

컴퓨터 세이더를 이용한
군집 알고리즘 구현

서강대학교 일반대학원
미디어공학 전공
유림

<목 차>

I . 서론

1.1 연구 배경

1.2 연구 내용

II . 본론

2.1 군집 알고리즘

2.2 컴퓨트 셰이더

2.3 컴퓨트 셰이더를 이용한 군집 알고리즘 구현

III . 결론

I. 서론

1.1 연구 배경

2020년 서울에서 세계 최초로 공개된 팀랩(teamLab)의 신작은 디지털 세계 속에 탄생한 동·식물들이 관객의 움직임에 반응하는 놀라운 경험을 전하고, 끊임없이 변화하는 작품들이 압도적 스케일의 공간을 통해 생명의 아름다움을 표현했다. DDP 배움터 디자인 전시관에 마련된 독립된 8개 어두운 공간에는 벽과 바닥, 천장에 꽃, 나비, 동물, 파도 등 자연을 주제로 한 영상과 이미지가 펼쳐지는데, 동물의 형상을 만들어 가거나 끊임없이 증식하는 꽃들, 사람들 발밑에서 태어나는 나비 떼, 입자 간 상호 작용을 하는 무수한 물 입자의 연속체가 그것이다. 이처럼 수많은 생명체가 끊임없이 변화하는 모습만으로도 관객으로 하여금 몰입감을 유발할 수 있음을 시사한다.



<그림-1>

군집 알고리즘은 뉴미디어 예술 작품에서 다양하게 쓰이면서 그 자체로 복잡계로서 의미를 지닌다. 새 떼와 물고기 떼의 군집 운동을 모델화하는 방법으로 개체의 운동 규칙을 세 가지로 구분하여 제시하고, 그 운동을 발생하게 하는 지각 범위를 지정하여 전체 운동을 효과적으로 설명하였다. 실제로 새 떼가 이와 같은 규칙으로 움직이는 것과는 별도로 군집 알고리즘은 군집 운동의 복잡한 패턴을 효과적으로 모델화한다. 이 알고리즘은 새 떼의 운동뿐만 아니라 물고기 떼 등의 운동을 묘사할 수 있기 때문에 컴퓨터 애니메이션(Compute Animation)이나 게임, 영화 CG(Computer Graphics) 분야에서 다양하게 활용되고 있다.¹⁾

하지만 군집 알고리즘을 기반으로 움직이는 수많은 개체들을 실시간 렌더링(Real-Time Rendering)을 위해서는 많은 물리 연산을 수행해야하므로 연산 성능의 저하가 심각한 상황이다. 이를 해결하기 위해 GPU(Graphics Processing Unit)를 활용해 연산을 수행하는 GPGPU(General Purpose Computing on Graphics Processing Units)가 많이 사용되고 있는데, 최근 컴퓨터 그래픽 하드웨어(Computer Graphics Hardware), 물리 기반 시뮬레이션 기술(Physically Based Simulation)의 발전으로 인해 빠른 속도로 수행이 가능하고, 안정성 부분에서도 많은 향상이 이루어지고 있다.

1.2 연구 내용

본 논문에서는 컴퓨터 셰이더를 이용한 군집 알고리즘 구현을 목표로 한다. 컴퓨터 셰이더를 이용하지 않고 수많은 개체들을 군집 알고리즘을 기반으로 움직이게 하기 위해서는 많은 물리 연산 수행이 필요하며 이 때문에 프레임 속도(Frame Rate)가 매우 떨어진다는 단점이 있다. 이 논문에서 컴퓨터 셰이더 및 GPU 인스턴싱(GPU Instancing)을 통해 군집 알고리즘을 구현하고, 이를 기반으로 나비 10,000 마리가 움직이는 모습을 실시간 렌더링 하여도 안정적인 프레임 속도를 보장할 수 있다는 점이 주목할 만하다.

