Motion Matching for Responsive Animation for Digital Humans

Matthias Karst
ETH Zürich
karstm@ethz.ch

Xue Xian Lim
ETH Zürich
xuelim@ethz.ch

J. Pablo Paniagua H.
ETH Zürich
ppaniagua@ethz.ch

Saatushan Sabesan
ETH Zürich
ssabesan@ethz.ch

Abstract

In this report, the implementation of a complete motion matching pipeline capable of real-time synthesis of realistic motions for a digital human character is presented. This is achieved with a trajectory controller, modeled with a spring-damper system, that receives the user's input and translates them to desired future positions and orientations. The LAFAN1 dataset is preprocessed and subsequently searched for the most appropriate animation sequence. To ensure smooth transitions, discontinuities are mitigated through inertialization, and the animation sequence is synchronized with the trajectory controller. A foot-locking mechanism is employed, and the motion is retargeted to a custom character mesh. The resulting animations exhibit a high level of credibility, showcasing natural characteristics while simultaneously providing responsive motions.

Keywords: Motion matching; character animation; animation; motion capture; foot-locking; inertialization

1. Introduction

In video games and movies, it is often crucial that digital human characters are animated with natural and realistic movements to maintain an immersive experience. This task is especially challenging in video games where the character has to respond in real-time to the user's inputs. Traditionally, this has been accomplished using a state machine, whereby animation clips and transitions are manually defined and mapped to one another. This process can become very time-consuming, especially as the number of states grows larger.

An alternative method, known as motion matching, has been developed by Ubisoft[5] in 2016. Motion matching queries a Motion Capture (mocap) database every few frames to find an animation clip that best matches the current pose and the desired future path for the character.

This paper presents a complete motion matching pipeline implementation using the Ubisoft La Forge Animation Dataset (LAFAN1) dataset[16].

2. Related Work

2.1. Motion Matching

Ubisoft introduced the first motion matching pipeline[5]. They recorded mocap data of actors following 'dance card' sequences with all the required movements. This data was refined and marked with event tags.

They then implemented a simple, timed logic state machine. The difference between this and traditional animation state machines is that the character logic is decoupled from the animation. That is, the logic state machine defines what subsequent actions the character could take along with the timings for these actions. It then sends a request to the motion matching algorithm for a suitable piece of animation that was evaluated via pose, trajectory and event matching. Unlike animation state machines, there is no longer a need to manually define animation transitions, as the motion matching algorithm accomplishes this.

The motion matching algorithm works by finding a frame in an animation clip which has a feature vector that minimizes the euclidean distance to the feature vector of the current pose. This feature vector is defined by the animator and commonly contains future trajectory positions and orientations, feet positions and velocity, and hip velocity - although more features (e.g. weapon position) can be added if necessary. Because every frame in the motion matching database is searched, various optimizations have been explored to speed up the algorithm for real-time usage. These include using a KD-tree, a voxel-based lookup table[13], clustering techniques[17] or principal component analysis.

2.2. Inertialization

At Game Developers Conference 2018, David Bollo introduced the concept of inertialization[3], a post-process animation blending technique that allows for a smooth transition between two unrelated animation states.

Traditionally, the blending process is done by interpolating between the old and the new animation during blend time, leading to twice the animation cost. Bollo, inspired by the concept of inertia in physics, proposes a method that instead saves the positional offset and inertial information for every joint at the transition and uses a polynomial to blend out the positional offset using the inertial information. The inertial information consists of the joint's speed and acceleration before the transition. Inertialization can be directly applied to joint position vectors and relative joint orientation quaternions.

Inertialization has multiple advantages over the traditional blending method. Firstly, the animation cost is reduced to the computation of the new animation and the polynomial. During the transition, the previous animation clip is no longer needed. Secondly, inertialization can be re-applied at any point during the blend process to transition to a new animation state. Lastly, inertialization can produce animations that are more physically plausible than traditional blending methods. Given the example of a character waving his hand transitioning to his hand hanging static at his side, both the traditional blending method and inertialization will produce a smooth transition of the character lowering his arm. However, in the traditional blending method, the character will continue to wave his hand while lowering his arm. In contrast using inertialization the character's hand will follow the inertia of the hand when stopping to wave while lowering it at the same time.

2.3. Foot-locking

A common artifact in motion matching is foot sliding. Foot sliding describes the character's feet sliding across the ground instead of remaining still whenever contact is made. The effect can be observed when the animation and trajectory speeds mismatch, during blending[6] or due to poor source animations. More data can be added to the dataset to reduce this effect, but this comes at an increased memory and processing cost. As an alternative, foot-locking - a technique consisting of locking the foot based on contact constraints and thresholds - is proposed in different sources[10,14], as well as being included in different plugins for Unreal Engine[7] and Deep Motion[1].

Traditionally, the proposed method forces the toes to lock in position whenever the foot is making contact with the ground. IK is used to adjust the leg until the foot is released[4]. Additional locking and release constraints such as the foot velocity while in proximity to the ground, and distance between locked and current root positions are further suggested in other implementations[10,14].

