Utilising Neural Networks for Character Control

Jan Szkaradek, MAI

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

MAI in Computer Engineering

Supervisor: Carol O'Sullivan

April 2024

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Jan Szkaradek

April 15, 2024

Utilising Neural Networks for Character Control

Jan Szkaradek, MAI in Computer Engineering

University of Dublin, Trinity College, 2024

Supervisor: Carol O'Sullivan

Creating smooth and realistic animations in video games has been an ongoing challenge, driving advancements in the field of real time animation since the early days of the industry. The state of the art methods provide promising results but suffer from limited scalability, as well as general lack of adaptability to unexpected movements. Most recently, data driven approaches utilising neural networks were proven to be a viable solution to these problems. The aim of this project was to investigate these solutions and demonstrate their potential by implementing a Learned Motion Matching character controller in Unreal Engine 5 to make it easier for other developers to use and understand these data driven methods. The project resulted in a successful implementation of a Learned Motion Matching system in the engine, with lower memory requirements and higher quality of animation compared to other character control methods investigated in this dissertation. The designed system is capable of setting up a fully working animation for a character from raw motion capture data. It allows to preprocess this motion capture data, train neural networks on that data, and then import those models into Unreal Engine and create smooth, realistic animations in real time.

Acknowledgments

I would like to thank my amazing parents Tomasz and Ewelina, my sister Julia and my friends Ajerico, Hubert and Ryan for being there for me these last 5 years. I would also like to thank Carol O'Sullivan for being a great help as my supervisor.

Jan Szkaradek

University of Dublin, Trinity College
April 2024

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

In the field of video games, character animation is an important aspect of the game. The movement of the character needs to accurately convey the intention of the player to make them more engaged in the game. The most common method of animating the character is to utilize multiple animation sequences of varying movements and synchronize their execution with the current state of the game through finite state machines. The state machine is used to swap between different sequences based on the information from the current state of the game. These sequences can be either completely hand-crafted or recorded as motion capture and later spliced and edited.

In its current state, this approach leaves a lot to be desired in terms of scalability and robustness. The problem comes from the poor scalability of finite state machines. As the number of states and transitions grows, the complexity of the state machine grows exponentially, making it hard to maintain and debug. However, the main issue with the current approach is the lack of smooth transitions between different animation sequences. This is due to the fact that the transitions are hardcoded and do not take into account the current state of the game. This can lead to the character performing unnatural movements and breaking the immersion of the player.

New developments in the field of data-driven character control show promise in solving these issues. One of the most promising approaches is the Learned Motion Matching (LMM) system. The LMM system is a data-driven approach to character control that uses neural networks to predict the next frame of animation based on the current state of the game. The system is trained on motion capture data and uses it to generate smooth and natural-looking animations. The LMM system is also scalable and robust, as it does not rely on hardcoded transitions between different animation sequences. Instead, it uses a series of neural networks running in sequence to generate the next frame of animation

based on the current state of the character. It also requires much less manual labor as the system can be trained on raw motion capture data, modified mainly using procedural methods.

As of now, there are not many tools utilizing this method, as most LMM solutions available are mostly self-contained demonstrations or research projects. By investigating how to implement this system in a more reusable and scalable way, Learned Motion Matching character control could become much more approachable for game developers to implement in their games, enhancing the user experience and immersion.

1.2 Objective

The goal of this project is to make the creation of Learned Motion Matching system more available to developers by implementing a pipeline to edit, train, and finally use the LMM character controler in the game.

1.2.1 Motion Capture Data Extraction

The first step of the proposed pipeline is to extract movement data from raw motion capture files to be used for the training of the neural networks. The main steps the program has to perform are to convert raw file coordinate systems into the one used in the game engine, extract positional and angular velocities, and finally label each frame with features necessary for the training of the neural networks in the next step.

1.2.2 Model Training

The pipeline should be able to use the processed animation data to train the neural network models that will be used in the character controller. Each NN model requires its own training method to serve its purpose in the larger system. The training program should also provide insights into the performance of the model at its given task for users to analyze.

1.2.3 Character Controller in Unreal Engine

The main part of the LLM pipeline is the character controller to be used in the game. The controller should be able to extract current character information from the scene, convert it into features used by the LMM, and utilize the previously trained models to generate the next frames of motion, creating continuous animation. The controller will also need to create smooth transitions between current and next character poses through blending.

1.3 Summary

The purpose of this document is to provide an insight into the final implementation of Learned Motion Matching system implemented for the purposes of this project. The document explains all concepts relevant to the implementation. It also goes over the research done in the field of data-driven animation to provide a better context for the reader. The methodology explains the theory behind the system and its consecutive steps, explaining in detail the necessary algorithms and equations. Specifically, it goes over how data is prepared, how the models are trained, and finally how the character controller utilizes these models to create the next pose. The implementation chapter goes over the solutions and tools used to create the system, explaining the environment setup, the dataset chosen for the project, as well as different nuances of the game engine the LMM is being implemented in. The evaluation chapter provides the results of the system, analysing its performance, quality of the animation and the insights to model training as well as the memory used compared to other character controller approaches. The document concludes with a discussion on the results and the future work that could be done to improve the system.

Chapter 2

Background

2.1 Relevant Concepts

2.1.1 Skeleton Based Animation

Skeleton based animations is a technique in computer animation in which the character is represented by a hierarchical skeleton of bones. A bone is defined as a point in space with its position, rotation and scale, that is connected to other bones in a chain like manner, forming a structure of joints related to each other in position and rotation. The bones are used to deform the character mesh, which is a 3D model of the character, by moving the vertices of the mesh according to the movement of the bones. This allows for the character to be animated by moving the selected joints of the skeleton, which in turn moves the mesh. The skeletal animation is a popular method of animating characters in video games, as it allows 3D artists to abstract complex vertex calculations into simple bone transformations.

In any given skeleton, each bone can have its parent bone and multiple child bones. The root bone is the topmost bone in the hierarchy, which is not connected to any other bone. The child bones are connected to the parent bone, forming a joint chain. The relationship between bones and thier children is defined by the bone's transformation matrix, which is a 4 by 4 matrix that defines the position, rotation and scale of the bone. The transformation matrix of the child bone is multiplied by the transformation matrix of the parent bone to get the final transformation.

With that in mind it is important to make the distinction between global and local space of the bones. The global space of the bone is the position, rotation and scale of the bone in the world space, while the local space is the position, rotation and scale of the bone relative to its parent bone. The local space of the bone is defined by the trans-

formation matrix of the bone, while the global space is defined by the transformation matrix of the bone and all of its parent bones. This allows for the bones to be moved in a hierarchical manner, where the movement of the parent bone affects the movement of the child bones.

In most game engines there is also a character space, which assumes the root bone is the origin of the character skeleton. This helps in abstracting the movement of the character from the world space, making it easier to animate. This is especially important in the in engine implmentation of this character controller, as it tries to predict the next frame of animation based on the current state of the character, isolated from the world space of the scene.

2.1.2 Animation State Machines

To adapt to different events in the game, the transition between animations has to be conditional to the player's actions and its current state in the game. The most common approach is to utilise finite state machines to transition animations in the correct manner. The state machine is a graph of states, where each state is associated with a specific animation sequence. The transitions between states are defined by the conditions that have to be met for the transition to occur.

An example of that can be a character in a running motion trying to crouch down. A player presses a button to crouch, and the game sets *isCrouching* variable to *true*. The state machine which is currently in a running state sees that the *isCrouching* changed from *false* to *true*, which is a condition for the transition to the crouching state. The crouching state then triggers the change in animation to crouching, most often interpolating between currently playing running animation and the crouching animation.

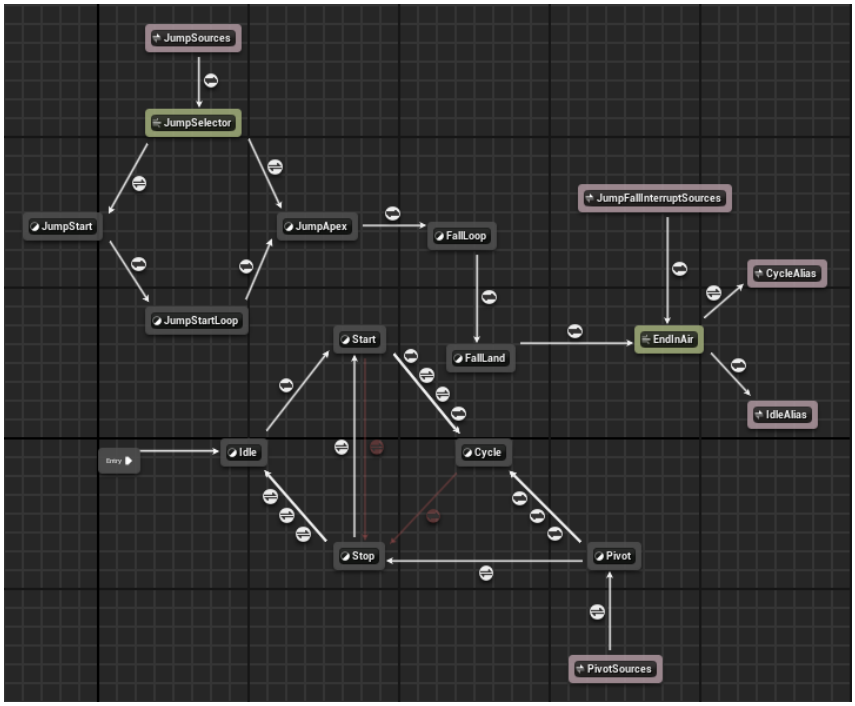


Figure 2.1: Animation State Machine used for locomotion in Lyra Starter Project fo Unreal Engine

Aside from simple truth/false variables, more sofisticated state machines can use more complex conditions for transitions. These can be based on the current state of the game, such as the position of the character, the state of the environment or the state of other characters. Depending on a complexity of the game, these state machines can become complex and hard to maintain effectively. Any addition of a new state or transition may need to be integrated into multiple other states as well. This can lead to a lot of manual labor and debugging.

2.1.3 Animation Transition Methods

Simply transitioning between two animations can lead to unnatural looking movements. It usually creates a sudden change in location of the character, which in context of humanoid characters fails to mimic the effects of mass and inertia associated with the movement of the human body. It also creates visual aritfacts in form of animation "popping" into place. To create a more natural transition between animations, several animation transition techniques have been developed.