II. 본론

2.1 군집 알고리즘

인공생명(Artificial Life, A-Life)은 생명 현상을 재창조 또는 모방하며 로봇, 컴퓨터 모델, 생화학으로 시뮬레이션을 하는 학문으로, 생명 현상의 특징인 유전 알고리즘을 응용해 만들어진다. 인공생명이란 용어는 1986년 이 분야의 산파로 불리는 Christopher Langton에 의해 만들어졌다. Christopher Langton는 인공생명의 핵심 개념을 창발적 행동(Emergent Action)으로 보고, 이러한 행동은 개별적 행동들 사이의 모든 국지적인 상호작용으로부터 출현하며, 인공생명이 채택한 방법론인 아래로부터 위로의, 분산된, 국지적인 행동에 따른 결정으로부터 나온다고 설명한다.

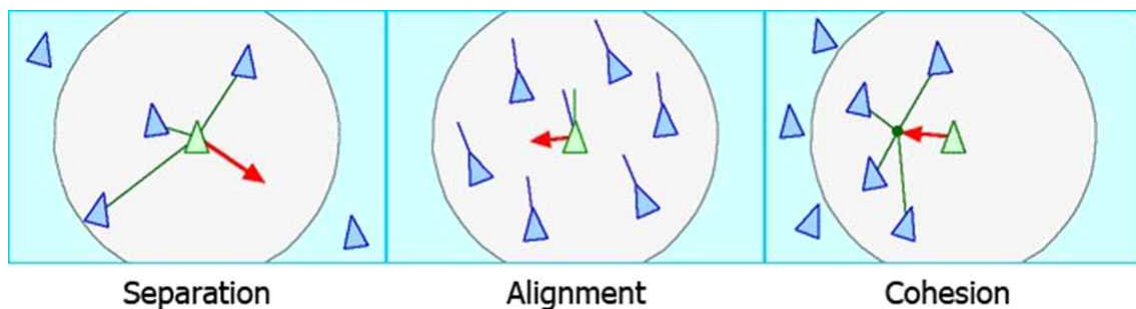
인공생명은 주요 3가지, 딱딱한 인공생명(Hard A-Life), 부드러운 인공생명(Soft

1) 오준호, “뉴미디어 예술 작품에 적용된 알고리즘의 미학적 함의 : 라이브 코딩을 중심으로”, 한국콘텐츠학회 논문지 제13권 3호

A-Life), 그리고 젖어있는 인공지능(Wet A-Life)로 분류된다. 이들 이름은 인공지능에 대한 접근 방법에서 따왔다. 컴퓨터 프로그램에서 적절한 모형을 만들어 생명의 형태를 탐구하는 부드러운 인공지능, 하드웨어에서 생명과 관련된 것을 구현하는 것으로써 로봇 공학과 밀접한 관계가 있는 단단한 인공지능, 생화학적 물질로부터 생명이 있는 계를 합성하려는 젖은 인공지능으로 구분된다.

부드러운 인공지능은 소프트웨어 프로그램 상으로 존재하는 인공지능을 말한다. 기존의 로봇의 동작이나 에이전트의 애니메이션은 사람이 일일이 움직임을 지정하여 시간 혹은 조건에 따라 움직여왔다. 또한 최적의 움직임을 위해 정교한 수식을 동원해야 하는데 이것은 많은 인력과 시간을 요구하며, 다양한 패턴의 행동을 만들기 어려운 한계가 있다. 그에 반해 유전자 알고리즘을 사용한 인공지능 로봇은 스스로 문제를 해결하고 최적화된 움직임을 찾아내는데서 기존의 인공지능 기반 로봇과 차별화를 가지고 있다.²⁾

군집 알고리즘(Flocking Algorithm)이란 1986년에 Craig Reynolds가 처음 소개하였으며 군집(Flock)에서의 각 개체의 행동 모델을 구현하는 알고리즘으로 보이드(Boid)라고 칭해지는 각각의 개체가 응집(Cohesion, 주변 보이드로 모이는 규칙), 정렬(Alignment, 주변 보이드와 같은 방향을 향하려는 규칙), 분리(Separation, 각 보이드가 너무 가까워지지 않으려는 규칙)의 세 가지 규칙을 이용하여 움직이는 것을 말한다.



