이전 k 개의 가장 가까운 이웃을 이용한 무리 짓기에 대한 공간분할 방법의 개선※

이재문*
한성대학교 멀티미디어공학과*
jmlee@hansung.ac.kr

An Improvement Of Spatial Partitioning Method For Flocking Behaviors By Using Previous k-Nearest Neighbors

Lee, Jae Moon*
Dept. of Multimedia Engineering, Hansung University*

요 약

본 논문에서는 무리 짓기에 대한 공간분할 방법의 성능을 개선하는 알고리즘을 제안한다. 핵심 개념은 무리 속에서 움직이는 개체인 보이드가 지속적으로 자신의 방향과 위치를 변경시키나 자신의 다음 방향의 결정에 영향을 주는 k개의 가장 가까운 이웃인 kNN은 자주 바뀌지 않는다는 사실을 이용하여 성능을 개선하는 것이다. 본 논문에서 이전의 kNN을 이용하여 새로운 kNN이 변경되었는지를 판별하는 방법이 제안되었고, 제안된 방법의 정당성은 정리를 통하여 증명되었다. 제안된 방법은 구현되었으며, 기존의 공간분할 방법과 성능이 비교되었다. 비교 결과로부터 제안된 알고리즘이 초당 프레임 수 관점에서 기존의 알고리즘보다 약 30% 개선 효과를 주는 것을 알 수 있었다.

ABSTRACT

This paper proposes an algorithm to improve the performance of the spatial partitioning method for flocking behaviors. The core concept is to improve the performance by using the fact that even if a moving entity, boid in flock continuously changes its direction and position, its k-nearest neighbors, kNN to effect on decision of the next direction is not changed frequently. From the previous kNN, the method to check whether new kNN is changed or not is proposed in this paper and then the correctness of the proposed method is proved by two theorems. The proposed algorithm was implemented and its performance was compared with the conventional spatial partitioning method. The results of the comparison show that the proposed algorithm outperforms the conventional one by about 30% with respect to the number of frames per a second.

Keyward : flocking behaviors, boid, k-nearest neighbors, spatial partitioning

접수일자 : 2009년 01월 29일
심사완료 : 2009년 03월 02일
※ 본 연구는 2008년 한성대학교 교내연구비 지원과제임.

1. 서 론

무리 짓기란 많은 자연에서 관찰되는 현상으로 떼를 지어 다니는 새들, 어군속의 물고기들, 목장에서 양들, 음식물을 찾아다니는 개미들 등의 움직임이 모두 무리 짓기에 속한다. 이러한 동물들의 그룹은 무리속의 각각이 지속적으로 그들의 방향과 형태를 바꿀지라도 전체적으로 하나의 응집된 형태를 가지는 특성을 보인다. 무리속의 하나의 움직이는 개체를 보이드라고 한다[1].

무리 짓기 알고리즘은 컴퓨터 애니메이션에서 새들의 무리를 모방하는 방법으로 [1]에 의하여 처음으로 제안되었다. 무리 짓기는 무리를 이끄는 대표도 없을 뿐만 아니라 전체를 제어하는 그 어떤 힘도 없이 자발적으로 나타나는 집단 행동의 대표적인 예이다. 각 보이드는 전체 환경의 지리적 특성을 감지하고, 주변의 몇몇 동료들과 상호 작용을 하면서 무리를 형성한다. 무리 짓기의 가장 기본적인 모델은 세 가지 단순한 조종행동(Steering Behavior)인 분리, 응집, 정렬로 구성된다. 분리는 다른 동료와 일정 거리를 유지하는 능력을 말하는데 이것은 다른 동료와 너무 가까이 가지 않음으로써 충돌을 피할 수 있도록 하는 행동이다. 응집은 주위 동료들과 그룹을 형성하고자 하는 행동이다. 응집은 주위의 동료들에 대한 위치 평균을 구함으로써 계산한다. 정렬은 주위의 동료들과 같은 방향을 유지하려는 행동이다. 이것은 주위 동료들의 평균 방향을 구함으로써 계산한다. 무리 짓기의 핵심은 이러한 조종행동의 계산에서 영향을 주는 동료 보이드들을 계산하는 것이다.

