< 캡스톤디자인 최종보고서 >

프로젝트 명: Learned MotionMatching 을 적용한 택배 배달 게임 개발

팀원(참여학생): 이재형(소프트웨어융합학과), 최진욱(소프트웨어융합학과)

요 약

MotionMatching 이란 가상 캐릭터 애니메이션을 생성하는 데 사용되고 있는 기술로, Clavet 와 Büttner 에 의해 연구되어 발표되었습니다.[1] MotionMatching 기술은 애니메이션 클립을 블렌드 트리(Blend Tree)나 상태 기계(State Machine)에 배열할 필요 없이, 캐릭터의 각 상황에 맞는 애니메이션 클립을 대규모 애니메이션 데이터베이스에서 찾아 매칭한 후 시퀀스들 간의 블렌딩과 전환을 연속적으로 수행하여 보다 자연스럽고 유연한 애니메이션을 생성할 수 있습니다. 이 때문에 어플리케이션 실행 시간에서 주어진 상황이 계속적으로 바뀌는 게임 산업에서 많이 활용되고 있습니다.

본 보고서에서는 기존 MotionMatching 의 단점을 알아보고, 뉴럴 네트워크를 활용한 딥러닝 기술을 도입하여 발전된 Learned MotionMatching[2]을 타깃 논문으로 하여 언리얼 5 엔진에 구현하는 것을 목표로 과제를 수행했습니다. 또한 구현된 "Learned MotionMatching" 기술을 바탕으로 사실적인 애니메이션을 출력하면서 몰입도를 높인 "모션 액션 택배 배달" 게임 콘텐츠를 제작했습니다.

1. 서론

1.1. 연구배경

상호작용 가능한 애플리케이션, 특히 콘솔 게임이나 PC 게임 등에서는 점점 기술이 발전함에 따라 이용자는 AAA 게임과 같은 더 사실적이고 몰입감 있는 동적인 세계를 원하며, 그들의 요구에 맞추어 동적인 세계 안에서 시각적으로 보이는 그래픽 기술들이 어떻게 하면 사실적으로 표현될지에 관해서 많은 연구와 개발을 통해 발전하고 있습니다. 여기서 시각적으로 보이는 그래픽 기술은 단지 완성도가 높은 게임 배경이나 많은 양의 파티클을 요구하는 화려한 이펙트뿐만 아니라 캐릭터의 움직임이 얼마나 사실적인가도 포함합니다.

이미 많은 사람들이 사실적인 캐릭터 애니메이션에 관하여 고민해왔고, 다양한 기술이 고안되었으며, 그 중 최근에 가장 많이 쓰이는 기술이 "MotionMatching"입니다. 하지만 "MotionMatching" 기술에는 여러 단점이 존재합니다.

첫 번째 단점은 애니메이션 데이터의 규모입니다. 모션 매칭은 방대한 양의 사전 녹화된 애니메이션 데이터를 필요로 합니다. 데이터베이스에 저장되는 애니메이션 클립이 많아질수록 메모리 사용량도 증가합니다. 메모리 사용량은 데이터 양에 따라 선형적으로 증가하여, 데이터가 많아질수록 더 많은 메모리를 요구합니다. 이는 고품질 애니메이션을 위해 많은 데이터를 필요로 하는 현대의 AAA 게임에서 특히 문제가 될 수 있습니다.

두 번째 단점은 실시간 검색 비용이 많이 듭니다. 최근접 이웃 검색(nearest neighbor search)을 실시간으로 수행해야 하므로, 많은 애니메이션 데이터가 있을 경우 런타임 성능이 저하될 수 있습니다. 리얼타임이 중요한 게임 애플리케이션에서는 치명적일 수 있습니다.

세 번째 단점은 데이터베이스 구축 비용입니다. 대규모 애니메이션 데이터베이스를 구축하고 유지하는 데 많은 시간과 노력이 필요합니다. 또한 "MotionMatching"에서는 애니메이션 품질에 크게 의존합니다. 원본 데이터가 부자연스럽거나 부정확하면 결과 애니메이션도 품질이 떨어질 수 있습니다.

네 번째 단점은 일반 "MotionMatching"을 사용했을 때 캐릭터 간의 상호작용, 캐릭터와 오브젝트 사이의 복잡한 상호작용, 또는 환경과의 복잡한 상호작용을 처리하는 데 어려움이 있습니다. 이는 더 복잡한 알고리즘이나 추가적인 데이터 처리 기술을 필요로 할 수 있습니다.

이러한 단점들 때문에 모션 매칭은 특정 상황에서는 매우 유용하지만, 모든 상황에서 최적의 솔루션은 아닙니다. 따라서 신경망 기반의 생성 모델을 활용하여 고안된 "Learned MotionMatching" 기술을 언리얼5에서 구현해보자는 과제 목표를 정했으며, 구현된 기술로 이용자가 즐길 수 있는 콘텐츠를 만들고자 했습니다.

1.2. 연구 목표

본 연구는 기존 "MotionMatching" 기술에서 뉴럴 네트워크를 활용한 딥러닝을 도입하여 향상된 "LearnedMotionMatching"을 언리얼5에서 구현하고, 동적으로 애니메이션을 생성하여 게임 속에서 자연스러운 캐릭터 모션을 출력합니다.

1.3. 기대 효과

기존 "MotionMatching"의 긍정적인 측면과 상황에 맞는 사실적인 동적 애니메이션 출력을 유지하면서 애니메이션을 데이터에서 출력을 하는 것이 아닌 신경망 기반의 딥러닝을 통해 상황에 맞는 적합한 애니메이션이 출력됨으로 써 한정된 메모리 안에서 다양한 애니메이션을 실시간으로 생성하고 출력하여 메모리와 런타임 성능을 개선할 수 있습니다.

2. 배경 지식 및 관련 연구

"Learned Motion Matching"은 캐릭터의 상황에 맞는 애니메이션을 매칭하기 위해 특징 벡터 구조를 사용합니다. 이 특징 벡터의 각 요소는 캐릭터의 미래 궤적 위치를 포함하며, 이 미래 궤적 위치는 Spring-Damper 알고리즘을 통해 계산됩니다. Spring-Damper 미래 궤적 예측 기법으로 계산된 캐릭터의 특징 벡터는 모션 매칭 단계에서 파라미터 값으로 사용되며, 학습에 필요한 애니메이션 특징 벡터 데이터셋인 "FeatureDataSet"을 구성하는 과정에서도 활용됩니다. "Learned Motion Matching"에서는 학습에 필요한 "FeatureDataSet" 뿐만 아니라, 특징 데이터에 대응되는 애니메이션 데이터셋 "PoseDataSet"도 구성해야 합니다.

이 두 가지 특징 벡터 데이터셋을 구성한 후, 다음 세 가지 신경망 기술의 조합으로 상황에 적합한 애니메이션을 동적으로 생성합니다:

1. **Compressor & Decompressor**
2. **Stepper**
3. **Projector**

이 세 가지 과정을 통해 특징 매칭이 이루어지고 이후 애니메이션이 생성됩니다. 그리고 생성된 애니메이션들을 이전 애니메이션과 다음 애니메이션을 매끄럽게 이어주는 Blending 알고리즘으로 Inertialization Blending 기법이 사용됩니다.

2.1.1. Ubsoft "LAFAN1" Dataset & BVH file Construction

학습에 필요한 애니메이션 데이터셋으로 Ubisoft에서 제공하는 "LAFAN1[3]" 데이터셋을 사용하였습니다. 이 데이터셋은 모션 캡처 기술로 제작되었으며, 파일 형식은 ".BVH" 포맷으로 구성됩니다.

