# AUTOMATION TOOLS FOR THE ANIMATION PIPELINE

Maksym Perepichka

A thesis
in
The Department
of
Computer Science  Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

May 2020

© Maksym Perepichka, 2020

# Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Maksym Perepichka**

Entitled: **Automation Tools for the Animation Pipeline**

and submitted in partial fulfillment of the requirements for the degree of

**Master of Computer Science**

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

| Name | Role |
|---|---|
| Sudhir Mudur | Chair |
| Sudhir Mudur | Examiner |
| Marta Kersten | Examiner |
| Tiberiu Popa | Supervisor |

Approved ______________________________

Latar Narayanan, Chair

Department of Computer Science and Software Engineering

______________ 2020 ______________________________

Amir Asif, Dean

Faculty of Engineering and Computer Science

# Abstract

Automation Tools for the Animation Pipeline

Maksym PEREPICHKA

Video-games and animated movies require a sophisticated multistage set of processes known as the animation pipeline for collecting animation data, beginning with actors in a Motion Capture Studio and ending in animated digital characters. Throughout the animation pipeline, varying levels of manual human intervention are typically needed to ensure animation quality. Passive markers used for Motion Capture require manual cleanup by trained MOCAP artists to fix issues such as marker occlusions, marker swaps, and noise. This thesis proposes a novel method to automate this process that works by identifying broken marker path segments and subsequently reconstructing broken markers using a kinematic reference. The result is a state-of-the-art method that outperforms existing solutions by being simultaneously more accurate as well as easier to integrate into existing animation pipelines. Once marker data is cleaned, studios will often want to retarget the captured data onto different characters, a step that usually requires the manual tweaking of various retargeting parameters in proprietary software. This thesis proposes a batch mesh-based retargeting algorithm that uses Jacobian Inverse Kinematics tracking mesh vertices to retarget animations between different skeletal rigs. This results in an efficient algorithm that is capable of retargeting multiple animation clips without requiring the manual tweaking of parameters specified.

# Acknowledgments

I would like to acknowledge the help of my supervising professor Tiberiu Popa, as well as my industry supervisor Daniel Holden for their enormous help and guidance throughout the last two years. My Master's degree would not have been a success without their feedback and ideas. I would also like to thank everyone at Ubisoft La Forge and Ubisoft Alice for providing resources, reference data, as well as vital feedback. I also acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).

"*Animation can explain whatever the mind of man can conceive. This facility makes it the most versatile and explicit means of communication yet devised for quick mass appreciation.*"

Walt Disney

# Contents


**List of Figures** viii

**List of Tables** xi

**1 Introduction** 1

1.1 Computer Animation . . . 1
1.2 Motion Capture . . . 2
1.3 Animation Pipeline . . . 4
1.3.1 MOCAP Shoot . . . 4
1.3.2 Marker Tracking . . . 4
1.3.3 Marker Solving . . . 4
1.3.4 Animation Retargeting . . . 5
1.3.5 Delivery . . . 5
1.4 Contribution . . . 5
1.4.1 Motion Capture Cleanup . . . 6
1.4.2 Animation Retargeting . . . 7

**2 Background** 9

2.1 Animation Representation . . . 9
2.2 Forward Kinematics . . . 10
2.3 Inverse Kinematics . . . 11
2.4 Linear Blend Skinning . . . 12

**3 Robust Marker Trajectory Repair for MOCAP using Kinematic Reference** 14

3.1 Introduction . . . 14

3.2 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.3 Method . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.3.1 Kinematic Solver Improvements . . . . . . . . . . . . . . . . . . 19
3.3.2 Marker Reconstruction . . . . . . . . . . . . . . . . . . . . . . 20
3.3.3 Erroneous Marker Interval Detection . . . . . . . . . . . . . . . 21
3.3.4 Gap Filling . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
3.4 Results and Discussion . . . . . . . . . . . . . . . . . . . . . . . . 24
3.4.1 Comparison . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
3.4.2 Parameter Selection . . . . . . . . . . . . . . . . . . . . . . . . 26
3.4.3 Performance . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

**4 Mesh Based Animation Retargeting** **34**
4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
4.2 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.3 Method . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
4.3.1 Subsampling . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
4.3.2 Mesh Mapping . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
4.3.3 Normalization . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
4.3.4 Blend Mask . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
4.3.5 Cumulative Weights . . . . . . . . . . . . . . . . . . . . . . . . 41
4.3.6 Jacobian . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
4.3.7 Error . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
4.3.8 Solving . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
4.4 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
4.5 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45

**5 Conclusion** **54**

**Bibliography** **56**

# List of Figures


1 Image of me in a MOCAP suit. . . . . . . . . . . . . . . . . . . . . . . 3
2 A mesh rigged with a skeleton, with mesh deformation being driven by the skeleton deformation via LBS. . . . . . . . . . . . . . . . . . . . . 12
3 Marker paths visualized alongside a kinematic solution in 3D . . . . . 14
4 Overview of entire pipeline. a: Initial corrupted marker data. b: Solved skeleton using NN. c: Solved skeleton using commercial software. d: Reconstructed marker paths using LBS. e: Comparison of kinematic solutions to determine erroneous markers. f: Filling of incorrect marker paths. g: Filled marker paths. h: Final kinematic solution . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
5 Example of marker filling from reference path . . . . . . . . . . . . . 30
6 Marker paths before and after gaps in valid data . . . . . . . . . . . . 30
7 3D visualization of the filled marker paths passed through a kinematic solver ($\bar{\mathbf{Y}}$). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
8 2D visualization of the left upper leg joint from the filled marker paths passed through a kinematic solver. . . . . . . . . . . . . . . . . . . . 31
9 Visualization of a marker along with a 3D visualization of the corresponding kinematic solution. . . . . . . . . . . . . . . . . . . . . . . 32
10 3D visualization of the filled marker paths passed through a kinematic solver ($\bar{\mathbf{Y}}$). Hard example with two characters wrestling and numerous erroneous markers. . . . . . . . . . . . . . . . . . . . . . . . . . 32
11 Visualization of a marker along with a 3D visualization of the corresponding kinematic solution. . . . . . . . . . . . . . . . . . . . . . . 33
12 Two rigged meshes overlayed on each-other, with the animation from the green character being retargeted onto the red character. . . . . . 34

