대표 보이드를 이용한 대규모 무리의 효율적인 무리짓기*

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

An Efficient Flocking Behaviors for Large Flocks by Using Representative Boid

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

요 약

본 논문에서는 임의적으로 움직이고 미리 정해진 위치가 없는 보이드들의 효율적인 무리짓기 대한 알고리즘을 제안한다. 하나의 보이드에 대하여 근사적으로 kNN을 찾고 행위특성의 값을 계산함으로써 제안하는 알고리즘은 기존의 공간 분할 알고리즘을 개선한다. 이를 위하여, 본 논문은 보이드들의 한 그룹에 대하여 평균 방향과 위치를 갖는 대표 보이드를 정의하여 사용한다. 제안하는 알고리즘은 구현되었으며 기존의 알고리즘과 실험적으로 비교되었다. 실험적 비교 결과로부터 제안하는 알고리즘이 기존의 알고리즘에 비하여 초당 렌더링 프레임 수 관점에서 약 -5~130%까지의 개선 효과가 있음을 알 수 있었다.

ABSTRACT

This paper proposes an algorithm for efficient behaviors of boids which freely move and have no predefined position. By finding the kNN and computing the value of behavioral characteristic of a boid approximately, the proposed algorithm improves the conventional spatial partitioning one. To do this, this paper defines and uses the representative boid which has the average direction and position for a group of boids. The proposed algorithm was implemented and compared with the conventional one experimentally. The results of the experimental comparisons show that the proposed algorithm outperforms the conventional one about -5~130% in terms of the ratio of the number of rendering frames per the second.

Keyword : Game AI, Boid, Flocking, Spatial Partitioning, kNN

접수일자 : 2008년 7월 8일
심사완료 : 2008년 8월 7일
* 본 연구는 2007년 한성대학교 교내연구비 지원과제임

1. 서 론

무리짓기란 메뚜기 떼와 같이 무리를 지어서 다니는 동종의 수많은 보이드(정해진 위치가 없이 자유롭게 움직이는 객체)들의 자동 움직임을 말한다. 이러한 무리의 특징은 무리를 이끄는 우두머리도 없을 뿐만 아니라 어디로 이동할지 예측하지 못한다는 것이다. 무리짓기는 본질적으로 임기응변적인 행동을 한다. 무리의 보이드들은 무리가 어디로 가는지에 대해서는 전혀 알지 못하지만, 모든 보이드들은 하나의 무리로서 움직이고 장애물과 적들을 피하며, 다른 보이드들과 보조를 맞춰서 유동적으로 이동한다. 최근 이러한 무리짓기가 게임, 만화 및 영화 등에 애니메이션으로 활용되어 이들에 대한 연구가 활발히 진행되고 있다[1, 2, 5, 7].

무리짓기의 가장 주목할 만한 특징은 각각의 보이드는 무리에 대한 어떠한 정보도 가지지 않는다는 점이다. 즉, 각각의 보이드는 매 순간마다 자신의 주변을 다시 평가할 뿐, 무리에 대한 정보는 전혀 가지고 있지 않다. [1]에서 이러한 무리짓기에 대하여 처음으로 다루었다. [1]에서 저자는 자동화된 에이전트들의 집단이 새 떼나 물고기 떼, 또는 벌 떼와 비슷한 집단행동을 보이도록 만들기 위하여 다음과 같은 세 가지 규칙을 적용하였다.

- 정렬(Alignment) 규칙: 주변 보이드들과 같은 방향을 가리키도록 노력한다.
- 응집(Cohension) 규칙: 주변 보이들과 평균 위치쪽으로 방향을 돌리려 노력한다.
- 분리(Separation) 규칙: 주변 보이드들과 충돌하지 않도록 방향을 돌리려 노력한다.