무리 짓기의 가장 쉬운 방법은 각 보이드에 대하여 나머지 모든 보이드들을 조종행동에 영향을 끼치는 동료로 고려하여 조종행동을 계산하는 것이다[1, 2, 3, 5, 8, 9]. 이것은 보이드의 수가 n일 때 $O(n^2)$의 시간 복잡도를 가진다. 이 시간 복잡도는 n의 크기에 따라 비용이 크게 늘어난다. 따라서 보이드 수가 작은 응용 프로그램에서는 큰 문제가 되지 않으나, 보이드 수가 커짐에 따라 성능은 급격히 저하된다. 이 문제를 해결하는 하나의 방법은 실제 상황에서처럼 하나의 보이드에 대하여 조종행동을 계산할 때 다른 모든 보이드들을 동료로 고려하는 대신에 가까이 있어 영향을 많이 주는 동료들만 고려하는 것이다. 이 경우 이 문제는 임의의 포인트에서 가장 가까운 k개의 이웃 포인트들을 찾는 k-*nearest neighbors*(kNN)[4, 6, 7] 문제와 같아진다. 무리 짓기에서 kNN을 찾는 것은 데이터의 모든 포인트가 보이드의 위치에 해당하고, 보이드는 지속적으로 움직이므로 항상 변하는 동적 데이터에 대하여 kNN을 찾아야 하는 문제이다. 이 문제 역시 인덱스와 같은 특별한 전처리가 없는 한, 하나의 보이드에 대한 kNN을 찾는 것이 $O(n)$이기 때문에 전체적으로는 여전히 $O(n^2)$의 시간 복잡도를 갖는다. [2, 3]에서는 이를 해결하기 위하여 공간분할 방법을 사용하였다. 이것은 무리에 속한 모든 보이드들을 그들의 위치에 따라 3차원 배열에 군집화(해싱)하여 저장하는 것이다. 임의의 보이드에 대하여 kNN을 찾고자하는 경우 모든 보이드를 대상으로 하여 kNN을 찾는 것이 아니라 그 보이드가 속한 셀에서부터 가장 가까운 셀부터 순차적으로 액세스하여 셀에 저장된 보이드를 찾음으로써 kNN을 찾는 비용을 줄이는 것이다. 이것은 시간 복잡도를 평균적으로 $O(kn)$으로 줄여준다[2, 3, 9, 10]. 여기서 k는 가장 가까운 이웃 보이드들의 수이다.

본 논문은 무리 짓기에서 공간분할 방법의 성능을 개선하는 것이다. 무리 짓기에서 보이드는 지속적으로 움직이므로 방향과 위치가 달라져 매 순간마다 kNN을 다시 계산하여야 한다. 그러나 조종행동의 특징 관찰하면 실제로는 kNN이 많이 변경되지 않는 것을 알 수 있다. 보이드 수 512에 대하여 kNN의 수를 16~128까지 변화시키면서 실험결과 82%~92%의 보이드가 시간 t에서의 그들의 kNN과 시간 $t+1$에서의 kNN이 동일함을 관찰할 수 있었다. 본 논문에서는 이러한 사실에 기반하여 임의의 보이드에 대하여 kNN의 변경여부를 탐지하여 변경되지 않은 경우 kNN을 다시 찾지

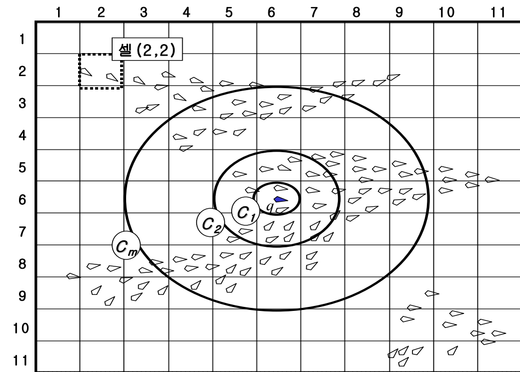
않음으로써 성능을 개선한다.

2장은 기존의 무리 짓기의 공간분할 방법과 무리 짓기의 특성을 소개하며, 3장에서는 임의의 보이드에 대하여 kNN 변경 여부를 탐지하는 방법을 소개하여 기존의 공간분할 알고리즘을 개선한다. 4장에서는 실험에 의한 성능을 비교하며 5장에서 결론을 논한다.