13 Autodesk MotionBuilder animation software. Character tool shown on right hand side, displaying mapping to an intermediate skeleton. . . . 36
14 Three subsampled versions of the same mesh. The squares indicate the vertices chosen by the subsampling algorithm. The leftmost mesh is subsampled at 128:1 vertices, middle one at 64:1, and rightmost one at 32:1. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
15 Illustration demonstrating the cumulative weights. $A, B, C, D, E$ represent joints in a hierarchy, while $V$ is a vertex. The $\mathbf{w}$ represents the skinning weight between the two while $\hat{\mathbf{w}}$ represents the cumulative skinning weights. . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
16 Illustration motivating the derivation of $\frac{\partial \mathbf{P}_t^i}{\partial \theta_j}$. The red points visualizes the vertex position $\mathbf{P}_t^i$ while the blue arm visualizes a joint hierarchy starting at $\mathbf{T}_t^j$. $\Delta\mathbf{P}_t^i$ represents the displacement of the vertex position as as a result of the rotation of joint $\mathbf{T}_t^j$ around axis $\mathbf{r}_t^j$ by angle $\Delta\boldsymbol{\theta}_t^j$ 43
17 Illustration motivating the derivation of $\frac{\partial \mathbf{N}_t^i}{\partial \theta_j}$. The red points visualizes the vertex normal $\mathbf{N}_t^i$ while the blue arm visualizes a joint hierarchy starting at $\mathbf{T}_t^j$. $\Delta\mathbf{N}_t^i$ represents the displacement of the vertex normal as as a result of the rotation of joint $\mathbf{T}_t^j$ around axis $\mathbf{r}_t^j$ by angle $\Delta\boldsymbol{\theta}_t^j$ 44
18 Figure demonstrating mesh based retargeting from the source character (green) to the target character (red). . . . . . . . . . . . . . . . . . . . 46
19 Figure demonstrating mesh based retargeting on different scale characters, from the source character (green) to the target character (red). Blue lines on the hand indicate vertex position constraints. . . . . . . 47
20 Figure demonstrating mesh based retargeting on different scale characters, from the source character (green) to the target character (red). 48
21 Figure demonstrating mesh based retargeting from the source character (green) to the target character (red). . . . . . . . . . . . . . . . . . . . 49
22 Figure demonstrating mesh based retargeting from the source character (green) to the target character (red) for a walking motion. . . . . . . 49
23 Figure demonstrating mesh based retargeting from the source character (green) to the target character (red) for a kneeling down motion. . . . 49
24 Figure demonstrating mesh based retargeting from the source character (green) to the target character (red) for a walking motion. . . . . . . 50

25 Figure demonstrating mesh based retargeting from the source character (green) to the target character (red) for a jogging motion. . . . . . . . 53

# List of Tables

1 Comparison between different marker filling methods in terms of the mean per-frame error between their kinematic solutions and the ideal path recreated by an artist. $\Delta_{\text{pos}}$ represents the error between the global joint positions (in cm) while $\Delta_{\text{rot}}$ represents the error between the global joint rotations (in degrees). . . . . . . . . . . . . . . . . . . . 24

2 Performance of our pipeline computed on a typical sequence of 10,920 frames. Percentages calculated using largest remainder method. . . . 27

3 Relative performance metrics for a single iteration of the retargeting algorithm, separated by the individual components. Percentages are rounded to the nearest integer. . . . . . . . . . . . . . . . . . . . . . . . 47

4 Performance metrics of processing time of one 4000 frame clip based on the subsampling rate. . . . . . . . . . . . . . . . . . . . . . . . . . 48

# Chapter 1

# Introduction

## 1.1 Computer Animation

Computer animation can be defined as the process of digitally generating an animated sequence, whether it be of images, 3D poses, or other formats. Much of modern computer animation take its roots from traditional animation, where artists would draw animated scenes frame-by-frame and then record these sequences of images. Another important source of inspiration is stop-motion animation, where artists would manipulate 3D objects by small increments, take still-pictures at each time-interval, and play them back frame-by-frame.

The digitization of this process allowed greater robustness for artists working in the field. For traditional 2D frame animation, this meant that animators could now save time that they would traditionally spend redrawing frames by using the computer to load previously created frames and slightly modify them. For 3D animations, the development of digital 3D graphics resulted in the ability to display realistic 3D scenes capable of accurately representing perspective. Furthermore, borrowing from the aforementioned work in stop-motion animation, animated 3D characters could now be created, allowing artists to manipulate theirs joints on a frame-by-frame basis.

This idea was expanded upon with the introduction of keyframe animation. Artists no longer had to define each individual frame of a particular animation. Instead, artists would define keyframes which would define transition points of an animation. For instance, if an artist wants to represent a human character moving their arm in a circular fashion for 1 minute at 24 FPS, instead of defining the motion using

$24 \times 60 \times 60 = 8640$ frames, they would simply define a handful of keyframes at spaced out intervals and the computer would automatically fill in the rest by interpolating between them.

This explosion in efficiency resulted in an increase of more ambitious projects being undertaken throughout the animation industry. With the release of *Toy Story* 1995 as the first fully computer-animated full feature, a new standard of quality was created.

Likewise, the gaming industry also took notice of this technological evolution. Throughout the 70-80s, videogames would usually represent 2D scenes, where animation was done using sprites. However, with the development of 3D graphics, games representing 3D scenes were now becoming possible. Two notable examples of this were *Wolfenstein 3D* 1992 and *Doom* 1993, which although not technically being first to-do-so, introduced 3D looking environments to players. Although these games didn't technically utilize 3D scenes as we define them today and used 2D sprites for much of their content, they forever changed the gaming landscape. Games that followed, such as *Quake* 1996, would now use full 3D scenes, where weapons, enemies and the surrounding world would be represented in full 3D. Most importantly, game characters were now represented in 3D, which greatly complicated the task of animating them compared to animating traditional sprites.

The new scale of projects in both film and videogames created a demand for 3D animation. Even with keyframing and other time-saving techniques, animating was a long and often tedious process. If only there were a way of capturing motion in the real world and transferring it to the 3D scenes and characters represented in these mediums. With this idea, motion-capture was born.

## 1.2 Motion Capture

Although the idea of capturing motion and transferring it to digital scenes was an obvious one, the implementation of this technique was less-so. Many competing techniques emerged as methods to undertake this task.