BVH (Biovision Hierarchy) 포맷은 모션 캡처 데이터를 저장하기 위한 파일 포맷으로, 텍스트 기반의 파일 형식입니다. BVH 포맷은 주로 두 부분으로 구성됩니다:

1. **계층 구조 (Hierarchy) 섹션:**
    - **ROOT:** 루트 노드는 뼈대의 최상위 부모 노드입니다. 캐릭터의 중심 (예: 골반)으로 간주됩니다.
    - **JOINT:** 각 조인트는 뼈대의 각 부분을 나타냅니다. 루트 노드 아래에 여러 조인트가 존재할 수 있으며, 계층 구조로 연결됩니다.
    - **OFFSET:** 각 조인트의 상대적 위치를 정의합니다. 이는 부모 조인트로부터의 상대적인 위치를 나타냅니다.
    - **CHANNELS:** 각 조인트의 모션 데이터를 설명합니다. 일반적으로 각 조인트는 3 개의 회전 축 (Xrotation, Yrotation, Zrotation)을 가집니다.

- **End Site:** 말단 조인트를 정의합니다. 더 이상 하위 조인트가 없는 조인트를 나타냅니다.

2. **모션 데이터 (Motion Data) 섹션:**
   - **Frames:** 총 프레임 수를 나타냅니다.
   - **Frame Time:** 각 프레임의 시간 간격을 나타냅니다.
   - **모션 데이터 값:** 각 프레임에 대한 모든 조인트의 채널 데이터를 포함합니다. 루트 노드부터 시작하여 계층 구조에 정의된 순서대로 각 조인트의 데이터를 나열합니다.

이와 같은 구조를 통해 BVH 포맷은 모션 캡처 데이터를 정확하게 저장하고 효과적으로 활용할 수 있습니다.

2.1.2. Feature & Pose Vector Construction

캐릭터의 특징 벡터와 애니메이션의 특징 벡터 데이터셋의 구조는 다음과 같습니다:

캐릭터의 특징 벡터 x:

- x = {t^t, t^d, f^t, f!^t, h!^t } ∈ R^27 으로 구성됩니다.
- t^t ∈ R^6 : Local 캐릭터로부터 지상에 투영된 미래 궤적 위치 벡터입니다. 각 요소는 애니메이션의 60Hz 기준으로 20, 40, 60 프레임의 정보를 가집니다. 이후 설명할 Spring-Damper 알고리즘 기법을 통해 구하게 됩니다.
- t^d ∈ R^6 : Local 캐릭터로부터 미래 궤적의 정면 방향 벡터입니다. 마찬가지로 각 요소는 60Hz 기준으로 20, 40, 60 프레임의 정보를 가집니다. 이후 설명할 Spring-Damper 알고리즘 기법을 통해 구하게 됩니다.
- f^t ∈ R^6 : Local 캐릭터로부터 두 발의 위치 벡터입니다.
- f!^t ∈ R^6 : Local 캐릭터로부터 두 발의 속도 벡터입니다.
- h^t ∈ R^3 : Local 캐릭터로부터 엉덩이의 속도 벡터입니다.

특징벡터 구조에서 주의해야 할 것이 있습니다. 특징 벡터는 각 요소의 크기가 다를 수 있기 때문에, 이를 정규화하는 것이 중요합니다. 프로젝트에서는 데이터셋 내 각 특징(예: 왼쪽 발 위치)을 표준 편차로 스케일링 합니다. 이후 스케일링은 검색에서의 중요성을 조정하기 위해 사용자 가중치를 통해 추가로 조정될 수 있습니다. 타겟 논문에서는 사용자 가중치를 1 로 두고 구현했습니다.

애니메이션의 특징 벡터 데이터셋 y:

- y = {y^t, y^r, y!^t, y!^r, r!^t, r!^r, o*} 으로 구성됩니다.
- y^t, y^r : 각 관절의 Local Translation 과 Rotations 입니다.
- y!^t, y!^r : 각 관절의 Local Translation 과 Rotational Velocity 입니다.
- r!^t, r!^r : 캐릭터 Root 의 Translation 과 Rotational Velocity 입니다.

- o* : 추가적인 정보를 담고 있으며, 예를 들어 발 접촉 정보, 같은 씬의 다른 캐릭터의 위치나 궤적, 캐릭터의 일부 관절의 미래 위치 등을 포함합니다. 이 정보는 다른 캐릭터나 사물, 환경과 상호작용을 고려하여 다양한 상황에 맞는 애니메이션을 생성하는 데 활용될 수 있습니다.

2.1.3. Spring-Damper (미래 궤적 예측 알고리즘)

Spring-Damper 로 미래 궤적을 예측하기 전에, 게임에서 자주 사용되는 보간 기법인 Damper 를 먼저 이해해야 합니다. 객체를 부드럽고 자연스럽게 이동시키기 위한 보간 방법으로, 이는 이동 뿐만 아니라 회전에도 적용될 수 있습니다. 두 점 A 와 B 가 있고, 각각의 좌표가 (x1, y1)과 (x2, y2)라면, 두 점 사이의 비율 t ($0 \le t \le 1$)에 대한 보간된 점 P 의 좌표 (x_p, y_p)는 다음과 같이 계산됩니다.

- $x\_p = x1 + t \times (x2 - x1)$
- $y\_p = y1 + t \times (y2 - y1)$

여기서 t는 보간 비율로, t=0이면 점 P는 점 A에 위치하고, t=1이면 점 P는 점 B에 위치하게 됩니다. t가 0과 1 사이의 값일 경우, 점 P는 A와 B 사이의 어느 한 지점에 위치하게 됩니다. 하지만 이 해결책에는 문제가 있습니다. 게임의 프레임 속도(또는 시스템의 시간 간격)를 변경하면 댐퍼의 동작이 달라집니다. 더 구체적으로 이야기하면, 프레임 속도가 낮을 때 객체는 더 느리게 이동합니다. 이 문제를 해결하기 위해 The Exact Damper[4]가 고안되었지만, 이 알고리즘에서는 목표 위치가 빠르게 변할 경우, 즉 이전 방향과는 상관없이 즉시 반대 방향으로 움직이는 경우 불연속성이 발생할 수 있습니다. 이는 시각적으로 급격한 움직임을 초래할 수 있습니다. 예를 들어, 물체가 한 방향으로 움직이고 있더라도 목표가 방향을 바꾸면 즉시 반대 방향으로 움직이게 됩니다. 이러한 여러 복합적인 문제를 해결해서 나온 선형 보간 알고리즘이 Spring-Damper입니다.

Exact Damper의 문제는 속도의 연속성이 없기 때문에 발생합니다. 즉, 이전 프레임에서 발생한 변화가 댐퍼의 동작에 전혀 반영되지 않고 항상 목표 방향으로 이동하려고 시도하기 때문에 발생하는 것입니다. 이를 해결하기 위해 Spring-Damper는 현재 속도에 부드럽게 영향을 미치는 방식으로 개선할 수 있습니다. 현재 속도에 목표 방향으로 향하는 속도를 스케일링된 Stiffness(가소성) 파라미터로 추가하는 방법이 제안됩니다. 또한 목표 속도를 나타내는 q 변수를 도입하여 현재 속도가 이 목표 속도로 점진적으로 접근할 수 있도록 Damping(감쇠) 파라미터로 조절하는 방법도 포함됩니다.