2. 무리 짓기의 공간분할 방법

2.1 알고리즘

공간분할 알고리즘은 조종행동의 계산에 필요한 k개의 가장 가까운 이웃, kNN을 효율적으로 찾기 위하여 [2]에서 제안되었다. 별도의 보조 데이터가 없는 상태에서 n개의 데이터에 대하여 모든 kNN을 찾는 것은 $O(n^2)$의 비용이 필요하다. 공간분할 방법은 매 순간 하나의 보이드에 대해서만 kNN을 찾는 것이 아니라 모든 보이드에 대하여 kNN을 찾아야 한다는 점에 착안하여 kNN을 찾기 전에 모든 보이드들에 대하여 보이드들의 위치를 근사적으로 군집화 시키는 것이다. 예를 들어 [그림 1]과 같이 2D에서 게임의 세계를 그리드로 나눈 후, 전처리 작업으로 각 보이드를 보이드의 위치를 포함하는 셀에 저장한다. [그림 1]에서 셀 (2,2)에는 두 개의 보이드가 저장되어 있다. 그런 후 임의의 지역에 존재하는 모든 보이드를 찾는 것은 단순히 그 지역에 매핑되는 그리드의 셀만 조사하면 된다. 이러한 공간분할된 그리드를 이용하여 보이드 q에 대한 kNN을 찾는 것은 가장 먼저 q의 위치에 매핑되는 그리드의 셀, 즉 [그림 1]에서 C_1에 포함되는 셀에 저장된 보이드를 찾고 순차적으로 C_2, C_3, ..., C_m에 포함된 셀들에 저장된 보이드를 찾으면서 처음으로 k개를 만족하는 C_m에서 멈춘다. 이렇게 하면 모든 보이드를 탐색하는 대신에 $k'(\geq k)$개의 보이드만 탐색하면 된다. 이 방법은 평균적으로 $O(kn)$의 시간 복잡도를 갖는다. 3D에서는 그리드 대신 3차원 배열을 사용한다.



[그림 1] 공간분할의 예

Algorithm FlockingPS: Inputs: Flock, k, gsize Outputs: None
01: Grid(gsize, gsize, gsize)= {ϕ}
02: foreach q in Flock {
03: (x, y, z)= MappingGrid(q.pos)
04: Grid(x, y, z).AddBoid(q)
05: }
06: foreach q in Flock {
07: S= getNearKnn(q, k, Grid)
08: kNN= getKnn(q, k, S)
09: q.newdir= ComputeSteer(q, kNN)
10: }
11: // 이동행동: 각 보이드에 대한 새로운 위치 계산

[그림 2] 무리 짓기의 공간분할 알고리즘

[그림 2]는 다음은 공간분할 무리 짓기에 대한 의사 코드이다. 알고리즘의 입력은 보이드의 집합인 Flock, kNN의 수인 k, 그리드의 크기를 나타내는 $gsize$이다. 라인 1~5까지가 모든 보이드를 그리드에 저장하는 전처리 작업이다. 여기서 $q.pos$는 보이드 q의 위치벡터이며, 함수 MappingGrid는 입력된 위치벡터에 대응하는 그리드의 셀 좌표를 리턴하는 함수이다. 라인 6~10 사이가 보이드 q에

대하여 조종행동을 계산하여 새로운 방향을 계산하는 과정이다. 라인 7에서 함수 getNearKnn은 [그림 1]에서 보이듯이 그리드를 이용하여 q와 가까운 $k'(\geq k)$개의 이웃 보이드를 찾는 것이고, 라인 8에서 getKnn은 S로부터 정확한 kNN을 찾는 함수이다. getKnn은 단순히 kNN을 q와의 거리를 기준으로 S의 요소들을 정렬하여 얻을 수도 있고, 우선순위 큐를 이용하여 얻을 수도 있다. 실험 결과 대부분 k가 작아 어떤 방법을 사용하여도 실행시간에 큰 영향을 주지 않았다. 또한 프로그램의 코딩에 따라서 getNearKnn과 getKnn은 하나의 함수로 만들 수도 있다. 라인 9에서 함수 ComputeSteer은 q에 대한 조종행동을 계산한 후 최종적으로 새로운 방향을 출력하는 것이다. 이에 대한 자세한 구현은 [3]에서 참고하였다. 라인 11은 [1]에서 제시한 이동행동을 계산하는 것이다.