무리짓기에서 대부분의 연구는 이러한 정렬 규칙, 응집 규칙 및 분리 규칙을 효율적으로 처리하는 데 초점을 맞춘다. 이들의 효율적인 처리는 하나의 보이드에 정렬 규칙, 응집 규칙 및 분리 규칙에 영향을 주는 주변 보이드들을 효율적으로 선택하는 문제와 같다[3, 4, 5, 6, 7]. 이에 대한 가장 일반적인 두 가지 접근 방식이 있다. 첫 번째 방식은 주어진 반경 내에 있는 모든 다른 보이드들을 선택하는 방법이고, 두 번째 방식은 k개의 가장 가까이 있는 보이드들을 선택하는 것이다[5]. 후자의 방법을 일반적으로 kNN(k-nearest neighbors)이라 부른다. 전자는 비교적 간단한 알고리즘에 비하여 임의의 보이드들에 대하여 영향을 주는 주변 보이드들의 수가 일정하지 않다는 단점을 갖고 있다. 후자는 전자의 단점을 보완하는 것으로 항상 k개의 가장 가까운 이웃 보이드들만 영향을 주도록 제어할 수 있는 반면 k번째 주변에 많은 보이드가 존재하는 경우 이들에 대하여 정확한 식별이 필요하므로 비용이 많이 든다. 임의의 보이드에 대하여 주변 보이드들을 선택하기 위해서 n개의 보이드가 존재하는 경우 $O(n)$의 시간 복잡도를 갖는다. 일반적으로 무리짓기에서는 모든 보이드에 대하여 주변 보이드를 선택하여야 하므로 한 무리의 보이드들의 순간 이동을 모두 처리하기 위해서는 $O(n^2)$의 시간 복잡도를 갖는다. [2, 3, 5]에서는 후자의 방법에 대하여 공간 분할 기법을 도입하여 시간 복잡도를 $O(kn)$으로 개선하였다.

본 논문에서는 후자의 방법에 대한 연구로써 근사적 kNN을 찾고, 이들에 대한 대표 보이드를 사용하여 정렬 규칙, 응집 규칙 및 분리 규칙을 근사적으로 계산하도록 하므로써 기존의 무리짓기 알고리즘의 성능을 개선한다. 2장에서는 정렬 규칙, 응집 규칙 및 분리 규칙의 계산 방법과 기존의 알고리즘에 대하여 설명하며, 3장에서는 새로운 알고리즘을 제안하며, 4장에서는 기존의 알고리즘과 제안하는 알고리즘의 성능을 비교 분석한다. 마지막으로 5장에서 결론을 논의한다.

2. 관련연구

2.1 기본적인 무리짓기 알고리즘

무리짓기에 대한 연구는 [1]에서 시작되었다. [1]에서는 자동적으로 움직이는 게임 에이전트의 움직임을 액션 선택, 조종(steering) 및 이동력(locomotion)의

세 가지 층으로 분리하고 이러한 세가지 층에서 가장 많은 계산을 요구하는 층이 조종층이라 하였다. 이것은 조종층의 계산이 공간상에 존재하는 모든 물체에 대하여 kNN을 찾아야 하고 이들을 이용하여 새로운 방향을 계산하여야 하기 때문이다. 본 논문에서는 무리짓기의 계산을 조종층의 계산으로 한정한다.

임의의 보이드에 대하여 kNN이 찾아진 경우 이 보이드에 대한 새로운 방향은 정렬 규칙, 응집 규칙 및 분리 규칙을 적용하여 결정하는데, [1]에서는 이들을 각각 벡터로 계산하여 통합하는 방법을 제시하였다. 임의의 보이드 b에 대하여 보이드 집합 $\{b_1, b_2, ..., b_k\}$를 보이드 b의 kNN이라 하자. 이때 보이드 b의 정렬벡터, 응집벡터, 분리벡터 및 새로운 방향벡터는 다음과 같이 계산된다.

$$\text{정렬 벡터} = \frac{\sum_{i=1}^{k} bi.dir}{k} - b.dir \quad (1)$$

$$\text{응집 벡터} = \frac{\sum_{i=1}^{k} b_i.pos}{k} \quad (2)$$

$$\text{분리 벡터} = \sum_{i=1}^{k} \frac{b_i.pos - b.pos}{|b_i.pos - b.pos|^3} \quad (3)$$

$$\text{새로운 방향 벡터} = C_a \times \text{정렬벡터} + C_c \times \text{응집벡터} + C_s \times \text{분리벡터} \quad (4)$$

여기서 $b_i.dir$은 객체 b_i의 방향 벡터를 의미하고, $b_i.pos$는 객체 b_i의 위치 벡터를 의미한다. 또한 C_a, C_c, C_s는 일반적인 상수이고, 이들의 값은 새떼, 메뚜기떼 등 응용분야 및 응용 환경에 따라 적절한 값으로 정한다.

[1]에서는 상기의 계산에 근거하여 가장 기본적인 무리짓기의 알고리즘을 제시하였다. 이는 공간상에 존재하는 모든 보이드들에 대하여 각각 kNN을 찾고 이들로부터 정렬벡터, 응집벡터 및 분리벡터를 계산하여 새로운 방향을 결정하는 것이다. [그림 1]은 이에 대한 알고리즘이다. 이 알고리즘의 입력은 보이드의 집합인 *Flocking*과 영향을 받는 이웃 보이드의 수를 제한하는 상수 k이다.