One approach involves using various forms of sensors to track human motion. An actor would wear a suit with gyroscopes, accelerators and other sensors embedded throughout. Motion could then be recreated by integrating the acceleration data

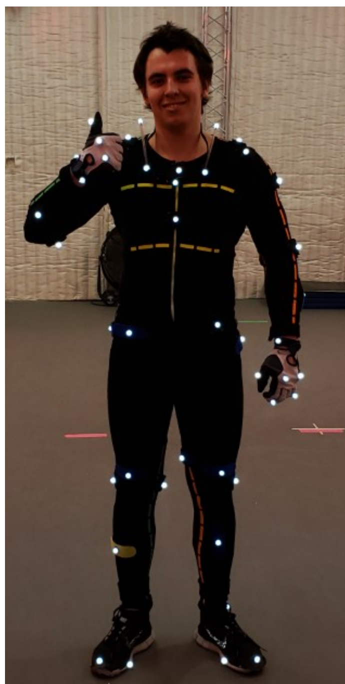

Figure 1: Image of me in a MOCAP suit.

received from these sensors.

Another approach was to use optical systems to track human motion. Since motion was difficult to extract from raw video, marker-based systems were created where human characters would wear suits with optical markers attached to them, such as the one seen in Figure 1. The system would then only need to track these markers in-order to recreate human motion. These systems could be further subdivided into both active and passive markers. Active markers would use different colors or patterns within the markers to allow the system to disambiguate between different markers. Passive systems would have identical markers, with the role of disambiguation left completely to the system.

Due to a combination of factors including ease of set-up, marker redundancy, and accuracy, passive marker-based optical systems remain the go-to method of motion capture for AAA (high production value) video-game studios today, over 20 years after their initial introduction. Video-game studios will often have their own in-house capture systems, being able to record actors performing various movements.

However, even with the benefit of being able to capture human motion and transfer it into digital 3D scenes, this does not make the whole process seamless. A large

number of individual steps must be undertaken before an actor's motion can be used in a video-game. The sum of these processes is known as the animation pipeline.

## 1.3 Animation Pipeline

### 1.3.1 MOCAP Shoot

The animation pipeline starts with a motion capture shoot. Studios will often list out particular actions that they want for theirs games and actors are hired to act out these sequences of motions. Arrays of static infrared cameras record the actors doing their movements, with a MOCAP technician typically being present to supervise this process. The zone being recorded by the cameras is typically referred to as the MOCAP volume. The number of cameras varies based on a number of factors, such as the desired precision and the size of the volume.

### 1.3.2 Marker Tracking

The next step after a motion is captured is known as the tracking phase. Marker tracking involves taking as input the video stream from the MOCAP cameras and solving for the marker positions in 3D space. This process is typically done in parallel with the MOCAP shoot so that technicians are able to spot issues as they arise. Once a shoot is completed, MOCAP artists will review the generated marker positions and correct any errors present in the data.

### 1.3.3 Marker Solving

The solving stage involves taking as input the 3D marker positions and generating a set of hierarchical joint rotations that define the movement of a character. Since character motion in most modern video-games is represented in this format, this is a vital step in the process. As with the last step, this is often done in parallel with the MOCAP shoot. An initial ROM (Range of Motion) is done in-order to determine certain characteristics such as joint lengths. Then, marker positions are mapped onto joint rotations. As with the marker tracking phase, once a shoot is complete, MOCAP artists need to review the data and correct any errors presents.

### 1.3.4 Animation Retargeting

While the data is now represented as a set of hierarchical joint transformations that is usable by games, the animation pipeline is not necessarily over. Game companies will often want to represent characters of differing proportions without necessarily hiring actors of those proportions. The animation retargeting phase of the pipeline involves the problem of transferring animation data from one skeletal structure to another. In the case where their proportions and scales match, this problem is trivial and simply involves copying over the joint angles. In all other cases involving differing proportions, scales, contacts, and props, this problem becomes much more challenging. For AAA games, an artist is usually involved in this step as well, as retargeting animation between sufficiently different characters can lead to ambiguity. For instance, if we wish to retarget an animation of a tall character holding a weapon at shoulder height onto a shorter character, it is ambiguous whether or not we want the weapon to move or the character to move.

### 1.3.5 Delivery

Finally, once all previous steps are completed, the animation data is delivered to a studio and ready for use. Sometimes, when a MOCAP shoot is done and animators aren't quite satisfied with the exact nature of a certain pose, they will manually edit sections of the animation data using keyframes in-order to better conform to their artistic vision.

## 1.4 Contribution

A fully automated animation pipeline has been the goal for animators and game studios. However, as described, the animation pipeline is large with several individual interdependent components. Each component requires various levels of manual intervention by specialized artists and has rigorous standards of quality. Any automated solution needs to not only solve the problems in each component, but do so without sacrificing quality and with sufficient robustness as to allow for manual intervention if necessary. All of this makes the automation of the entire pipeline a challenging endeavor. This thesis works towards the goal of an automated pipeline by tackling

two specific challenges: MOCAP cleanup and Animation Retargeting.

### 1.4.1 Motion Capture Cleanup

During the marker tracking and solving phases, several issues can arise with the input marker data. Optical markers are prone to becoming occluded from camera view, due to limbs, other actors, or props blocking the camera's line of sight. Obstructed markers can cause problems in two ways. If the system fails to properly detect a marker as obstructed, it will retain the position it had when it was last seen. When the marker reappears, it will suddenly pop to its new positions, resulting in jittery unrealistic movements. Second, even if the system properly detects a marker as being obstructed, it can still fail if enough markers get obstructed simultaneously, since there won't be enough marker data to input into the marker solving phase. Markers coming within close proximity with one another are susceptible to being swapped. For instance, if an actor passes his wrist alongside his hips, the system might believe that the wrist marker is the hip marker and vice-versa. In this scenario, the system will then utilize this erroneous information to construct an unusable skeleton animation. Swaps can occur with only one actor in the scene and result in corrupted data, but are even worse in situations with multiple actors. For instance, in the case where two actors are wrestling, markers can get swapped between the two actors, resulting in broken motion for both characters. An example of this can be seen in Figure 10 from Chapter 3. Markers are also vulnerable to high-frequency noise, which render their positions inaccurate and result in a corrupted skeleton animation. Such noise can from several sources, including but not limited to dirty or unfocused camera lenses. Existing methods to solve these problems are problematic, either generating solutions of insufficient quality or generating solutions that are hard to integrate into existing animation pipelines. All these issues typically require manual intervention, with artists manually analyzing marker curves and correcting them on a case-by-case basis.

