Journal of Korea Multimedia Society Vol. 13, No. 5, May 2010(pp. 763-770)

대규모 무리 짓기에서 이웃 에이전트 탐색의 개선된 알고리즘

이재문[†], 정인환[††]

요 약

본 논문은 무리 짓기에서 공간분할 방법의 성능을 개선하는 알고리즘을 제안한다. 무리 짓기에서 여러 특성중의 하나는 두 에이전트가 공간적으로 가깝게 있다면 많은 공동 이웃들을 공유한다는 것이다. 본 논문은 이 특성을 적용하여 공간분할 방법을 개선한다. 기존의 공간분할 방법이 한 번에 하나의 에이전트에 대한 k개의 가장 가까운 이웃 에이전트들을 찾는 것에 반하여, 제안하는 방법은 에이전트들이 공간적으로 가까이 있다면 그들에 대하여 동시에 k개의 가장 가까운 이웃 에이전트들을 계산한다. 제안된 알고리즘은 구현되었으며, 그것의 성능은 기존의 공간분할 방법과 실험적으로 비교되었다. 비교의 결과로부터 제안하는 알고리즘이 기존의 방법을 평균적으로 33%정도 개선한다는 것을 알 수 있었다.

An Improved Algorithm of Searching Neighbor Agents in a Large Flocking Behavior

Jae Moon Lee[†], In Hwan Jung[††]

ABSTRACT

This paper proposes an algorithm to enhance the performance of the spatial partitioning method for a flocking behavior. One of the characteristics in a flocking behavior is that two agents may share many common neighbors if they are spatially close to each other. This paper improves the spatial partitioning method by applying this characteristic. While the conventional spatial partitioning method computes the k-nearest neighbors of an agent one by one, the proposed method computes simultaneously the k-nearest neighbors of agents if they are spatially close to each other. The proposed algorithm was implemented and its performance was experimentally compared with the original spatial partitioning method. The results of the comparison showed that the proposed algorithm outperformed the original method by about 33% in average.

Key words: Flocking behavior(무리 짓기), Spatial partitioning(공간분할), Agent(에이전트), k-nearest neighbors(k개의 가장 가까운 이웃들)

1. 서 론

무리 짓기는 자연에서 쉽게 볼 수 있는 행동이다. 예를 들어 무리를 지어 날아다니는 새들, 어항속의 물고기들 수영, 훈련된 개에 의하여 조종되는 양들의 움직임 등이 있다. 이러한 행동에서 많은 무리들은 하나의 일정한 형태를 유지하는 구조를 갖는다[1,4,9]. 'V'자 형태로 날아다니는 새들의 행동이 그 예이다.

※ 교신저자(Corresponding Author) : 이재문, 주소 : 서울 성북구 삼선동2가 389 한성대학교 멀티미디어공학과(136-792), 전화 : 02)760-4135, FAX : 02)760-4488, E-mail : jmlee@hansung.ac.kr

접수일 : 2009년 11월 30일, 수정일 : 2010년 1월 24일
완료일 : 2010년 1월 25일

[†] 종신회원, 한성대학교 멀티미디어공학과 교수
[††] 종신회원, 한성대학교 컴퓨터공학과 부교수
(E-mail : ihjung@hansung.ac.kr)

※ 본 연구는 2009년 한성대학교 교내연구비 지원과제임.

무리 짓기는 리더도 없고 중앙 제어도 없는 특징을 가진다. 그것은 가까운 에이전트들끼리 상호작용에 의하여 나타나는 현상이다. 무리 짓기에 대한 첫 연구는 Reynolds에 의하여 컴퓨터를 이용한 새들의 비행에 대한 시뮬레이션으로 시도 되었다. 여기서 각 에이전트는 무리의 전체에 대한 어떠한 정보도 없이 단순히 지형에 대한 정보와 이웃 에이전트들의 영향만 고려하여 비행한다. [1-3]에서는 이웃들의 영향을 분리힘, 정렬힘 및 결합힘으로 구성하였다. 분리힘은 이웃 에이전트간 충돌을 피하기 위한 힘이며, 정렬힘은 이웃 에이전트들과 같은 방향으로 움직이려는 힘이며, 마지막으로 결합힘은 이웃 에이전트들과 너무 멀리 떨어지지 않으려는 힘이다. 이러한 힘들의 계산에서 가장 큰 부하를 주는 것이 이웃 에이전트를 탐색하는 것이다.