Algorithm ***SteeringBasic***: Inputs: *Flocking*, *k*, Outputs: *None*

```
01: foreach boid in Flocking{
02:     kNNPQ={φ};
03:     foreach neighbor in Flocking{
04:         kNNPQ.AddBoid(boid, neighbor, k);
05:     }
06:     boid.newdir= ComputeSteerWithBoid(boid, kNNPQ);
07: }
08: // 이동층: 각 보이드에 대한 새로운 위치 계산
```

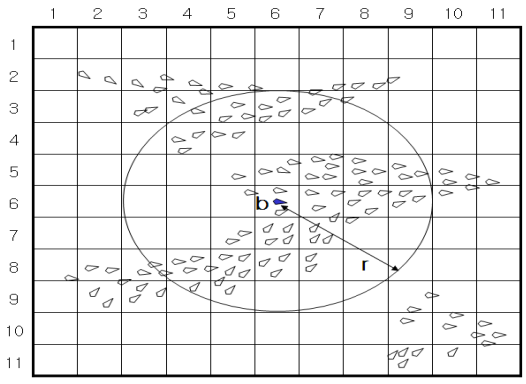
[그림 1] 기본적인 무리짓기 알고리즘

상기 알고리즘에서 라인 03~05는 하나의 보이드에 대하여 우선순위 큐(kNNPQ)를 사용하여 kNN을 찾는 과정이다. 라인 06에서 함수 *ComputeSteerWithBoid*은 하나의 보이드에 대하여 찾아진 kNN을 이용하여 식(1)~(3)를 계산하여 최종적으로 새로운 방향 식(4)를 계산한다. 상기 알고리즘의 최대 병목현상은 모든 보이드에 대하여 kNN을 찾는 것이고 이것이 $O(n)$이므로 전체적으로 이 알고리즘의 시간 복잡도는 $O(n^2)$이 된다. 여기서 n은 *Flocking*에서 보이드의 수이고, kNNPQ의 연산 비용은 무시하였다.

2.2 공간 분할 무리짓기 알고리즘

[2, 4]에서는 무리짓기 알고리즘에서 kNN을 효율적으로 찾기 위하여 셀 공간 분할 자료구조를 사용하였다. 이것은 게임 공간을 일정한 크기로 분할하고, 모든 보이드들을 위치에 따라 분할된 공간에 그룹핑하므로써 효율적으로 kNN을 찾고자 하는 것이다. [그림 2]는 공간 분할에 대한 2차원 예이다. [그림 2]에서와 같이 게임 공간을 11×11 배열로 분할하고, 모든 보이드들을 그들의 위치에 따라 배정하였다고 가정하자. 예를 들어 (0, 0)-셀에서는 어떠한 보이드도 할당되지 않았으며, (2, 3)-셀 에는 3개의 보이드가 할당되었다. 여기서 할당되었다는 것은 임의의 (x, y)-셀에 대하여 단순

히 이 셀만 엑세스하면 이 셀에 할당된 보이드들을 알 수 있다는 의미이다.



[그림 2] 무리짓기에서 공간 분할의 예

이러한 공간 분할된 보이드로 kNN을 찾는 것은 매우 단순하다. 예를 들어 [그림 2]에서 보이드 b에 대한 kNN을 찾고자 하는 경우 보이드 b를 중심으로 하는 원의 반경을 나타내는 r을 점차적으로 셀의 크기 단위로 증가하면서 반경 r의 원안에 포함된 셀을 엑세스하여 그 셀안에 존재하는 보이드들에 대하여 kNN을 찾는다. 따라서 이 경우 모든 보이드들을 대상으로 kNN을 찾는 것이 아니라 반경 r의 원안에 포함된 보이드들만을 대상으로 kNN을 찾기 때문에 매우 효율적으로 kNN을 찾게 된다.

[그림 3]은 공간 분할을 사용한 무리짓기 알고리즘이다. 이 알고리즘은 [그림 1]의 기본 무리짓기 알고리즘의 입력에다 두개의 정수 *xsize*, *ysize*을 입력으로 추가로 더 받는다.

```
Algorithm SteeringWithPS: Inputs: Flocking, k, xsize, ysize, Outputs: None
01:  Gird(xsize, ysize)= {ϕ};
02:  foreach boid in Flocking{
03:    (x, y)= MappingGrid(boid.pos);
04:    Grid(x, y).AddBoid(boid);
05:  }
06:  foreach boid in Flocking{
07:    kNNPQ={ϕ};
08:    (x, y)= MapingGrid(boid.pos);
09:    r= -1;
10:    while(r++>=0){
11:      for(i=x-r;i<=x+r;i++){
12:        for(j=y-r;j<=y+r;j++){
13:          if(i>x-r && i<x+r && j>y-r && j<y+r)
               continue;
14:          Cell= Grid(i, j);
15:          foreach neighbor in Cell{
16:            kNNPQ.AddBoid(boid, neighbor, k);
17:          }
18:        }
19:      }
20:      if(kNNPQ.Size()==k) break;
21:    }
22:    boid.newdir= ComputeSteerWithBoid(boid, kNNPQ);
23:  }
24:  // 이동층: 각 보이드에 대한 새로운 위치 계산
```