Interpolation

Interpolating between animation frames is the most common approach in creating smooth transitions between frames in an animation sequence. The blending is done by calculating the weighted average of the two frames, where the weight is determined by the time between the two frames. The interpolation is performed on positions, rotations and scales of the bones in the skeleton. Various interpolation methods can be used, such as linear interpolation or *lerp*, cubic interpolation or spherical linear interpolation. The choice of interpolation method depends on the type of animation, the desired effect as well as performance. This removes the discontinuity between the two frames and makes the animation independent of the frame rate. It is especially beneficial in the context of computer games, where the frame rate can vary depending on the performance of the hardware.

Crossfading

While interpolation creates smooth transition between frames, crossfading is used to create smooth transitions between animation sequences. This works on the same principle, as the future frames of two different sequences are blended together, where one sequence is faded out and the other is faded in. Multiple sequences can be crossfaded together at different time frames depending on the implementation. This is especially useful in the context of character animations, where the character can transition between different states (running, jumping, crouching, etc.) at the same time. This also helps in relaying the intent of the player and its input to the character, allowing for more responsive and immersive gameplay.

Inverse Kinematics

While calculating the position of a final joint in the chain through transforming it by its parent joints, the reverse can also be utilised in the form of inverse kinematics. In cases where the position of the end effector, that be a foot or a hand joint, needs to be at a specific position and rotation, the position of the parent joints can be calculated based on the position of the end effector. This is mainly used for a more accurate foot placement, where the position of the foot needs to be on the ground.

2.1.4 Spring Dampening

In context of character controllers, spring dampening can be defined as a specific implementation of exponentilal decay, targeted towards smoothing or stabilising the changes in velocity and positions of the objects in the game, replicating the behaviour of a spring. The version of the spring dampening that is specific to this project comes from resources made available by Daniel Holden (Holden (2024)). Given the position x_i, the velocity v_i of the object and time interval dt, the new position x_{i+1} and velocity v_{i+1} of the object, whose goal is position x_{goal} can be calculated as follows:

$$j_0 = x_i - x_{\text{goal}} \tag{2.1}$$

$$j_1 = v_i + j_0 \cdot \lambda \tag{2.2}$$

$$x_{i+1} = e^{-\lambda \cdot dt} \cdot (j_0 + j_1 \cdot dt) + x_{\text{goal}} \tag{2.3}$$

$$v_{i+1} = e^{-\lambda \cdot dt} \cdot (v - j_1 \cdot y \cdot dt) \tag{2.4}$$

The function is used throughout the project for purposes such as inertialisation as well as computing trajectories. It gives a straightforward way to create smooth, continous changes in movement of the object, without much overhead in terms of computational speed.

2.2 Related Work

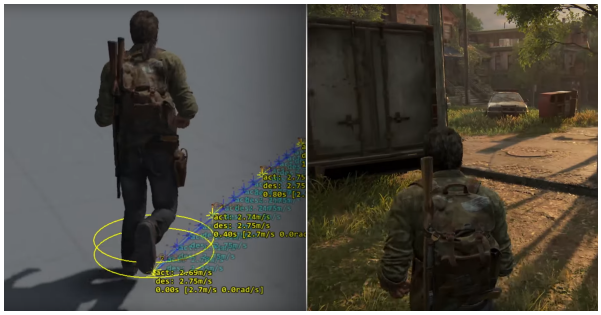
The conventional method of designing finite state machine and assigning each state to a sequence of animations can be cumbersome and hard to maintain the more complex the system is. The transitions between these sequences are also limited in flexibility and also leave a lot to be desired in terms of realism and fluidity of motion. In recent years, new methods have been developed that help developers mitigate those concerns by transitioning from code to data driven systems, and utilising neural networks and machine learning systems in character control.

2.2.1 Motion Matching

The method of motion matching can be traced back to the initial paper of motion graphs (Kovar et al. (2002)). The idea was to create a graph from a database raw motion capture animations, split the sequences into smaller parts and generate the transiotions between them. The controller would then be able to jump between the animations through those transitions, creating a smooth movements while also applying the control inputs of the

player. Later on, motion fields approach was introduced (Lee et al. (2010)), providing a framework to create movements that are more responive to player input. Instead of the graph, the relationships between animations can be encapsulated into vector field of positions, rotations and velocities of joints in every frame. Reinforcment learning can then be used to find the best policy for which frame to choose based on the current state. More importantly, transitions take place between the animation frames rather than sequences, resulting in animations that orient themselves toward the trajectory of character much faster than motion graphs. Unfortunately due to the complexity of the system as well as the computational cost of finding the next animation pose/frame, the method was not widely adopted in the industry.

Motion matching is a technique developed by researchers at Ubisoft (Clavet (2016)) that builds upon motion fields principles while being much more approachable in terms of implementation in games. The vector field of the model does not include the whole pose, but rather essential joints such as feet and hips, as well as the trajectory of the pose, usually at points of 20, 40 and 60 frames forward. In motion matching terms, those variables are called features, where each set of features corresponds to a given pose. The vector field itself is also not a unique entity. It is embedded into the animation database at each frame. The most important distinction is the simplification of the policy of finding the next frame, by performing a simple greedy search through a database rather than creating policies based on reinforcment learning. This allows for a much faster and more efficient way of finding the next frame, while still maintaining the responsiveness and flexibility of the system. The animations become much more dynamic and much more realistic in terms of transitions between movements. Since its inception this approach has been widely adopted in the industry. *For Honor* (Clavet (2016)) was the first game to adopt this method, with later tiltes such as *The Last of Us part II* (Michal Mach (2021)) and most recently *Fortnite* (Nilson et al. (2024)) replacing all of its animations with motion matching.



(a) Motion Matching in game "The Last of Us Part II"

(b) Animations in game "For Honor"

Figure 2.2: Examples of motion matching in recently released games

While motion matching remains a powerful tool, it still comes with some limitations. The main drawback of this method is memory. To make it work effectively, a big set of animations is needed, which can take up a lot of space in memory. This is especially true in the context of video games, where memory is a limited resource, reserved mainly for quick access to 3d assets and textures.

2.2.2 Phase Functioned Neural Networks

It was later shown that neural network based character controllers can perform just as well in generating dynamic movements with much lower memory footprint. Specifically the set of methods expanding on the use of phase function neural networks (Holden et al. (2017)). A single network trained on labeled motion capture data is capable of generating continous frames of animation that provide much better results than conventional methods. The first iteration showed much promise in creating controllers capable of traversing rough surface terrains as well as creating transitions similar in realism to motion matching approach. The raw motion capture data is first processed to extract the features of each pose. The key feature that distinguishes this method among others is the extraction of a phase variable. This property determines at which state of motion the pose currently is. It is important for the training of the network to capture the cycle of foot and hand movements, to ensure that the model maintains the offsets in rotation between right and left joints of both feet and hands, providing continuity to the cycle of movement. The animations are also duplicated and translated onto rough terrain surfaces to create a more diverse dataset and making the network capable of traversing rougher terrains. Overview of the method from the first paper is presented in Figure 2.3.

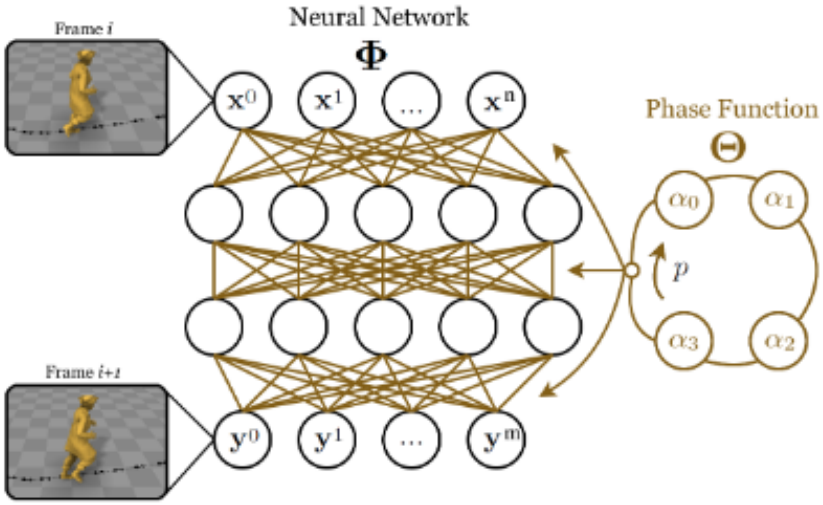


Figure 2.3: Visual representation of the Phase Functioned Neural Network from the original paper (Holden et al. (2017))

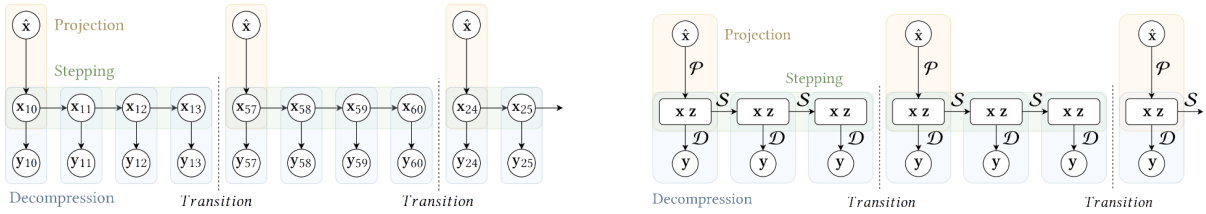
The later advancements in that area showed models capable of extracting phase using neural networks themselves (Zhang et al. (2018)), and also adapt to more unique situations and states such as being able to pick up objects of variable size and walk through variable size openings and holes in walls Starke et al. (2019). It also proved to work well in context of quadruped motion, where creating specific motion capture on trained animal can be rather difficult. The model was also adapted to more complex scenarios of movement by labeling each joint with a local phase, rather than extracting a phase of the whole frame (Starke et al. (2020)). This allowed for more complex movements in terms of synchronisations, such as bouncing a basketball while running or pushing away enemy opponents while walking.

While powerful and capable of generating highly adaptable movements, this technique is once again not very straightforward to implement reliably in games. As with most neural network approaches, it is hard to predict and correct the output of the model. If the features diverge significantly enough from the training data, the model is not able to generate correct poses. Any game requiring deterministic and predictable movements such as competetive multiplayer games would not utilise this method due to its unpredictability. Futhermore, depending on the amount of animations and features, the model can be quite large and computationally expensive to train and run. While the first iteration is a simple set of dense layers, the later iterations require more complex architectures to produce accurate movements reliably.