결국, Spring-Damper는 간단한 댐퍼의 움직임을 흉내내면서도 목표 방향으로의 부드러운 이동을 제공합니다. 이 방식은 두 개의 "힘"을 사용하여 해결될 수 있습니다 - 목표 위치로 당기는 힘과 목표 속도로 당기는 힘입니다. 이를 통해 이전의 댐퍼와 비교하여 속도의 연속성을 유지하면서도 부드러운

운동이 가능해집니다.

우리는 이 개념을 확장하여 미래 궤적을 예측할 수 있습니다. 예를 들어, 캐릭터의 현재 위치를 기준으로 하여, 조이스틱의 입력 값을 적용하여 목표 위치를 정할 수 있습니다. 그리고 기준 위치에서 목표 위치까지의 궤적을 감쇠와 보간 방식을 사용하여 구할 수 있습니다. 이는 이동뿐만 아니라 회전에서도 동일한 방법으로 적용되며, 프레임마다 특징 벡터의 미래 궤적 정보를 업데이트하는 데 사용됩니다.

2.1.4. Quaternion Rotation

애니메이션에서 사용하는 회전 방식은 크게 Euler angles 방식과 Quaternion 방식을 사용합니다. 두 방식에는 단점과 장점이 분명하여 필요한 상황에 따라 어떤 방식을 사용할지 결정해야 합니다. Quaternion 회전은 3 차원 공간에서 Euler angles 을 사용하는 것보다 효율적으로 회전을 표현하고 계산할 수 있는 방법입니다. Quaternion 은 하이퍼 복소수(hypercomplex number)로, 벡터와 스칼라 부분으로 구성됩니다. 일반적으로 q = w + xi + yj + zk 로 표현됩니다. 여기서 w, x, y, z 는 실수입니다.

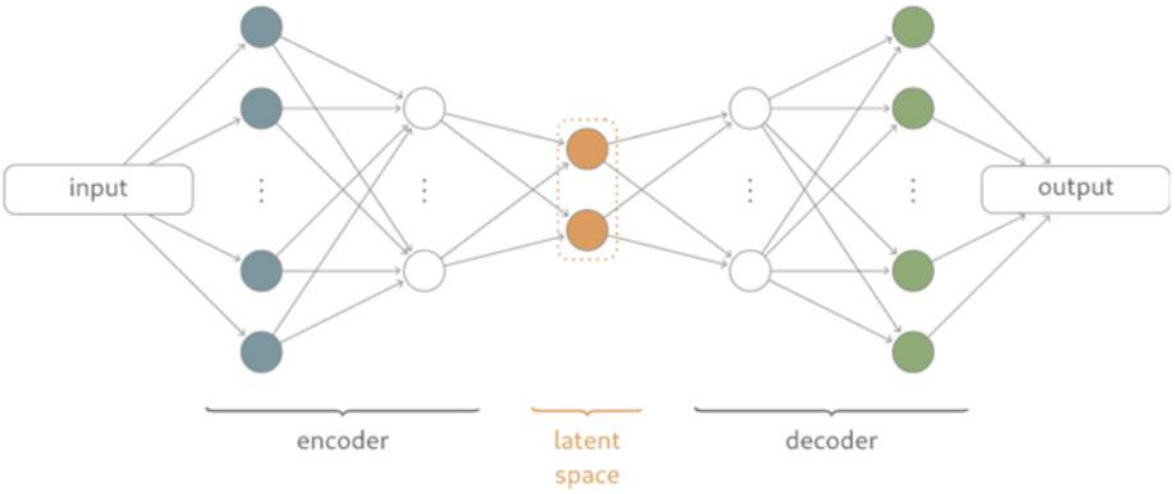
Quaternion 회전을 사용하면 효율적인 계산뿐만 아니라 Euler angles 의 치명적인 단점인 짐벌락 현상을 해결할 수 있습니다. 그렇기 때문에 Quaternion 방식은 공간에서의 정밀한 회전 제어가 필요한 많은 응용 분야에서 선호됩니다.

본 프로젝트에서 타겟 논문으로 진행된 "Learned Motion Matching"에서의 학습 단계에서 애니메이션 데이터셋을 뉴럴 모델에 학습시킬 때 Quaternion 방식을 사용하게 되고, 또한 뉴럴 네트워크를 거쳐 나온 출력 값 또한 Quaternion 회전 값으로 출력합니다. 우리는 이 쿼터니언으로 출력된 회전값을 언리얼 5 포멧에서 캐릭터에 적용할 때 Euler angles 각 변환을 이용해서 적용하였습니다.

2.1.5. Compressor & Decompressor

Decompressor는 원래의 자세 데이터 Y를 메모리에 저장하지 않고도, 특정 프레임 xi의 특징 벡터를 이용하여 해당 자세 yi를 효과적으로 생성하고 관리하는 기술입니다. 이를 가능하게 하기 위해 우리는 "Compressor"라고 불리는 네트워크를 사용하여 자세 yi를 저차원 잠재 공간 표현 zi로 매핑합니다. 이 저 차원 잠재 공간 표현 zi는 특징 벡터 xi와 연결되어 "Decompressor"에 입력으로 제공됩니다. “Decompressor”는 이 입력을 기반으로 원래의 자세 yi를 재구성합니다. 이 과정에서 우리는 특징 벡터 x에 부족한 추가 정보가 무엇인지를 학습하고, 이 정보를 잠재 변수 zi에 인코딩하여 보다 정확하고 완전한 자세 재구성을 목표로 합니다. 이 과정은 오토인코더와 유사한 구조로 이루어져 있습니다.

오토인코더(Autoencoder)란 입력 데이터를 최대한 압축한 후, 압축된 데이터를 원래의 입력 형태로 복원하는 신경망입니다. 이 과정에서 입력 데이터를 압축하는 부분을 Encoder라고 하고, 복원하는 부분을 Decoder라고 합니다. 압축 과정에서 추출한 의미 있는 데이터 Z는 보통 잠재 벡터(latent vector)라고 불리며, 이 외에도 '잠재 변수(latent variable)', '코드(code)', '특성(feature)' 등의 용어로 사용될 수 있습니다.



< Variational autoencoders. [7]>

“Decompressor”의 핵심 요소 중 하나는 사용하는 손실 함수입니다. 단순한 평균 제곱 오차 손실은 움직임이 흔들림이나 낮은 품질로 나타날 수 있기 때문에 이를 대체하기 위해 시각적으로 인식되는 오차를 최소화하는 손실 함수를 설계합니다. 이 손실 함수에는 Forward Kinematics 를 사용하여 캐릭터 공간에서의 오차를 측정하고, 자세가 시간에 따라 부드럽게 변하도록 하는 속도 기반 손실도 포함됩니다. 훈련 절차는 주어진 애니메이션 데이터베이스 Y 와 매칭 데이터베이스 X 로부터 두 개의 프레임을 사용하여 진행됩니다. “Compressor” C 를 사용하여 잠재 변수 Z 를 찾고, 이를 “Decompressor” D 를 사용하여 원래의 자세를 재구성합니다. 각 샘플링(두 프레임의 쌍)에 대해 이 절차를 설명하며, 미니 배치의 각 요소에 적용하여 네트워크 매개 변수 θC 와 θD 를 업데이트할 때 결과를 평균화합니다. 연산자

⊖는 두 자세 사이의 차이를 계산하는 데 사용되며, 회전 차이에 대해서는 회전 매트릭스를 사용하여 계산합니다. 속도 손실에서는 자세의 속도 차이를 계산하지 않습니다.