The contribution to this problem is a novel method for detecting and repairing broken segments of MOCAP marker data. Unlike existing methods, it results in high quality marker cleanup while allowing for artist intervention in cases of system failure. Using a state-of-the-art kinematic solver as a reference to detect broken segments of marker data, broken markers are subsequently reconstructed using Linear Blend

Skinning from the kinematic solver and blended with the remaining markers using a spline-based blending algorithm. The result is a set of markers where corrupted markers are reconstructed and uncorrupted markers are untouched. This work was were submitted to MIG 2019 under the title "Robust Marker Trajectory Repair for MOCAP using Kinematic Reference".

### 1.4.2 Animation Retargeting

Animation retargeting is a process that requires a significant amount of manual work by artists, which can be attributed to multiple factors. Typically, existing retargeting solutions can achieve impressive result, but require manual selection of dozens to hundreds of parameters. These parameters control aspects of the retargeting such as which sections of the skeleton are given priority in the solving process. These parameter aren't trivial to select, and will vary depending on input character structure. Furthermore, even for an identical character structure, there exists a range of viable parameters. For example, if retargeting an animation of a short character shooting a weapon onto a tall character, one can think of two different sets of parameters. The first set would preserve the weapon's global position by sacrificing animation fidelity, while the second would preserve animation fidelity by sacrificing the weapons position. This inherent ambiguity makes animation retargeting a difficult area to automate. This problem is magnified when old animations are reused. Unlike with new MOCAP shoots where it is possible to justify the cost of manual animation retargeting, this is not the case for old animation data. In these cases, a batch retargeting algorithm is desirable, that can efficiently process hundreds of animated clips and output decent quality animations.

The contribution to this problem is a mesh based retargeting algorithm that is sufficiently robust to choice of input skeleton and in capable of retargeting many different animation clips in an efficient manner. Starting with a source skeleton containing the animation and a target skeleton onto which we wish to retarget, each skeleton is rigged with its own mesh. The mesh is subsampled and a mapping is created between the source and target meshes. Then, Jacobian Inverse Kinematics is used to compute joint angles for the target skeleton results in the target mesh vertices tracking the source mesh vertices. The result is a retargeting algorithm that is robust to different skeletal structures and requires minimal manual intervention. Although

not always rivaling the quality of manually selected parameters, it allows for a good quality automatic solution for cases where manual intervention is not possible.

# Chapter 2

# Background

Within the computer animation community, there are several omnipresent techniques that deal with processing animation data. This chapter aims to recap some of the concepts that appear in Chapters 3 and 4.

## 2.1 Animation Representation

There exists a multitude of ways for representing animated 3D characters. One popular method used predominantly in the games industry relies on storing animation on a hierarchical structure known as a skeleton. Skeletons can take on various sizes, proportions, and structures, but for humanoid characters, usually roughly resemble a simplified version of the human skeleton. Animated sequences are then expressed as skeletal deformations, with joints being able to rotate with 3 degrees of freedom. For a joint hierarchy with $nj$ number of joints, a pose can be expressed in terms of joint translations and rotations with respect to their parent joints $\mathbf{LP} \in \mathbb{R}^{nj\times 3}, \mathbf{LR} \in \mathbb{R}^{nj\times 4}$. Note that joint rotations $\mathbf{LR}$ are expressed using quaternions. These translations and rotations are described with respect to the reference frame of their parent joint, and thus known as local transformations. Translation and rotation information can also be combined into a homogeneous transformation matrices $\mathbf{T} \in \mathbb{R}^{nj\times 3\times 3}$.

The notation can be extended to animated sequences by adding another axes to our representation, defining $\mathbf{LR} \in \mathbb{R}^{n\times nj\times 4}$, where $n$ represents the number of frames in a particular animation. Although it is possible for joint translations to also vary over time, due to constraints imposed by many game-engines, joint lengths are

assumed to be fixed in this thesis.

## 2.2 Forward Kinematics

Although describing animation using local reference frames is convenient, it is often useful to have access to the global joint positions. The process of going from local joint transformations to global joint transformations in a hierarchical structure is known as forward kinematics (**FK**). Although there are multiple algorithms for achieving this, the simplest method is to iterate over a joint hierarchy starting at the root joint, computing the global transformations of each of the roots descendants one at a time, as seen in Algorithm 1. Note that this particular implementation assumes that the joints are stored in a preorder traversal of the joint hierarchy. Note that the function $Parent(i)$ returns the index of joint $i$'s parent within the hierarchy.

**Algorithm 1** Pseudocode for a forward kinematics algorithm. Taking as input the set of local translations $\mathbf{LP}$ as well as quaternion rotations $\mathbf{LR}$ it outputs the set of global translations and rotations $\mathbf{GP}, \mathbf{GR}$.

**Function** *FK* $(\mathbf{LP} \in \mathbb{R}^{nj\times 3}, \mathbf{LR} \in \mathbb{R}^{nj\times 4})$
  // Create initial empty global transformation arrays
  $\mathbf{GP} \in \mathbb{R}^{nj\times 3}, \mathbf{GR} \in \mathbb{R}^{nj\times 4} \leftarrow \emptyset$
  // Fills first values
  $\mathbf{GP}^{0,\cdots}, \mathbf{GR}^{0,\cdots} \leftarrow \mathbf{LP}^{0,\cdots}, \mathbf{LR}^{0,\cdots}$
  // Iterate over predefined number of joints
  **for** $j = 1...nj$ **do**
    // Get index of parent joint
    $pj \leftarrow \text{Parent}(j)$
    // Compute global positions
    $\mathbf{GP}^{j,\cdots} \leftarrow \mathbf{GR}^{pj,\cdots}\mathbf{T}^{j,\cdots} + \mathbf{GP}^{pj,\cdots}$
    // Compute global rotations
    $\mathbf{GR}^{j,\cdots} \leftarrow \mathbf{GR}^{pj,\cdots}\mathbf{LR}^{j,\cdots}$
  **end for**
  **return** $\mathbf{GP}, \mathbf{GR}$
**End**