2.2.3 Learned Motion Matching

And so we converge onto the most recent iteration of the data driven character control that is learned motion matching (Holden et al. (2020)). The technique combines the principles of motion matching and phase functioned neural nets to minimise the drawbacks of both, effectively improving on the memory requirements, model training speed, as well as predictability of the model. The methodology is quite straightforward, as it simply replaces the procedural stages of motion matching with learned alternatives.

By examining Figure 2.4 we can see that the core functionality of motion matching revolves around three stages, *Projection*, *Stepping* and *Decompression*. All of these steps can be replaced with neural networks of relatively simple architectures compared to the PFNN models. The key improvement of this method is changing the way the motion matching accesses the database of animations by directly encoding it in the *Decompression* network. This removes the need for the database after the model has been trained, saving considerable amounts of memory while still maintaining the quality of the animations. The model is also much more predictable and reliable, as it still follows the same algorithm as motion matching, but with the added benefit of being able to adapt to more unique situations and states. Features passing through the projector and stepper are much more reliable in terms of continuity and alignment with the features of the database, working even for the unseen data while not suffering from the incorrect poses due to divergence form the base data.



(a) Motion Matching (b) Learned Motion Matching

Figure 2.4: Motion Matching and Learned Motion Matching algorithms side by side

Chapter 3

Methodology

The basic idea of the Motion Matching based system is to create a database of animations where each frame is embedded with a set of features that describe the pose of that frame. The goal of the system is to always play the frame from the database that most closely matches the desired features of the current pose. The desired features change based on the player input and the current state of the character in game.
The system consists of three steps: *Projection*, *Decompression* and *Stepping*. Almost every frame *Stepping* advances the current animation by one frame. Every N frames, the *Projection* is used to project the current pose into the feature space. The projected features are then used to search the database for the best matching frame. If a frame with a lower cost than the current frame is found, a transition is inserted, and the new animation is blended in. The *Decompression* is used retrieve the full pose from the database using the features extracted from the *Projection*.

The method for this project is almost directly replicated from the original paper (Holden et al. (2020)) in terms of definition of the components. Feature vector of every frame is defined as

$$x = \{\mathbf{t_t}, \mathbf{t_d}, \mathbf{f_t}, \hat{\mathbf{f}}_\mathbf{t}, \hat{\mathbf{h}}_\mathbf{t}\}$$

Where $\mathbf{t_t}$ is the trajectory position, $\mathbf{t_d}$ is the trajectory direction, $\mathbf{f_t}$ are the positions of both feet, $\mathbf{f_v}$ are their translational velocites and $\mathbf{h_v}$ is the hip translational velocity. All of the joint components are represented in local space. The one difference between the original paper is the exclusion of predicting foot contact informations, helping with foot placement via inverse kinematics. In most cases it the inverse kinematics tools provided by the engine are much faster and more accurate compared to the proposed method. In case of implementing a smaller demo project, it would be more reasonable.

The pose information in the database is also stored as a vector formulated as below.

$$y = \{\mathbf{y_t}, \mathbf{y_r}, \hat{\mathbf{y}}_\mathbf{t}, \hat{\mathbf{y}}_\mathbf{r}, \hat{\mathbf{r}}_\mathbf{t}, \hat{\mathbf{r}}_\mathbf{r}\}$$

Where $\mathbf{y_t}$ and $\mathbf{y_r}$ are the positions and rotations of the feet, $\hat{\mathbf{y}}_\mathbf{t}$ and $\hat{\mathbf{y}}_\mathbf{r}$ are their translational and rotational velocities. The $\hat{\mathbf{r}}_\mathbf{t}$ and $\hat{\mathbf{r}}_\mathbf{r}$ represent the translational and rotational velocities of the root bone of the skeleton, relative to the position of trajectory of the current frame. The rotations themselves are stored as a 2 axis rotation matrix representaion, recomputing the final axis when converting to a quaternion. (Zhang et al. (2018))

The one thing that is different in Learned Motion Matching compared to conventional Motion Matching is the addition of a set of latent variables $\mathbf{z}$. The vector $\mathbf{z}$, extracted during training, encodes the relationships between poses and features that were obtained by the neural network, serving as an additional component indexing the database. It is important to establish the system based on motion matching definitions, as it essentially replicates each step of the process, but with providing learned alternatives to the key steps by introducing three neural nets to replace them: *Stepper*, *Projector* and *Decompressor*.

The LMM system mimics finding and retrieving poses from the database of frame data. For each frame there exist feature, latent and pose vectors, from which the animation pose can be retrieved. The key difference here is that it is not necessary to store the databeas in memory after the models have been trained, saving space in memory. There are also more solution specific processes taking place, such as the blending of the poses as well as calculating the cost of the transition between poses.

3.1 Database Design

To train the models necessary for the system to work, the training data in form of the animation database has to be generated. As the latent variables are computed during training, the training data consists of feature and pose vector tuples assigned for each frame in the motion capture data. The raw motion capture is processed to extract the pose information, which is then used to extract the features. This section elaborates on different conversions and encodings of specific features and variables of the database.

3.1.1 Trajectory

The trajectory is the direction is defined a general direction the pose of the current frame is directed towards. In most adata driven control systems it is constructed form the position as well as direction. There are numerous solutions to computing the trajectory of the pose, mostly through extracting trajectory values based on the orientation of th3 pose. The trajectory extraction for this project was done through obtaining position based on the spine bone, and the direction based on the current direction of the hip bone.

Position

Trajectory positions were extracted by projecting the positions of the *Spine2* bone onto the x-y plane. The Spine2 bone was chosen as it is the most stable bone in the skeleton, and its position is the most reliable for the trajectory computation.

Direction

The direction of the trajectory of the pose was obtained from the rotation of the *Hip* bone. The forward vector of the hip bone was projected onto the x-y plane to obtain the direction of the trajectory. To small high frequency movements of the hip, the direction also needed to be smoothed out. In this case, it was done by applying the Savitzky-Golay filter (Savitzky and Golay (1964)) onto the direction values over time. This results in smooth trajectory estimation with minimal noise.

3.1.2 Joint Velocities

To simulate inertia when blending the final poses, the inertialization function needs to know the velocities of the joints in the pose over time. Both the translational and angular velocities are computed on the basis of the central difference, extrapolating the values of position/rotation of *i-1* to i and i to *i+1* frames.

Translational Velocity

To obtain the translational velocity at point t, the following formula is used:

$$\mathbf{v}_i = \frac{\mathbf{p}_i - \mathbf{p}_{i-1}}{2} + \frac{\mathbf{p}_{i+1} - \mathbf{p}_i}{2} * \mathbf{f} \tag{3.1}$$

Where $\mathbf{v}$ is the velocity, $\mathbf{p}$ is the position, and $\mathbf{f}$ is the frame rate of the animation. It is essentially the extrapolation of local velocities at frames **i-1** and **i+1** relative to the current frame **i**. The frame rate is used to scale the velocity to seconds.

Angular Velocity

In a similar manner, angular velocity is calculated by extrapolation of the scaled angle axis representation of local velocities of rotations at frames **i-1** and **i+1**. The formula is presented as follows:

$$\mathbf{v}_i = \frac{toScaledAngleAxis(\mathbf{q}_i \cdot \mathbf{q}_{i-1}^t)}{2} + \frac{toScaledAngleAxis(\mathbf{q}_{i+1} \cdot \mathbf{q}_i^t)}{2} * \mathbf{f} \tag{3.2}$$

Where **v** is the velocity, **q** is the rotation, and **f** is the frame rate of the animation. The *toScaledAngleAxis* function converts the quaternion into a scaled angle axis representation. The quaternion multiplication is used to obtain the relative rotation between the frames **i** and **i-1** and **i** and **i+1**.

The scaled angle representation is simply the conversion of the quaternion into the angle axis representation, with the axis component scaled by the angle. As for angle axis computation itself, by defining the quaternion as $q = w + xi + yj + zk$, The axis vector is simply the x, y, and z components of the quaternion, while the angle is computed as $2 * acos(w)$. It is also common to take the absolute value of w to avoid the angle flipping when the quaternion is negated.

3.1.3 Feature Definition

The specific breakdown of the features used is presented in the table below.

Property	Number of elements
Trajectory positions at frames 20,40,60	3 xy values
Trajectory directions at frames 20,40,60	3 xy values
Right/Left foot joints position	2 xyz values
Right/Left foot joints translational velocity	2 xyz values
Hip joint translational velocities	1 xyz value

Table 3.1: Breakdown of the feature vector used

Feature selection is an important step in creating the database for the LMM system. The features have to be selected in such a way that provides the most information about the pose, specifically the cycle of movement in specific joints. In PFNN, this was done through the phase variable, but the motion matching solution is much simpler. The joint information extracted as features are the positions and velocities of *Right/LeftFoot* joints to obtain the cycle of movement, as well as a velocity closest to the center of mass of the object to capture the high frequency oscillations of the pose. In this case, the *Hip*

bone. The final component is the trajectory, which is used to correlate the animation with player input in the game. Most common implementations also take a number of future trajectories as features, usually 10 to 20 frames ahead of the current frame. This is done to maintain continuity between animations as well as provide more responsiveness to the player input.

3.2 Model Training

Each step of Learned Motion Matching requires its own training setup to perform correctly. It is also important to ensure that the *Decompressor* is trained before the other two models. This is due to the computation of latent variables when training the *Decompressor*. As the *Stepper* and *Projector* predict the latent variables based on input features, the *Decompressor* needs to be trained first to provide the latent variables for the other two to predict. The training algorithms for each model remain unchanged from the original paper (Holden et al. (2020)).

3.2.1 Decompressor

The purpose of the *Decompressor* is to take in features and latent vectors generated by the *Projector* or *Stepper* and return the pose vector corresponding to this input. The *Decompressor* training is based on compressing and then decompressing the pose vectors. The *Compressor* network is used to compress the input pose vector to a feature vector $\mathbf{x}$ and a latent vector $\mathbf{z}$. The *Decompressor* network is then used to decompress the feature and latent vectors back into the pose vector. The training process is done by minimising the difference between the input pose vector and the output pose vector. That way, the information about the pose that is not encoded into $\mathbf{x}$ is stored in $\mathbf{z}$. The latent variable is specific to Learned Motion Matching, as it is capable of retrieving the pose more accurately then through the feature vector alone like in Motion Matching. The loss is computed by the using positions and velocites of the pose, "as the velocities are too noisy to be useful" (Holden et al. (2020)). The character space positions and rotations are also used, computed through forward kinematics of the skeleton joints. This ensures the pose is correctly reconstructed in the context of local space of the character.