가중치 w*는 모든 자세 기반 손실에 대해 거의 동일한 가중치를 부여하고, 정규화 손실에는 작은 가중치를 부여하여 설정됩니다. "Compressor"에게 로컬 및 캐릭터 공간 입력을 모두 제공함으로써 정확도를 높일 수 있었으며, 이는 "Compressor"가 특징이 유용하다고 판단하면 직접적으로 잠재 공간으로 복사할 수 있음을 의미합니다. 훈련이 완료된 후, "Decompressor"는 다른 네트워크 없이도 이미 유용하게 사용될 수 있습니다. z 의 차원을 작게 만들어 각 프레임 i 에 대해 잠재 변수 zi 를 계산하고, Y 대신 Z = [z_0, z_1, ..., $z_{n^{-1}}$]를 저장함으로써 중요한 메모리 절약을 달성할 수 있습니다. 이는 Motion Matching 알고리즘의 동작에는 전혀 영향을 미치지 않습니다.

최종적으로 "Decompressor"의 활용 목표는 PoseDataSet 인 Y 를 메모리에 저장할 필요성을 제거합니다.

본 구현 과정에서 참고한 Compressor&Decompressor 에 대한 pseudocode 는 다음과 같습니다.

Algorithm 1: Our training algorithm for Decompressor $\mathcal{D}$.

Function TrainDecompressor($\mathbf{X}, \mathbf{Y}, \theta_C, \theta_{\mathcal{D}}$):

/* Compute forward kinematics */
$\mathbf{Q} \leftarrow \text{ForwardKinematics}(\mathbf{Y})$
/* Generate latent variables Z */
$\mathbf{Z} \leftarrow C([\mathbf{Y}\ \mathbf{Q}]^T; \theta_C)$
/* Reconstruct pose $\tilde{\mathbf{Y}}$ */
$\tilde{\mathbf{Y}} \leftarrow \mathcal{D}([\mathbf{X}\ \mathbf{Z}]^T; \theta_{\mathcal{D}})$
/* Recompute forward kinematics */
$\tilde{\mathbf{Q}} \leftarrow \text{ForwardKinematics}(\tilde{\mathbf{Y}})$
/* Compute latent regularization losses */
$\mathcal{L}_{lreg} \leftarrow w_{lreg} \| \mathbf{Z} \|_2^2$
$\mathcal{L}_{sreg} \leftarrow w_{sreg} \| \mathbf{Z} \|_1$
$\mathcal{L}_{vreg} \leftarrow w_{vreg} \left\| \frac{\mathbf{Z}_0 - \mathbf{Z}_1}{\delta t} \right\|_1$
/* Local & character space losses */
$\mathcal{L}_{loc} \leftarrow w_{loc} \| \mathbf{Y} \ominus \tilde{\mathbf{Y}} \|_1$
$\mathcal{L}_{chr} \leftarrow w_{chr} \| \mathbf{Q} \ominus \tilde{\mathbf{Q}} \|_1$
/* Local & character space velocity losses */
$\mathcal{L}_{lvel} \leftarrow w_{lvel} \left\| \frac{\mathbf{Y}_0 \ominus \mathbf{Y}_1}{\delta t} - \frac{\tilde{\mathbf{Y}}_0 \ominus \tilde{\mathbf{Y}}_1}{\delta t} \right\|_1$
$\mathcal{L}_{cvel} \leftarrow w_{cvel} \left\| \frac{\mathbf{Q}_0 \ominus \mathbf{Q}_1}{\delta t} - \frac{\tilde{\mathbf{Q}}_0 \ominus \tilde{\mathbf{Q}}_1}{\delta t} \right\|_1$
/* Update network parameters */
$\theta_C\ \theta_{\mathcal{D}} \leftarrow \text{RAdam}(\theta_C\ \theta_{\mathcal{D}}, \nabla \sum_* \mathcal{L}_*)$

end

< DANIEL HOLDEN, Ubisoft La Forge, Ubisoft, Learned Motion Matching[5] >

2.1.6. Stepper

초기에 매칭된 잠재 특성 벡터 x_i, z_i 가 주어진 상황에서 우리는 다음 프레임의 애니메이션을 정보를 얻기 위해 연속된 애니메이션 벡터에 인덱스 값을 증가시켜서 검색해야 합니다. "Learned MotionMatching"이 아닌 기존 "MotionMatching"에서는 이과정이 어렵지 않습니다. 단순히 인덱스 "i"를 증가시키고 X 또는 Z 의 새로운 행을 참조하면 됩니다. 하지만 "Learned MotionMatching"에서는 X 와 Z 를 메모리에 저장하지 않고 다음 애니메이션 행을 찾는 것이 목적이기 때문에 이 과정은 비교적 단순 하지 않습니다. 우리는 이 과정을 Stepper 라는 신경망 기반 학습 모델을 통해 진행하려고 합니다. 이는 특정 프레임 x_i, z_i 에서 입력으로 받아들이고, 다음 프레임에서의 특성 벡터 x_i+1, z_i+1 을 생성할 수 있는 델타를 출력하는 네트워크입니다.

여기서 우리는 자기 회귀적 방식으로 네트워크를 훈련시키며, 길이 S 의 짧은 창 내에서 특성 벡터 X 와 잠재 변수 Z 를 반복적으로 예측하여 다음 프레임에서 그들을 피드합니다. 훈련 샘플 하나에 대해 절차를 설명하였지만, 미니배치의 각 요소에 적용하고 θS 를 업데이트할 때 결과를 평균화합니다. 가중치 w∗ 는 모든 손실에 거의 동일한 가중치를 부여하기 위해 설정됩니다.
훈련을 마치면, Stepper 는 파이프라인의 스텝핑 부분을 대체하는 데 사용될 수 있으며, X 나 Z 에 의존하지 않고 일치 및 잠재 특성 벡터의 스트림을 생성할 수 있습니다. 압축기를 사용하여 초기 시작 상태를 찾을 수 있지만, 가장 가까운 이웃 검색에는 X 와 Z 를 메모리에 유지해야 합니다.
본 구현 과정에서 참고한 Stepper 에 대한 pseudocode 는 다음과 같습니다.

Algorithm 2: Our training algorithm for Stepper $\mathcal{S}$.

Function TrainStepper($\mathbf{X}, \mathbf{Z}, s, \theta_{\mathcal{S}}$):
  /* Set initial states */
  $\tilde{\mathbf{X}}_0, \tilde{\mathbf{Z}}_0 \leftarrow \mathbf{X}_0, \mathbf{Z}_0$
  /* Predict $\tilde{\mathbf{X}}$ and $\tilde{\mathbf{Z}}$ over a window of s frames */
  for $i \leftarrow 1$ **to** s **do**
    /* Predict deltas for $\tilde{\mathbf{X}}$ and $\tilde{\mathbf{Z}}$ */
    $\delta\tilde{\mathbf{x}}, \delta\tilde{\mathbf{z}} \leftarrow \mathcal{S}([\tilde{\mathbf{X}}_{i-1}\ \tilde{\mathbf{Z}}_{i-1}]^T; \theta_{\mathcal{S}})$
    $\tilde{\mathbf{X}}_i \leftarrow \tilde{\mathbf{X}}_{i-1} + \delta\tilde{\mathbf{x}}$
    $\tilde{\mathbf{Z}}_i \leftarrow \tilde{\mathbf{Z}}_{i-1} + \delta\tilde{\mathbf{z}}$
  end
  /* Compute losses */
  $\mathcal{L}_{xval} \leftarrow w_{xval} \left\| \mathbf{X} - \tilde{\mathbf{X}} \right\|_1$
  $\mathcal{L}_{zval} \leftarrow w_{zval} \left\| \mathbf{Z} - \tilde{\mathbf{Z}} \right\|_1$
  $\mathcal{L}_{xvel} \leftarrow w_{xvel} \left\| \frac{\mathbf{X}_{0\to s-1} - \mathbf{X}_{1\to s}}{\delta t} - \frac{\tilde{\mathbf{X}}_{0\to s-1} - \tilde{\mathbf{X}}_{1\to s}}{\delta t} \right\|_1$
  $\mathcal{L}_{zvel} \leftarrow w_{zvel} \left\| \frac{\mathbf{Z}_{0\to s-1} - \mathbf{Z}_{1\to s}}{\delta t} - \frac{\tilde{\mathbf{Z}}_{0\to s-1} - \tilde{\mathbf{Z}}_{1\to s}}{\delta t} \right\|_1$
  /* Update network parameters */
  $\theta_{\mathcal{S}} \leftarrow \text{RAdam}(\theta_{\mathcal{S}}, \nabla \sum_* \mathcal{L}_*)$
end