무리 짓기에서 이웃 에이전트를 결정하는 방법에는 크게 두 가지가 있다[7]. 하나는 일정 반경 r을 정하여 대상 에이전트를 중심으로 반경 r내에 존재하는 모든 에이전트들을 이웃 에이전트로 정의하는 방법이며, 다른 하나는 임의의 상수 k를 정하여 대상 에이전트를 중심으로 가장 가까이 존재하는 k개의 에이전트들을 이웃으로 정의하는 것이다. 두 방법 사이에는 trade-off가 있다[5]. 전자는 일정한 범위 내에 존재하는 모든 에이전트들이 이웃 에이전트가 되므로 이웃 에이전트를 찾는 것이 비교적 쉬운 장점이 있으나, 최악의 경우 모든 에이전트가 이러한 반경 내에 존재하는 경우 모든 에이전트가 모든 이웃 에이전트로 고려되어야 하는 단점이 있다. 이것은 상기 세 가지 힘을 계산할 때 속도 면에서 심각한 문제를 야기한다. 후자의 경우 주변에 존재하는 에이전트가 적은 경우 매우 먼 거리의 에이전트도 포함하여야 하므로 이웃 에이전트를 찾는 비용이 클 수도 있는 단점이 있는 반면, 항상 이웃의 수가 k개로 고정되어 있기 때문에 세 가지 힘을 계산하는 속도가 일정하다는 장점이 있다. 따라서 무리 짓기의 시뮬레이션 환경에 따라 두 방법 중의 하나가 선택될 수 있다. 본 논문은 후자의 경우에 대하여 효율적으로 이웃 에이전트를 탐색하는 알고리즘 제안한다.

n개의 에이전트들이 게임 공간에 존재할 때 매 프레임마다 각각의 에이전트에 대하여 k개의 이웃 에이전트를 탐색하여야 한다. KD-트리 등과 같은 특별한 보조 자료구조가 없는 경우 하나의 에이전트에 대한 k개의 가장 가까운 이웃을 찾는 비용은 $O(n)$이다. 이것은 하나의 에이전트에 대하여 n-1개의 다른 모든 에이전트와 거리를 계산하고 그 중에서 가장 가까운 거리에 있는 k개의 에이전트를 탐색하여야 하기 때문이다. 따라서 모든 에이전트에 대하여 이를 계산하여야 하므로 그 비용은 $O(n^2)$이 된다[1,6,10]. 이러한 비용은 매 프레임마다 계산되어야 한다. 즉, 게임이 최소한 초당 30프레임의 화면을 보인다면 초당 30번씩 $O(n^2)$ 만큼 계산되어야 한다. 따라서 대규모 무리 짓기와 같이 n이 큰 경우 이 비용은 매우 크다. 이러한 비용을 줄이기 위하여 [1,9,10]에서는 공간분할 방법을 사용해 왔다. 이 방법은 매 프레임마다 에이전트들을 분할된 공간에 매핑하므로써 이웃 에이전트들을 효율적으로 탐색하도록 한다. [1,9,10]에 의하면 이 비용은 $O(kn)$이 된다. 대부분의 경우 k가 n에 비하여 매우 작으므로 이 방법은 무리 짓기의 성능을 크게 향상시킨다. 본 논문은 별도의 추가 비용 없이 공간분할에서 이웃 에이전트를 보다 효율적으로 탐색하는 알고리즘을 제안한다.

2장에서는 기존의 무리 짓기 방법인 공간 분할 방법을 소개하며, 이러한 무리 짓기의 특성을 분석한다. 3장에서는 2장에서 분석된 무리 짓기의 특성을 활용하여 효율적으로 이웃 에이전트를 탐색하는 알고리즘을 제안하며, 4장에서는 기존의 방법과 실험적으로 성능 비교를 한다. 마지막으로 5장에서 결론을 논한다.

2. 공간분할 무리 짓기 및 무리 짓기의 특성

2.1 공간분할 무리 짓기 알고리즘

무리 짓기에서 에이전트들은 지속적으로 움직이기 때문에 매 프레임마다 위치가 변한다. 이러한 사실 때문에 무리 짓기에서는 KD-트리 등과 같이 삽입/삭제 비용이 비싼 자료구조는 사용할 수 없다. 공간분할은 가장 단순한 구조이다[3,8,9]. 먼저 게임 공간을 하나의 큰 큐빅으로 나타내고, 하나의 큰 큐빅을 여러 개의 작은 셀로 분할한다. 다음 모든 에이전트에 대하여 그들의 위치에 근거하여 이들을 큐빅내 해당 셀에 할당한다. 이렇게 하면 게임 공간상의 모든 에이전트들은 그들의 위치에 따라 큐빅의 하나의 셀에 할당되게 되는 것이다. 분할된 셀의 수나 특정한 위치로 에이전트들의 쏠림 현상 등으로 하나의 셀에

여러 에이전트가 할당될 수도 있다. 그림 1은 2차원 공간에 대한 공간 분할의 예이다. 공간은 10*10로 분할되었다. 셀 [1, 1]에는 한 마리의 새가 할당되었지만 셀 [6, 4]에는 세 마리의 새가 할당되었다.