3.2.2 Projector

The *Projector* takes in the feature vector computed from the current pose and player input and projects it to one more closely matching a given entry of the database. It also returns a latent vector associated with the projected feature vector. It essentially emulates

the search for the matching feature by using the neural network model. It is necessary to train the *Decompressor* first as the network uses the latent variables computed by the *Decompressor* to predict the latent variables of the projected feature. The *Projector* network is trained by adding noise to the features from the database, finding the closest matching features to them, and passing it through the network, taking the differences in projected feature and latent variables as the loss.

3.2.3 Stepper

The *Stepper* networks replicates the action of transitioning the current animation to the next frame. In this case it takes the current feature and latent variables as input and outputs the features and latent variables of the next frame in the animation. The training of this step is done through batches of inputs rather than single frames. A segment of sequential frames is extracted as a batch and passed through the network. The average loss of the batch is then fed into the loss function and the network weights are updated.

3.3 Character Controller

3.3.1 Real Time Feature Extraction

The features extracted by an in-engine character controller have to represent the current state of the character as well as the changes in player input. In this and most data driven controllers, the input is taken from the gamepad the direction of movement taken using the rotation of the joystick. While current features related to the pose can be computed manually, it is most common to retrieve the features computed by the projector or stepper in the previous frame. This maintaing the continuity in computation as well as increses the speed of computation.

The key feature to be extracted is the trajectory determined by the user input. The position of current trajectory is simply the position of the root, and the direction is the x and y value of the controller joystick direction multiplied by the velocity. The walking speed, accelleration as well as other components are more related to the movement controller and can vary depending on the implementation. The future trajectories are predicted by a spring equation (Equation 2.4). In this particular case the computation is less intuitive, as the $\mathbf{x}$ is the velocity of the movement and $\mathbf{v}$ is the acceleration, defined by the controller. The direction is computed in a similar manner with anglular velocity and acceleration extracted from the direction of the joystick.

3.3.2 Learned Motion Matching Logic

The controller logic for Learned Motion Matching utilises the **Projector**, **Stepper** and **Decompressor** networks to compute the pose of the character in real time. The features are extracted based on the previous joint features and the current trajectory of the pose.

Most of the time the system simply updates to the next frame through the **Stepper** network, decompressing the next pose $\mathbf{y}$ through its updated feature and latent vectors $\mathbf{x}$ and $\mathbf{z}$.

When the timer elapses or the current trajectory deviates too much from the previous one, the **Projector** is used to project the current pose into the feature space and search for the best matching frame in the database. If a frame with a lower cost than the current frame is found, a transition is inserted, and the new animation is blended in. The **Decompressor** is used to retrieve the full pose from the database using the features extracted from the **Projector**. The system is presented in the algorithm below.

Algorithm 1 Learned Motion Matching

Given: $\mathbf{x}, \mathbf{z}, \mathbf{y}$ from the previous frame

$\mathbf{x_{curr}}, \mathbf{isForceSearch} \leftarrow \mathit{UpdateFeatures}(\mathbf{t'_t}, \mathbf{t'_d}, \mathbf{f_t}, \hat{\mathbf{f}}_\mathbf{t}, \hat{\mathbf{h}}_\mathbf{t})$ ▷ Get new features

if searchTimer.finished **or** isForceSearch **then**
  $\mathbf{x_{curr}}, \mathbf{z_{curr}}, \mathbf{isTransition} \leftarrow \mathit{Projector}(\mathbf{x_{curr}})$ ▷ Project the features
  if isTransition **then**
    $\mathbf{y_{curr}} \leftarrow \mathit{Decompressor}(\mathbf{x_{curr}}, \mathbf{z_{curr}})$ ▷ Decompress the pose
    $\mathit{InertialiserTransition}(\mathbf{y}, \mathbf{y_{curr}})$ ▷ Update the offsets of the inertialiser
  end if
  $\mathit{searchTimer.reset}()$
end if

$\mathit{searchTimer.tick}()$ ▷ Tick the search timer

$\mathbf{x}, \mathbf{z} \leftarrow \mathit{Stepper}(\mathbf{x}, \mathbf{z})$ ▷ Step to the next frame index
$\mathbf{y_{curr}} \leftarrow \mathit{Decompressor}(\mathbf{x}, \mathbf{z})$ ▷ Decompress the pose

$\mathit{InertialiserUpdate}(\mathbf{y}, \mathbf{y_{curr}})$ ▷ Update current pose with the new pose
SetCurrentPose($\mathbf{y}$) ▷ Set the pose of the character

Each step of the algorithm helps in providing continous motion. Without the **Stepper**, the pose would not be able to advance forward in time. Without the **Projector**, the pose would not take into account the current information about the scene. It is the oscillation between those two steps that provides the most accurate pose for the character. It is also important to acknowledge that stepping occurs every frame, while the projection is done occasionally.

The frequency of the projection step is controller by the **searchTimer** and **isForceSearch** variables, which play important role in maintaining the continuity of the pose. Essentially the **searchTimer** ensures that the projection is done at a regular interval. The **isForceSearch** variable is used to force the projection step to occur in cases of sudden changes in the trajectory of the character. It helps to maintain the responsiveness of the character without waiting for the **searchTimer** to elapse.

3.3.3 Inertialisation

The component that plays a key role in smoothing the computed frames over time is the inertialiser. Inertialisation is a recently proposed method (Bollo (2018)) of creating transitions between poses through offset decay. The offsets are computed by taking the difference between the current pose and the previous pose, and then applying a spring damper decay equation to the offsets (Equation 2.4). The principle is to keep track of the offsets between the current pose and the next through gradually blending the two poses using these offsets. It simulates the effects of inertia and momentum of the character, making the transitions between poses appear more smooth and natural than traditional crossfading or interpolation. As seen in Algorithm 1, the inertialiser consists of two steps, **Transition** and **Update**.

Transition

The transition steps updated the offsets of the inertialiser based on the difference between the current pose and the new pose computed by the **Projector**. The general equation to recompute the offset of variable **a** is simply.

$$\mathbf{offset}_v = (\mathbf{v} + \mathbf{offset}_v) - \mathbf{v}_{new} \tag{3.3}$$

That also applies to the rotations, where the quaternion offset is recomputed by the following equation.

$$\mathbf{offset}_q = \mathbf{offset}_q \cdot \mathbf{q} \cdot \mathbf{q}_{new}^{-1} \tag{3.4}$$

Because the projection step usually outputs poses that vary more than the one from the stepping step, the transition step updates the context of the inertialiser to significantly different offsets, preventing sharp changes and animation popping in the next pose.

Update

This step is used to create a new pose from the current pose and the offsets of the inertialiser. First the offsets are passed through the spring damper decay function to gradually reduce them over time. Given offsets for position, rotation, translational and angular velocity $\mathbf{offset}_x$, $\mathbf{offset}_v$, $\mathbf{offset}_r$, $\mathbf{offset}_a$, the dampening coefficient λ, and time difference $\mathbf{dt}$, the position and velocity offsets are scaled down in the following manner.

$$\mathbf{j}_1 = \mathbf{offset}_v + \lambda * \mathbf{offset}_x \tag{3.5}$$

$$\mathbf{offset}_x = \exp(-\lambda * dt) * (\mathbf{offset}_x + dt * j_1) \tag{3.6}$$

$$\mathbf{offset}_v = \exp(-\lambda * dt) * (\mathbf{offset}_v - dt * \lambda * j_1) \tag{3.7}$$

Similarily the rotation offsets are scaled down in the following manner.

$$\mathbf{j}_0 = QuatToScaledAngleAxis(\mathbf{offset}_r) \tag{3.8}$$

$$\mathbf{j}_1 = \mathbf{offset}_a + \lambda * \mathbf{j}_0 \tag{3.9}$$

$$\mathbf{offset}_r = ScaledAngleAxisToQuat(\exp(-\lambda * dt) * (\mathbf{j}_0 + dt * \mathbf{j}_1)) \tag{3.10}$$

$$\mathbf{offset}_a = \exp(-\lambda * dt) * (\mathbf{offset}_a - dt * \lambda * j_1) \tag{3.11}$$

The output pose is then computed by adding the offsets to the pose computed by **Decompressor** using the **Stepper** input.

$$\mathbf{x}_{new} = \mathbf{x} + \mathbf{offset}_x \tag{3.12}$$

$$\mathbf{v}_{new} = \mathbf{v} + \mathbf{offset}_v \tag{3.13}$$

$$\mathbf{r}_{new} = \mathbf{r} \cdot \mathbf{offset}_r \tag{3.14}$$

$$\mathbf{a}_{new} = \mathbf{offset}_a + \mathbf{offset}_r.RotateVector(\mathbf{a}) \tag{3.15}$$

Lambda Computation

The λ value is a key component of the inertialiser. It is used to control the speed of the decay of the offsets. The higher the value of λ, the faster the offsets decay. As it will be seen later in the evaluation, the λ needs to be carefully tuned with respect of the frequency of the projection step to ensure its value does not accumulate over time. The best way to control the λ is to set it based on the desired time of the decay of the offsets. For the purposes of this project, the λ is controlled based on the time it takes for the offsets to be reduced to 99% of their original value. The formula for computing the λ can be computed based on the formula for exponential decay.

$$N(t) = N_0 \cdot e^{-\lambda t}$$

where $N(t)$ is the remaining value after time t, N_0 is the initial value, λ is the decay constant, t is the time, and e is the base of the natural logarithm.

At the time t 99% of the initial value has decayed and only 1% of the initial value remains, so $N(t) = 0.01N_0$. Substituting this into the exponential decay formula gives:

$$0.01N_0 = N_0 \cdot e^{-\lambda t}$$

$$0.01 = e^{-\lambda t}$$

Taking the natural logarithm of both sides results in:

$$\ln(0.01) = -\lambda t$$

$$t = \frac{\ln(0.01)}{-\lambda}$$

$$\lambda = -\frac{\ln(0.01)}{t}$$

$$\lambda = \frac{\ln(100)}{t} = \frac{4.605}{t}$$

This way the λ can be computed based on the desired time of almost total the decay of the offsets.