## 2.3 Inverse Kinematics

Inverse kinematics (**IK**), as the name suggests, is essentially the opposite of forward kinematics. Instead of going from local joint transformations to global joint transformations, we wish to go from global joint positions to local joint angles. Unlike FK which is relatively straight-forward, computing IK is usually more complicated. For the simplest cases where a joint hierarchy only has 2 joints, IK can be computed analytically. For joint hierarchies such as the ones used for character animations, iterative approaches are usually used to approximate a solution. Depending on the problem, an exact solution might not exist. One such method for numerically solving this problem is known as Jacobian Inverse Kinematics, extensively detailed in S. Buss, 2004 and summarized in this section. The Jacobian $\mathbf{J} \in \mathbb{R}^{3nt \times 3nj}$ is defined as a matrix of partial derivatives of $nt$ number of end-effectors translations with respect to changes in $nj$ joint angles $\theta$. End-effectors are defined as as a set of joints within the hierarchy that are chosen to be tracked. Typically, $nt <= nj$, but this is not necessarily the case (as seen in Chapter 4). $nt$ is multiplied by 3 for each positional axes (x,y,z), while $nj$ is multiplied by 3 for each rotational axes of the given joint. The individual entries of the Jacobian matrix can be expressed as follows:

$$\mathbf{J}^{i,j} = \frac{\partial \mathbf{p}_i}{\partial \boldsymbol{\theta}^j} \tag{1}$$

The derivation of the individual partial derivatives will vary according to several factors, most notably the format of the joint rotations. The follow is the the partial derivative for rotational joints:

$$\frac{\partial \mathbf{p}^i}{\partial \boldsymbol{\theta}^j} = \mathbf{r}^j \times (\mathbf{s}^i - \mathbf{p}^j) \tag{2}$$

In equation 2, $\mathbf{r}^j$ represents the rotation axis of joint $j$, $\mathbf{s}^i$ represents the end-effector position, $\mathbf{p}^j$ represents the joint positions, and $\times$ represents the cross-product operation.

From this, we wish to derive the joint angles $\boldsymbol{\theta}$ such that the previously mentioned end-effectors $\mathbf{s}$ match their target positions $\mathbf{s} \in \mathbb{R}^{nt \times 3}$. This can be achieved through an iterative approach, solving the equation:

$$\mathbf{t}^i - \mathbf{s}^i = \mathbf{J}\Delta\boldsymbol{\theta} \tag{3}$$

Various methods exists for solving this equation, one of which is known as Damped Least Squares:

$$\Delta\theta = \mathbf{J}^\top(\mathbf{J}\mathbf{J}^\top + \lambda I)^{-1}(\mathbf{t} - \mathbf{s}) \tag{4}$$

In this equation, $\lambda$ is known as the damping factor, a small non-zero constant that ensures that the resulting matrix is invertible, as well as adding stability to the system.

## 2.4 Linear Blend Skinning

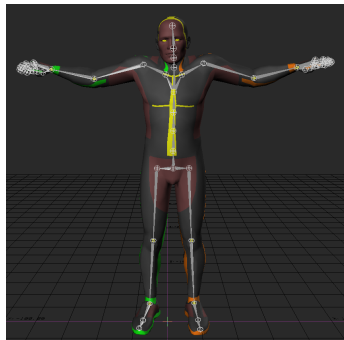

Figure 2: A mesh rigged with a skeleton, with mesh deformation being driven by the skeleton deformation via LBS.

Although joint hierarchies are used to represent characters in videogames, they aren't seen by the end-user. Instead, a process known as rigging is used to fit the skeleton into a mesh (seen in figure 2). The deformation of this mesh is then driven by the skeletal deformations, through a process called skinning. Linear Blend Skinning (**LBS**), also known as pose space deformation as described in Lewis, Cordner, and Fong, 2000, is a simple skinning technique that is used generate mesh deformations from skeletal deformations. LBS defines mesh vertex positions are a linear combination of joint deformations. For a joint hierarchy with transformation matrices $\mathbf{T} \in \mathbb{R}^{nj\times3\times3}$, a mesh with vertex positions $\mathbf{v} \in \mathbb{R}^{nv\times3}$ vertices, a set of skinning weights $\mathbf{w} \in (0,1)^{nj\times nv}$ is used to determine how much each joint affects each vertex.

The modified vertex position $\hat{\mathbf{v}} \in \mathbb{R}^{v \times 3}$ can be derived as follows:

$$\hat{\mathbf{v}}^j = \sum_{i=1}^{nj} \mathbf{w}^{i,j} \mathbf{T}^j \mathbf{v}^j \tag{5}$$

Another interesting problem is computing normals for mesh vertices, labeled $\mathbf{n} \in \mathbb{R}^{v \times 3}$. Typically, due to a problem described by Tarini, Panozzo, and Sorkine-Hornung, 2014, the following does not hold:

$$\hat{\mathbf{n}} j = \sum_{i=1}^{nj} \mathbf{w}^{i,j} \mathbf{T}^j \mathbf{n}^j \tag{6}$$

Although this would normally be a problem, through experimentation, we determined that this does not impact this particular use case in Chapter 4, due to large amount of normals being tracked and the relatively low rate at which the errors occur. As such, equation 6 is used in Chapter 4 even though it does not necessarily hold in the general case.

The LBS algorithm is further simplified with an assumption that each vertex is only affected by a maximum of $n$ joints, where $n = 4$ for most practical applications. Due to its simple nature, there are many issues with LBS which have resulted in many other skinning techniques being proposed in academic literature. However, the simplicity of LBS coupled with its ease of implementation has resulted in it still being one of the principal methods used by game companies.

# Chapter 3

# Robust Marker Trajectory Repair for MOCAP using Kinematic Reference

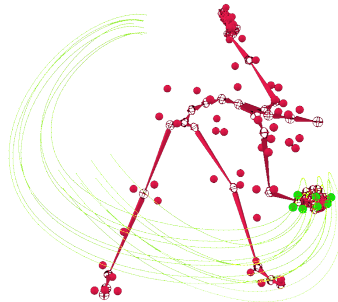

Figure 3: Marker paths visualized alongside a kinematic solution in 3D

## 3.1 Introduction

Marker based motion capture systems (MOCAP) are still the most popular way to capture kinematic motion. Current commercial MOCAP systems achieved commercial maturity and scalability such that they are used ubiquitously in movie and game productions. Some MOCAP systems now can deploy hundreds of cameras and track in real-time a very large numbers of markers from several simultaneous actors.

One inherent problem with marker based systems is that their temporal trajectories have significant gaps and errors. Gaps in the data are usually due to occlusions and self-occlusions, while trajectory errors frequently occur during capture when marker paths cross each other creating an ambiguity that can lead to the MOCAP system swapping their trajectories (Begon, Wieber, and Yeadon, 2008). Trajectory errors also frequently occur when markers can drop from the body during high energy motion. To address these issue there are two problems to be solved: detecting the markers and the time interval where trajectory errors occur and fill in the trajectory data when it is missing or it is incorrect.