3. Proposed Method

Motion matching is a data-driven algorithm used to generate animated virtual characters. It searches through an animation database for the best pose sequence that aligns with the current pose as well as the intended character trajectory. An overview of the implementation pipeline can be seen in Figure 1.

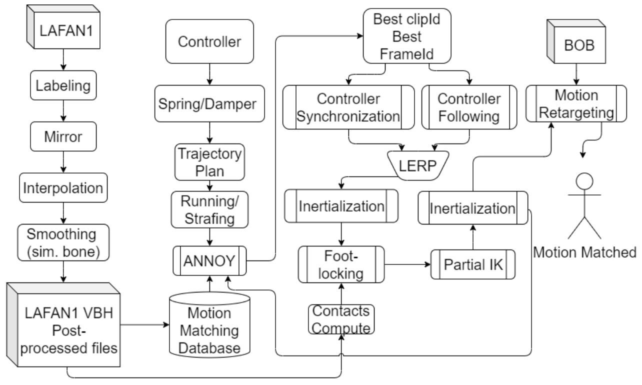

Figure 1. Overview of motion matching pipeline

The implementation of the controller trajectory and motion matching pipeline closely follows the one presented by Holden[10,12]. The final implementation can be found in GitHub [i].

3.1. Gamepad Controlled Trajectory

The trajectory controller converts the user's input into a trajectory with future positions and orientations. It is modeled after a mass-spring-damper system described by the differential equation $m\ddot{x} + b\dot{x} + kx = 0$. With initial conditions $x_0 = X_0$ and $\dot{x}_0 = V_0$ where at time $t = 0$, the critically damped solution is

$$x = e^{-\lambda t}\left(X_0 + (V_0 + \lambda X_0)\, t\right) \quad \text{where } \lambda = \frac{b}{2m} \tag{1}$$

Rather than taking x to be the position, it is more appropriate to replace x with the velocity v and $\dot{x}$ with the acceleration a instead[11]. This is a more representative model of the character's dynamics since the character has inertia and takes a finite amount of time to change its velocity. Furthermore, to drive v towards a desired velocity V_{des} instead of 0, v is replaced by the relative velocity $v - V_{des}$. The V_{des} value is obtained via a predefined mapping of the gamepad's analog stick position to desired character velocity. The modified Equation 1 is thus

$$v = \; e^{-\lambda t}\left(V_0 - V_{des} + (A_0 + \lambda (V_0 - V_{des}))\, t\right) + V_{des} \tag{2}$$

Integrating and differentiating Equation 2 yields the position and acceleration respectively.

$$x = \; e^{-\lambda t}\left(-\frac{K_1}{\lambda^2} - \frac{K_0 + K_1 t}{\lambda}\right) + \frac{K_1}{\lambda^2} + \frac{K_0}{\lambda} + V_{des} t + X_0 \tag{3}$$

$$a = e^{-\lambda t}\left(A_0 - K_1 \lambda t\right) \tag{4}$$

$$\text{where} \quad \begin{aligned} K_0 &= V_0 - V_{des} \\ K_1 &= A_0 + \lambda (V_0 - V_{des}) \end{aligned}$$

Equations 2, 3, 4 are also adapted for future orientations by replacing x, v and a with the angular position θ, angular velocity ω and angular acceleration α respectively. The damping factor λ has been tuned to achieve a balance between smoothness and responsiveness.

Strafing is implemented as well by locking the character's orientation towards the camera direction while the positional kinematics continues to be governed by Equations 2, 3 and 4. Additionally, by altering V_{des}, the character is able to walk, jog or sprint.

3.2. Motion Captured Data Pre-Processing

Since motion matching does not synthesize new poses, the quality of the initial database is essential to obtain a realistic and natural animation. The quantity, accuracy, continuity and variation of animations in a database affect not only the resulting motion, but also the memory and processing cost. For this implementation, the Ubisoft La Forge Animation Dataset (LAFAN1)[8,16] was selected. The dataset offers 77 sequences (33 which directly involve walking or running), for a total of 496,672 motion frames at 30 Frames per Second (FPS) (roughly 4.6 hours) in BioVision Hierarchy (BVH) format.

Before building the motion matching database from the dataset (see Section 3.3), the data was pre-processed in Python to get better quality poses to match. The first step of the pre-processing consists of a manual selection of animations, whereby a more condensed dataset focused primarily on walking, running and strafing is obtained. The animation selection removes undesired context specific actions (i.e. aiming, dancing and jumping).

All the clips from the condensed dataset are then mirrored along the ZY-plane to obtain, effectively, twice as many poses to match. The mirroring is performed as described in Equations 5. Additionally, the indices of the BVH files for left and right joints are inverted, and then used to rearrange the mirrored positions and rotations.

$$\begin{aligned} z_{mirror} &= -z_{original} \\ \beta_{mirror} &= -\beta_{original} \\ \gamma_{mirror} &= -\gamma_{original} \end{aligned} \tag{5}$$

$$\text{where} \quad \beta := \text{rotation around y-axis}$$
$$\gamma := \text{rotation around z-axis}$$

The dataset is up-sampled to 60 FPS to provide a smoother result that can still be handled in real-time with the available computational power. This step is performed by a cubic interpolation of the positions and rotations of each joint.