Chapter 4

Implementation

4.1 Game Engine

The choice of the game engine was made based on popularity, as well as sofistication of tools available for the developer. The minimum that is required is the support for importing and executing Neural Networks into the game, as well as availability of procedural skeleton animation tools that can be utilised to perform the necessary LMM related joint transformations.

The main contenders were Unity, Unreal Engine and Godot game engines. Due to lack of tools in Godot as well as the dwindling support of the community towards Unity due to its recent policy changes (Wired (2023)), the Unreal Engine was chosen as the main platform for the project. It provides official support for running neural networks in game through its Neural Network Evaluator plugin, as well as a robust animation system that can be used to implement the LMM controller.

It is also important to mention that due to its complexity and number of tools and frameworks, Unreal Engine is the hardest to implement the new features as an outside developer. Almost all of its logic is written in C++ with most of the game related logic abstracted into its system of Blueprints. And while the focus is to make the blueprint system approachable to novice developers, the C++ side of the engine is very complex and hard to navigate. Implementing a complicated system such as the LMM controller in a C++ part of the engine is an achievement in itself, but will provide developers with a clearer view of the inner workings of the engine, hopefully making it easier to implement similar systems in the future.

The project itself was created from the Third Person Movement template provided by the engine. It provides all the necessary blueprints and assets to set up the enviornment in which we can assess the performance of the LMM and the baseline. Figure 4.1 shows the view from the created scene with imported character mesh and animations.

Figure 4.1: View from the scene in Unreal Engine

4.2 Motion Capture Dataset

To evaluate that the pipeline is working correctly, the animations that are being fed into it have to be selected. In this case, the LAFAN1 animation dataset was used. It is an open source database of raw motion capture animations made available by Ubisoft. The expanded version of said dataset was actually used in training of the networks in the original paper (Holden et al. (2020)). It provides all the necessary movements, specifically the ground locomotion such as walking, running and turning.

The animations from the dataset were also imported into the engine to provide a baseline for the LMM system to evaluate against. Unreal Engine requires each animation to be assigned a skeleton that it will be played on. And so the skeletal mesh used in the dataset was also imported into the engine. LAFAN1 already provides an fbx file with the character that can be easily imported into the Unreal. The animations themselves had to be adjusted to fit the Unreal Engine coordinate system, as well as the skeleton structure.

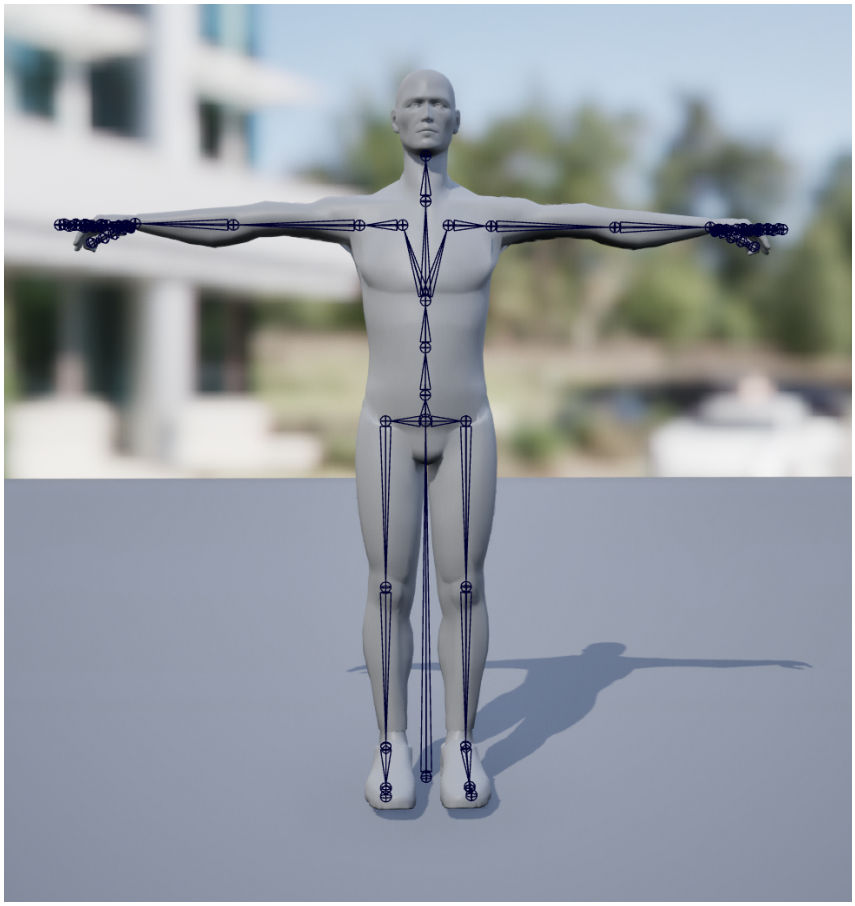


Figure 4.2: LAFAN1 Character Skeleton Mesh imported into Unreal Engine

4.3 System Overview

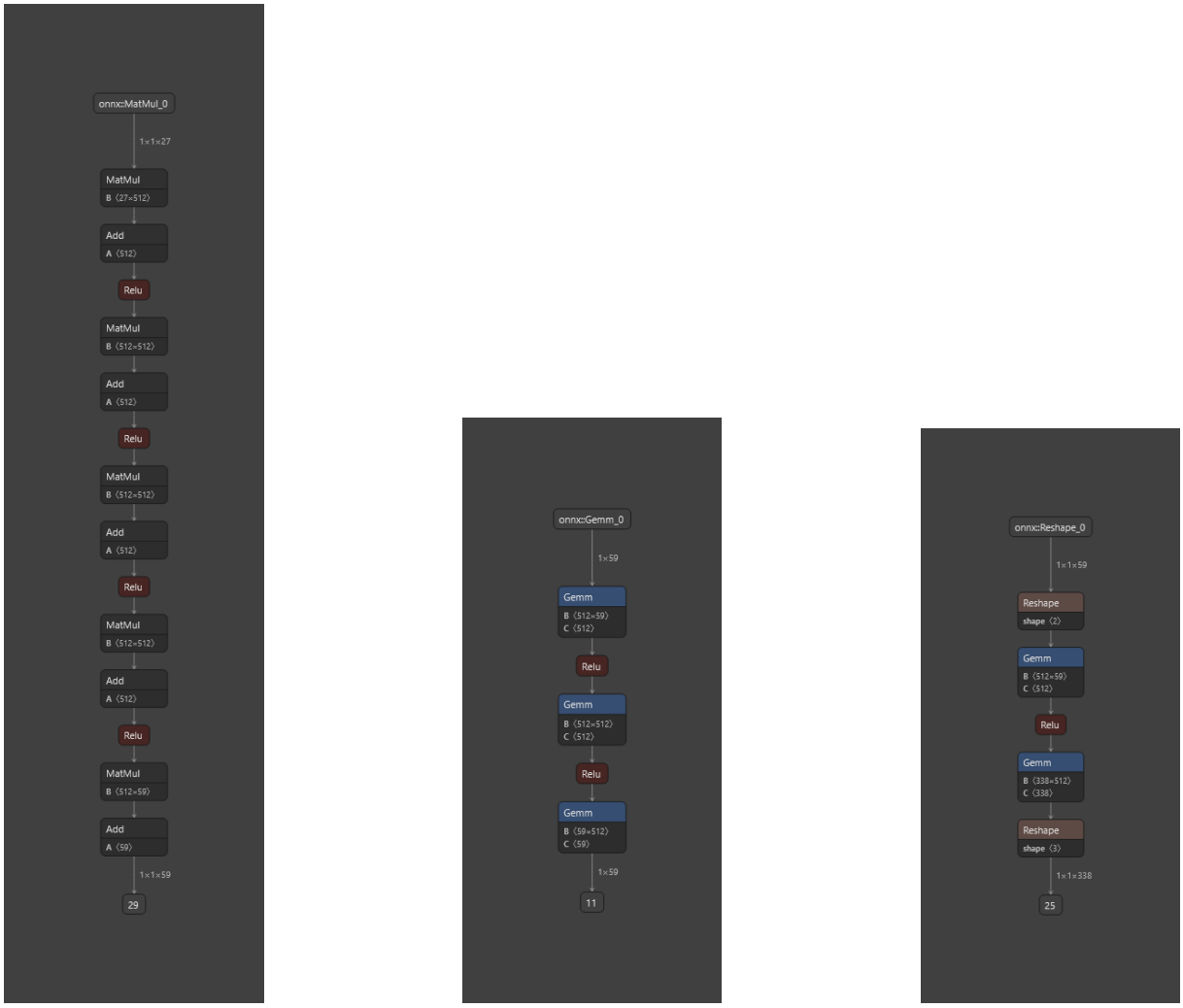
The implementation consists of three key stages: data extraction, model training and character controller. The motion capture is edited and adjusted for the neural network training, then the models are trained and finally the character controller is implemented in Unreal Engine. The Unreal Engine project provides a sandbox environment to run tests, as well as provide a baseline in form of built-in motion matching system.

4.3.1 Data Extraction

The LAFAN1 dataset consists of a set of files in bvh format. The data is converted into fbx format and remapped to Z-up orientation of the Unreal Engine, by rotating the root 90 degrees in the x-axis. The conversion is done using the FBX SDK provided by Autodesk. The data is then used to create the database using methods explained in chapter 3.

4.3.2 Model Training

Model training was done through PyTorch and the models were saved in ONNX format. The models architectures used for the project are shown in Figure 4.3. **Projector**, **Stepper** and **Decompressor** all consist of simple linear layers with ReLU activation functions. The models were trained on a single GTX 3070Ti GPU.



(a) Projector (b) Stepper (c) Decompressor

Figure 4.3: Model layouts used in the project. Extracted from using Netron ONNX viewer

4.3.3 Character Controller

The character controller was designed to work with Unreal Engine Animation Blueprint system by designing a custom animation node written in C++ (Figure 4.4). The node takes in 5 inputs: the previous pose captured by the engine, three ONNX model files, and the character trajectory computed through Motion Trajectory plugin. The node then computes the features from the previous pose and the trajectory, feeds them into the models and computes the next pose. The pose is then blended with the previous pose to create a smooth transition. The pose created by the node is then fed back as the Output Pose. The benefit of this approach is that the node can be easily added to the existing animation graph in Unreal Engine, allowing anyone else using this LMM system to easily integrate it into their projects.