대부분의 구현에서는 성능 향상을 위하여 공간분할을 위한 큐빅의 생성은 초기에 한번만 하고, 매 프레임마다 셀의 위치가 변경된 에이전트들에 대하여 기존의 셀에서 삭제를 하고, 새로운 셀에 삽입을 한다. 이러한 과정은 트리 형태의 자료구조에 비하여 훨씬 단순하고 빠르다. 그림 2는 공간분할 무리 짓기 알고리즘의 의사코드이다.

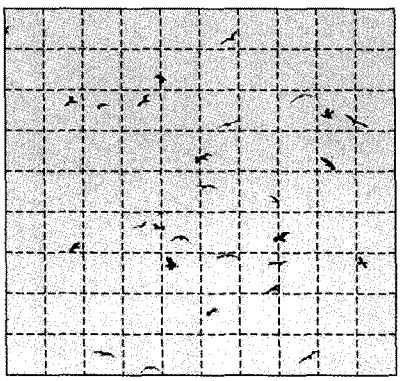


그림 1. 2차원 공간에서 공간분할의 예

Algorithm *FlockingPS*: **Inputs:** *agents, k, t* **Outputs:** *None*

```
01: // 공간분할 업데이트
    foreach q in agents{
02:   if(Cubic(q.oldLoc) != Cubic(q.newLoc)){
03:     Cubic(q.oldLoc).Delete(q);
04:     Cubic(q.newLoc).Insert(q);
05:   }
06:   q.oldLoc= q.newLoc;
07: }
08: // 이웃 에이전트 탐색 및 조종힘 계산
    foreach q in agents {
09:   Q= GetNearKnn(q, k, Cubic);
10:   kNN= GetKnn(q, k, Q);
11:   q.force= ComputeSteerForce(q, kNN)
12: }
13: // 에이전트의 새로운 위치 계산
    foreach q in agents {
14:   a= q.force/q.mass;   // f=ma
15:   q.newLoc= q.oldLoc+at+a t^2/2;
16: }
```

그림 2. 공간분할 무리 짓기 알고리즘

그림 2의 알고리즘 *FlockingPS*에서 입력은 에이전트의 무리인 *Agents*, 이웃 에이전트의 수를 나타내는 k, 이전 프레임과 현 프레임간의 시간차이인 t이다. 상기 알고리즘은 크게 3 부분으로 나뉜다. 첫 부분은 공간분할을 업데이트하는 것이다. 앞에서 언급하였듯이 모든 에이전트들은 매 프레임마다 그들의 위치를 변경하므로 그들이 매핑되는 큐빅의 셀도 변경될 가능성이 있다. 따라서 알고리즘에서는 이를 검사하여 필요한 경우 변경시켜야 한다. 그림 2에서 1-7라인 사이의 코드가 이것을 한다. *Cubic(location)*은 주어진 위치 *location*에 매핑되는 큐빅의 셀을 리턴하는 함수이다. 3-4라인은 셀이 변경된 에이전트에 대하여 과거의 셀에서 삭제를 하고 새로운 셀에 삽입하는 과정을 나타낸다. 두 번째 부분은 각각의 에이전트에 대하여 이웃 에이전트를 탐색하고, 이를 이용하여 분리힘, 정렬힘 및 결합힘으로 구성되는 조종힘을 계산하는 것이다. *GetNearKnn*은 큐빅을 이용하여 해당 에이전트와 가장 가까운 $k'(\geq k)$개의 에이전트를 찾는 함수이며, *GetKnn*은 9라인의 S에 저장된 k'개의 에이전트로부터 k개의 가장 가까운 이웃들을 찾는 함수이다. 이 함수는 단순히 k'개의 에이전트들을 해당 에이전트와의 거리를 기준으로 정렬하는 것이다. *GetNearKnn*과 *GetKnn*은 하나의 함수로 구현되어도 된다. *ComputeSteerForce*은 조종힘을 계산하는 함수이다. 이에 대한 계산은 [2, 3]을 참고로 하였다. 마지막 부분은 각 에이전트에 대하여 새로운 위치를 계산하는 것이다. 각 에이전트들은 자신의 질량(대부분의 경우 에이전트들의 질량은 동일함)을 갖고 있다. 따라서 뉴턴의 힘의 법칙에 따라 조종힘을 이용하여 이 에이전트에 가해지는 가속도(a)를 구하고 이를 이용하여 새로운 위치를 계산한다.