In the final step, an artificial root bone, referred to as simulation bone, is added. This bone is created to more accurately represent the trajectory of the character. The simulation bone has 3 Degrees of Freedom (DoFs), consisting of the rotation around the up-axis of the *hips* and the position of the $spine_2$ joint projected onto the ground.

The simulation bone is normalized and filtered to remove the oscillation introduced by the natural swing of the hips in human motion. This is done with a third order polynomial Savitzky-Golay Filter[15] with a filter width of 31 for the positions and 61 for the rotations. The hip positions and orientations are adjusted to account for the new root bone and saved in BVH format for use in the application.

3.3. Motion Matching

The motion matching implementation consists of two main components: the motion matching database and the motion matching algorithm.

Motion Matching Database

The motion matching database uses the already preprocessed mocap data to create a 27-dimensional feature vector for each frame of each animation clip. A feature vector $v_{clip,frame} = (t_p, t_d, f_p, f_v, h_v)$ is the concatenation of the following five features, chosen in accordance to Holden et al.[12]:

- **Trajectory Positions:** $t_p = \left(t_p^{(20)}, t_p^{(40)}, t_p^{(60)}\right)$
  2D trajectory positions 20, 40 and 60 frames into the future
- **Trajectory Directions:** $t_d = \left(t_d^{(20)}, t_d^{(40)}, t_d^{(60)}\right)$
  2D normalized trajectory directions 20, 40 and 60 frames into the future
- **Foot Positions:** $f_p = \left(f_p^{left}, f_p^{right}\right)$
  Current left and right 3D foot positions
- **Foot Velocities:** $f_v = \left(f_v^{left}, f_v^{right}\right)$
  Current left and right 3D foot velocities
- **Hip Velocity:** h_v
  Current 3D hip velocity

These features are computed relative to the simulation bone. Each feature vector is indexed by the animation clip and frame it was computed from. To allow for control over feature importance in the motion matching algorithm, the database is feature-wise normalized before user defined weights are applied.

To ensure fast matching times, Spotify's annoy[2] library was used. Annoy is a C++ library that allows for fast approximate nearest neighbor search. The euclidean distance was chosen for the annoy database to compute the nearest neighbor. In order to prevent the motion matching algorithm from reaching the end of an animation clip, the last 30 frames of each animation clip are omitted from the annoy database.

Motion Matching Algorithm

The matching algorithm finds the nearest neighbor in the annoy database given a query vector q. To ensure that the feet and hip poses match the current character pose as closely as possible, the query vector is created by first looking up the database entry of the current frame $v_{clip,frame} = (t_p, t_d, f_p, f_v, h_v)$. Then the trajectory positions t_p and directions t_d are replaced with the desired trajectory positions t_p^* and directions t_d^* from the controller.

Note: *t_p^* and t_d^* need to be normalized and weighted with the same mean, standard deviation and weights as the database entries.* The resulting query vector $q = \left(t_p^*, t_d^*, f_p, f_v, h_v\right)$ is then used to find the nearest neighbor in the annoy database. Finally, the corresponding animation clip and frame indices are used to update the currently playing animation.

To ensure responsive controls the motion matching algorithm is called every 200 ms. This allows for a maximum delay of 12 frames between the current character pose and the user input. In practice this is barely noticeable as long as the controller trajectory stays similar to the animation trajectory during the 200 ms interval. If however the controller trajectory changes abruptly, the delay becomes more noticeable. To prevent this, a search is also triggered whenever the controller orientation or the acceleration is greater than some pre-defined threshold.

3.4. Inertialization

When the currently playing animation changes due to a motion matching call, pose discontinuities can occur. This is especially noticeable in the upper body and arms of the character, as only the feet positions are used for the motion matching.

Note: *Not every motion matching call necessarily results in a discontinuity. As long as the controller trajectory remains in proximity to the animation trajectory, the motion matching algorithm will return the same clip and frame index as before and the current animation will continue to play. Therefore, a discontinuity only occurs if the clip or frame index change.*

To smooth out these discontinuities, the inertialization algorithm is used. The implementation closely follows the one presented by Bollo[3] in GDC18. Inertialization is implemented as a post processing step to the motion matching and works in two phases:

- Computing the inertialization information when a discontinuity occurs
- Inertialization of a pose following the discontinuity

Computing the Inertialization Information

Whenever a discontinuity occurs the inertialization information is updated for every joint. The initialization information consists of the following components:

- **Discontinuity offset:** $\vec{x}_0$
- **Discontinuity offset magnitude:** x_0
- **Velocity in discontinuity offset direction:** v_0
- **Acceleration in discontinuity offset direction:** a_0
- **Blend time:** t_1
- **Coefficients:** A, B, C

These components are designed to work with both vectors and quaternions.