< DANIEL HOLDEN, Ubisoft La Forge, Ubisoft, Learned Motion Matching[5] >

2.1.7. Projector

위 챕터에서의 Stepper 라는 신경망 모델로 시간에 따라 특징 벡터를 진행시킬 수 있었습니다. 하지만 Stepper 만을 이용해서 특징데이터셋인 X 에 대해서 메모리를 독립시킬 수 없었습니다. 왜냐하면 쿼리 벡터 x^에 수행되는 매칭 검색 과정에서는 여전히 X 벡터가 사용되야 하기 때문입니다. 이것을 해결하기 위해 타겟 논문에서는 Projector 라는 신경망 모델을 사용합니다. Projector 의 목표에 대하여 설명하면 프로젝터(Projector)는 주어진 입력 벡터에 대해 매칭 데이터베이스에서 가장 가까운 이웃을 찾아 그에 해당하는 특징 벡터와 잠재 변수를 출력하는 네트워크입니다. 이 네트워크는 일반적으로 최근접 이웃 검색을 대체하고, 학습된 데이터를 기반으로 쿼리 벡터와 가장 유사한 벡터를 찾아주는 기능을 하게 됩니다.

과정은 이렇습니다. 매칭 데이터베이스의 특징 벡터 x 를 제공하면, 우리는 노이즈 크기 nσ 를 샘플링하여 이를 가우스 노이즈 벡터 n 에 스케일링을 합니다. 이를 x 에 추가하여 xˆ을 생성하고, 가장 가까운 이웃 k∗를 찾습니다. 그런 다음 프로젝터는 해당 특징 벡터와 잠재 변수 xk∗ 및 zk∗를 출력하도록 훈련됩니다 훈련 샘플에 대해 여기서는 단일 절차를 제시했지만, 미니배치의 각 요소에 적용하고 θP 를 업데이트할 때 결과를 평균화합니다. 다양한 노이즈 크기를 샘플링함으로써 프로젝터를 다양한 크기의 교란에 강화시킵니다. 가중치 w∗는 모든 손실에 대해 대략적으로 동일한 가중치를 제공하도록 설정됩니다. .

최종적으로 Projector 가 훈련되면, 학습된 모션 매칭 파이프라인이 완성됩니다. 최근접 이웃 검색 대신, 매 N 프레임마다 사용자 쿼리 xˆ를 프로젝터 P 를 통해 전달합니다. 그런 다음 각 프레임마다, “Stepper” S 를 사용하여 찾은 매칭 및 잠재 특징 벡터를 전진시키고, 디코딩하여 “Decompressor” D 를 사용하여 포즈를 생성합니다.

이제 Project 를 사용함으로써 메모리에 X 와 Z 를 저장할 필요가 없도록 구성할 수 있습니다.

본 구현 과정에서 참고한 Projector 에 대한 pseudocode 는 다음과 같습니다.

Algorithm 3: Our training algorithm for Projector $\mathcal{P}$.

Function TrainProjector($\mathbf{x}, \mathbf{X}, \mathbf{Z}, \theta_{\mathcal{P}}$):

/* Sample uniform noise magnitude n^{σ} */

$n^{\sigma} \sim \mathcal{U}(0, 1)$

/* Sample gaussian noise vector $\mathbf{n}$ */

$\mathbf{n} \sim \mathcal{N}(\mathbf{0}, \mathbf{1})$

/* Add noise to feature vector */

$\hat{\mathbf{x}} \leftarrow \mathbf{x} + n^{\sigma}\mathbf{n}$

/* Find nearest neighbor */

$k^{*} = \text{Nearest}(\hat{\mathbf{x}}, \mathbf{X})$

/* Project feature vector */

$\tilde{\mathbf{x}}, \tilde{\mathbf{z}} \leftarrow \mathcal{P}(\hat{\mathbf{x}}; \theta_{\mathcal{P}})$

/* Compute losses */

$\mathcal{L}_{xval} \leftarrow w_{xval} \parallel \mathbf{x}_{k^{*}} - \tilde{\mathbf{x}} \parallel_1$

$\mathcal{L}_{zval} \leftarrow w_{zval} \parallel \mathbf{z}_{k^{*}} - \tilde{\mathbf{z}} \parallel_1$

$\mathcal{L}_{dist} \leftarrow w_{dist} \left\| \parallel \hat{\mathbf{x}} - \mathbf{x}_{k^{*}} \parallel_2^2 - \parallel \hat{\mathbf{x}} - \tilde{\mathbf{x}} \parallel_2^2 \right\|_1$

/* Update network parameters */

$\theta_{\mathcal{P}} \leftarrow \text{RAdam}(\theta_{\mathcal{P}}, \nabla \sum_{*} \mathcal{L}_{*})$

end

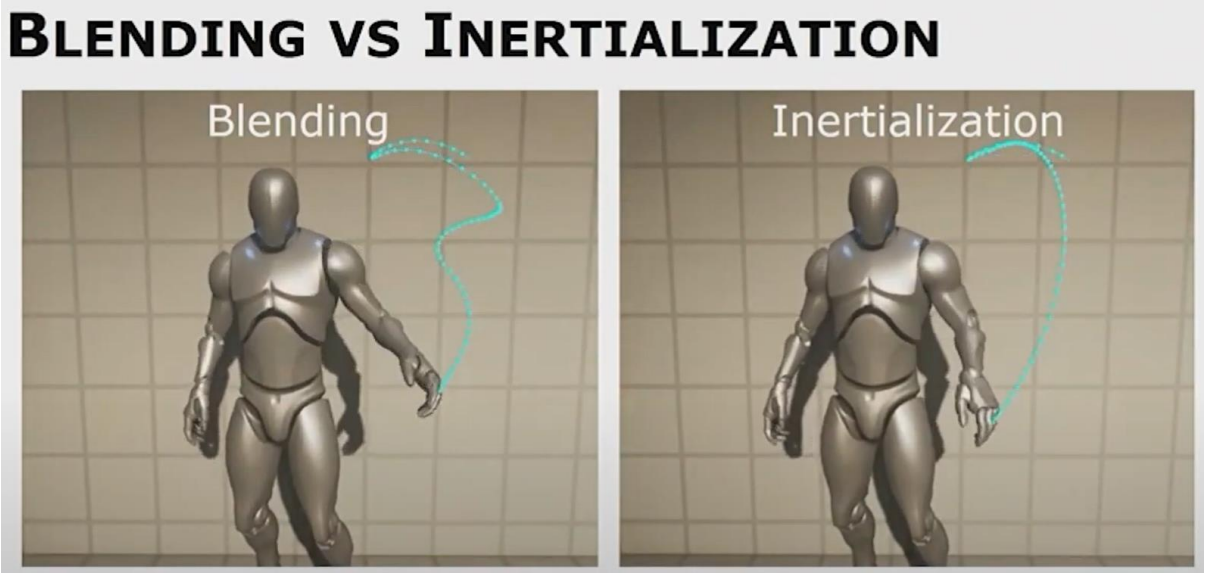
< DANIEL HOLDEN, Ubisoft La Forge, Ubisoft, Learned Motion Matching[5] >