*FlockingPS*에서 가장 큰 부하를 가진 함수가

Function GetNearKnn: Inputs: *agent, k, Cubic* Outputs: *agents*

```
01: agent가 매핑된 셀을 중심셀로 포함하고 k개 이상의
    에이전트를 포함하는 서브큐빅 Cm을 찾는다
02: Cm에서 agent와 가장 멀리 떨어진 에이전트 p를 찾
    고 두 에이전트 간 거리 (r)을 계산한다.
03: agent를 중심점으로 반경 r내에 존재하는 모든 에이
    전트를 찾아 agents에 저장한다.
04: agents를 리턴한다
```

그림 3. 함수 GetNearKnn의 의사코드

*GetNearKnn*이며, 그림 3은 이에 대한 의사 코드이다. 이 함수는 가장 먼저 *Cubic*을 이용하여 *k*개 이상의 에이전트를 포함하는 서브큐빅을 찾는다. 이것은 단순히 현재의 에이전트인 *agent*가 속한 셀로부터 점차적으로 그 범위를 확대하면서 찾으면 된다. 다음 이 서브큐빅에 속한 에이전트 중 *agent*와 가장 먼 거리에 있는 에이전트 *p*를 찾고 *q*와 *p*사이의 거리 *r*을 계산한다. 마지막으로 *agent*를 중심으로 반경 *r*인 구내부에 존재하는 모든 에이전트를 찾아서 이들을 리턴한다. 이 알고리즘은 *agent*와 가장 가까운 *k*개의 에이전트가 포함되어 있음을 보장한다.

2.2 무리 짓기의 특성

*FlockingPS*는 무리 짓기의 어떠한 특성도 반영하지 않은 가장 기본적인 알고리즘이다. 본 절에서는 무리 짓기에서 흔히 나타날 수 있는 특징을 예측해 보고, 이를 실험적으로 분석해 보기로 한다. 무리 짓기의 가장 큰 예가 무리를 지어서 날아다니는 새떼들이다. 이들이 무리를 지어서 날아갈 때 일정한 형태를 형성하면서 무리를 지어서 날아간다. 이러한 무리속의 에이전트들은 주변의 에이전트들간 이웃 에이전트를 공유하는 특성을 갖는다[9]. 본 절에서는 무리 짓기에서 무리들의 이러한 특성을 실험적으로 분석하였다.

무리 짓기에서 임의의 순간 에이전트 *q*와 가장 가까운 에이전트를 *p*라고 하자. 표 1은 에이전트 *q*와 *p*가 서로 공유하는 이웃 에이전트의 비율을 실험적으로 분석한 것이다. 실험은 에이전트 수(*n*) 128, 256, 512, 1024에 대하여 이웃 에이전트의 수(*k*)를 31, 127로 변화시키면서 공유 이웃 에이전트 수를 계산한 것이다. 표 1에서 비율은 다음 식과 같이 계산되었다.

$$rate = \frac{\text{가장 인접한 이웃 에이전트와 공유하는 평균 이웃 에이전트의 수}}{\text{이웃 에이전트의 수}} \times 100(\%)$$

표 1. 공유 이웃 에이전트의 수 및 비율

n	*k*=31		*k*=127	
	s	*rate*(%)	*s*	*rate*(%)
128	26.2	84.4	124.8	98.3
256	26.0	83.9	120.4	94.8
512	25.6	82.5	117.3	92.4

표 1에서 첫 번째 행의 경우 128개의 에이전트가 있고, 조종힘을 계산하기 위하여 주변의 31개의 에이전트를 찾았다는 의미이다. 그런데 이러한 128개의 에이전트와 가장 가까운 이웃 에이전트와 임의의 순간 그들의 이웃 에이전트를 비교한 결과 평균 26.2개의 에이전트들은 두 이웃 에이전트에 공통으로 포함되어 있다는 것이다. 이것을 비율로 나타내면 약 84.4%이다. 즉 84.4%의 에이전트들을 서로 공유하고 있다는 것이다. 이 공유 비율은 이웃 에이전트의 수가 증가하면 더욱 많아지는 것을 표 1을 통하여 알 수 있다. 본 논문은 무리 짓기의 이러한 특성을 이용하여 하나의 이웃 에이전트를 탐색할 때 가능하다면 가장 가까운 에이전트의 이웃 에이전트도 동시에 탐색함으로써 탐색 비용을 줄일 수 있도록 하는 것이다.

3. 효율적인 이웃 탐색

3.1 기본 개념

앞 절에서 분석으로부터 많은 에이전트들은 임의의 순간에 이웃 에이전트를 서로 공유하고 있다는 것을 알 수 있다. 이러한 사실로부터 각각의 에이전트에 대하여 독립적으로 이웃 에이전트를 구하는 것이 아니라 많은 이웃 에이전트들을 공유하는 에이전트들에 대해서는 한 번에 이웃 에이전트를 구하는 경우 보다 효율적일 수 있다는 것을 알 수 있다. 이것이 본 논문의 기본 개념이다. 제안하는 알고리즘은 다음 정리 1과 2에 기반한다. 정리를 간단히 표시하기 위하여 다음과 같은 기호를 도입하기로 한다.