For vectors: the discontinuity offset $\vec{x}_0$ is computed as the difference between the position of the joint before the discontinuity $\vec{x}_{prev}$ and the position of the joint after the discontinuity $\vec{x}_{curr}$. As well as the difference $\vec{x}_{-1}$ between the last two occurring positions ($\vec{x}_{prev}$ and $\vec{x}_{prev-1}$) before the discontinuity.

$$\vec{x}_0 = \vec{x}_{prev} - \vec{x}_{curr}$$
$$\vec{x}_{-1} = \vec{x}_{prev-1} - \vec{x}_{prev}$$

The magnitude of the discontinuity offset x_0 is given by the vector norm.

$$x_0 = \|\vec{x}_0\|$$

The velocity v_0 is only relevant in the direction of the discontinuity offset. Therefore, the magnitude x_{-1} of the projection of $\vec{x}_{-1}$ onto the discontinuity offset direction is first computed via dot product.

$$x_{-1} = \vec{x}_{-1} \cdot \frac{\vec{x}_0}{\|\vec{x}_0\|}$$

And then, the velocity v_0 is computed by dividing the difference in magnitudes over the time between the previous and current frame Δt.

$$v_0 = \frac{x_{-1} - x_0}{\Delta t}$$

The acceleration a_0 is computed by the following equation using the context specific constants specified by Bollo[3]:

$$a_0 = \frac{-8\, v_0\, \Delta t - 20\, x_0}{\Delta t^2}$$

Since the desired effect of the velocity is to reduce the discontinuity offset magnitude x_0, v_0 is set to zero if they point in the same direction. Analogously since the desired effect of the acceleration is to reduce v_0, a_0 is set to zero if they point in the same direction.

$$v_0 = \begin{cases} 0, & \text{if} \quad v_0\, x_0 >= 0 \\ v_0, & \text{otherwise} \end{cases}$$

$$a_0 = \begin{cases} 0, & \text{if} \quad a_0\, v_0 >= 0 \\ a_0, & \text{otherwise} \end{cases}$$

To prevent overshooting of the target position, the blend time t_1 is limited by:

$$t_1 = \min\left(\frac{-5\, x_0}{v_0}, t_1\right)$$

Then the coefficients A, B, C are computed by the following equations:

$$A = -\frac{a_0\, t_1^2 + 6\, v_0\, t_1 + 12\, x_0}{2\, t_1^5}$$

$$B = \frac{3\, a_0\, t_1^2 + 16\, v_0\, t_1 + 30\, x_0}{2\, t_1^4}$$

$$C = -\frac{3\, a_0\, t_1^2 + 12\, v_0\, t_1 + 20\, x_0}{2\, t_1^3}$$

For quaternions: the orientation difference q_0 between the previous and current orientation (q_{prev} and q_{curr}) is computed, aswell as the difference q_{-1} between the last two occuring orientations (q_{prev} and q_{prev-1}) before the discontinuity.

$$q_0 = q_{prev}\ q_{curr}^{-1}$$
$$q_{-1} = q_{prev-1}\ q_{prev}^{-1}$$

Then, the discontinuity offset $\vec{x}_0$ is obtained by taking the vector part of q_0 and the magnitude of the discontinuity offset x_0 is set to the angular part of q_0.

$$\vec{x}_0 = \text{vec}(q_0) \quad x_0 = \text{angle}(q_0)$$

While the magnitude of the rotation difference before the discontinuity x_{-1} is computed as:

$$x_{-1} = 2 * \arctan(\frac{\text{vec}(q_{-1}) \cdot \vec{x}_0}{\text{angle}(q_{-1})})$$

The rest of the computations are as in the vector case.

Inertialization of a Pose

For both vectors and quaternions the idea is to use velocity v_0 and acceleration a_0 to reduce the magnitude of the discontinuity offset x_0 to zero over the blend time t_1.

The inertialization algorithm uses a fifth degree polynomial to achieve this, where t is the time since the discontinuity occurred:

$$x_t = A\,t^5 + B\,t^4 + C\,t^3 + \frac{a_0}{2}\,t^2 + v_0\,t + x_0$$

The inertialized vector $\vec{x}_t$ is then obtained by:

$$\vec{x}_t = x_t \frac{\vec{x}_0}{x_0} + \vec{x}_{curr}$$

And the inertialized quaternion q_t is obtained by:

$$q_t = \text{quat}(\vec{x}_0, x_t)\ q_{curr}$$

3.5. Synchronization

The animation clips received by the motion matching pipeline do not always strictly follow the controller's trajectory. This can lead to the character lagging behind the controller's position or overshooting it, which makes the character feel unresponsive. One approach to solve this issue is synchronization. Synchronization forcefully sets the character's position to the controller's position, giving a very responsive feeling to the character. Unfortunately, since the character is effectively pulled across the floor towards the controller position, this introduces a lot of foot sliding (see Section 2.3). An improved approach, and the one used in the final implementation, is to interpolate between the synchronized position and the natural trajectory positions of the animation.

$$\vec{x} = (1-\alpha)\,\vec{x}_{animation} + \alpha\,\vec{x}_{controller}$$
$$\text{where} \quad \alpha := \text{ the synchronization factor}$$

This trade-off reduces the amount of foot sliding while retaining a responsive feeling of the character control.