Both these problems can be handled fundamentally in two ways: at the kinematic level (Holden, 2018) or at the marker level (Aristidou, Cohen-Or, et al., 2018). A kinematic level solution does not detect or fill in the geometric trajectories of specific markers, but rather computes directly the kinematic motion in a holistic way using all the information available, including the potentially missing and erroneous trajectories. A marker level solution focuses on each marker trajectory individually to detect trajectory errors and fills in the erroneous or missing trajectories in their geometric space.

For most MOCAP applications the marker tracking is simply a transitional step in obtaining the kinematic motion of an articulated model that is usually expressed as joint angles. Therefore, kinematic approaches have the advantage that the reconstruction can be framed as a regression problem that is more robust as the marker set used contains many markers and has some redundancy built in. The disadvantage of this approach is that the new kinematic motion does not always conform to the marker trajectories, even for the markers which had initially correct trajectories. This is a major problem for production teams that sometimes require the actor to do a precise motion that may be lost in the process. Addressing the problem at a marker level is very challenging because marker trajectories are highly correlated. Another issue is that, for cases where the kinematic approach fails to create an adequate solution, the motion becomes difficult to manually fix, as manual marker tracking artists are trained to work by modifying the marker paths which are no longer available.

In this work we present a novel method for the detecting and filling of marker trajectories that leverages the benefits of both approaches. It uses a state-of the-art robust kinematic solver (Holden, 2018) to construct an initial kinematic motion.

Using this motion as a reference, our method detects erroneous trajectories and fills erroneous and missing trajectories by transferring the paths from the kinematic solver in a shape preserving way. We show that our method is robust and outperforms state-of-the-art techniques.

The rest of the paper is structured as follows. Section 3.2 presents related work. Section 3.3 presents an overview of the method. Section 3.4 presents ours results and comparisons with alternatives methods. Chapter 5 concludes and presents possible avenues for future work.

## 3.2 Related Work

MOCAP data is typically acquired as time-synchronized 3D-trajectories of a fairly large number of markers. The problem of MOCAP clean-up is identifying the portions of these trajectories that contain errors and correcting them.

There are several ways to look at this data. One is as a high-dimensional time-series and frame the problem as a smoothing problem as seen in X. Liu et al., 2014. Another is to model the problem as a dynamic system and use variations of Kalmann filters (Julier and Uhlmann, 1997; Wan and Van Der Merwe, 2000) to optimize for the unknown trajectories (H. J. Shin et al., 2001; Tak and Ko, 2005; Li, McCann, N. S. Pollard, et al., 2009; Aristidou and Lasenby, 2013). Kalman filter methods can be improved by adding additional kinematic constraints, seen in Herda et al., 2000; Gleicher, 2001; Li, McCann, N. Pollard, et al., 2010.

Aggregating all trajectories into one data-stream makes the data monolithic, unscalable and difficult to handle. In contrast, the data can be decoupled based on markers and be seen as a set of time-parameterized 3D curves yielding to a geometric view of the problem. However, naive geometric filling (Lee and S. Y. Shin, 1999) (i.e. spline curves) is suitable for very short segments, but it fails for long sequences because the trajectories over large periods of time are complex and highly correlated with the trajectories of adjacent markers. This trajectory correlation can be modeled in several ways. The correlation between marker trajectories can be modeled using PCA, seen in G. Liu and McMillan, 2006; P. A. Federolf, 2013; Gløersen and P. Federolf, 2016 or using probabilistic model averaging estimating the distance from one marker knowing the trajectories of the rest of the markers (Tits, Tilmanne, and

Dutoit, 2018). Most of these methods work very well if the data corruption is limited to one or two markers, but they are understandably less robust in the presence of multiple marker failures.

Another approach to address this limitation is to use a database of existing MOCAP clips and extract from them the best fitting trajectory (Hsu, Gentry, and Popović, 2004; Baumann et al., 2011; Shen et al., 2012; X. Wang, Chen, and W. Wang, 2014; Zhang and Panne, 2018). This works well if the motion already exists in the database. To address the existence of a database, Aristidou, Cohen-Or, et al., 2018 present a data-driven method based on self-similarity.

With the recent advances in deep-learning there are a number of methods that frame the problem using neural networks, such as Fragkiadaki et al., 2015; Jain et al., 2016; Mall et al., 2017; Butepage et al., 2017; Kucherenko, Beskow, and Kjellström, 2018; Holden, 2018.

Some of these methods such as Mall et al., 2017; Holden, 2018 use a recursive neural network or a feed forward neural network to directly obtain the kinematic output (i.e. the skeleton joints and their angles). As mentioned before, the methods that directly generate the kinematic information of the skeleton are generally more robust and produce feasible poses, but sometimes deviate from the motion executed by the mocap actor. This is an undesirable production artifact.

Many of the methods mentioned above assume that the interval where the data is incorrect is given as an input and focus on the repair step of the problem. However, the detection step is equally important, but, it has received far less focus. While it is true that many of the capture problems occur when the markers are occluded (a case easily detectable), there are a number of common scenarios which are harder to detect, such as marker swaps, where maker paths are corrupted, often in a cluster with other markers.

The most common approach is to use statistical analysis to detect when the motion is natural (Ren et al., 2005; W. Kim and Rehg, 2008). However, these methods operate at the kinematic level and they will provide the interval when the motion is unnatural, but not which markers are the culprits. Invalidating all markers in a certain interval is unpractical. Aristidou, Cohen-Or, et al., 2018 proposes a novel method based on self similarity that allows for more granularity in detecting anomalies in the motion captured data.