$||S||$: 에이전트 집합인 S에서의 에이전트 수.
$|p-q|$: 에이전트 p와 q 사이의 거리.
$|S-q|$: 에이전트 집합 S와 에이전트 q에 대하여, S에서 q와 거리가 가장 멀리 떨어진 에이전트와 q 사이의 거리, 즉 p가 q와 가장 멀리 떨어져 있는 S의 요소라고 하면 $|S-q|$는 $|p-q|$ 같음.

정리 1. S_Q와 S_P를 각각 중심이 에이전트 q와 p이고 임의의 반경을 가진 구라고 하고, Q와 P는 각각 S_Q와 S_P에 포함된 모든 에이전트들의 집합이라 하자. $|Q-q| \geq |p-q|+|P-p|$가 성립하면, S_P는 항상 S_Q내부에 존재한다.

증명 : $|Q-q| \geq |p-q|+|P-p|$를 만족하기 위해서는

최소한 $|Q-q| \geq |p-q|$가 만족하여야 한다. $|Q-q| \geq |p-q|$가 성립한다는 것은 p가 S_Q내에 존재한다는 것을 의미한다. 그림 4와 같이 q에서 시작하여 p를 통과하는 반직선 l_{qp}를 고려해 보자. q'와 p'를 각각 구 S_Q와 S_P와 반직선 l_{qp}와 만나는 점이라고 하자. $|Q-q| \geq |p-q|+|P-p|$를 $|q'-q| \geq |p-q|+|p'-p|$로 쓸 수 있다. 하나의 반직선상에 놓여 있는 q, p, q', p'가 $|q'-q| \geq |p-q|+|p'-p|$을 만족하기 위한 유일한 조건은 p'는 p와 q'사이에 있어야 한다. 따라서 p가 S_Q내에 있고, p'가 p와 q'사이에 있으므로 S_P는 S_Q내에 존재하여야 한다.

정리 1에서 Q와 P는 각각 S_Q와 S_P내의 존재하는 모든 에이전트들에 해당하므로, $|Q-q| \geq |p-q|+|P-p|$가 성립하면 P는 항상 Q의 부분집합이다.

정리 2. FlockingPS에서 임의의 에이전트 q에 대하여 GetNearKnn의 결과를 Q라 하자. GetNearKnn의 정의에 따라 $\|Q\| \geq k$이다. q가 아닌 $p \in Q$ 인 에이전트 p에 대하여, P를 Q에서 찾아진 p와 가장 가까운 k개의 에이전트들이라 하자. 이때 $|Q-q| \geq |p-q|+|P-p|$가 성립하면, P는 Q가 아닌 전체 에이전트들에 대한 p의 가장 가까운 k개의 이웃 에이전트들과 같다.

증명 : $P=\{p_1, p_2, ..., p_k\}$라 하자. 또한 S_Q와 S_P를 각각 중심이 q와 p이고 반경이 $|Q-q|$와 $|P-p|$인 구라고 하자. P가 전체 에이전트들에 대하여 p의 가장 가까운 k개의 이웃 에이전트들이 아니라는 것을 가정하기 위하여 $p_i \in P$인 p_i보다 p에 더 가까운 $p' \notin Q$인 p'가 존재한다고 가정하자. 따라서 $|p_i-p|>|p'-p|$여야 하고, 이것은 $|P-p| \geq |p_i-p|$이므로 $|P-p|>|p'-p|$여야 한다. $|P-p|>|p'-p|$는 p'가 S_P내에 있다는 것을 의미한다. 정리 1에 따라서 $|Q-q| \geq |p-q|+|P-p|$는 S_P가 S_Q내에 있다는 것을 의미하므로 p'는 S_Q내에 존재하여야 한다. 이것은 $p' \in Q$라고 가정하였으므로 모순이다. 그러므로 S_Q밖에 존재하는 어떠한 에이전트 p' $(p' \notin Q)$도 $|P-p| \geq |p'-p|$ 조건을 만족하지 못하기 때문에 P는 전체 에이전트들에 대한 p의 가장 가까운 k개의 이웃 에이전트들이 된다.