3.6. Foot-Locking and Inverse Kinematics

One way to counteract the remaining foot sliding is through foot-locking. The implemented foot-locking algorithm keeps track of the character's foot positions and locks them into place whenever contact in the currently playing animation clip is made. Contact is defined as the height h and velocity v of the feet being below some pre-defined thresholds t_h and t_v. The foot-locking system will release the foot whenever one of the following criteria is met:

- The contact information of the current frame reports no contact, i.e.

$$h > t_h \quad \text{or} \quad v > t_v$$

- The character's heading changes drastically.
- The distance between the character's foot position in the current animation p_{foot} and the fixed contact point p_{fixed} is above a threshold t_{dist}, i.e.

$$\|p_{fixed} - p_{foot}\| > t_{dist}$$

Inverse Kinematics (IK)

All of this can lead to unnatural leg and foot motion, which is solved by applying a simple two joints IK, as implemented by Holden[9]. This analytical method only takes the hip, knee and heel joint positions ($x_{hip}, x_{knee}, x_{heel}$) and orientations ($q_{hip}\ q_{knee}, q_{heel}$) into account.

First the closest point (p_{target}) to the fixed contact point (p_{fixed}) is calculated. If the fixed point is in reach of the leg, the target is set to that fixed point. The leg is prevented from fully extending using a small buffer ϵ.

$$max_{ext} = \|(x_{hip} - x_{knee})\| + \|(x_{knee} - x_{heel})\| - \ \epsilon$$
$$\text{if } \|p_{fixed} - x_{hip}\| \leq max_{ext} \text{ then}$$
$$p_{target} = p_{fixed}$$

Otherwise the closest reachable point is set as the target.

$$\begin{aligned} x_{normalized} &= \text{normalize}(x_{hip} - p_{fixed}) \\ p_{target} &= x_{hip} + max_{ext}\ x_{normalized} \end{aligned} \tag{6}$$

Afterwards the problem can be represented by two triangles as shown in Figure 2. The current and the target angle (θ_c and θ_t) for the knee are highlighted in the image. The rotation $\theta_{diff} = \theta_t - \theta_c$ along the rotation axis of the knee is calculated and then applied to the knee. A similarly computed rotation is applied to the hip's orientation. Finally the heels are rotated towards the heading direction. This is to ensure that the feet point in the direction of travel - artifact that is sometimes unresolved by the analytical IK solver.

After foot-locking and IK is applied, some discontinuities can appear as the character's leg snaps in or out of the locked state. For this reason inertialization (see Section 3.4) can be applied again to smooth out these transitions.

Uneven Terrain

The implementation of IK also allows to have the charac-

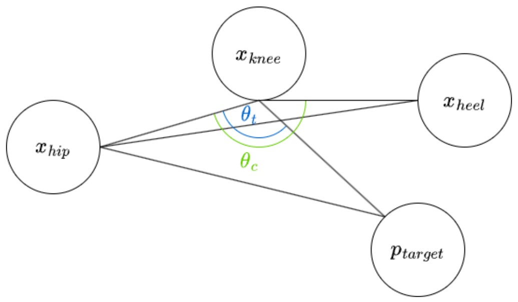

Figure 2. A conceptual image of the triangles, which are looked at during the two joint IK during foot-locking.

ter move over slightly uneven terrain. The feet will not intersect or go through the ground in a significant manner, as once the contact information from the animation clip reports a contact for a foot, the foot-locking system will lock it to a point on the terrain which has the same (x,z)-coordinates as the character's foot position. Additionally every frame the simulation bone's world position (which is 0 on a flat terrain) is ray-casted onto the terrain and then set to the retrieved point.

3.7. Animation Retargeting

To fully integrate the motion matching algorithm into the provided code base the skeleton was retargeted to a character mesh (named Bob). As the lower half of the LAFAN1 mocap skeleton and the Bob skeleton have very similar dimensions, retargeting the animations is a matter of applying the joint angles to Bob. IK is therefore not necessary in this phase.

However, the local rotation quaternions of the joints in the mocap skeleton could not simply be mapped to Bob's joints for several reasons. Firstly, the two skeletons have a different number of joints. Therefore, the rotations of some joints in the mocap skeleton needed to be concatenated to fit the reduced number of joints in Bob.

Secondly, Bob's joints are all revolute, thus requiring the rotation quaternions to be decomposed into separate Euler angles about the joints' local axes.

Thirdly, the x, y and z directions in the coordinate systems of the two skeletons were different. It was hence required for the rotation axes of Bob's joints to be expressed in the frame of reference of the mocap joints before its rotation quaternion could be decomposed into the correct Euler angles.

Lastly, the default pose of the two skeletons were different (see Figure 3). Hence, the angular offset between the mocap joints and Bob's joints needed to be added back to the decomposed Euler angles to ultimately match Bob to the mocap skeleton.