<그림-2>

각각의 보이드는 매 순간마다 자신의 주변을 다시 평가할 뿐 무리에 대한 정보는 가지고 있지 않다. 무리의 보이드는 어디로 가는지에 대해서 전혀 알지 못하지만 모든 보이드는 하나의 무리로서 움직이고, 장애물과 적들을 피하며, 다른 보이드와 보조를 맞춰서 유동적으로 이동하게 된다. 이는 단순한 규칙의 적용으로 복잡한 행

2) 위키백과

동이 나타나는 창발적 특징을 보여주며, 복잡해 보이는 자연 현상이 실제로는 단순한 규칙들의 상호작용들로 이루어졌을지도 모른다는 가정을 갖게 한다.

영화에 컴퓨터 그래픽으로 거대한 군집의 움직임을 구현할 때 보통은 하나하나를 특정한 위치에 그린다. 하지만 인공생명 기법으로 생명체의 군집 행동을 모방하여 움직이도록 만들 수 있다. 예를 들어 개체들 사이의 거리만 생각해볼 때 하나의 개체가 바로 주위의 다른 개체와 너무 가깝지도, 너무 멀지도 않아야 한다. 이 조건을 컴퓨터 그래픽에 적용해 각 개체가 복잡 미묘하게 움직이게 하고 전체 군집의 행동을 자연스럽게 만드는 것이 인공생명 기법이다.

군집 알고리즘은 응집 힘(주변 보이드들과 가까운 쪽으로 가려는 힘), 정렬 힘(주변 보이드들과 같은 방향을 가리키려는 힘), 분리 힘(주변 보이드들과 먼 쪽으로 가려는 힘)의 세 벡터에 상수를 곱한 값과 진행 방향의 벡터를 합한 값으로 보이드의 다음 위치를 계산한다. 개발자는 임의의 실험에 따라 상수의 값을 조절하여 원하는 값을 찾아내야 한다. 위 규칙을 이용하여 구현된 군집은 각각의 보이드가 자신의 360도 범위 내에 있는 모든 보이드를 고려해야 하므로 자원 소모 문제가 발생하는 단점이 있어 컴퓨트 셰이더를 이용해 이 문제를 극복하려 한다.

2.2 컴퓨트 셰이더

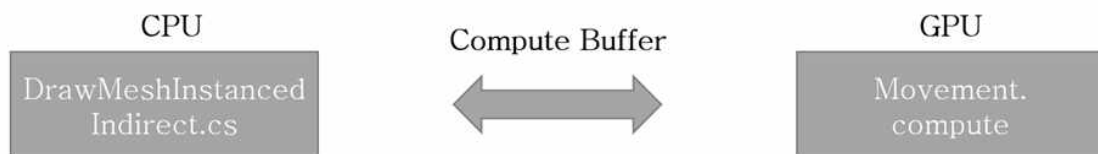
최근 그래픽 아키텍처(Graphics Architecture)의 부동 소수점 벡터 연산 지원과 이러한 연산에 최적화된 하드웨어 설계를 통해 비약적인 발전으로 GPU의 부동 소수점 연산 속도는 CPU(Central Processing Unit)의 연산 속도보다 압도적으로 빠르다. 기존의 GPU는 그래픽 렌더링(Graphics Rendering)만을 위해서 사용되었으나, 최근 이러한 자원을 활용하여 그래픽스 목적이 아닌 범용적인 용도로 사용하는 기술인 GPGPU가 등장했다. GPGPU는 CPU에 비해 월등히 많은 코어(Core)로 이루어진 GPU에 작업을 할당하여, 병렬 프로세싱으로 작업을 진행한다.