[그림 3] 공간 분할 무리짓기 알고리즘

공간 분할 알고리즘은 먼저 모든 보이드들을 분할된 공간에 할당하는 것으로 시작한다. 라인 02~05까지가 이것을 하는 것이다. 라인 03에서 *MappingGrid* 함수는 보이드의 위치를 그 위치가 속하는 그리드의 위치로 변환하는 단순한 함수이다. 이렇게 분할 배정된 보이드들에 대하여 라인 06-23까지는 모든 보이드에 대하여 kNN을 계산하고, 이들을 기초로 *ComputeSteerWithBoid*에서 새로운 방향을 계산하는 것이다. 이것은 [그림 1]에서 라인 01~07까지에 대응된다. [그림 3]에서는 kNN을 찾기 위하여 먼저 현재 보이드가 속한 셀의 위치를 찾고, 이 셀 *Grid(x, y)*에서부터 r을 증가하면서 가장 가까운 셀에 속한 보이드들을 대상으로 kNN을 찾는다. 라인 13의 조건문은 작은 r의 값에 대하여 이미 반영된 셀은 반복하여 반영하지 않기 위함이다. [7]에서는 효율적으로 인접 셀을 찾는 알고리즘을 제시하고 있다. 알고리즘으

로부터 k가 전체 보이드의 수 n에 대하여 작은 값일 경우 매우 빠른 성능을 보일 것을 알 수 있다. [그림 3]에서 라인 10~21이 k번 반복되므로 이 알고리즘의 전체 시간 복잡도는 O(kn)이 된다.

3. 새로운 무리짓기 알고리즘

3.1 기본 개념

[그림 3]에서 보이는 공간 분할 무리짓기 알고리즘의 가장 큰 병목요소는 임의의 보이드 b에 대하여 kNN을 찾는 것과 이러한 kNN을 이용하여 정렬벡터, 응집 벡터 및 분리 벡터를 계산하는 *ComputeSteerWithBoid*이다. 본 논문은 이들에 대한 계산을 근사적으로 함으로써 공간분할 알고리즘의 성능을 개선하고자 하는 것이다. 이를 위하여 본 논문에서는 다음과 같이 대표 보이드를 정의하여 사용한다.

정의 1: 임의의 보이드 집합 $\{b_1, b_2, ..., b_m\}$에 대하여 대표 보이드 b_{rep}는 다음과 같이 정의한다.

$$b_{rep}.pos = (\sum_{i=1}^{m} b_i.pos)/m \text{ 및 } b_{rep}.dir = \sum_{i=1}^{m} b_i.dir$$

정리 1: 보이드 b에 대한 kNN이 $\{b_1, b_2, ..., b_k\}$라 하고, $\{b_1, b_2, ..., b_k\}$의 대표 보이드를 b_{rep}라 하면 b의 정렬벡터 식(1)은 $\frac{b_{rep}.dir}{k} - b.dir$와 같다.

증명 1: *정의에 따라* $\frac{b_{rep}.dir}{k} - b.dir$는 $\frac{\sum_{i}^{k} b_i.dir}{k} - b.dir$*로 표현되고 이것은 식(1)과 같다.*

정리 2: 보이드 b에 대한 kNN이 $\{b_1, b_2, ..., b_k\}$라 하고, $\{b_1, b_2, ..., b_k\}$의 대표 보이드를 b_{rep}라 하면 b의 응집벡터 식(2)는 b_{rep}의 위치 벡터와 같다.

증명 2: *정의에 따라* b_{rep}*의 위치 벡터는* $\frac{\sum_{i=1}^{k} b_i.pos}{k}$ *이고 이것은 식(2)와 같다.*

정리 3: 보이드 b에 대한 kNN이 $\{b_1, b_2, ..., b_k\}$라 하고, $\{b_1, b_2, ..., b_k\}$의 대표 보이드를 b_{rep}라 하면 b의 분리벡터 식(3)은 $\frac{b_{rep}.pos - b.pos}{|b_{rep}.pos - b.pos|^3}$이 아니다.