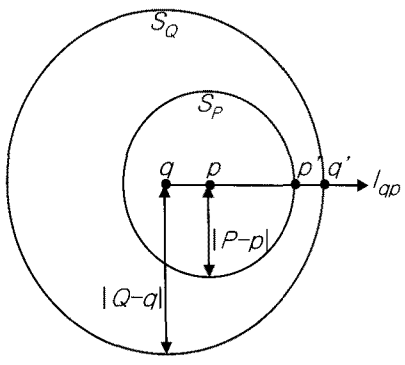


그림 4. 구 S_Q, S_P와 반직선 l_{qp}

3.2 제안하는 알고리즘

본 절에서는 앞 장에서 보인 무리 짓기에서 에이전트의 특성과 정리 2를 이용하여 개선된 공간분할 알고리즘을 제안한다. 제안하는 알고리즘은 단순하다. 에이전트 q와 p가 매우 인접하여 있다고 하자. 2장의 무리 짓기 특성에 의하며 이들의 이웃 에이전트들의 대부분은 q와 p에 공유될 것이다. 그럼에도 불구하고 알고리즘 *FlockingPS*는 q와 p에 대하여 *GetNearKnn*을 각각 호출하여 이웃 에이전트를 계산한다. 제안하는 알고리즘은 q에 대하여 *GetNearKnn*을 호출한 경우 그 리턴 값을 Q라고 할 때 Q에서 q의 이웃 에이전트를 찾고, 또한 Q를 이용하여 p의 이웃 에이전트를 찾는 것이다. 즉, Q내의 에이전트들에서 p와 가장 가까운 k개의 에이전트 집합 P를 찾고, 정리 2에 따라 $|Q-q| \geq |p-q|+|P-p|$를 체크하여 만족하는 경우 P를 p의 이웃 에이전트 집합으로 처리 하는 것이다. 이러한 경우 *GetNearKnn* 호출을 한번 줄일 수 있다. 반면 $|Q-q| \geq |p-q|+|P-p|$를 만족하지 못하는 경우 p에 대하여 *GetNearKnn*을 호출하여야 한다.

그림 5는 제안하는 알고리즘의 의사 코드이다. 알고리즘 *ImprovedPS*도 *FlockingPS*와 같이 3부분으로 나누어져 있다. 첫 번째 부분인 공간분할 업데이트와 마지막 부분인 에이전트의 새로운 위치 계산은 *FlockingPS*의 것들과 동일하다. 9번째 라인과 14번째 라인에서 *q.force*가 0이 아니라는 의미는 새로운 힘이 이미 계산되었다는 것을 의미한다. 라인 10~12은 에이전트 q에 대한 이웃 에이전트를 찾고 이를 이용하여 조종힘을 구하는 것으로 *FlockingPS*와 동일하다. 라인 13~19가 q와 가까운 에이전트 p에 대하여 이웃 에이전트를 찾고 조종힘을 계산하는 것이다. 라인 16은 정리 2에 따라 P가 이웃 에이전트를 포함하는 집합인지를 체크하는 것이다.

*ImprovedPS*은 라인 13-19에서 에이전트 p가 라

```
Algorithm ImprovedPS: Inputs: agents, k, t Outputs: None
01- // 공간분할 업데이트: FlockingPS와 동일
07: foreach q in agents{ ... }
08: // 이웃 에이전트 탐색 및 조종힘 계산
    foreach q in agents {
09:   if(q.force != 0) continue; //이미 힘이 계산되었다면
10:   Q= GetNearKnn(q, k, Cubic);
11:   kNN= GetKnn(q, k, Q);
12:   q.force= ComputeSteerForce(q, kNN)
13:   foreeach p in Q{
14:     if(p.force != 0) continue; //이미 힘이 계산되었다면
15:     P= GetKnn(p, k, Q);
16:     if(|Q-q| ≥ |p-q|+|P-p|){
17:       p.force= ComputeSteerForce(p, T);
18:     }
19:   }
20: }
21- // 에이전트의 새로운 위치 계산: FlockingPS와 동일
23: foreach q in agents { ... }
```

그림 5. 제안하는 알고리즘의 의사코드

인 16의 조건을 만족하지 못하는 경우 라인 15의 *GetKnn*의 실행은 불필요한 실행이므로 이것은 *ImprovedPS*의 단점이다. 이러한 문제는 *ImprovedPS*에서 Q의 모든 요소들에 대하여 라인 15의 *GetKnn*을 실행하기 때문이다. 그런데 Q의 에이전트들 중에는 라인 16의 조건을 통과할 가능성이 많은 에이전트와 가능성이 적은 에이전트가 있다. 예를 들어 p_i, $p_j \in Q$인 두 에이전트에 대하여 $|p_i-q| \approx 0$ 이고 $|p_j-q| \approx |Q-q|$라고 하자. 즉, p_i는 q에 매우 인접해 있고 p_j는 q에 매우 멀리 떨어져 있다는 것을 의미한다. P_{p_i}, P_{p_j}를 각각 p_i와 p_j에 대한 라인 15의 *GetKnn* 결과라 하면 이들이 라인 16을 통과하기 위해서는, $|P_{p_i}-p| \approx |Q-q|$가 되고 $|P_{p_j}-p| \approx 0$ 되어야 한다. 이것의 의미는 반경 $|Q-q|$인 구내에 k개의 에이전트가 존재할 가능성과 반경 거의 0에 가까운 작은 구내에 k개의 에이전트가 존재할 가능성은 큰 차이가 있다는 것이다. 따라서 $|p_j-q| \approx |Q-q|$인 에이전트를 사전에 제거함으로써 *GetKnn* 호출을 최소화할 수 있다. 이것은 라인 14와 라인 15사이에 다음과 같은 코드를 추가하면 된다: *if*($|p-q| > \alpha|Q-q|$) *continue*. 여기서 $0 \le \alpha \le 1$이다. α가 1이면 *FlockingPS*와 같고, α가 0이면 Q의 모든 에이전트에 대하여 *GetKnn* 호출을 호출하도록 하는 것과 같다. 실험 결과 대부분의 경우 α가 0.65인 경우 좋은 성능을 보였다.