게임 성능의 결정적인 부분은 CPU와 GPU의 역할이 크며 이들의 한계와 제약을 극복하기 위한 방법으로 Unity3D에서 병렬 프로그래밍을 이용하여 최적화 방법을 한다. CPU에서 몇 개의 코어를 이용하거나 Thread를 이용한 병렬 처리와 GPU의 프로그래밍이 가능한 Shader를 이용한 병렬 처리로 성능 향상을 하는 데는 한계점이 있다. 하지만 최근 병렬 프로그래밍 방법에는 GPU에서의 수많은 코어를 이용하여 프로그래밍 가능한 GPGPU를 이용하는 방법으로 병렬 처리의 성능 향상을 극대

화 할 수 있다. GPGPU의 특화된 언어로는 CUDA(Compute Unified Device Architecture), OpenCL(Open Computing Language), Compute Shader(DirectCompute)가 있다.³⁾

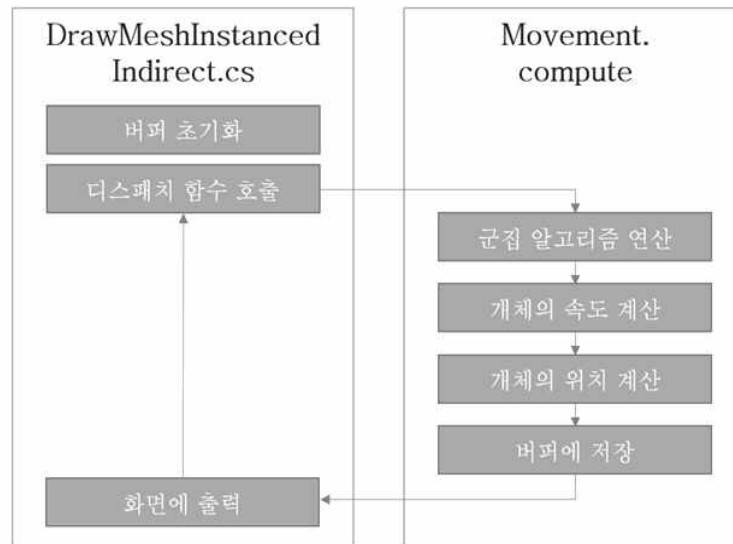
컴퓨터 셰이더는 그래픽 프로세서(Graphics Processor)를 통한 범용 연산을 지원하는 API(Application Programming Interface)이다. 일반 셰이더와 비슷하며 *.compute를 확장자로 가지는 에셋(Asset) 파일이다. DirectX11의 HLSL(High Level Shader Language) 언어로 작성되어 있으며 프라그마(pragma) 지시문을 사용하여 커널(Kernel) 사용 여부를 지정한다. 셰이더의 핵심은 커널로 셰이더로의 진입점이 되고, 스크립트(Script)에서 디스패치(Dispatch) 함수를 호출하여 셰이더를 동작하게 한다.

본 논문에서는 GPU 인스턴싱으로 동일한 메쉬(Mesh)를 한 번에 렌더링(Rendering) 할 수 있는 DrawMeshInstancedIndirect.cs 스크립트를 이용한다. 이 스크립트에서 메쉬의 물리적인 특성이 담긴 버퍼(meshPropertiesBuffer)를 초기화한 뒤 업데이트(Update) 함수에서 디스패치 함수를 호출하면 Movement.compute 스크립트가 동작하게 된다. 이 스크립트에는 각 개체의 응집 속도, 정렬 속도, 그리고 분리 속도를 구할 수 있는 군집 알고리즘 연산이 포함되어 있으며 개체의 속도와 위치를 계산하여 버퍼에 저장하게 된다. 이어 DrawMeshInstancedIndirect.cs 스크립트에서 graphics.DrawMeshInstancedIndirect() 명령어로 화면에 개체를 출력하는 과정을 반복, 군집 알고리즘을 기반으로 하는 수많은 메쉬를 한 번에 렌더링 할 수 있다.