증명 3: *정의에 따라* $b_{rep}.pos = \frac{\sum_{i=1}^{k} bi.pos}{k}$ *이므로 아래가 성립한다.*

$$\begin{aligned} b_{rep}.pos - b.pos &= \frac{\sum_{i=1}^{k} bi.pos}{k} - b.pos \\ &= \frac{\sum_{i=1}^{k} b_i.pos - k \times b.pos}{k} \\ &= \frac{\sum_{i=1}^{k} (b_i.pos - b.pos)}{k} \end{aligned} \quad (5)$$

따라서 식(5)를 원식에 적용하면 다음과 같다.

$$\frac{b_{rep}.pos - b.pos}{|b_{rep}.pos - b.pos|^3} = k^2 \times \frac{\sum_{i=1}^{k} (b_i.pos - b.pos)}{\left|\sum_{i=1}^{k} (b_i.pos - b.pos)\right|^3} \quad (6)$$

식(6)은 분리 벡터의 정의인 식(3)과 다르다.

정리 4: 보이드 b에 대한 kNN이 $\{b_1, b_2, ..., b_k\}$라 하고, $\{b_1, b_2, ..., b_k\}$의 대표 보이드를 b_{rep}라 할때 모든 i에 대하여 $b_i.pos - b.pos$이 동일한 값이면 b의 분리벡터 식

(3)은 $\frac{b_{rep}.pos - b.pos}{|b_{rep}.pos - b.pos|^3}$이 된다.

증명 4: *모든 i에 대하여 $b_i.pos - b.pos$가 동일한 값이면*

$$\sum_{i=1}^{k}(b_i.pos - b.pos) = k \times (b_t.pos - b.pos) \quad (7)$$

이 항상 성립한다. 여기서 t는 k보다 작거나 같은 임의의 자연수이다. 또한 식(7)을 이용하여 다음과 같이 식(8)을 쉽게 유도할 수 있다.

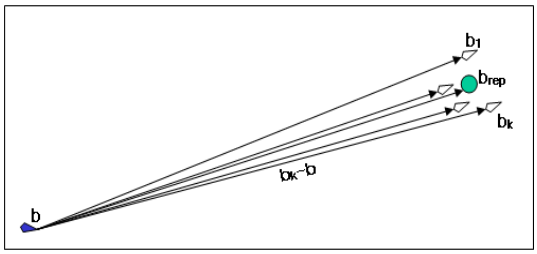
$$\frac{\left|\sum_{i=1}^{k}(b_i.pos - b.pos)\right|^3}{k^2} = \frac{k^3 \times |b_t.pos - b.pos|^3}{k^2}$$
$$= k \times |b_t.pos - b.pos|^3$$
$$= \sum_{i=1}^{k}|b_i.pos - b.pos|^3 \quad (8)$$

식(8)의 결과를 식(6)에 적용하면 다음과 같은 식을 얻는다.

$$\frac{b_{rep}.pos - b.pos}{|b_{rep}.pos - b.pos|^3} = k^2 \times \frac{\sum_{i=1}^{k}(b_i.pos - b.pos)}{\left|\sum_{i=1}^{k}(b_i.pos - b.pos)\right|^3}$$
$$= \frac{\sum_{i=1}^{k}(b_i.pos - b.pos)}{\sum_{i=1}^{k}|b_i.pos - b.pos|^3}$$
$$= \sum_{i=1}^{k}\frac{b_i.pos - b.pos}{|b_i.pos - b.pos|^3} \quad (9)$$

식(9)의 결과는 식(3)과 정확히 일치한다.

정리 4는 다른 각도에서 고려하면 매우 단순한 사실이다. 즉 "*모든 i에 대하여 $b_i.pos - b.pos$이 동일한 값*"이라는 의미는 $b_i.pos$와 $b.pos$가 위치 벡터임을 고려하면 kNN내의 모든 보이드가 동일 위치에 있다는 의미이고, 따라서 이런 경우 당연한 결과이다. 그러나 이러한 단순 사실에도 불구하고 정리 4로부터 다음과 같은 사실을 유추할 수 있다. kNN내의 보이드 b_1, b_2, ..., b_k가 서로 매우 인접한 거리에 있고, 이들이 모두 보이드 b와 매우 멀리 떨어져 있는 경우 $b_1.pos - b.pos \approx b_2.pos - b.pos \approx ... \approx b_k.pos - b.pos$가 성립함을 유추할 수 있다. [그림 4]는 이러한 경우에 대한 예이다. 여기서 화살표가 붙은 라인이 $b_i.pos - b.pos$를 의미한다. 그림에서 쉽게 알 수 있듯이 b_1, b_2, ..., b_k가 서로 매우 인접한 거리에 있으므로 그들의 방향($b_i.pos - b.pos$의 단위 벡터)은 유사할 것이고, b_1, b_2, ..., b_k가 b와 매우 멀리 떨어져 있으므로 그들의 크기($b_i.pos - b.pos$의 크기)는 비슷할 것이다. 따라서 $b_1.pos - b.pos \approx b_2.pos - b.pos \approx ... \approx b_k.pos - b.pos$가 성립함을 유추할 수 있다.



