used to make simple geometric shapes for the agents to attempt to conform to.

I implemented two static formations in the demo: a line of agents abreast of the leader, and a flying vee formation. These produce very nice looking results at minimal effort: with a simple inheritance relationship, an AI behavior writer literally only has to specify a tiny 'getSpot' function. Here's the function I wrote to define the line behavior.

```
pos = k.pos + k.vel.perp(Vec3f(0,1,0)).unit() * slot * (slot % 2 ? -1 : 1);
```

Given a kinematic k with position and velocity information of the leader, we define a new position vector that's orthogonal to the leader's velocity vector, alternating left and right, and scaled by the current agent's slot.

2.4 Dynamic Formations

Dynamic formations are similar to static formations in intent, but they offer more flexibility in terms of managing lots of agents in one formation all at onec. Instead of assigning a relative position to an agent based on only its slot number, dynamic formations also take into account the number of total agents participating in the system. This allows for more interesting resizing or changing of the formation. I implemented a simple resizable circle formation to demonstrate the dynamic resizability of the algorithm. This is easily demonstratable by adding or removing agents while the defensive circle behavior is still active.

2.5 Emergent Formations

Emergent formations encompass behaviors like flock, whose interesting and nuanced behavior emerges from preconditions that alone do not indicate such behavior. Another example of an emergent formation is the emergent vee behavior from Millington and Funge.

The idea of the emergent vee is that agents try to find other agents that are ahead of them, and seek to a target that's behind and to the left or right of that agent, depending on whether another agent is already occupying that slot. According to the text, this simple steering behavior leads to a recursive vee not unlike a sparsely populated binary tree. I wasn't able to get this behavior to work, however. The difficulty is that, as the text describes, it's hard to prevent agents from fighting over a slot: more often than not, agents would simply clump up behind an agent that was once in front. I think this could be solved with a more sophisticated geometrical definition of where the vee-slots really are in relation to neighbors.

3 Implementation techniques

This project was a joy to implement, partially because of the framework that I had been previously working on with the rest of z-fight-club, and partially because such AI behaviors easily fall into an object-oriented-style inheritance relationship.

Borrowing from the group project, *Steerables* are controlled by *AIControllers* using *SteerInfo* structures, which contain an acceleration vector and a rotation. For this demo I didn't need to use a a separate heading; instead I treated the unitized velocity vector as the heading. AIControllers are managed by the *AIManager*, which is in charge of changing everyone's strategy when the user requests it.

AIControllers own a pointer to a single *AIStrategy*, which is the top of the hierarchy that controls the lion's share of the AI programming. The AIStrategy hierarchy follows the "Strategy" design pattern, in that all of its subclasses are drop-in replacements for AIControllers' AIStrategy pointers. This is accomplished by keeping a uniform API, in which calling getSteering() on any AIStrategy will return the SteerInfo structure it requests from the steerable character.

Subclasses of AIStrategy implement their own functionality as well. Notably, as mentioned above, the *StaticFormation* class makes it easy to define a new static formation by simply overriding the getSpot() method.

Finally, the BlendedStrategy subclass implements an interface for weighted strategy blending. Children or owners are allowed to add a new SteerInfo output structure along with an associated weight to the blend. When getSteering() on the BlendedStrategy instance is called, the weighted average of the outputs is taken and returned. This makes it easy to implement more complex blends of AI behaviors, like in the Flock demo.

4 Sources

1. Individual-based models - Craig Reynolds - `www.red3d.com/cwr/ibm.html`
2. Steering behaviors - Craig Reynolds - `www.red3d.com/cwr/steer/`
3. Path planning algorithm of maintaining CGF team formation - Zheng et al. - `www.computer.org/portal/web/csdl/doi/10.1109/CSIE.2009.495`
4. Team AI: probabilistic pathfinding - John et al. - `portal.acm.org/citation.cfm?id=1234341.1234372&coll=ACM&dl=ACM&CFID=86476434&CFTOKEN=46450234`
5. Artificial Intelligence for Games - 2nd Ed - Millington and Funge