The animation system of the engine also comes with the benefit of only needing to compute transforms of each joint in local space, as the engine handles the global transforms of the skeleton automatically at the later stage of the pipeline.

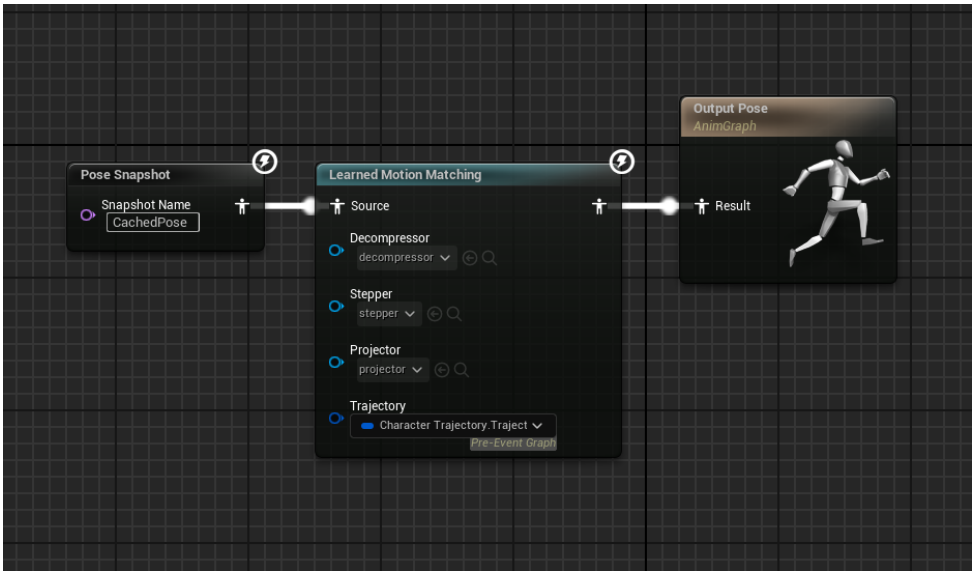


Figure 4.4: Learned Motion Matching Animation Graph

The Learned Motion Matching node also provides access to other parameters of the LMM through the properties window (Figure 4.5). Variables such as search times as well as network weights can be adjusted in real time to make testing of the controller easier.

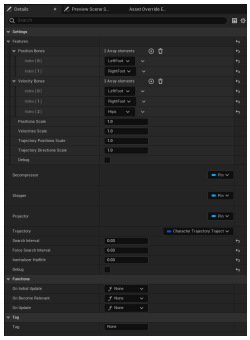


Figure 4.5: Learned Motion Matching Animation Graph Properties Window

4.4 Other character control methods

To compare the model effectively, the baseline motion matching system was implemented in Unreal Engine. The engine already provides tools to implement motion matching through Motion Trajectory and Pose Search plugins. The motion capture animations were imported into the project and the Motion Matching AnimNode was set up to use them. The animation setup itself consists of simple Motion Matching animation node as well as the Pose History node that keeps track of joints used for features. The features used remain the same as in motion matching to provide a fair comparison. The trajectory computation also remains the same. Another character control method the LMM is evaluated against it the Finite State Machine system, which is taken directly from the project template.

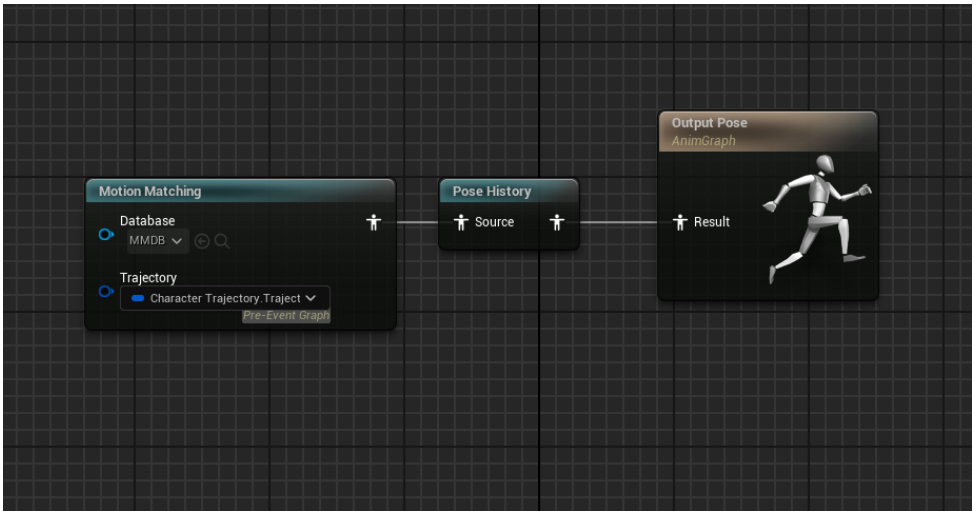


Figure 4.6: Motion Matching Animation Graph

Chapter 5

Evaluation

The result of this project is a fully implemented Learned Motion Matching(LMM) character controller, capable of extracting data from motion capture datasets, training the neural network models on it and utilizing the character controller using those models to create responsive and realistic animations. This chapter will evaluate the performance of the character controller in terms of the quality of the generated animations, the performance of the controller and the memory usage.

5.1 Comparison Methods

5.1.1 Against other character control methods

To compare the LMM controller, the evaluation was done against other methods of character control, as well as the LMM model that was trained on wider range of animations. The methods the LMM is compared against are the conventional Motion Matching (MM) implemented in Unreal, as well as a standard Finite State Machine (FSM) solution automatically generated with the Unreal Project. The comparsion methods were chosen to show the differences between data driven and code driven control, and also compare the LMM to the Motion Matchign solution, that is the closest to the LMM in terms of the method used to generate the animations. The controllers used for comparison, as well as the animation sequences that they use are shown in Figure 5.1. All of the animations aside from the FSM are from the LAFAN1 dataset. The FSM animations are the default UE5 Mannequinn animations included with a project template.

Controller Name	Animations Used
LMM	“pushAndStumble1_subject5”, “run1_subject5”, “walk1_subject5”
LMM_Big	“jumps1_subject1”, “jumps1_subject2”, “run1_subject2”, “run1_subject5”, “run2_subject1” “run2_subject4”, “sprint1_subject2”, “sprint1_subject4”, “walk1_subject1”, “walk1_subject2”, “walk1_subject5”, “walk2_subject1”, “walk2_subject3”, “walk2_subject4”, “walk3_subject1”, “walk3_subject2”, “walk3_subject4”, “walk3_subject5”, “walk4_subject1”
MM	“jumps1_subject1”, “jumps1_subject2”, “run1_subject2”, “run1_subject5”, “run2_subject1” “run2_subject4”, “sprint1_subject2”, “sprint1_subject4”, “walk1_subject1”, “walk1_subject2”, “walk1_subject5”, “walk2_subject1”, “walk2_subject3”, “walk2_subject4”, “walk3_subject1”, “walk3_subject2”, “walk3_subject4”, “walk3_subject5”, “walk4_subject1”
FSM	“MF_Idle”, “MF_Run_Fwd”, “MF_Walk_Fwd”

Table 5.1: Animations used in Controllers of different character control methods

5.1.2 Against different LMM parameters

The analysis of different LMM parameters was also investigated by creating different versions of the LMM controller, with different parameters. Th changed parameters were the **searchTime**, **forceSearchTime** as well as inertializer's **Decay Rate**, as shown in Figure 5.2. All controllers except LMM_Big utilise the same models and animations as the standard LMM controller

Controller Name	Search Time	Force Search Time	Decay Rate
LMM	0.1	0.1	0.1
LMM_Big	0.1	0.1	0.1
LMM_No_Inertia	0.1	0.1	0.0
LMM_High_Search	1.0	1.0	0.10
LMM_Low_Search	0.01	0.01	0.01
LMM_High_Search_High_Decay	1.0	1.0	1.0
LMM_High_Decay	0.10	0.10	1.0
LMM_Low_Decay	0.1	0.1	0.01

Table 5.2: Parameters of LMM Controllers Tested (time in seconds)

5.1.3 With Disabled Inertialiser

The LMM controller was also tested with the inertialiser disabled, to see the effect the removal of the inertialiser transitions has on the animations, and also to evaluate how much the inertialiser contributes to the overall quality of the animations.

5.2 Training Time

Figures 5.3 and 5.4 show the evaluation metrics extracted during training. Because the ONNX model is saved every 1000 iterations, all of the iterations are multiples of that number. The loss is the final loss recorded before terminating the training program

The results from Figure 5.3 show that with a small amount of training data, the training of the model is relatively fast, with the total training time of 10 minutes and 23 seconds. This result shows the advantages of using smaller models, as prototyping with different animations can be done quickly.

Model	Training Time	Iterations	Loss
Decompressor	5 minutes 49 seconds	10000	2.553
Stepper	1 minute 37 seconds	3000	2.443
Projector	2 minutes 57 seconds	4000	1.641

Table 5.3: Model Training Benchmark for the LMM model

Training of the LMM_Big controller was much more time consuming (Figure 5.4), with the total training time of 1 hour 3 minutes and 29 seconds. Each step suffers from the

same problem, as the training time is proportional to the amount of data that needs to be processed. The results show that the training time is significantly longer, but the loss is lower, which is expected as the model has more data to learn from. The iteration per second does not change significantly, aside from the projector model, with much lower amount of iterations given the time it was trained for. This is due to how the model is trained. Evary iteration, the output of the model is compared against the result of a nearest neighbour search in the whole feature space, to find the most suitable feature vector for the given input. As the number of frames in the database increases, so does the number of features. This means that the projector model has to search through a larger amount of data, which increases the time it takes for each iteration to run.

Model	Training Time	Iterations	Loss
Decompressor	19 minutes 29 seconds	15000	3.248
Stepper	17 minute 36 seconds	30000	1.798
Projector	26 minutes 24 seconds	1000	1.820

Table 5.4: Model Training Benchmark for the LMM_Big model

5.3 Animation Quality

The quality analysis of all LMM,MM and FSM controllers was designed to test the generated animations in terms of their responsiveness as well as quality. Each version of the LMM, as well as the other controller methods it was compared against were tested by running a set of motions designed to test different aspects of ground locomotion movement. The first set of motions shows 45/90/180 degree turns, done through arrow keys. This method allows to see the responsiveness of the controller to sudden changes of movement. The other motion set evaluates gradual movements and continous turns based on the rotation of the camera using a standard Xbox One game controller. This helps to see how the character controller handles more subtle movements and how it maintains continuity between frames.