2.2 무리 짓기의 특성 분석

무리 짓기는 모든 보이드들에 대하여 매 순간마다 조종행동의 계산으로 새로운 방향을 결정하고, 이를 이용하여 이동행동에서 새로운 위치를 계산한다. 이동행동에서 새로운 위치를 계산하는 것은 매우 단순하다. 각 보이드들에 대하여 새로운 방향($\overrightarrow{ndir}$), 보이드의 속도(v), 소요된 시간(dt) 및 현재의 위치($\overrightarrow{pos}$)를 이용하여 새로운 위치 $\overrightarrow{npos} = \overrightarrow{pos} + \overrightarrow{ndir} \times v \times dt$로 계산한다. 이것은 모든 보이드들은 매 순간마다 그들의 위치를 지속적으로 변화시키고 따라서 그들의 kNN도 매 순간마다 변경될 가능성이 있다는 것이다. 이러한 변경 가능성 때문에 *FlockingPS*는 매 순간마다 모든 보이드에 대하여 kNN을 다시 찾아야 한다.

무리 짓기에서 보이드들은 지속적으로 움직이나 그들의 조종행동에 영향을 주는 이웃 보이드들, kNN은 크게 변하지 않는다는 것을 직관적으로 알 수 있다. 이 절에서는 앞에서 제시한 *FlockingPS*를 이용하여 각 보이드들에 대하여 시간 t에서 구해진 kNN과 시간 $t+1$에서 구해진 kNN을 비교하여 얼마나 많은 보이드들에 대하여 그들의 kNN이 변경되었는가를 분석한다. [표 1]은 보이드 q가 직전의 kNN과 현재의 kNN이 동일할 비율을 실험을 통하여 분석한 내용이다. 실험은 kNN에서 k의 값을 16, 32, 64, 128로 변화시키면서 보이드 수(n) 256, 512, 1024에 대하여 시뮬레이션하여 아래 식을 계산하여 [표 1]의 결과를 얻었다.

$$\text{비율} = \frac{kNN(t)\text{와}\,kNN(t+1)\text{이 동일한 보이드 수}}{kNN(t+1)\text{을 구한 수(전체 보이드 수)}} \times 100(\%)$$

[표 1] 동일 kNN에 대한 비율

k	n		
	256	512	1024
16	94.3%	92.3%	87.8%
32	91.6%	89.9%	83.1%
64	88.4%	88.5%	83.1%
128	89.0%	82.4%	76.9%

상기 식에서 $kNN(t)$는 임의의 보이드에 대한 시간 t에서 kNN을 의미한다. $kNN(t+1)$을 구한 수는 알고리즘 FlockingPS의 라인 8을 실행한 수와 같다. $kNN(t)$와 $kNN(t+1)$이 동일한 보이드 수는 라인 8과 라인 9 사이에서 q의 이전 kNN과 라인 8에서 구해진 kNN을 비교하여 두 kNN이 동일한 보이드 수이다. [표 1]로부터 n이 256일때에는 89% 이상의 보이드가 동일한 kNN을 가지며, n이 1024인경우에는 76%이상의 보이드가 동일한 kNN을 가지는 것을 알 수 있다. 이러한 결과는 대부분 보이드들에 대한 kNN이 자주 변경되지 않음을 보인다. 이것은 무리 짓기의 운행 현상을 관찰할 때 느끼는 직관과 일치한다.

3. 개선된 공간분할 알고리즘

3.1 기본 개념

앞장에서 분석된 [표 1]은 공간분할 알고리즘의 개선에 대한 아이디어를 제공하고 있다. 즉, 만

약 시간 $t+1$ 시점에서 시간 t에서 구해진 kNN을 알고 있고, 그리고 간단한 부과적인 처리를 통하여 kNN이 변경되지 않았다는 사실을 알 수 있다면 시간 $t+1$에서 구태여 kNN을 구하지 않고 시간 t에서 구해진 kNN을 그대로 사용함으로써 *FlockingPS*의 성능을 개선할 수 있을 것이다. 이 경우 앞의 실험 결과에 따르면 약 76%이상 kNN을 구하는 비용을 줄일 수 있다. 이것이 본 논문의 기본 개념이다. 다음 정리 1과 2는 이러한 기본 개념을 이용할 수 있는 근거를 제시한다.