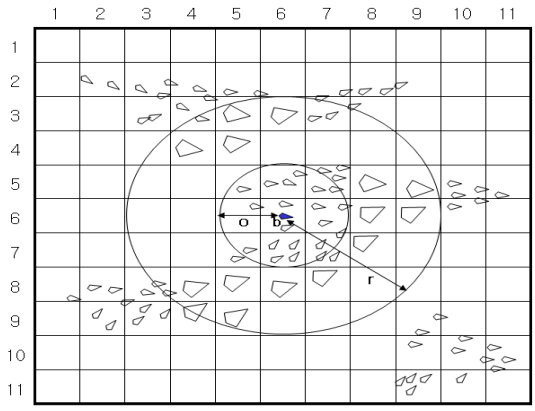
[그림 4] 대표 보이드

본 논문에서는 임의의 보이드 b의 kNN에 대하여 정리 1, 2, 4의 성질을 만족하는 부분적 보이드들에 대하여 대표 보이드를 계산하고, 이를 이용하여 근사적인 kNN, 정렬벡터, 응집벡터 및 분리벡터를 구함으로써 공간분할 무리짓기의 속도를 개선한다.

3.2 대표 보이드를 사용한 무리짓기 알고리즘

본 논문은 공간 분할 무리짓기 알고리즘에서 동일한 셀에 속하는 보이드들에 대한 대표 보이드를 정의하고 이를 사용하므로써 [그림 3]에서 kNN을 찾는 것과 *ComputeSteerWithBoid*를 효율적으로 계산하도록 하여 공간 분할 무리짓기 알고리즘의 성능을 개선한다. [그림 5]는 [그림 2]에서 보이는 공간 분할 무리짓기에서 부분적으로 대표 보이드를

도입하여 대표 보이드를 표시한 것이다. [그림 5]에서 작은 그림의 보이드는 원래의 보이드를 의미하며, 큰 그림의 보이드는 대표 보이드를 의미하고, 반경 r은 kNN에 포함되는 원을 표시하며, 반경 o는 보이드 b로부터 비교적 멀리 떨어져 있는 보이드들의 최소 거리를 표시하고 있다. 예를 들어 (4, 4)-셀에서의 대표 보이드는 [그림 2]에서 (4, 4)-셀에 있는 3개의 보이드에 대한 대표 보이드이다. 제안하는 알고리즘은 보이드 b에 대한 kNN을 구할 때 모든 kNN을 구하는 대신에 [그림 5]에서와 같이 반경 o와 r사이에 있는 보이드들에 대해서 필요하다면 대표 보이드를 구함으로써 kNN을 탐색하는 비용을 줄이고 *ComputeSteerWithBoid*를 효율적으로 계산하는 것이다. 반경 o와 r사이에 있는 보이드들에 대하여 대표 보이드를 적용하는 것은 정리 4의 성질을 이용하고자 하는 것이다. 물론 이러한 경우 정리 3과 4의 결과로부터 예측할 수 있듯이 보이드의 자연스러움이 떨어 질 수 있다.



[그림 5] 공간분할 무리짓기에서 대표 보이드

이러한 성질을 이용함에 있어서 대표 보이드를 생성하는 것 자체가 많은 계산량을 요구할 수 있다. 본 논문에서는 이의 비용을 줄이기 위하여 kNN을 찾을 때마다 대표 보이드를 생성하는 것이 아니라 알고리즘의 초기화의 일환으로 모든 셀에 대하여 대표 보이드를 사전에 생성하고, 이들을 kNN을 찾을때 마다 반복하여 사용하도록 한다. [그림 6]은 대표 보이드를 이용한 공간분할 무리짓기 알고리즘이다. 이 알고리즘은 [그림 3]의 공간분할 무리짓기 알고리즘에 비하여 입력으로 정수 o를 더 입력 받는다.