2.1.8. inertialization

Inertialization[6] 이란 David Bollo 가 2016 년 GDC 2018 에서 'Inertialization: High-Performance Animation Transitions in Gears'라는 주제로 발표한 캐릭터 애니메이션 Blending 기법입니다. Blending 기법이란 기본적으로 두 가지 다른 애니메이션 상태 사이의 자연스러운 전환을 가능하게 합니다. 예를 들어, 캐릭터가 걷는 상태에서 달리는 상태로 전환될 때, 이 과정이 자연스럽고 부드럽게 이루어지게 됩니다. 이는 게임에서 플레이어 경험을 향상시키는 데 중요한 역할을 합니다.

기존의 Blend 기법은 보통 애니메이션 상태 사이의 선형 보간(Linear Interpolation)을 사용하여 전환합니다. 이는 간단하고 빠르지만, 전환 과정에서 갑작스러운 변화가 발생하여 플레이어 경험을 낮추기도 합니다. 하지만 Bavid Bollo 가 고안한 'Inertialization Blending 기법은 물리적 관성을 모델링하여 애니메이션 전환을 부드럽게 만듭니다. 예를 들어, 캐릭터가 움직임을 멈출 때도 일정한 관성을 고려하여 부드럽게 변화하도록 만듭니다. 이는 보다 자연스러운 애니메이션 전환을 가능하게 합니다. 또한 이 기술의 핵심 장점 중 하나는 높은 성능을 제공한다는 점입니다. 게임에서 매우 많은 양의 애니메이션 전환이 발생하여 성능 문제를 일으킬 수 있는데, 'Inertialization' 기술은 이를 최대한

효율적으로 처리하는 알고리즘으로 게임의 부드러운 플레이를 유지할 수 있습니다.



< David Bollo. 2016. Inertialization: High-Performance Animation Transitions in 'Gears of War'. In Proc. of GDC 2018 [6] >

위 참조 그림과 같이 캐릭터 팔이 위에서 아래로 내려올 때 기존 Blending 방식과 Inertialization 은 궤적 차이를 보입니다. Inertialization Blending 기법이 조금 더 자연스럽게 팔이 떨어지는 것을 볼 수 있습니다.

2.1.9. 기존 MotionMatching 문제점 및 LearnedMotionMatching 에서의 해결 방안

기존의 Motion Matching 알고리즘은 몇 가지 주요 문제점을 가지고 있습니다. 이러한 문제점들을 해결하기 위해 Learned Motion Matching 알고리즘에서 어떤 접근 방식을 취하는지에 대해 알아보겠습니다.

1. 메모리 사용량 : 기존 Motion Matching 은 매칭 데이터베이스 X 와 애니메이션 데이터베이스 Y 를 메모리에 유지해야 하며, 이는 많은 애니메이션, 포즈 특성 및 매칭 특성이 추가될수록 메모리 오버헤드를 초래할 수 있습니다.

해결 방법 : Learned Motion Matching 에서는 뉴럴 네트워크 "Decompressor" "Stepper" "Projector"를 이용하여 X 와 Y 에 대한 의존성을 제거하고, 대신 특성 벡터와 잠재 변수만을 사용하여 알고리즘을

구현합니다. 즉, 데이터베이스를 메모리에 저장할 필요 없이 네트워크를 통해 필요한 정보를 생성합니다.

2. 실시간 처리와 성능: 기존의 경우, 매 N 프레임마다 매칭과 포즈 생성을 수행해야 하며, 이는 실시간 애니메이션 시스템에서 성능 문제를 초래할 수 있습니다.

해결 방법: Learned Motion Matching 은 학습된 네트워크를 사용하여 쿼리 벡터를 처리하고, 최적의 매치를 찾고, 시간에 따라 데이터를 전진시키며, 최종적으로 캐릭터 포즈를 생성합니다. 이러한 접근 방식은 기존의 비교적 직접적인 방법보다 효율적이며, 성능을 향상시킬 수 있습니다.

3. **유연성과 확장성**: 기존 Motion Matching 은 새로운 애니메이션을 추가하거나 기존 애니메이션을 수정할 때 시간과 비용이 많이 들 수 있습니다. 또한, 다양한 환경이나 캐릭터 특성에 적응하기 어려울 수 있습니다.

해결 방법: Learned Motion Matching 은 데이터베이스의 의존성을 줄이고, 대신 학습된 모델을 사용하여 특성 벡터와 잠재 변수를 생성하므로, 새로운 애니메이션을 추가하거나 환경을 변화시키는 데 유연하고 확장 가능합니다. 또한, 학습된 모델을 업데이트하거나 조정함으로써 다양한 요구사항에 쉽게 적응할 수 있습니다.

---

3. 추진 내용

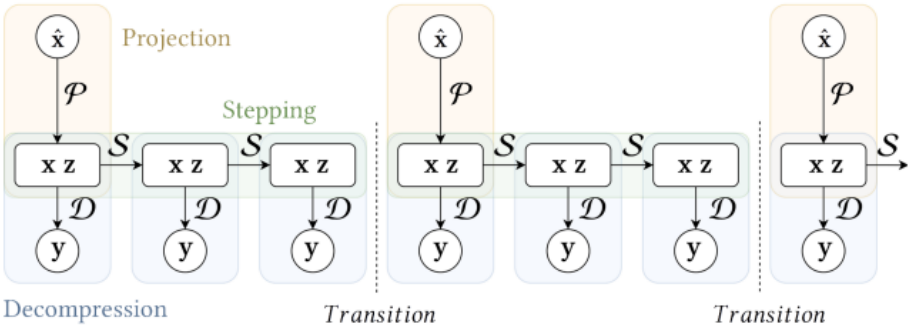
3.1. 팀 구성 및 역할

이름	역할	수행업무
이재형	팀장	Motion matching 알고리즘 구현, 게임 로직 구현, 일정 및 리소스 관리
최진욱	팀원	Motion matching 알고리즘 구현, 게임 로직 구현, 일정 및 리소스 관리

3.2. 전체 시스템 구성

이 프로젝트는 모션 캡처 데이터인 BVH 파일에서 추출한 애니메이션 데이터를 기반으로 한 데이터셋 구축과, PyTorch 를 이용하여 구현한 Projector, Stepper, Decompressor 네트워크를 통해 언리얼 엔진에서의 모션매칭 시스템을 개발하는 것을 주요 목표로 합니다. 데이터셋 구축 단계에서는 BVH 파일의 관절 위치와 회전을 처리하고, 좌우 반전과 같은 데이터 증강 기법을 적용하여 다양한 pose 데이터를 생성합니다. 이 데이터는 60fps 로 변환되고, 보간을 통해 부드럽게 처리됩니다.