<그림-3>

3) 이기수, “Unity3D에서 CPU/GPU 부하분산 최적화 기법”, 학위논문 사항



<그림-4>

3.3 컴퓨터 셰이더를 이용한 군집 알고리즘 구현

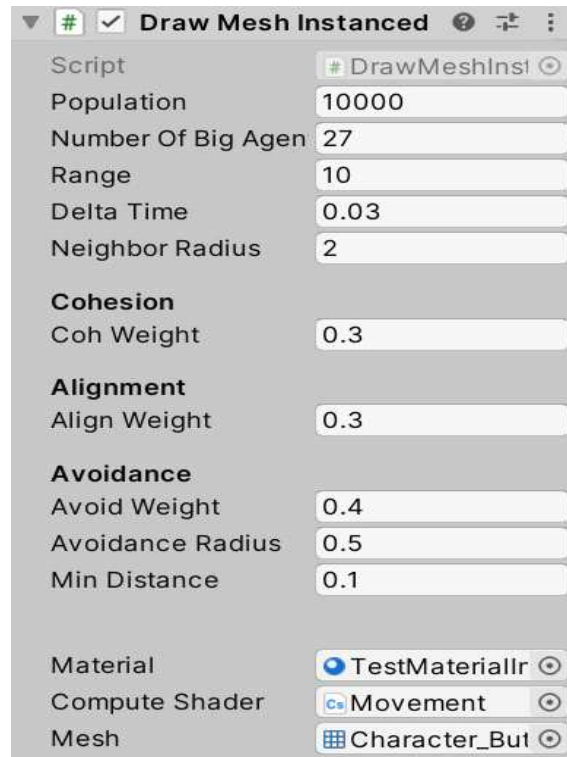
군집 알고리즘에서 **응집 속도(Cohesion Velocity)**는 한 개체를 중심으로 사용자가 정한 `_neighborRadius` 반경 내에 존재하는 보이드의 질량 중심(centerOfMass)를 구한 다음, 이 질량 중심을 향하는 속도 벡터(cohTargetVelocity)를 계산하여 구할 수 있다. **정렬 속도(Alignment Velocity)**는 한 개체에 대해 사용자가 정한 `_neighborRadius` 반경 내에 존재하는 보이드의 속도를 모두 더한 다음 (sumOfOtherAgentVelocity), 보이드의 개수(neighborCount)로 나누어 구할 수 있다. **분리 속도(Separation Velocity)**는 한 개체를 중심으로 사용자가 정한 `_avoidanceRadius` 반경 내에 가까이 존재하는 보이드로부터 멀어지는 속도 (avoidVelocity)를 구한 다음, 이 개체에 대해 보이드가 가까이 있을수록 avoidVelocity가 높게 책정될 수 있도록 avoidVelocity를 개체와 보이드 간의 거리 (dist)로 나누고, 이를 총합(sumOfValidAvoidVelocity), 보이드의 개수(nAvoid)로 나누어 구할 수 있다.

군집 알고리즘	스크립트
응집(Cohesion)	<pre> float3 myPosition = getAgentPosition(myIndex, _Properties); float3 myVelocity = _Properties[myIndex].velocity; float neighborCount = 0.0; float3 sumOfOtherAgentPosition = float3(0.0, 0.0, 0.0); float3 cohVelocity = float3(0.0, 0.0, 0.0); for (int index = 0; index < _population; index++) { float3 otherAgentPosition = getAgentPosition(index, _Properties); if (distance(myPosition, otherAgentPosition) < _neighborRadius) { sumOfOtherAgentPosition += otherAgentPosition; neighborCount += 1; } } if (neighborCount == 0.0) { cohVelocity = myVelocity; } float3 centerOfMass = sumOfOtherAgentPosition / neighborCount; float3 cohTargetVelocity = centerOfMass - myPosition; cohVelocity = cohTargetVelocity; return cohVelocity; </pre>
정렬(Alignment)	<pre> for (int index = 0; index < _population; index++) { float3 otherAgentPosition = getAgentPosition(index, _Properties); if (distance(myPosition, otherAgentPosition) < _neighborRadius) { float3 otherAgentVelocity = getAgentVelocity(index, _Properties); sumOfOtherAgentVelocity += otherAgentVelocity; neighborCount += 1.0; } } alignVelocity = sumOfOtherAgentVelocity; if (neighborCount == 0.0) { alignVelocity = myVelocity; } else // neighborCount != 0.0 { alignVelocity = sumOfOtherAgentVelocity / neighborCount; if (length(alignVelocity) == 0.0) { alignVelocity = myVelocity; } } return alignVelocity; </pre>
분리(Seperation)	<pre> for (int index = 0; index < _population; index++) { float3 otherAgentPosition = getAgentPosition(index, _Properties); if (distance(myPosition, otherAgentPosition) < _avoidanceRadius) { float3 avoidVelocity = myPosition - otherAgentPosition; float dist = length(avoidVelocity); if (dist < _minDistance) { dist = _minDistance; } sumOfValidAvoidVelocity += avoidVelocity / dist; nAvoid += 1; } } if (nAvoid == 0.0) { avoidVelocity = myVelocity; } if (nAvoid > 0.0) { avoidVelocity = sumOfValidAvoidVelocity / nAvoid; } return avoidVelocity; </pre>