Algorithm ***SteeringWithRB***: Inputs: *Flocking, k, xsize, ysize, o*, Outputs: *None*

```
01: Gird(xsize, ysize)= {ϕ};
02: foreach boid in Flocking{
03:   (x, y)= MappingGrid(boid.pos);
04:   Grid(x, y).AddBoid(boid);
05: }
06: rGrid(xsize, ysize)= {ϕ};
07: // 각 셀에 대하여 대표 보이드 계산
    for(i=0;i<xsize;i++){
08:   for(j=0;j<ysize;j++){
09:     RepBoid.Create(Grid(i, j));
10:     rGrid(i, j).AddBoid(RepBoid);
11:   }
12: }
13: foreach boid in Flocking{
14:   kNNPQ={ϕ};
15:   (x, y)= MapingGrid(boid.pos);
16:   r= -1;
17:   while(r++>=0){
18:     for(i=x-r;i<=x+r;i++){
19:       for(j=y-r;j<=y+r;j++){
20:         if(i>x-r && i<x+r && j>y-r && j<y+r)
              continue;
21:         Cell= (|x-i|>=o or |y-j|>=o)?
                rGrid(i, j): Grid(i, j);
22:         foreach neighbor in Cell{
23:           kNNRPQ.AddBoid(boid, neighbor, k);
24:         }
25:       }
26:     }
27:     if(kNNRPQ.Size()>=k) break;
28:   }
29:   boid.newdir=ComputeSteerWithVirtualBoid(boid, kNNPQ);
30: }
31: // 이동층: 각 보이드에 대한 새로운 위치 계산
```

[그림 6] 대표 보이드를 사용한 공간 분할 무리짓기 알고리즘

[그림 6]의 알고리즘은 기존의 공간 분할 무리짓기의 알고리즘에서 라인 06~12 부분과 라인 21부분이 추가된 것이다. 라인 06~12 부분은 동일한 셀에 존재하는 모든 보이

드들에 대하여 초기에 하나의 대표 보이드(*RepBoid*)를 생성하여 대표 그리드(*rGrid*)에 할당하는 것이고, 라인 21은 o이상의 거리에 있는 보이드에 대해서는 대표 보이드를 선택하도록 하는 것이다. *kNNRPQ*는 대표 보이드가 삽입될 때에는 그 크기가 t(대표 보이드에 포함된 보이드의 수)만큼 증가한다. 알고리즘 *SteeringWithRB*는 시간 복잡도면에서는 알고리즘 *SteeringWithPS*와 동일하다. 그러나 4장에서 보이겠지만 실행속도는 *SteeringWithPS*를 크게 개선한다. 이 알고리즘은 라인 27에서 볼 수 있듯이 찾아진 kNN에 k개 이상의 보이드가 존재할 수 있다. 따라서 근사적 kNN이라 한다.

4. 성능비교

본 논문에서는 제안하는 알고리즘 *SteeringWithRB*의 성능을 측정하고 이를 비교하기 위하여 알고리즘 *SteeringWithPS*와 *SteeringWithRB*를 직접 구현하였다. 구현은 MS사의 비주얼 스튜디오 2005를 사용하여 C++로 구현하였고 OpenGL을 사용하였다. 실험 환경으로는 펜티움4 2.5GHZ, 메모리 2GB의 PC를 사용하였고 운영체제로는 윈도우즈 XP를 사용하였다.

실험은 임의의 환경에서 두 알고리즘의 초당 실행 프레임 수를 비교하는 것으로 하였다. 무리짓기의 성능을 측정하고 비교하는 다양한 방법이 있겠으나 논문에서는 공간분할의 수를 변경하면서 성능을 측정하기로 한다. 이를 위하여 전체 보이드의 수는 1,024개로 고정하였으며, 2,000 프레임까지 렌더링하는 시간을 측정하여 초당 평균 렌더링 수를 측정하였다. 제안하는 알고리즘의 성능을 다양하게 측정하기 위하여 k를 변화시키면서 성능을 측정하였다.

[표 1]은 1024개의 보이드가 운행중인 공간을 256(16×16)개의 셀로 분할하여 두 알고리즘의 성능을 측정한 것이다. 여기서 SWPS는 *SteeringWithPS*의 초당 프레임 렌더링 수를 의미하며, SWRB는 *SteeringWithRB*의 초당 프레임 렌더링 수를 의미한다. 두 알고리즘의 성능은 kNN을 100에서 200까지 증가시키면서 측정하였고, 제안하는 알고리즘 *SteeringWithRB*에 대해서는 알고리즘에서 제시한 o를 1, 2, 3으로 변화시키면서 측정하였다. o가 증가하면서 *SteeringWithRB*의 성능이 낮아지는 것은 대표 보이드의 적용이 적어지고 따라서 알고리즘 라인 06~12에서 대표 보이드의 생성이 오버헤드가 되기 때문이다. 예를 들어 o가 3이고, kNN이 100인 경우에는 오히려 제안한 알고리즘의 성능이 더 나쁘게 나타났는데 이것은 알고리즘에서 제시한 o와 r이 같아서 대표 보이드를 생성은 하였지만 전혀 사용하지 못하기 때문에 대표 보이드 생성 비용으로 오히려 성능이 낮아진 것이다. 그렇지만 o가 작다는 것은 더 많은 보이드에 대하여 대표 보이드를 적용하는 것이고 따라서 정리 3과 4의 결과로부터 예측할 수 있듯이 보이드들의 행동에 대한 자연스러움이 떨어질 수 있다. 실험 결과 o가 3인 경우 제안하는 알고리즘에서의 보이드들의 움직임과 기존의 알고리즘에서의 보이드들의 움직임이 다르다는 것을 시각적으로 구별할 수 있는 정도였고, o가 2 이하인 경우 두 알고리즘에서의 보이드들에 대한 움직임의 차이를 시각적으로 구별할 수 없었다.