정의1. *$kNN_q(t)$={q_1, q_2, ..., q_k}를 시간 t 에서 보이드 q의 kNN이라 정의한다.*

정의2. *$|q'-q|_t$는 시간 t에서 보이드 q'와 q사이의 거리로 정의한다.*

정의3. *임의의 보이드 집합 S={s_1, s_2, ..., s_n}에 대하여 $|S-q|_t$는 시간 t에서 q의 위치를 중심으로 S의 모든 요소를 포함하는 최소의 반경으로 정의한다.*

정의4. *보이드 집합 S={s_1, s_2, ..., s_n}에 대하여 $\|S\|$는 집합 S의 요소수로 정의한다.*

정의 2, 3에 따라 $|S-q|_t$는 max($|s_1-q|_t$, $|s_2-q|_t$, ..., $|s_n-q|_t$)이다. 여기서 max는 입력 중 가장 큰 값을 리턴하는 함수이다. 상기 정의를 바탕으로 다음과 같은 정리를 할 수 있다.

정리1. *$S_q(t_2)$는 시간 t_2에서 q를 중심으로 반경 $|kNN_q(t_1)-q|_{t_2}$의 구내에 존재하는 보이드의 집합이라고 하면, $kNN_q(t_2) \subseteq S_q(t_2)$는 항상 성립한다. 여기서 $t_1 < t_2$이다.(증명: 부록 참조)*

FlockingPS에서 시간 t_2에서 $kNN_q(t_1)$을 알고 있다면 정리 1을 이용하여 함수 getNearKnn에서 구해진 S에 대하여 $|kNN_q(t_1)-q|_{t_2}$를 구하여 이보다 멀리 떨어져 있는 보이드는 $kNN_q(t_2)$을 구하는 것에서 제외 할 수 있음으로 S의 크기를 줄일 수 있다.

정리2. *$S_q(t_2)$는 시간 t_2에서 q를 중심으로 반경 $|kNN_q(t_1)-q|_{t_2}$의 구내에 존재하는 보이드의 집합이고 $\|S_q(t_2)\|=k$이면 $kNN_q(t_2) \equiv kNN_q(t_1)$이다.(증명: 부록 참조)*

정리 2는 q에 대하여 t_1, t_2에서 kNN이 동일할 조건을 제시하는 것이다. 따라서 정리 2의 결과를 이용하면 FlockingPS의 라인 8에서 getKnn의 실행을 줄일 수 있다.

3.2 개선된 알고리즘

본 절에서는 정리 1, 2를 사용하여 개선된 *FlockingPS*를 제안한다. 앞에서도 언급하였듯이 정리 1, 2는 기본적으로 모든 보이드들은 이전의 kNN을 가지고 있어야 한다. 본 논문에서는 각 보이드마다 k개의 포인터를 저장하는 배열을 두어 이를 관리하는데 사용한다. 이를 *prevKnn*으로 표시한다. [그림 3]은 개선된 알고리즘 *EnhancedPS*의 의사 코드이다.

Algorithm EnhancedPS: Inputs: Flock, k, gsize Outputs: None

```
01: Grid(gsize, gsize, gsize)= {φ}
02: foreach q in Flock {
03:     (x, y, z)= MappingGrid(q.pos)
04:     Grid(x, y, z).AddBoid(q)
06: }
07: foreach q in Flock {
08:     S= getNearKNN(q, k, Grid)
11:     r= |q.prevKnn - q|      // |kNN_q(t_1)-q|_{t_2}
13:     S′= {φ}
14:     foreach s in S{
15:         if(|q - s| <= r) S′= S′ U {s}
```

Algorithm EnhancedPS: Inputs: Flock, k, gsize Outputs: None	
16:	}
17:	if(‖S′‖ > k){
18:	kNN= getKnn(q, k, S′)
19:	q.prevKnn= kNN
20:	}else{ // $kNN_q(t_2) \equiv kNN_q(t_1)$
21:	kNN= q.prevKnn
22:	}
23:	q.newdir= ComputeSteer(q, kNN)
25:	}
26:	// 이동행동: 각 보이드에 대한 새로운 위치 계산