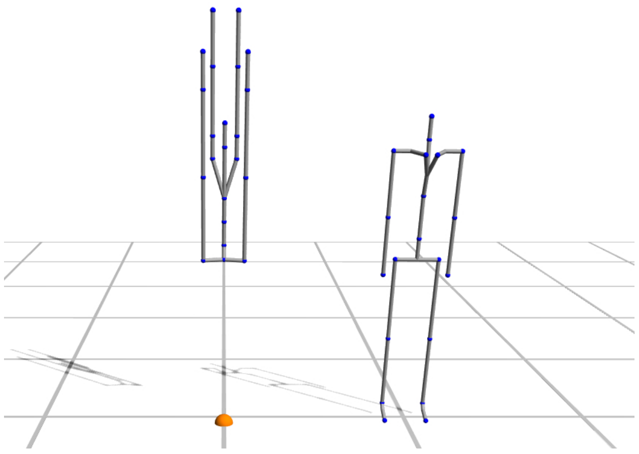

Figure 3. Default zero joint angle pose of the mocap and Bob skeletons. All limbs are pointed upwards in the mocap skeleton.

4. Experiments

4.1. Dataset Pre-Processing Contributions

One of the primary factors used for the motion matching query is the trajectory and position of the root bone after 20, 40 and 60 frames. When using the dataset without the pre-processing described in Section 3.2, motion matching was still achieved, but an increased number of transitions between clips was noticeable. This is undesirable, since the larger the number of transitions, the greater the potential for unwanted artifacts in the final animation.

The filtered simulation bone in the pre-processing module provides a more stable trajectory and position that reduces the number of clip transitions - and thus of blending required. Figure 4 shows a comparison between the hips x and z position (in green), and the simulation bone positions both before (in blue) and after filtering (in orange). The result, though subtle, more accurately represents the motion of the character on the clip. Additionally, the change introduces a root-mean-square error of only $5.9683 \cdot 10^{-3}$, relative to the originally used positions, over the entire aiming1_subject1_walk clip of the LAFAN1 dataset (arbitrarily chosen as an example).

A greater effect can be seen in Figure 5, where a smoother angle transition is present when performing normalization and filtering. The new angles, however, still follow the original general motion of the clip. The resulting bearing is more informative of the one followed by the animation and provides better matches and fewer clip transitions. Similarly, the filtering and smoothing action present a mean-square-error over the original rations of the root of $2.8945 \cdot 10^{-3}$ over the entire aiming1_subject1_walk clip of the LAFAN1 dataset, and thus does not introduce a large deviation.

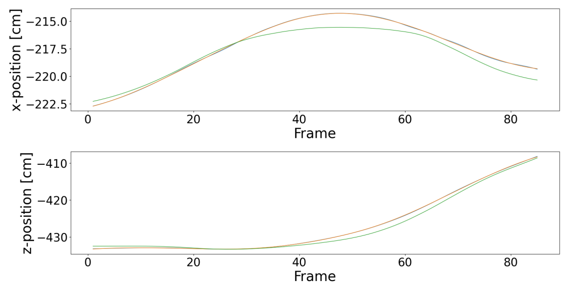

Figure 4. x and z position [cm] plots of 85 frames of the aiming1_subject1_walk clip from the LAFAN1 dataset (up-sampled to 60 FPS). Hips position in green, simulation bone positions in blue and filtered simulation bone positions in orange.

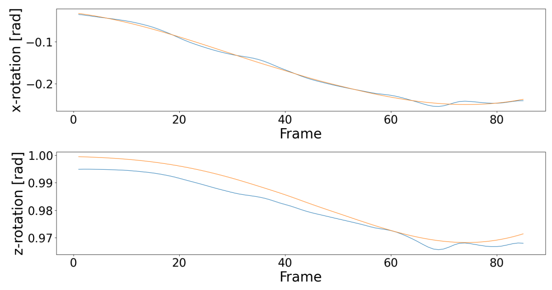

Figure 5. Rotations around x and z-axis [rad] plots of 85 frames of the aiming1_subject1_walk clip from the LAFAN1 dataset (up-sampled to 60 FPS). Simulation bone rotations in blue and filtered simulation bone rotations in orange.

4.2. Motion Matching

The most impactful parameters of the motion matching algorithm are the motion matching frequency, dictating how often the mocap database is searched, and the database feature weights, controling the feature importance on the matching process.

Motion Matching Frequency

Lowering the matching frequency allows each animation to play for longer and look completely natural. A long playtime can, however, result in the character feeling unresponsive. Therefore, a higher motion matching frequency is desirable. On the tested hardware, increasing the motion matching frequency has a negligible effect on computational demand. Compared to a matching frequency of once per second, a matching frequency of 10 times per second increases the average processing time from 0.7% to just 0.9% of the time between frames. Unfortunately, a very high matching rate can lead to unnatural motion where a very short animation section is looped several times, making the character appear almost frozen for some time. This likely occurs because the time passed since the last matching is so short that the character has barely changed its pose and the last match is still the best fit to the desired trajectory. Through experimenting it was determined that a matching rate of 5 times per second yields a good trade-off between responsiveness and natural looking motion.