The quality evaluation is based on a set of video clips provided as a supplemental material (Appendix A). The videos show the character controllers performing the movements described in the previous paragraph. The videos were recorded in the Unreal Engine 5, with the default quality settings, at 60 frames per second, with motion blur disabled and VSync enabled.

5.3.1 LMM vs other character control methods (Appendix A, Videos 1-6)

Overall the LMM provides the most responsive and smooth animations, with the character moving in a more natural way than the other controllers. During development it was assumed that the LMM solution would yield similar results to the MM controller, however the MM suffers from artifacts from the dataset, resulting in the character stumbling and tripping, as well as bouncing its head during the run. While most animation sequences in LAFAN1 are clearly labeled, some clips are not as clean as others. Both LMM and MM do show great responsiveness to sudden turns as well as gradual changes in the camera. The FSM controller on the other hand, while being the most stable, lacks the responsiveness of the other controllers, as it is not designed to handle sudden changes in movement, creating a more complex state machine might yield better results, but with added drawback of extended development time. Figure 5.1 highlights those differences in the event of the gradual movement of the camera. The MM and LMM shift the centre of mass to account for the turning angle, while the FSM simply changes the facing direction. It can also be seen that the MM controller exaggerates that movement compared to the LMM controller.

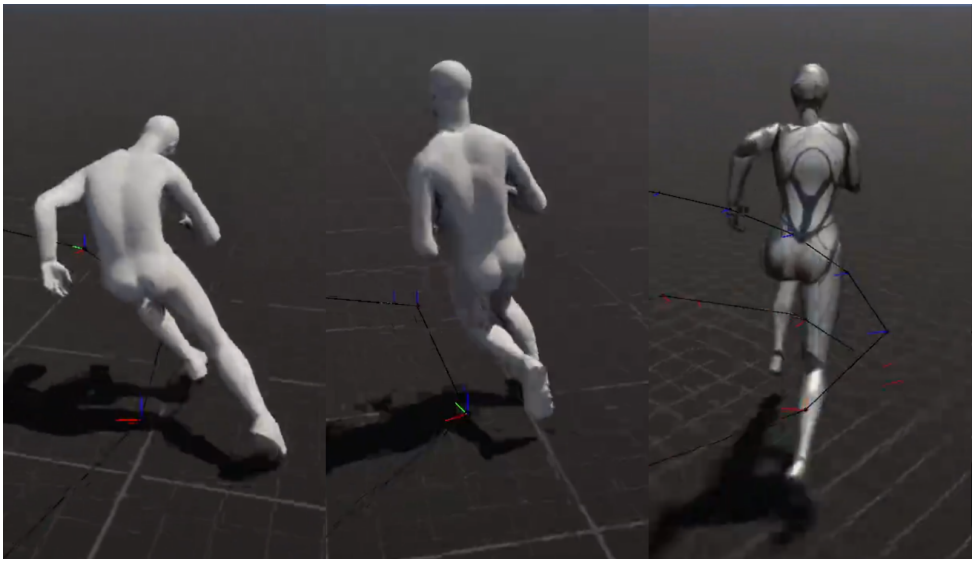


Figure 5.1: Gradual camera turn to the left shown for the MM (left), LMM (Middle) and FSM (Right) character controllers

5.3.2 Effect of varying projection frequency (Appendix A, Videos 12-17)

The effect of varying projection frequency was tested by creating controllers with different search times. In case of LMM_High_Search, the controller suffers from weaker responsiveness to the changes in movement, and the controll of the character feels more sluggish compared to other controllers. With high search time, the projection is not able to fit the character pose to changes in trajectory, making the Stepper model generate poses for the same trajectory over a long peroid of time. It is especially noticeable in sharp turns, where the LMM_High_Search rotates in a manner more similar to the FSM than to the other LMM controllers, without any shift in leg movement or shifting of its centre of mass (Figure 5.2).

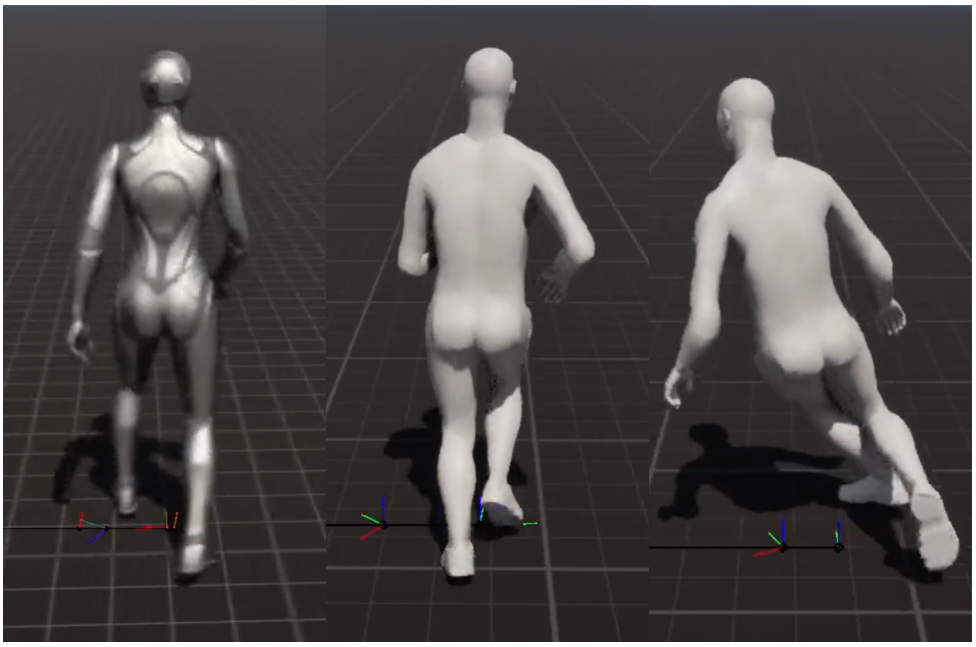


Figure 5.2: Middle frame of 180 truns in character controllers FSM (left), LMM_High_Search (middle) and LMM (right)

5.3.3 Effect of varying decay rate (Appendix A, Videos 9-11 and 14-15)

The decay rate controlls how fast the animation transitions into the next pose, and helps to smooth out the sudden changes in the pose caused by the projection step. This value needs to be finely tuned, as too high of a value can cause the character to become unstable, as shown in the LMM_High_Decay controller. The offsets do not decay fast enough and accumulate over time as more projections are taking place. The makes the pose unstable, as the character distorts and its joints twist and move further and further away from each other. Increasing the search time helps to prevent that behavior but also creates some artifacts. This can be seen with LMM_High_Search_High_Decay controller, where the character is able to maintain stability, but the offsets still accumulate in some frames, which is especially noticeable in the 180 degree turn, where some end effectors like feet and arms rotate in a way that is not natural. The LMM_Low_Decay controller on the other hand, while being more stable, decays the offsets to fast, showing the popping behaviour similar to LMM_No_Inertia controller.

5.3.4 Small vs Large Dataset (Appendix A, Videos 5-8)

Unintiutivelly, providing bigger dataset with longer training did not result in better quality of generated animations. The LMM_Big controller shows much smaller range of movements and changes in the character pose compared to the standard LMM. This is mainly due to the fact that while the dataset increased, the model architecture and feature vector design remained unchanged. The model is not able to learn the more complex movements from the dataset, as its feature set is too small to capture the differences between different movements. Same goes for the model architecture. The **Decompressor**'s linear layers are not able to capture the complexity of different poses, resulting in a more averaged out movement.

5.3.5 Inertialiser (Appendix A, Videos 5-6 and 18-19)

This comparison serves to demonstrate the capability of the inertialiser in terms of transitions. In videos of LMM_No_Inertia controller we can see exactly where the projection is being poerformed, as the pose suddenly snaps to the desired location. While the LMM controller, with inertialiser enabled, smoothly transitions from one pose to another, without any popping taking place. In some cases this jitterry movement is desired, as it can be used for more stylised animations. Most recently, "Spider man: Across the Spider-Verse" mimiced the comic book animation style through the use of popping transitions Dimian et al. (2019), by duplicating frames and adding a slight offset to the character pose.

5.4 Performance

Performance was measured using internal tools provided by Unreal Engine. Specifically the "Anim" statistics option (Figure 5.3), allowing to see the times taken by specific parts of the animation pipeline.

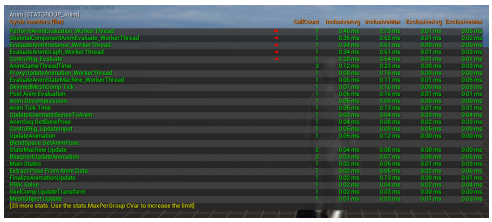


Figure 5.3: "Anim" Performance Window in Unreal Engine 5

The one that is most important for the evaluation is the "PerformAnimEvaluation_WorkerThread" time, as it shows the total time it takes for the animation blueprint to run from start to finish. The results can be seen in Figure 5.5. All metrics were recorded in scene with default quality settings, at 60 frames per second.

Controller	Average Time	Max Time
LMM	0.37	0.63
MM	0.13	0.28
FSM	0.12	0.14
LMM_High_Search	0.37	0.57
LMM_Low_Search	0.42	0.58
LMM_No_Inertia	0.36	0.62

Table 5.5: Performance of different character control methods. Time measuered in miliseconds

Against other controller methods, the standard LMM performs much worse than others. While FSM is faster due to its simplicity, MM is still faster as it does not need to run through 3 sets of neural networks to generate the next pose. The performance problems of LMM may also arise from lack of optimisation in the implementation, compared to the other methods which were implemented by the developers of the engine. As for LMM models themselves, the performance varies as well. The high search time model performs the same as the standard LMM, but the low search time model performs slightly worse as it does more projection computations on top of stepping step than other LMM controllers. Disabling inertia also does not affect the performance of the controller significantly, as the time taken by the inertialiser is negligible.

5.5 Memory Usage

The memory usage was analysed based on the amount of space taken by the essential elements of evaluated controllers. In case of LMM and LMM_Big, the size of the ONNX models as well as the size of feature binary was taken into account, as these are the only objects used by the controller. In case of Motion Matching, the size was established based on the space taken by the animations that were used in the Motion Matching Database. The animation size was measures after it was converted to Unreal Engine UAsset format. The results can be seen in Figure 5.6.