[그림 3] 개선된 공간분할 알고리즘

상기 알고리즘의 입력과 라인 1~6은 FlockingPS와 정확히 동일하다. 라인 11~16사이의 내용이 정리 1을 적용하는 코드이다. 현재의 시점이 t_2라고 하면 라인 11에서 r은 정리 1에서 $|kNN_q(t_1)-q|_{t_2}$를 의미한다. 라인 13~16은 정리 1의 성질을 이용하여 S의 크기를 줄이는 과정이다. 라인 17~22 사이의 코드가 정리 2를 적용하는 것이다. $S' \subseteq S$이지만 정리 1에 따라 $\| S' \| \geq k$는 항상 만족된다. 따라서 $\| S' \| > k$인 라인 17~20의 경우에는 S'내에 $kNN_q(t_2)$에 속하지 않는 보이드가 있다는 것이므로 함수 getKnn을 이용하여 $kNN_q(t_2)$를 구하여야 하며 또한 시간 $t_3(>t_2)$에서 EnhancedPS의 실행을 위하여 prevKnn에 저장하여야 한다. $\| S' \| = k$인 라인 21의 경우에는 정리 2의 성질에 따라 $kNN_q(t_2)$를 구할 필요 없이 prevKnn을 kNN으로 사용한다. 이 경우가 대부분 76%이상 될 것이다.

4. 실험에 의한 성능 비교

성능 비교를 하기 위하여 기존의 *FlockingPS*와 제안하는 *EnhancedPS*를 구현하였다. 두 알고리즘은 윈도우즈상에서 Visual Studio 2005를 사용하여 구현하였다. 구현 언어로는 C++와 STL (standard template library)을 사용하였으며 보이드에 대한 렌더링을 위하여 OpenGL을 사용하였다. 실험은 펜티움4 2.5GHZ CPU와 2GB 메모리를 가진 개인용 컴퓨터에서 실행하였다.

```
s= getCurrentTime();
for{count=0;count<1,000;count++){
    // (1) Process input
    (2) Update Boids // Call FlockingPS
                     // or EnhancedPS
    // (3) Render Boids and Terrain
}
duration= getCurrentTime()-s;
frames= count/duration;
```

[그림 4] 성능측정 방법

전체적으로 프로그램의 실행은 [그림 4]와 같은 형식으로 구성하였다. [그림 4]에서 알 수 있듯이 성능은 초당 몇 번 *FlockingPS*(또는 *EnhancedPS*)를 실행할 수 있는가를 측정하는 것이다. 모든 실험은 1,000번씩 알고리즘을 실행하여 소요된 시간으로 나누어서 초당 실행 수를 측정하였다. [그림 4]에서 “(1) Process Inputs"는 일반적으로 사용자가 가하는 입력에 대한 처리이다. 실험동안 사용자 입력이 있는 경우 성능 측정에 영향을 줄 수 있다. 또한 “(3) Render Boids and Terrain" 부분은 보이드의 형태(물고기, 새)나 게임 공간의 복잡도에 따라 실행 시간이 달라진다. 따라서 정확한 성능 비교를 하기 위해서 [그림 4]에서 (1)과 (3)을 실제 성능 측정에서 제외하였다.

성능 측정의 변수로 보이드 수 n과 kNN의 k를 사용하였다. 이것은 보이드 수는 게임 내용이나 게임 규모에 따라 정해지며, kNN의 k는 보이드의 행동을 제어하는데 사용될수 있기 때문이다. f_{FPS}는 *FlockingPS*의 초당 실행 프레임수를 표시하며, f_{EPS}는 *EnhancedPS*의 초당 실행 프레임수를 표시한다. 또한 *FlockingPS*에 대한 *EnhancedPS*의 성능 개선 비율을 다음과 같이 계산한다.

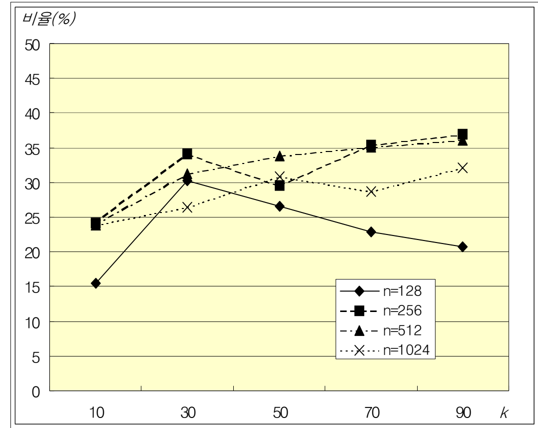
$$\text{성능개선 비율}(\xi)=\frac{f_{EPS}-f_{FPS}}{f_{EPS}}\times 100\ (\%)$$