Trajectory Position & Orientation Weight

The trajectory position and orientation weight control how well the animation trajectory matches the controller trajectory. If the weight is too low, the animation can drift far from the target trajectory and face in directions not aligned with the desired motion. This can be alleviated using synchronization. However, a high degree of synchronization can cause other issues such as foot sliding (see Section 3.5).

Foot Position & Velocity Weight

While it is important that the animation follows the desired trajectory it is equally important that the foot positions and velocities of the found pose match the previous. The importance of the positional weight is very intuitive. The significance of matching velocity can be well illustrated by the fact that a static mid-walk pose is, due to the pendulous leg movement, not indicative of the direction the legs are moving in. Therefore, if only the feet positions are considered for matching, it can very well happen that the walking motion abruptly changes direction.

Hip Velocity Weight

Finally, the hip velocity weight is important to maintain consistency on the type of movement the character performs. The motions for starting to run, walking and stopping can all have very similar trajectories. The hip velocity weight ensures that the type of movement does not suddenly switch to a starting to run animation in the middle of a walk.

Through experimentation it was found that the following weight combination results in visually adequate performance: *trajectory position* = 1.5, *trajectory orientation* = 0.8, *foot position* = 2.0, *foot velocity* = 2.0 *hip velocity* = 2.0.

4.3. Inertialization

The effect of inertialization is demonstrated in the following clip [ii]. The video shows the same motion matching result with and without inertialization. The animation without inertialization (on the left) shows clear discontinuities whenever the animation is changed. The inertialized animation (on the right) shows a smooth transition between animations, resulting in natural motion. The maximum blend time was set to 0.4 seconds which seems to be sufficient for most transitions.

However, the method is not perfect since it is still possible for a short drastic change in pose - such as an arm being raised above the head and then lowered to the side of the body again - to result in smooth but erratic motion of the character. This could be alleviated by filtering out clips that contain arm poses that are too different from conventional walking or running arm poses, however, inclusion of some of these clips can also result in interesting and natural looking motion.

4.4. Synchronization

The balance of synchronization is paramount for a correct trade-off between responsiveness and artifacts reduction. This trade-off can be observed in this clip [iii]. It shows the same motion matching result with different synchronization factor α (from left to right): 0.0, 1.0 and 0.5.

The clip shows that with $\alpha = 0.0$ the character lags behind the controller a lot and when the controller comes to a stop, the character overshoots and corrects itself afterwards. With $\alpha = 1.0$ there is no more lag between the character and the controller. However, the feet of the character slide on the ground, instead of remaining still. With $\alpha = 0.5$ the character does not follow the controller as strictly anymore, especially on turns, where the character's turning radius is larger than the fully synchronized character. But this also makes the turns more realistic and introduces very little foot sliding. At the same time the character does not overshoot too much while still feeling responsive.

4.5. Foot-locking

This clip [iv] shows the effect of applying foot-locking in combination with inertialization. The video shows three skeletons (from left to right): no foot-locking, with foot-locking and with foot-locking & inertialization. The skeleton without foot-locking slides slightly on the floor, which is especially visible on turns. The skeleton with only foot-locking enabled does not slide anymore, but some severe discontinuities are introduced. This demonstrates why inertialization needs to be applied again, the result of which can be seen on the final skeleton. It doesn't slide and is considerably smoother.

The result, while an improvement, doesn't provide a completely smooth motion. Some jitter persists, most certainly, due to the underlying foot-locked animation not being smooth, something that can not be smoothed out by inertialization. This could potentially be alleviated by using an improved IK system over the simple two joints IK.

5. Conclusion

In summary, a working motion matching pipeline that produces natural and realistic locomotion animations for a digital human character was implemented - the repository can be found in GitHub [i]. The character is capable of walking and running in any direction, as well as strafing. Additionally, the character is able to walk on slightly uneven terrain via IK.

A final demonstration of the achieved motion matching animation can be seen in this clip [v] .

Limitations

The quality of the produced motion is difficult to assess objectively. While it looks adequate, small artefacts caused by bad transitions or foot-locking still occurr. Further fine tuning of the parameters is likely to improve the motion and reduce artefacts.

When it comes to steep terrain, the character might have unnatural motions due to the lack of consideration of the environment within the implemented pipeline.

This can be generalized to a broader limitation of motion matching - the quality of the movements are only as good as the quality of the mocap dataset. Gaps in the pose space lead to unnatural motion. This could be overcome by recording more mocap data for a wider range or scenarios.

Future Work

A potential area for future work could be to investigate learning of the mocap dataset using machine learning models. This could not only help to reduce the required memory, but also help fill some of the gaps in the pose space.

Additionally, the utility of motion matching can be explored for actions beyond simple locomotion. These actions might involve extensive interactions with the environment (e.g. climbing stairs, jumping over obstacles) or objects (e.g. kicking a ball, sitting on a chair). Such interactions will require additional features such as collision detection and possibly even physics-based character dynamics.