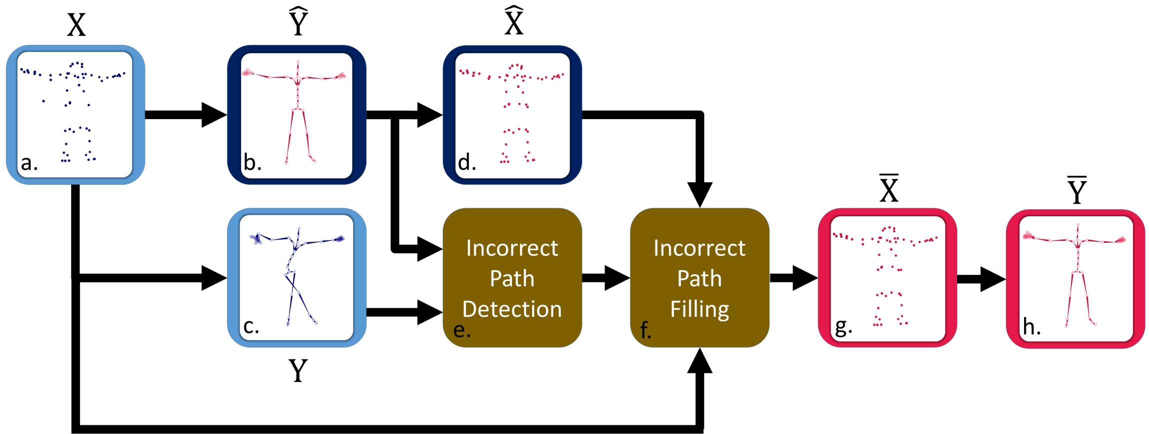


Figure 4: Overview of entire pipeline. a: Initial corrupted marker data. b: Solved skeleton using NN. c: Solved skeleton using commercial software. d: Reconstructed marker paths using LBS. e: Comparison of kinematic solutions to determine erroneous markers. f: Filling of incorrect marker paths. g: Filled marker paths. h: Final kinematic solution

## 3.3 Method

Our method is designed to complement a standard MOCAP system taking a set of marker paths and a set of joint transforms. Given a character consisting of $m$ markers for $n$ frames, the set of marker positions is denoted by $\mathbf{X} \in \mathbb{R}^{n \times m \times 3}$ (Fig. 4a) where $X_i^j$ is the $3D$ position of marker $j$ at frame $i$. For $j$ joints, we define a set of transforms $\mathbf{Y} \in \mathbb{R}^{n \times j \times 3 \times 4}$ (Fig. 4c) where the $Y_i^j$ is a $3 \times 4$ transformation matrix of the marker $j$ at frame $i$. The set of joint transforms is a naive kinematic solution generated using commercial software (Software, 2019) on the raw marker data. It contains numerous errors but the associated marker paths coincide with the original input.

To correct the erroneous marker paths, a second state-of-the-art kinematic solution is used as reference. In our experiments we used an modified version of Holden, 2018, as seen in (Fig. 4b) and elaborated on in section 3.3.1. However, our method is not exclusively tied to using this solver; any sufficiently robust kinematic solver can be used instead. While the original, naive, kinematic solution conforms to the original markers, it can contain erroneous poses. To the contrary, the robust solution contains only feasible poses, but it does not conform to the original markers even over time intervals where the marker paths are correct. We denote the output of this kinematic solver $\hat{\mathbf{Y}} \in \mathbb{R}^{n \times j \times 3 \times 4}$.

We use the second kinematic path to regenerate markers trajectories using linear blend skinning (Fig. 4d), which is denoted by $\hat{\mathbf{X}} \in \mathbb{R}^{n\times m\times 3}$. This process is explained in section 3.3.2. We determine the time intervals where the original marker paths are invalid by looking at the difference between poses (Fig. 4e), explained in section 3.3.3. We then recompute the invalid markers paths by augmenting their paths using the data from the robust kinematic solver, preserving the position and velocities at the boundaries of the time interval (Fig. 4f) section 3.3.4. This results in a set of marker paths $\bar{\mathbf{X}} \in \mathbb{R}^{n\times m\times 3}$ that is a blend of the other two (Fig. 4g).

Finally, we regenerate a kinematic solution for the filled markers (Fig. 4h) in-order to conduct our result analysis in section 3.4. This is done using the same process that is used to go from $\mathbf{X}$(Fig. 4a) to $\mathbf{Y}$(Fig. 4b), resulting in a kinematic solution that we denote as $\bar{\mathbf{Y}} \in \mathbb{R}^{n\times j\times 3\times 4}$. The following sections will describe the individual components of the process in more detail.

### 3.3.1 Kinematic Solver Improvements

The robust neural network based kinematic solver method as presented in Holden, 2018 works well for many examples, but as mentioned in the original paper, has certain failures cases when the input marker paths represent poses that were not covered in the training set or when the rigid body fitting process fails. In order to improve its performance, we slightly modify the algorithm from Holden, 2018 by using artificial data augmentation. This is done by computing correlations of individual local joint rotations, positions and scales from the training set and using this data to augment existing poses on a per-joint basis, with the magnitude being controlled by a standard Gaussian distribution. The following demonstrates this process for the position component, with the process for the other two components being similar: We define $\mathbf{L}_{\text{pos}} \in \mathbb{R}^{n\times 3j}$ as being the local space representations of the positional component of $\mathbf{Y}$, computed using an inverse kinematics method based on S. Buss, 2004. We then apply the following process:

$$\mathbf{A}_{\text{pos}} \sim \mathcal{N}(0, a_{\text{pos}}) \in \mathbb{R}^{n\times 3j} \tag{7}$$

Here $\mathbf{A}_{\text{pos}}$ controls the amount of local positional perturbations that are applied, with $a_{\text{pos}}$ is a manually set scalar value which controls the average magnitude of these

perturbations.

$$\mathbf{L}_{\text{pos}} \leftarrow \mathbf{L}_{\text{pos}} + \mathbf{A}_{\text{pos}}\mathcal{C}_{\text{pos}} \tag{8}$$

Here $\mathcal{C}_{\text{pos}} \in \mathbb{R}^{3j\times 3j}$ represents the lower triangular matrix from the Cholesky decomposition of the covariance matrix computed on the local joint positions found within the entire data set. After applying positional, rotational and scaling perturbations, the global rotations and positions are recomputed from the modified local values using forward kinematics. This method enforces correlations between joint transformations found within the data set onto the perturbations that are added in the preprocessing phase. Additionally, we also modify the artificial marker corruption process presented in Holden, 2018, increasing the likelihood that clusters of nearby markers are corrupted. This is used to simulate scenarios such as marker occlusion, where multiple nearby markers will often be occluded simultaneously.