[표 2] k=50에 대한 성능 개선 비율

n	f_{FPS}	f_{EPS}	$\xi(\%)$
128	117.9	160.4	26.5
256	38.9	55.2	29.6
512	14.8	22.4	33.8
1024	7.1	10.2	30.8

[표 2]는 k를 50으로 고정한 후 다양한 n에 대하여 두 알고리즘의 초당 실행 프레임 수 및 성능 개선 비율을 보이고 있다. 두 알고리즘 모두 n이 증가함에 따라 초당 실행 프레임 수가 줄어드는 것을 볼 수 있다. 이것은 두 알고리즘 모두 $O(kn)$의 시간 복잡도를 갖기 때문이다. 제안된 알고리즘이 약 30%정도의 성능 개선이 있다는 것을 [표 2]로부터 알수 있다. [표 1]에 의하면 약 76%의 성능 개선이 기대되나 약 30%밖에 안되는 것은 *EnhancedPS*의 라인 8에서 *getNearKnn*의 비용이 적지 않기 때문이다.

[그림 5]은 k의 변화에 따른 다양한 성능 개선 비율을 보이고 있다. 그래프에서 가로축은 k의 값을 표시하며, 세로축은 성능 개선 비율(ξ)을 표시한다. n이 128이고 k가 10인 경우 성능 개선 비율이 약 15%로 가장 낮게 나타나는데 이 경우에는 k와 n값이 작아서 두 알고리즘의 실행시간이 모두 작아 성능이 거의 비슷하게 나오기 때문이다. n이 256이고 k가 90인 경우에는 약 37%이 성능 개선 효과를 보이고 있다. n이 128인 경우 성능 개선 효과가 15%~30%로 비교적 변화가 큰 편이나, 그 외의 경우 대부분 약 25%~35%사이의 성능 개선 효과를 보이고 있다. 이것은 제안된 *EnhancedPS*가 기존의 *FlockingPS*보다 약 30% 정도의 성능 개선을 준다고 할 수 있다.



[그림 5] k의 변화에 따른 성능 개선 비율

5. 결 론

본 논문에서는 무리 짓기에서 가장 잘 알려진 공간분할 방법에 대한 개선된 알고리즘을 제안하였다. 제안한 알고리즘의 핵심 개념은 일반적으로 무리 짓기에서 무리 속의 개체인 보이드들이 지속적으로 움직여 방향과 위치가 달라지나 그들의 다음 방향을 결정하는 k개의 가장 가까운 이웃인 *kNN*은 자주 바뀌지 않는다는 사실을 이용하여 성능을 개선하였다. 이를 위하여 모든 보이드들은 이전에 구했던 *kNN*을 다음 *kNN*을 구할 때까지 저장하도록 하였으며, 이것을 이용하여 새로운 *kNN*을 구하는 비용을 줄였다. 제안된 방법의 정당성은 정리를 통하여 증명하였다. 실험을 통한 성능 비교 결과 제안된 방법이 기존의 방법보다 약 30%의 성능 개선 효과를 주는 것을 알 수 있었다.

참고문헌

[1] Reynolds, C. W. “Flocks, Herds, and Schools: A Distributed Behavioral Model”, SIGGRAPH, 21(4), pp. 25-34, 1987.

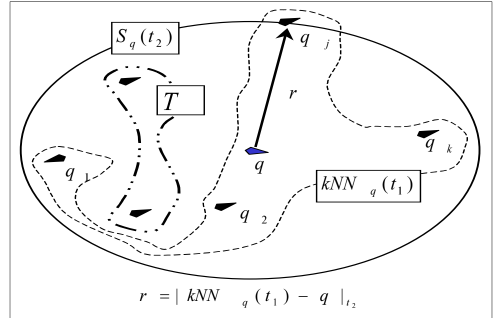
[2] C.W. Reynolds, “Interaction with Groups of Autonomous Characters”, In Proc. of Game Developers Conference, pp. 449-460, 2001.