Controller	**Memory Usage**	**Memory Usage(without feature vector)**
LMM	10.7 MB	5.3 MB
LMM_Big	110 MB	7.4 MB
MM	557 MB	557 MB
FSM	1.6 MB	1.6 MB

Table 5.6: Memory Usage of different character control methods

It is quite clear that the motion matching methods take up significantly less memory space than the Motion Matching solution. This is an evidence that encoding the aniamtion data into a set of onnx models is a more memory efficient solution, highlighting the advantage of the Learned Motion Matching method. The direct comparison can be seen against LMM_Big and MM controllers, as they use the exact same animation sequences. The lowest memory is taken by the FSM solution but that is expected considering the minimal amount of movements and lack of resonsiveness of the implemented method
Memory usage without features were also taken into account. The inclusion of features binary as essential part of the LMM controller comes from the fact that the features generated by the controller need to be normalised and scaled to be used by the neural network. The features binary contains the both the features and the normalisation parameters. The features themselves are not necessary, as they can be generated on the fly, but due to time constraints the solution does not generate a binary just with the normalisation parameters. Showing the memory usage without the features binary gives a better idea of the memory usage of the controller itself, without unnecessary features. It is especially important in the case of LMM_Big, where the features binary takes up a significant amount of space, but are never used. And the models themselves are significantly smaller.

Chapter 6

Conclusion

6.1 Contribution

The main contribution of this project is the implementation of a Learned Motion Matching character controller in Unreal Engine. The designed system is capable of setting up a fully working animation for a character from raw motion capture data. It allows to preprocess this motion capture data, train neural networks on that data, and then import those models into Unreal Engine and create smooth, realistic animations in real time. This paper also provides a thorough analysis of the system's performance and compares it to other character controllers such as Finite State Machine and Motion Matching.

Furthermore, the controller is capable of generating animations better in quality and responsiveness compared the other methods of character control with a benefit of lower memory footprint with comparable performance. It can be easily integrated into existing Unreal Engine projects, allowing developers to quickly implement Learned Motion Matching character control in their games. By making it available to a wider audience, the system can be further developed and improved upon, making it more accessible and easier to use.

6.2 Future Work

6.2.1 Synchronisation, Clamping and Inverse Kinematics

The character controller can be further improved by adding features such as synchronisation, clamping and inverse kinematics. Synchronisation can be achieved by comparing the current frame of the animation with the current frame of the motion capture data and adjusting the animation speed accordingly. Clamping can be achieved by limiting

the range of motion of the character to prevent unnatural movements. Inverse kinematics can be used to adjust the position of the character's feet to remove drifting artifacts when making sharp turns.

6.2.2 Dead Blending and Transitions using Neural Networks

The inertialisation method of transitioning between animations can be improved upon in terms of performance and fluidity. Other transition methods can be applied, such as dead blending. Instead of storing an offset between source and destination animations which we fade out, we instead extrapolate forward in time the source animation clip past the point of transition, and insert a normal cross-fade blend between the extrapolated source clip and the destination clip. This method is much simpler to implement with the added benefit of potentially faster computation times.
Both the inertialiser and dead blending can also be performed by a trained model, which would predict the transition pose based on the current pose and the destination pose. This provides an additional advantage of capturing less obvious relationships between poses and transitions, which can be hard to define using manual methods.

6.2.3 Extending the controller

The implemented controller can be adjusted through the use of different features to support a wider range of movements than ground locomotion. The LAFAN1 dataset provides a wide range of movements, such as jumping, crouching and climbing. The controller can be adjusted to support these movements by adding the z component to the trajectory features, as well as modifying data extraction to detect jumps and climbing. Most obvious solution would be to evaluate if both feet are above the ground and define the trajectory height as the average distance between the feet and the ground. The trajectory system itself would also have to be modified to snap onto surfaces and to make it replicate the movement before the jump or climb to the one seen in the data. This would be enough to create a rough version of vertical locomotion system. Other methods can include more scene interaction by keeping track of other objects in the scene and adjusting the trajectory to avoid them or to interact with them. That could be done by creating additional variables in the $\mathbf{y}$ pose information, mainly as additional joints or locations of points in space representing different items or contact points.

Bibliography

Bollo, D. (2018). Inertialization: High-performance animation transitions in 'gears of war'. Game Developers Conference (GDC) Vault. Available online: `https://www.gdcvault.com/play/1025165/Inertialization`.

Clavet, S. (2016). Motion matching and the road to next-gen animation. Game Developers Conference (GDC) Vault. Available online: `https://www.gdcvault.com/play/1023280/Motion-Matching-and-The-Road`.

Dimian, D., Beveridge, J., Clair, B. S., and Basantani, G. (2019). Swing into another dimension: the making of "spider-man: Into the spider-verse". In *ACM SIGGRAPH 2019 Production Sessions*, SIGGRAPH '19. Association for Computing Machinery.

Holden, D. (2024). Spring roll call - controllers. Available online: `https://theorangeduck.com/page/spring-roll-call#controllers`.

Holden, D., Kanoun, O., Perepichka, M., and Popa, T. (2020). Learned motion matching. *ACM Transactions on Graphics (TOG)*, 39(4).

Holden, D., Komura, T., and Saito, J. (2017). Phase-functioned neural networks for character control. *ACM Transactions on Graphics (TOG)*, 36(4).

Kovar, L., Gleicher, M., and Pighin, F. (2002). Motion graphs. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*. ACM SIGGRAPH, ACM.

Lee, Y., Wampler, K., Bernstein, G., Popović, J., and Popović, Z. (2010). Motion fields for interactive character animation. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia)*, 29(5).

Michal Mach, M. Z. (2021). Motion matching in the last of us part ii. Game Developers Conference (GDC) Vault. Available online: `https://www.gdcvault.com/play/1027118/Motion-Matching-in-The-Last`.

Nilson, F., Buck, J., Lemmer, S., Walker, P., Villarroel, J., and Rigamonti, S. (2024). Unreal engine 5.4: Animation deep dive (presented by epic games). https://schedule.gdconf.com/session/unreal-engine-54-animation-deep-dive-presented-by-epic-games/904103.

Savitzky, A. and Golay, M. J. E. (1964). Smoothing and differentiation of data by simplified least squares procedures. *Analytical Chemistry*, 36(8):1627–1639.

Starke, S., Zhang, H., Komura, T., and Saito, J. (2019). Neural state machine for character-scene interactions. 38(6).

Starke, S., Zhao, Y., Komura, T., and Zaman, K. A. (2020). Local motion phases for learning multi-contact character movements. *ACM Transactions on Graphics*, 39(4):54.

Wired (2023). Unity walks back policies, but can it regain lost trust?

Zhang, H., Starke, S., Komura, T., and Saito, J. (2018). Mode-adaptive neural networks for quadruped motion control. *ACM Trans. Graph.*, 37(4).

Appendix A

The analysis of the quality of the animation explained in the **Evaluation** chapter requires these video clips to be viewed. All videos can be found in supplementary material with the submission. The video clips are also available in the following link: `https://drive.google.com/drive/folders/1iDPjdCkwviZ5REluJXmyq46wjgkIPKQe`

- **Video 1 Filename:** `FSM_arrow_keys.mp4` - Video clip of the Finite State Machine character controller performing 45/90/180 degree turns.
- **Video 2 Filename:** `FSM_gamepad.mp4` - Video clip of the Finite State Machine character controller performing gradual movements with a gamepad.
- **Video 3 Filename:** `MM_arrow_keys.mp4` - Video clip of the Motion Matching character controller performing 45/90/180 degree turns.
- **Video 4 Filename:** `MM_gamepad.mp4` - Video clip of the Motion Matching character controller performing gradual movements with a gamepad.
- **Video 5 Filename:** `LMM_arrow_keys.mp4` - Video clip of the Learned Motion Matching character controller performing 45/90/180 degree turns.
- **Video 6 Filename:** `LMM_gamepad.mp4` - Video clip of the Learned Motion Matching character controller performing gradual movements with a gamepad.
- **Video 7 Filename:** `LMM_Big_arrow_keys.mp4` - Video clip of the Learned Motion Matching character controller with a big dataset performing 45/90/180 degree turns.
- **Video 8 Filename:** `LMM_Big_gamepad.mp4` - Video clip of the Learned Motion Matching character controller with a big dataset performing gradual movements with a gamepad.
- **Video 9 Filename:** `LMM_High_Decay.mp4` - Video clip of the Learned Motion Matching character controller with a high decay rate becoming unstable due to offset propagation.

- **Video 10 Filename:** `LMM_Small_Decay_arrow_keys.mp4` - Video clip of the Learned Motion Matching character controller with a small decay rate performing 45/90/180 degree turns.
- **Video 11 Filename:** `LMM_Small_Decay_gamepad.mp4` - Video clip of the Learned Motion Matching character controller with a small decay rate performing gradual movements with a gamepad.
- **Video 12 Filename:** `LMM_High_Search_arrow_keys.mp4` - Video clip of the Learned Motion Matching character controller with a high search rate performing 45/90/180 degree turns.
- **Video 13 Filename:** `LMM_High_Search_gamepad.mp4` - Video clip of the Learned Motion Matching character controller with a high search rate performing gradual movements with a gamepad.
- **Video 14 Filename:** `LMM_High_Search_High_Decay_arrow_keys.mp4` - Video clip of the Learned Motion Matching character controller with a high search rate and high decay rate performing 45/90/180 degree turns.
- **Video 15 Filename:** `LMM_High_Search_High_Decay_gamepad.mp4` - Video clip of the Learned Motion Matching character controller with a high search rate and high decay rate performing gradual movements with a gamepad.
- **Video 16 Filename:** `LMM_Low_Search_arrow_keys.mp4` - Video clip of the Learned Motion Matching character controller with a low search rate performing 45/90/180 degree turns.
- **Video 17 Filename:** `LMM_Low_Search_gamepad.mp4` - Video clip of the Learned Motion Matching character controller with a low search rate performing gradual movements with a gamepad.
- **Video 18 Filename:** `LMM_No_Inertia_arrow_keys.mp4` - Video clip of the Learned Motion Matching character controller without inertia performing 45/90/180 degree turns.
- **Video 19 Filename:** `LMM_No_Inertia_gamepad.mp4` - Video clip of the Learned Motion Matching character controller without inertia performing gradual movements with a gamepad.