Decompressor 는 특정 프레임의 feature vector 를 입력으로 받아 자세를 생성하며, Stepper 는 현재 프레임의 feature vector 와 latent 변수를 이용하여 다음 프레임을 예측합니다. Projector 는 주어진 쿼리 벡터로부터 가장 적합한 매칭 피처 벡터를 생성하여, 캐릭터의 자연스러운 걷기 혹은 달리기 모션을 얻습니다. 메인 시스템은 언리얼 엔진의 좌표계와 BVH 데이터의 호환성을 유지하기 위해 좌표계를 통합하고, 사용자의 입력 처리를 담당하며 이를 기반으로 모션 매칭을 실행하여 자연스러운 애니메이션을 출력합니다.



< DANIEL HOLDEN, Ubisoft La Forge, Ubisoft, Learned Motion Matching[5] >

3.3. 핵심 기능 개발

3.3.1 데이터셋 구축 프로그램 구현

BVH (Biovision Hierarchy) 파일 형식으로 된 모션 캡처 데이터를 처리하여 애니메이션 데이터베이스를 생성하는 작업을 수행합니다. BVH 파일은 인체의 관절 위치와 회전을 프레임 단위로 저장하는데, 이를 불러와 다양한 처리를 한 후 바이너리 파일로 저장합니다.

우선, Data augmentation 을 위해서 주어진 애니메이션 데이터를 좌우 반전합니다. 각 관절의 위치와 회전을 미러링하여 반환하고 이를 기존 데이터에 추가합니다.

처리할 BVH 파일 목록을 다음과 같이 정의하고, 데이터베이스에 필요한 리스트들을 초기화합니다.

bone_positions **bone_velocities** **bone_rotations** **bone_angular_velocities,**	**bone_parents** **bone_names** **range_starts** **range_stops** **contact_states**

이후 데이터를 60fps 로 변환하고, cubic 보간을 통해 데이터를 부드럽게 만듭니다. 또한 Simulation Bone 정보를 추출하여 저장합니다. 최종적으로 처리된 모든 데이터를 frame 단위로 구분되는 하나의 배열로 결합한 후 바이너리 파일로 저장합니다.

3.3.2 Projector, stepper, decompressor 학습 코드 구현

Projector, stepper, decompressor는 pytorch를 이용해 구현했습니다.

(Decompressor)

Decompressor의 주된 목표는 특정 프레임의 feature vector를 받아서 대응하는 자세를 직접 생성하는 것입니다. 피처 벡터는 일반적으로 대응하는 자세에 대한 중요한 정보를 포함하지만, 완전히 재구성하기에는 정보가 부족할 수 있습니다. 이를 보완하기 위해 추가적인 잠재 변수를 도입합니다. 이러한 구조는 Autoencoder와 유사합니다.

Compressor 네트워크를 사용하여 pose 정보 y^i를 저차원 표현 z^i로 변환합니다. 이 z^i를 x^i와 연결한 후, Decompressor에 입력으로 제공하여 원래의 자세 y^i를 복원합니다. 이를 통해 피처 벡터 x에서 누락된 추가 정보를 z에 인코딩하는 방법을 학습합니다.

단순한 평균 제곱 오차(Mean Squared Error, MSE) 손실을 사용하면 떨림이 있는 모션이 발생할 수 있습니다. 대신, 시각적으로 인식되는 오류를 최소화하기 위해 다음과 같은 Forward Kinematics를 사용하여 손실함수를 설계합니다.

(Stepper)

Stepper 네트워크는 현재 프레임의 피처 벡터와 잠재 변수에서 다음 프레임의 피처 벡터와 잠재 변수를 예측하는 역할을 합니다. 이 네트워크는 주어진 프레임의 피처 벡터 x^i와 잠재 변수 z^i를 입력으로 받아서, 다음 프레임의 피처 벡터 x^{i+1}와 잠재 변수 z^{i+1}를 생성합니다. 이를 위해 현재 프레임의 피처 벡터와 잠재 변수에 델타 값을 더하는 방식으로 다음 프레임의 벡터를 예측합니다. 짧은 길이의 피처 벡터 시퀀스 X와 잠재 변수 Z를 사용하여, 네트워크가 다음 피처 벡터와 잠재 변수를 반복적으로 예측하고 이를 다음 프레임의 입력으로 사용합니다. 손실 함수는 여러 요소를 고려하여 구성되며, 각 손실 항목에 대해 균등한 가중치를 줍니다. 미니 배치 내의 모든 샘플에 대해 적용하여 평균 결과를 통해 네트워크 매개변수 θ_s를 업데이트합니다.

(Projector)

Projector는 가장 가까운 이웃 검색 기능을 emulate하여, query vector $\hat{x}$와 가장 가까운 항목 x_k^*을 X에서 찾는 대신, 직접 매칭되는 피처 벡터와 잠재 변수를 생성합니다.

매칭 데이터베이스의 피처 벡터 x를 주어진 노이즈 크기 n_σ로 스케일된 가우시안 노이즈 벡터 n을 추가하여 쿼리 벡터 $\hat{x}$를 생성합니다. 이 쿼리 벡터 $\hat{x}$로부터 최근접 이웃 k^*를 찾습니다. Projector 네트워크는 x_k^*와 z_k^*를 출력하도록 훈련됩니다. 미니 배치 내의 모든 샘플에 대해 적용하고 평균 결과를 통해 네트워크 매개변수 θ_p를 업데이트합니다. 다양한 노이즈 크기를 샘플링하여, Projector가 다양한 크

기의 변형에 대해 견고하도록 만듭니다. 손실 항목에 대해 균등한 가중치를 부여합니다.

최종적으로, Decompresso, Stepper, Projector를 각각 학습시키고 통합합니다. 이를 통해 매칭 데이터베이스에 의존하지 않고도 기능을 수행할 수 있습니다. 구체적인 절차는 다음과 같습니다:

사용자 쿼리 처리:

매 프레임마다 사용자 쿼리 $\hat{x}$를 Projector에 전달하여, 가장 가까운 피처 벡터 x_k^*와 잠재 변수 z_k^*를 찾습니다.

Stepper와 Decompressor 사용:

각 프레임마다 Stepper를 사용하여 매칭 피처 벡터와 잠재 변수를 갱신합니다.

갱신된 피처 벡터와 잠재 변수를 사용하여 Decompressor를 통해 포즈를 생성합니다.

3.3.3 Unreal Engine에서 구동되는 모션매칭 시스템 구현

학습시킨 Projector, stepper, decompressor와 사용자 입력을 기반으로 게임 캐릭터의 애니메이션을 출력하는 모션매칭 시스템을 언리얼 엔진에서 구현했습니다. 핵심 기능들을 간략하게 설명합니다.

desired_gait_update(): 이 함수는 현재 캐릭터가 걷기에서 뛰기로 전환할 필요성이 있는지를 결정합니다. 플레이어가 게임패드를 누르면 걷기에서 뛰기로 전환할 수 있습니다.

desired_velocity_update(): 게임패드의 입력 및 카메라 방향 등을 고려하여 캐릭터의 원하는 속도와 방향을 계산합니다. 이는 캐릭터가 어디로 이동하고 있는지를 나타내며, 예를 들어 게임패드의 스틱 입력에 따라 전방, 후방, 좌우로의 이동을 결정합니다.

desired_rotation_update(): 캐릭터의 원하는 회전과 방향을 계산합니다. 예를 들어, 플레이어가 캐릭터를 특정 방향으로 회전시키는 입력을 하면, 이에 맞춰 캐릭터가 회전하도록 설정됩니다.

trajectory_desired_rotations_predict(): 이 함수는 캐릭터의 예상 회전을 계산합니다. 현재 입력 및 이전 회전 상태를 고려하여 다음에 캐릭터가 어떻게 회전할지 예측합니다.

trajectory_rotations_predict(): 이 함수는 캐릭터의 회전을 예측하고, 이전 회전 상태와 원하는 회전 상태를 보간하여 자연스러운 애니메이션 전환을 유도합니다.

database_search(): 데이터베이스에서 가장 적합한 애니메이션 프레임을 검색합니다. 현재 상태에 가장 적합한 걷기 애니메이션 또는 뛰기 애니메이션 프레임을 찾습니다.

projector_evaluate(): 이 함수는 현재 애니메이션 상태와 새로운 프로젝션 사이의 평가를 수행하여, 새로운 애니메이션 프레임을 결정합니다. 새로운 애니메이션 프레임이 이전 프레임과 충분히 다르면, 캐릭터의 움직임을 업데이트합니다.