<표-1>

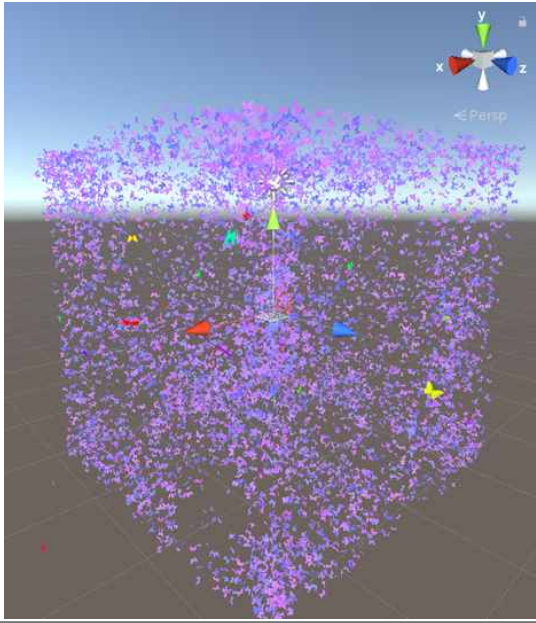
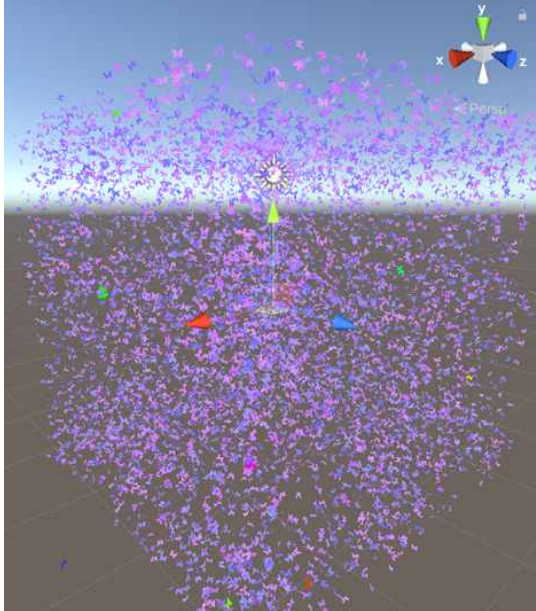
실험에 앞서 유니티 인스펙터(Inspector) 창을 통해 군집 알고리즘을 제어할 수 있다. Population은 나비 군집을 형성하는 개체의 숫자로, 이 실험에서는 10,000으로 설정하였다. 유니티 인스펙터 창에 원하는 개체의 숫자를 적으면 그 수에 따라 나비 개체가 그려지게 된다. Range는 나비 군집이 생성되는 위치 범위인데, 만약 Range가 10일 경우 정육면체 중심으로부터 각 면에 내린 수선의 길이가