[3] Mat Buckland, “Programming Game AI by

Example", ISBN 1556220782, Wordware Publications, 2005.
[4] N. Bell1, Y. Yu1 and P. J. Mucha2, "Particle-Based Simulation of Granular Materials", Eurographics/ACM SIGGRAPH Symposium on Computer Animation, 2005
[5] Reynolds, C., "Big Fast Crowds on PS3", In Proceedings of Sandbox (an ACM Video Games Symposium), Boston, Massachusetts, July 2006.
[6] Jagan Sankaranarayanan, Hanan Samet, Amitabh Varshney, "A fast all nearest neighbor algorithm for applications involving large point-clouds", Computers & Graphics pp 157-174, 2007.
[7] Nicolas Brodu, "Query Sphere Indexing for Neighborhood Requests", http://nicolas.brodu.free.fr/common/recherche/publications/QuerySphereIndexing.pdf, 2007.
[8] 이재문, "대규모 보이드를 이용한 대규모 무리의 효율적인 무리짓기", 한국게임학회 논문지, 제8권 제3호, 87-96, 2008
[9] 박현진, 이재문, "대규모 보이드들의 무리짓기에서 공간분할 방법에 대한 성능분석", 한국게임학회 추계 학술발표대회 발표지, 2008

부 록

정리1의 증명: $kNN_q(t_1)$={q_1, q_2, ..., q_k}이라 하면 $|kNN_q(t_1)-q|_{t_2}$= $\max(|q_1-q|_{t_2}$, $|q_2-q|_{t_2}$, ..., $|q_k-q|_{t_2})$이다. 임의의 q_j에 대하여 $|q_j-q|_{t_2}$= $|kNN_q(t_1)-q|_{t_2}$라 하자. 이것의 의미는 시간 t_2에서 q_1, q_2, ..., q_k중에서 q_j가 q와 가장 멀리 떨어져 있다는 의미이므로 q를 중심으로 반경 $|q_j-q|_{t_2}$의 구내에 q_1, q_2, ..., q_k가 존재한다는 것을 의미한다. 즉, $S_q(t_2)$=$kNN_q(t_1)\cup T$라고 할 수 있다(그림 6 참조). 여기서 T는 $kNN_q(t_1)\cap T$={ϕ}인 임의의 보이드 집합이다. 따라서 $\| S_q(t_2) \|$ = $\| kNN_q(t_1) \|$ + $\| T \|$ = k + $\| T \| \geq k$이다. 이것은 시간 t_2에서 q를 중심으로 반경 $|kNN_q(t_1)-q|_{t_2}$내에는 k개 이상의 보이드가 존재한다는 것을 의미한다. 따라서 $kNN_q(t_2)$의 정의에 따라 이에 속하는 k개의 보이드도 이 반경내에 속하게 된다. 그러므로 $kNN_q(t_2)\subseteq S_q(t_2)$ 성립한다.



[그림 6] 시간 t_2에서 $kNN_q(t_1)$과 T의 예

정리2의 증명: [그림 3]과 같이 $S_q(t_2)$= $kNN_q(t_1)\cup T$라고 할 수 있다. 여기서 T는 $kNN_q(t_1)\cap T$={ϕ}인 임의의 보이드 집합이다. 따라서 $\| S_q(t_2) \|$ = $\| kNN_q(t_1) \|$ + $\| T \|$ =k + $\| T \| \geq k$가 성립한다. 여기서 $\| S_q(t_2) \|$ = k라고 하였으므로 $\| T \|$ =0이 되고 T= {ϕ}가 된다. 따라서 $S_q(t_2)$=$kNN_q(t_1)\cup T$= $kNN_q(t_1)$되므로 $\| S_q(t_2) \|$ = $\| kNN_q(t_1) \|$ = k 이다. 또한 정리 1에 따라 $kNN_q(t_2)\subseteq S_q(t_2)$가 성립하므로 $kNN_q(t_2)\subseteq kNN_q(t_1)$가 성립한다. 두 집합 $kNN_q(t_1)$와 $kNN_q(t_2)$에서 $\| kNN_q(t_2) \|$ = $\| kNN_q(t_1) \|$ 이고 $kNN_q(t_2)\subseteq kNN_q(t_1)$인 경우는 $kNN_q(t_2)\equiv kNN_q(t_1)$ 경우 밖에 없다.

이재문(Lee, Jae Moon)

1986 한양대학교 전자공학과(학사)
1988 한국과학기술원 전기및전자공학과(석사)
1992 한국과학기술원 전기및전자공학과(박사)
1994~ 한성대학교 멀티미디어공학과 교수

관심분야 : 데이터베이스, 기계학습, 게임프로그래밍