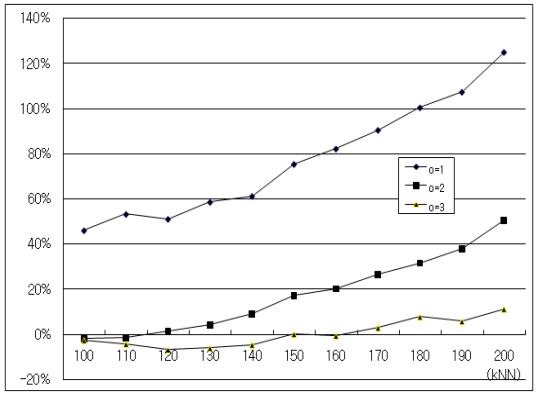
[표 1] *SteeringWithPS*와 *SteeringWithRB*의 초당 프레임 렌더링 수

kNN	*SWPS*	*SWRB*		
		o=1	o=2	o=3
100	10.8	15.8	10.6	10.6
120	10.4	15.6	10.5	9.7
140	9.3	15.1	10.2	8.9
160	8.2	14.9	9.9	8.2
180	7.3	14.7	9.6	7.9
200	6.3	14.2	9.5	7.0

[그림 7]은 [표 1]에 대한 데이터를 그래프로 나타낸 것이다. [그림 7]에서 가로 축은 kNN의 변화를 나타내며, 세로축(g)은 식(10)에 의하여 계산되었다.

$$g = \frac{(SWRB - SWPS)}{SWPS} \times 100(\%) \qquad (10)$$

여기서 g는 기존 알고리즘에 비하여 제안한 알고리즘의 상대적 성능 개선 비율을 말한다. [표 1]에서도 알 수 있었듯이 o가 증가하면서 개선 효과는 급격히 줄어들고 있으며, 또한 kNN이 증가하면서 개선 효과가 급격히 증가하고 있음을 알 수 있다. 이것은 o가 증가하면 대표 보이드의 사용이 줄어들어 개선 효과가 감소하는 것이고, kNN이 증가하면 r이 커지고 따라서 대표 보이드의 사용이 증가하기 때문에 개선 효과가 증가 하는 것이다. 즉, 이것으로부터 대표 보이드에 따라 성능이 매우 민감함을 알 수 있다.



[그림 7] *StreeringWithPS*와 *StreeringWithRB*의 성능비율

5. 결 론

본 논문에서는 임의적으로 움직이는 보이드들에 대한 무리짓기 알고리즘을 제안하였다. 기존의 널리 알려진 공간 분할 무리짓기 알고리즘을 개선하는 것으로 근사적 kNN과 분리 특성에 대한 근사적 값을 계산하여 성능을 개선한다. 이를 위하여 대표 보이드를 정의하였으며, 이를 이용하여 근사적 kNN을 찾고 무리짓기의 특성을 계산하도록 하였다. 제안하는 알고리즘의 성능을 측정하기 위하여 이를 구현하였으며, 기존의 알고리즘과 성능을 비교하였다. 성능 비교 결과 환경에 따라 -5~130%까지의 개선 효과를 주었다.

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] Ting Liu, Andrew Moore, Alexander Gray, Ke Yang, “An Investigation of Practical Approximate Nearest Neighbor Algorithms”, Advances in Neural Information Processing System, 2004.

[4] Mat Buckland, “Programming Game AI by Example”, ISBN 1556220782, Wordware Publications, 2005.

[5] Reynolds, et al, “OpenSteer: Steering Behaviors for Autonomous Characters”, http://opensteer.sourceforge.net, 2006.

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

[7] Nicolas Brodu, “Query Sphere Indexing for Neighborhood Requests”, http://nicolas.brodu.free.fr/common/recherche/publications/QuerySphereIndexing.pdf, 2007.

이재문 (Jae Moon Lee)

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

관심분야 : 데이터베이스, 기계학습, 게임프로그래밍 등