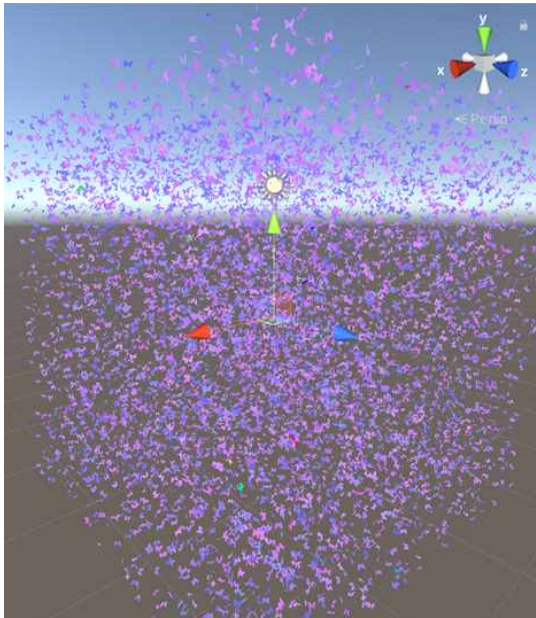
10[m]라는 의미이기 때문에 한 변의 길이가 20[m]인 정육면체 내에 나비 군집이 생성된다. 군집 알고리즘의 3가지 규칙(응집, 정렬, 그리고 분리)에 대해 각 가중치를 조절하여 사용자가 원하는 군집 운동을 만들어낼 수 있다.



<그림-5>

첫 번째 실험에서는 응집 속도에 큰 가중치, 나머지 두 속도에 동일한 가중치를 두고 실험을 진행하였다. 한 변의 길이가 20[m]인 정육면체 내에 생성된 10,000마리의 나비 군집이 Neighbor Radius(2[m]) 반경으로 빠르게 응집되는 것을 볼 수 있다. 두 번째 실험에서는 정렬 속도에 큰 가중치, 나머지 두 속도에 동일한 가중치를 두었으나, 이 경우에는 정렬 규칙보다는 응집 규칙이 두드러져 보이는 모습이었다. 세 번째 실험에는 분리 속도에 큰 가중치, 나머지 두 속도에 동일한 가중치를 두었고, 예상대로 나비 군집이 서로 멀어지는 모습을 확인할 수 있었다. 여러 실험 끝에 응집 가중치(Cohesion Weight)가 0.3, 정렬 가중치(Alignment Weight)가 0.3, 분리 가중치(Separation Weight)가 0.4일 때 나비 군집이 조화롭게 군집 운동을 유지하는 것을 볼 수 있었다. 이때 프레임 속도 또한 30[FPS] 이상으로 안정적이었다.

실험 결과	Weight
	<p> Cohesion Weight = 0.8 Alignment Weight = 0.1 Separation Weight = 0.1 </p>
	<p> Cohesion Weight = 0.1 Alignment Weight = 0.8 Separation Weight = 0.1 </p>

	<p>Cohesion Weight = 0.1 Alignment Weight = 0.1 Separation Weight = 0.8</p>
	<p>Cohesion Weight = 0.3 Alignment Weight = 0.3 Separation Weight = 0.4</p>

<표-2>

III. 결론

컴퓨터 셰이더를 이용하지 않고 수많은 개체들을 군집 알고리즘을 기반으로 움직이게 하기 위해서는 많은 물리 연산 수행이 필요하며 이 때문에 프레임 속도 (Frame Rate)가 매우 떨어진다는 단점이 있다. 하지만 본 논문에서는 컴퓨터 셰이더 및 GPU 인스턴싱을 통해 군집 알고리즘을 구현하고, 이를 기반으로 나비 10,000 마리가 움직이는 모습을 실시간 렌더링, 30[FPS] 이상의 프레임 속도를 보장할 수 있다는 점이 주목할 만하다.

실제 실험에서 한 변의 길이가 20[m]인 정육면체 내에 10,000 마리의 나비 군집을 생성하고 Neighbor Radius를 2[m]로 설정하여 각각의 성분에 대해 큰 가중치를 부여, 군집 운동을 관찰하려 하였으나 정렬 속도에 큰 가중치를 두었을 때 군집 운동에서는 정렬 규칙보다는 응집 규칙이 두드러져 보이는 모습이었는데 그 이유에 대해서는 추후 검증이 필요해 보인다.

Reference

- [1] 오준호, “뉴미디어 예술 작품에 적용된 알고리즘의 미학적 함의 : 라이브 코딩을 중심으로”, 한국콘텐츠학회 논문지 제13권 3호
- [2] <https://ko.wikipedia.org/wiki/인공생명>
- [3] 이기수, “Unity3D에서 CPU/GPU 부하분산 최적화 기법”, 학위논문 사항