4. 실험적 성능비교

기존의 공간분할 방법과 제안하는 방법의 성능을 비교하기 위하여 *FlockingPS*와 *ImprovedPS*를 구현하였다. 두 방법은 윈도우즈상에서 Visual Studio 2005를 사용하여 구현되었다. 사용된 컴퓨터 언어로는 C++와 STL(standard template library)을 사용하였으며 에이전트에 대한 렌더링을 위하여 OpenGL을 사용하였다. 구현 및 실험은 펜티움4 3GHZ CPU와 2GB 메모리를 가진 개인용 컴퓨터에서 실행하였다.

대부분의 컴퓨터 게임 프로그래밍과 같이 무리 짓기의 시뮬레이션도 (1) 사용자 입력처리, (2) 에이전트에 대한 새로운 정보를 계산, (3) 렌더링하는 부분으로 나누어 구성된다. 그림 6은 이에 대한 의사코드이다. 알고리즘의 정확한 성능 비교를 위해서 (1)과 (3)는 성능 측정에 포함하지 않았다. 무리 짓기와 같은 애니메이션에 대한 성능 측정은 한 번의 실행에 대한 성능보다는 동작 전체의 실행에 대한 성능 측정이 보다 중요하다. 따라서 본 논문에서는 각 알고리즘을 연속적으로 *1,000*번 실행하였을 때 소요된 시간을 측정하여 성능을 비교하였다.

성능비교에서 t_o, t_p는 각각 *FlockingPS*, *ImprovedPS*를 *1,000*번 실행하였을 때의 시간을 말하는 것으로 단위는 초이다. 또한 *FlockingPS*에 대한 *ImprovedPS*의 상대적 성능 개선 비율을 다음과 같이 계산한다: 성능개선 비율(ξ)=$\frac{t_o - t_p}{t_o} \times 100(\%)$. 표 2는 k와 n의 다양한 값에 대하여 제안하는 알고리즘이 기존의 공간 분할 알고리즘에 대한 성능 개선 비율을 보이고 있다. 여기서 k는 이웃 에이전트들의 수이며, n은 전체 에이전트의 수이다. 하나의 n에 대하여 k가 증가함에 따라 t_o, t_p가 증가하고 있음을 알 수 있다. 예를 들어 n=256에 대하여 k가 10에서 100으로 증가할 때 t_o는 2.7초에서 25.5초까지 증가하고 있음을 알

```
starttime= getCurrentTime();
for{count=0;count<1,000;count++){
  // (1) Process User Inputs
  (2) Update Agents // Call FlockingPS
        // ImprovedPS, ImprovedPS'
  // (3) Render Agents and Terrain
}
duration= getCurrentTime()-starttime;
```

그림 6. 성능측정 방법

표 2. k와 n의 변화에 따른 성능개선 비율

k	n=128		n=256		n=512		n=1024	
	(초)	ξ(%)	t_o(초)	ξ(%)	t_o(초)	ξ(%)	t_o(초)	ξ(%)
10	1.4	35	2.7	33	6.3	35	14.3	29
20	2.5	35	5.1	35	9.9	30	25.6	23
30	4.2	38	8.0	36	17.9	31	39.9	24
40	4.4	35	10.2	33	21.2	26	51.9	27
50	6.4	41	12.5	33	26.1	32	59.0	29
60	7.2	36	15.0	36	29.7	32	72.2	30
80	9.4	34	18.4	36	39.5	35	93.5	35
100	14.9	36	25.5	41	54.6	36	123.7	38

수 있다. 표 2를 통하여 t_p가 증가하고 있다는 것은 직접적으로 알 수 없으나, 성능개선 비율이 거의 일정함을 고려하면 역시 t_p도 증가하고 있음을 예측할 수 있다. 또한 하나의 k값에 대하여 n이 증가하는 경우도 t_o, t_p가 증가하고 있음을 알 수 있다. k=50에 대하여 n이 128에서 1024로 증가할 때 t_o는 6.4초에서 59초까지 증가한다. 이것은 두 알고리즘이 모두 O(kn)의 시간 복잡도를 갖기 때문이다. 이러한 시간 복잡도에도 불구하고 제안하는 알고리즘은 k와 n의 값에 따라 약간의 변화는 있으나 평균적으로 33%정도의 성능개선이 있음을 표 2를 통하여 알 수 있다. 즉 이것은 제안하는 알고리즘이 기존을 알고리즘에 비하여 평균 30%정도 빠르다는 것을 의미한다.