6. Contributions

The contributions for each module of the motion matching pipeline implementation can be seen in Table 1. Responsible members of each module are stated in the corresponding row, but contribution of all team members was present across each module.

Module	Matthias	Xue Xian	J. Pablo	Saatushan
Gamepad Controller Trajectory		x		
Preprocessed Motion Capture Data			x	
Motion Matching	x			
Inertialization	x			
Synchronization	x			x
Foot-Locking and IK			x	x
Uneven Terrain			x	x
Animation Retargeting		x		

Table 1. Lead responsible members for pipeline modules (x)

Glossary

BVH	BioVision Hierarchy. 3
DoFs	Degrees of Freedom. 3
FPS	Frames per Second. 3, 7
IK	Inverse Kinematics. 2, 5, 6, 8
LAFAN1	Ubisoft La Forge Animation Dataset. 1, 3, 6, 7
mocap	Motion Capture. 1, 3, 6, 7, 8

Links

[i] https://github.com/karstm/motion-matching
[ii] https://drive.google.com/file/d/12Xfv4nOljHPxLRfuKk8dk_ogMDwfH4G3/view
[iii] https://drive.google.com/file/d/1ib9P37oMMmVRFV8glqU94ZrT8t1e3g0F/view
[iv] https://drive.google.com/file/d/1P7FFQ9NKkXreTd31FdH-XqIPle4vkHqk/view
[v] https://drive.google.com/file/d/1hrhJ9yl7PZ5fW10v8ZTqapq7VGh2NFem/view

References

[1] Animate 3D. Animate 3d - v2.6 release: Foot locking modes. https://blog.deepmotion.com/2021/04/22/v26/, 2021. Accessed: 2023-06-10. 2

[2] Erik Bernhardsson. Annoy (approximate nearest neighbors oh yeah). https://github.com/spotify/annoy. Accessed: 2023-06-10. 3

[3] David Bollo. Inertialization: High-performance animation transitions in 'gears of war'. https://www.gdcvault.com/play/1025331/Inertialization-High-Performance-Animation-Transitions, 2018. Last accessed: 2023-06-13. 1, 4

[4] Kenneth Claassen. Foot locking & motion symphony. https://www.wikiful.com/@AnimationUprising/motion-symphony/foot-locking. Accessed: 2023-06-10. 2

[5] Simon Clavet. Motion matching and the road to next-gen animation. https://www.gdcvault.com/play/1023280/Motion-Matching-and-The-Road, 2016. Accessed: 2023-06-10. 1

[6] Jonathan Cooper. *Game Anim: Video game animation explained*, chapter Our Project: Polish and Debug, pages 251–255. CRC PRESS, 2021. Accessed: 2023-06-10. 2

[7] Unreal Engine. Motion symphony. https://marketplace-website-node-launcher-prod.ol.epicgames.com/ue/marketplace/en-US/product/motion-symphony, 2021. Accessed: 2023-06-10. 2

[8] Felix G. Harvey, Mike Yurick, Derek Nowrouzezahrai, and Christopher Pal. Robust motion in-betweening. 39(4), 2020. Accessed: 2023-06-10. 3

[9] Daniel Holden. https://theorangeduck.com/page/simple-two-joint, Jan 2017. Accessed: 2023-06-15. 5

[10] Daniel Holden. https://theorangeduck.com/page/code-vs-data-driven-displacement, Sep 2021. Accessed: 2023-06-10. 2

[11] Daniel Holden. Spring-it-on: The game developer's spring-roll-call dataset. https://theorangeduck.com/page/spring-roll-call, 2021. Accessed: 2023-06-10. 2

[12] Daniel Holden, Oussama Kanoun, Maksym Perepichka, and Tiberiu Popa. Learned motion matching. *ACM Transactions on Graphics*, 39(4), 2020. Accessed: 2023-06-10. 2, 3

[13] David Hunt, Richard Lico, and Michael Buttner. Topics in real-time animation. In *ACM SIGGRAPH 2018 Courses*, SIGGRAPH '18, New York, NY, USA, 2018. Association for Computing Machinery. 1

[14] Jose Luis Ponton. *Motion Matching for Character Animation and Virtual Reality Avatars in Unity*. PhD thesis, Universitat Politecnica de Catalunya, 2022. Accessed: 2023-06-10. 2

[15] Ronald W. Schafer. What is a savitzky-golay filter? [lecture notes]. *IEEE Signal Processing Magazine*, 28(4):111–117, 2011. Accessed: 2023-06-10. 3

[16] Ubisoft. Ubisoft la forge animation dataset ("lafan1"). https://github.com/ubisoft/ubisoft-laforge-animation-dataset, 2017. Accessed: 2023-06-10. 1, 3

[17] Gwonjin Yi and Junghoon Jee. Search space reduction in motion matching by trajectory clustering. In *SIGGRAPH Asia 2019 Posters*, SA '19, New York, NY, USA, 2019. Association for Computing Machinery. 1