Transition: 현재 프레임과 예상 프레임(프로젝션) 사이의 차이를 평가하여, 적절한 애니메이션 전환을 결정하는 조건입니다.

decompressor_evaluate: 예상 프레임에 대한 자세를 평가하여 필요한 모든 캐릭터의 뼈 위치 및 회전을 계산합니다.

Features_curr 및 **Latent_curr**: 현재 캐릭터의 특징 및 숨겨진 변수를 업데이트하여 다음 프레임의 평가에 사용됩니다.

inertialize_pose_transition(): 이 함수는 애니메이션 프레임 간의 자연스러운 전환을 담당합니다. 이전 애니메이션 상태에서 새로운 애니메이션 상태로의 움직임을 부드럽게 전환하여 캐릭터의 움직임이 자연스럽게 보이도록 합니다.

stepper_evaluate(): 이 함수는 애니메이션 스텝(보행 주기)을 평가하고, 다음 스텝으로 전환합니다.

3.3.4 게임 구현

모션 매칭을 이용해서 게임패드로 입력을 받고, 매우 자연스러운 걷기와 달리기 모션을 출력합니다. 이 시스템을 이용하여 캐릭터가 주어진 목적지로 물건을 배달하는 간단한 게임을 제작하였습니다. 이를 통해 모션 매칭 시스템이 원활하게 작동하는 것을 시각적으로 확인할 수 있습니다.

3.4 이슈 및 대응

3.4.1 좌표계 변환

애니메이션 data인 BVH에서 애니메이션 정보를 추출할 때 공간 좌표계를 오른손 좌표계(Yup, Zforward)로 세팅하였습니다. 그러나 Unreal engine은 공간 좌표계가 왼손 좌표계(Zup, Xforward)로 설정이 되어있습니다. 때문에 이 둘 간의 호환을 위해 좌표계를 변환하는 작업을 진행했습니다.

3.4.2 캐릭터 밀림 현상 방지(simulation object와 character mesh의 연계 문제)

기본적으로 시뮬레이션 객체는 코드에 의해 이동하고, 캐릭터 엔티티(캐릭터 Mesh)는 애니메이션 데이터에 따라 움직입니다. 이 두 이동 방식 간의 차이를 최소화하기 위해 다양한 방법을 활용해야 합니다. 다음 링크의 글을 참고하여 문제를 해결하였습니다.

< https://theorangeduck.com/page/code-vs-data-driven-displacement>

4. 결과

아래 유튜브 링크에서 결과물을 확인할 수 있습니다.

<https://youtu.be/t7G776BkS1k?si=Q6glNqOB58JoqDHl>

현재 보여지는 결과물은 ‘걷기’와 ‘달리기’ 모션을 학습하여 얻은 것입니다. 학습은 15 시간 정도 소요되었습니다.

Loss 는 다음과 같습니다.

decompressor -> loss: 1.5141

stepper -> loss: 1.020

projector -> loss: 0.730

사용자의 입력에 적절히 반응하여 매우 자연스러운 애니메이션을 출력하는 것을 확인할 수 있습니다.

5. 결론

본 프로젝트에서는 기존 Motion Matching 알고리즘의 단점을 세 가지 뉴럴 네트워크 모델을 통해 개선한 Learned Motion Matching 을 타겟 논문으로 하여 알고리즘을 언리얼 5 엔진에서 구현하는 과정을 거쳤습니다. 타겟 논문에서는 기존 Motion Matching 알고리즘의 메모리 효율성과 런타임 성능을 향상시키는 것을 목표로 하였으며 논문에서 수치적으로 이러한 개선점을 입증했습니다.

본 프로젝트에서 타겟 논문의 알고리즘을 언리얼 엔진 5 에 구현한 결과, 런타임에서 대체로 자연스러운 애니메이션을 생성하였고, 두 프레임 간의 시간차 성능 또한 0.08 로 유저 입장에서 전혀 이질감 없는 수치를 보였습니다.

그러나 타겟 논문에서 구현한 Motion Matching 알고리즘은 캐릭터의 이동을 3 차원이 아닌 2 차원으로 가정하고 있습니다. 따라서 캐릭터의 이동 궤적을 계산할 때 높낮이(Z 축)을 고려하지 않고, XY 평면에서의 움직임만을 이용합니다. 이로 인해 모션을 학습할 때 '걷기'와 '뛰기' 모션만을 학습하게 됩니다. 캐릭터가 위아래로 움직이는 특징은 고려하지 않고 있기 때문입니다. 이러한 한계를 극복하기 위해, 프로젝트 이후 우리는 Z 축 움직임을 고려한 특징을 추가하고, 점프 모션을 추가로 학습하도록 구현하려고 합니다. 특징이 늘어남에 따라 연쇄적으로 수정해야 할 부분이 많아질 것으로 예상됩니다. 현재로서는 이 부분이 가장 중요한 과제이며, 이를 완료한 후 다음 단계로 넘어갈 예정입니다.

프로젝트에서는 타겟 논문의 "Learned Motion Matching"을 구현한 이후, 유저에게 조작감에 더한 재미를 주기 위해 간단한 게임인 "모션 액션 택배 배달" 게임을 제작하였습니다. 게임의 룰과 요소는 매우 간단하며, 유저가 "Learned Motion Matching"으로 동적으로 생성된 애니메이션을 콘텐츠 요소와 함께 즐길 수 있도록 구현하였습니다. 또한 콘텐츠를 개발하면서 점프 애니메이션뿐만 아니라 택배처럼 무언가를 집는 데이터셋이 있다면 특징 벡터를 추가하여 물건을 집는 상황에 맞는 애니메이션도 생성할 수 있을 것 같습니다.

참고문헌

[1] Michael Büttner and Simon Clavet. 2015. Motion Matching - The Road to Next Gen Animation. In Proc. of Nucl.ai 2015. https://www.youtube.com/watch?v=z_wpgHFSWss&t=658s

[2] DANIEL HOLDEN, Ubisoft La Forge, Ubisoft, Canada SIGGRAPH 2020 – learned Motion Matching https://youtu.be/16CHDQK4W5k?si=CfZw2QgU5GXvKqGY

[3]Ubsoft-"LAFAN1"Dataset. https://github.com/ubisoft/ubisoft-laforge-animation-dataset

[4] The Exact Damper. https://theorangeduck.com/page/spring-roll-call#exactdamper

[5] DANIEL HOLDEN, Ubisoft La Forge, Ubisoft, Canada. https://theorangeduck.com/media/uploads/other_stuff/Learned_Motion_Matching.pdf

[6] David Bollo. 2016. Inertialization: High-Performance Animation Transitions in 'Gears of War'. In Proc. of GDC 2018 https://youtu.be/BYyv4KTegJI?si=GDP1qliaLHo5fBDH

[7] Variational autoencoders. https://www.jeremyjordan.me/variational-autoencoders/