또한 실험의 전반에 걸쳐 두 알고리즘의 m번째 프레임에서 각 에이전트의 상태가 동일함을 확인함으로써 *ImprovedPS*가 올바르게 동작함을 알 수 있었다.

5. 결 론

본 논문은 최근 컴퓨터 그래픽기술의 응용으로 게임, 영화 등에서 많이 사용되고 있는 무리 짓기에 대한 개선된 알고리즘을 제안하였다. 개선된 알고리즘을 제안하기 위하여 본 논문은 무리 짓기에서 무리들의 특성을 분석하였다. 실험적 분석을 통하여 공간적으로 가까운 무리들은 80%이상 그들의 이웃 에이전트들을 공유한다는 사실을 알았고, 제안하는 알고리즘은 이 특성을 적용하여 기존의 방법의 성능을 개선하였다. 기존의 무리 짓기 방법인 공간분할 방법을 소개하였고, 이 방법에 무리 짓기의 특성을 반영하여 새로운 알고리즘을 제안하였다.

제안된 방법의 성능은 실험적으로 기존의 방법과 비교되었다. 다양한 에이전트의 수와 이웃 에이전트의 수에 대하여 실험을 시행하였으며, 그 결과 제안하는 알고리즘이 기존의 공간분할 알고리즘에 비하여 상대적으로 약 33%정도 성능을 개선한다는 사실을 알 수 있었다.

참 고 문 헌

[1] Reynolds, C. W., "Interaction with Groups of Autonomous Characters," In Proceedings of the Game Developers Conference 2000, San Francisco, California, pp. 449-460, 2000.

[2] Iain D. Couzin, Jens Krause, Richard James, Graeme D. Ruxton and Nigel R. Franks, "Collective Memory and Spatial Sorting in Animal Groups," *J. theory Biol.*, pp. 1-11, 2002.

[3] Mat Buckland, *Programming Game AI by Example,* ISBN 1556220782, Wordware Publications, 2005.

[4] Reynolds, C. W., "Big Fast Crowds on PS3," In Proceedings of Sandbox (an ACM Video Games Symposium), Boston, Massachusetts, pp. 113-121, 2006.

[5] Julien Pettre, Pablo de Heras Ciechomski, Jonathan Maim, Barbara Yersin, Jean-Paul Laumond and Daniel Thalmann, "Real-time navigating crowds: scalable simulation and rendering," *Comp. Anim. Virtual Worlds*, pp. 445-455, 2006.

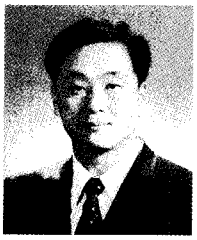
[6] Jagan Sankaranarayanan, Hanan Samet, Amitabh Varshney, "A fast all nearest neighbor algorithm for applications involving large point-clouds," *Computers & Graphics* 31, pp. 157-174, 2007.

[7] M. Ballerini, N. Cabibbo, R. Candelier, A. Cavagna, E. Cisbani, I. Giardina, V. Lecomte, A. Orlandi, G. Parisi, A. Procaccini, M. Viale, and V. Zdravkovic, "Interaction ruling animal collective behavior depends on topological rather than metric distance: Evidence from a

field study," *PNAS* 105, pp. 1232-1237, 2008.
[8] 이재문, "대규모 보이드를 이용한 대규모 무리의 효율적인 무리 짓기," 한국게임학회 논문지, 제8권 3호, pp. 87-96, 2008.
[9] 이재문, 권대중, 엄종석, 정인환, "무리 짓기에서 효율적인 이웃 에이전트 탐색," 한국멀티미디어학회 추계학술발표대회 논문집, 제12권2호, pp. 77-80, 2009.
[10] Wikipedia, "Algorithmic complexity of Flocking Behavior," *http://en.wikipedia.org/wiki/Flocking_(behavior)Algorithmic_complexity*.

이 재 문

1986년 한양대학교 전자공학과 졸업
1988년 한국과학기술원 전기 및 전자공학과 석사
1992년 한국과학기술원 전기 및 전자공학과 박사
1992년~1994년 한국전기통신공사 선임연구원
1994년 3월~현재 한성대학교 멀티미디어공학과 교수
관심분야: 게임프로그래밍, 게임인공지능, 기계학습, 데이터베이스



정 인 환

1984년 한양대학교 원자력공학과 학사
2000년 한국과학기술원 정보 및 통신공학과 박사
1985년~1998년 삼성전자(주) 시스템사업부 부장
1999년~2000년 두루넷 팀장
2001년 3월~현재 한성대학교 컴퓨터공학과 부교수
관심분야: 분산시스템, 멀티미디어통신 및 응용