### 3.3.2 Marker Reconstruction

From a valid kinematic solution $\hat{\mathbf{Y}}$, we reconstruct the markers using the same approach as in Holden, 2018: a linear blend skinning function is used, which will generate the reconstructed marker data $\hat{\mathbf{X}} = \text{LBS}(\hat{\mathbf{Y}}, \mathbf{Z})$, defined as:

$$\text{LBS}(\hat{\mathbf{Y}}, \mathbf{Z}) = \sum_{i=0}^{j} \mathbf{w}_i \odot (\hat{\mathbf{Y}}_i \otimes \mathbf{Z}_i) \tag{9}$$

where $\mathbf{Z} \in \mathbb{R}^{j\times 3}$ contains the rest pose marker offsets, $\mathbf{w} \in \mathbb{R}^{m\times j}$ contains the marker weights, $\odot$ represents component-wise multiplication and $\otimes$ represents matrix multiplication. Alternative skinning methods could be used instead, however it would also likely require their use in the training process of the neural network that generates $\hat{\mathbf{Y}}_i$. The output of this process is $\hat{\mathbf{X}}$, which represents the set of reconstructed marker paths. Due to their generation via linear blend skinning from a cleaned series of joint transforms, these marker paths will necessarily have a valid kinematic solution. However, due to limitations with the solver, the reconstructed marker paths will have two main issues. Firstly, small details in motion present in the original marker data such as finger movement will often be lost, potentially due to its use of a Savitzky-Golay filter  Savitzky and Golay, 1964. Second, the recreated marker paths will oftentimes be offset from the original marker paths, even when those sections in the

original marker paths are not corrupted.

### 3.3.3 Erroneous Marker Interval Detection

**Algorithm 2** Given two kinematic solutions, an erroneous one and a clean one, determine the set of missing markers paths

**Function** *DetectBad* ($\mathbf{Y} \in \mathbb{R}^{n \times j \times 3 \times 4}$, $\hat{\mathbf{Y}} \in \mathbb{R}^{n \times j \times 3 \times 4}$)

// Get differences between two kinematic solutions

$\Delta\mathbf{P} \in \mathbb{R}^{n \times j} \leftarrow ||\hat{\mathbf{Y}} - \mathbf{Y}||$

$\Delta\mathbf{R} \in \mathbb{R}^{n \times j} \leftarrow \text{Angle}(\hat{\mathbf{Y}}\mathbf{Y}^{\top})$

// Get joints surpassing allowed threshold

$$\mathbf{J} \in (0,1)^{n \times j} \leftarrow \begin{cases} 1, & \text{if } \Delta\mathbf{P} - \Delta\text{P}_{\text{max}} >= 0 \\ & \text{or } \Delta\mathbf{R} - \Delta\text{R}_{\text{max}} >= 0 \\ 0, & \text{otherwise} \end{cases}$$

// Get markers associated with joints surpassing threshold

$$\hat{\mathbf{w}} \in (0,1)^{m \times j} \leftarrow \begin{cases} 1, & \text{if } \mathbf{w} > 0 \\ 0, & \text{otherwise} \end{cases}$$

$\mathbf{M} \in (0,1)^{n \times m} \leftarrow \mathbf{J}\ \hat{\mathbf{w}}$

**return M**

**End**

In parallel to the marker reconstruction, another component of the process is used to detect erroneous sections in the marker paths $\mathbf{X}$ and remove them, leaving behind gaps in the marker data. This process is described in Algorithm 2 and further elaborated on here. To start with, all sections of marker paths detected as being outliers in Holden's method Holden, 2018 are added to the list of gaps. Subsequently, the two kinematic solutions $\hat{\mathbf{Y}}$ and $\mathbf{Y}$ are used to determine additional gaps. The positional and rotational components of the two joint transforms are compared. When either the rotational or positional differences between the two joints exceed their respective thresholds ($\Delta\text{P}_{\text{max}}$ and $\Delta\text{R}_{\text{max}}$), the violating joint is marked as being erroneous for the duration of the time that the threshold is exceeded. There is an inverse correlation between the magnitudes of the selected thresholds and the amount of frames marked as missing. The exact values chosen as well as the reasoning behind them is given in section 3.4.2. After determining the invalid frames on a per-joint basis, we determine which markers are to be marked as erroneous. The mapping from joints to markers is done using the same marker weight matrix $\mathbf{w}$ seen in equation 9,

by setting markers as missing if they have weights greater than 0 with respect to the violating joints.

The missing values are combined with the originally obtained outlying marker values to get a combined matrix of missing markers $\mathbf{M} \in (0, 1)^{n \times m}$, where 0 indicates the presence of the marker and 1 indicates its absence at a given keyframe.

### 3.3.4 Gap Filling

Before the marker paths can be fixed, some processing steps are performed to improve the system's performance. Firstly, in-order to simplify the system of marker gaps, the Boolean mask $\mathbf{M}$ is converted to a list of marker gaps $\mathbf{G} \in \mathbb{N}^{g \times 3}$, where $g$ is the total number of gaps present across all markers. For some gap $i$, $\mathbf{G}_i \in \mathbb{N}^3$ represents the index of the frame where the gap begins, the index of the frame where the gap ends, and the index of the associated marker, respectively.

Depending on the underlying reason for the gap in the tracked markers, the markers found at the frames before the start of the gap as well as the frames found after the gap are of limited reliability. As seen in Fig. 6, markers will often fly off erratically before and after disappearing. It is therefore useful to remove the keyframes found immediately before and after any given gap in a marker path. A simple approach is to cut off a preset amount of frames from both sides of all gaps in a given take. If set to a sufficiently high value, this method will remove most erroneous marker paths found near marker gaps. However, this will also cut off a decent number of useful keyframes, as well as not guaranteeing that the remaining marker path will be able to be correctly filled with the reference path in the next section. For this, a more sophisticated solution is used, cutting enough frames until the difference in slopes between the last uncut frame on the original path and the corresponding point on the reference path is minimized. A threshold can be set in-order not to cut out too many frames. Our chosen value is given in section 3.4.2.

Small tracks and gaps will often appear within the marker gap data. Small tracks are short sections of potentially valid marker frames that are surrounded by missing markers, and small gaps are short gaps surrounded by valid marker data. These artifacts can be present in the marker data due to issues propagated from the actual MOCAP system, due to the gap detection algorithm, where a kinematic solution can

repeatedly enter and leave the threshold limit, or due to the smart padding. Regardless of the underlying reason, small gaps are filled with a naive polynomial spline. Small tracks of marker data are simply removed and considered as missing, as they are of limited reliability. The thresholds for these two processes are tweakable, with our chosen values and reasoning behind them described in section 3.4.2. The order of these two operations is important, as it will generate different results depending on which one is run first. In our pipeline, we run the marker track removal first and then subsequently fill in any small gaps, which results in more gaps and less tracks filled in using the polynomial spline. This order is chosen since the subsequent step in our pipeline will use a more sophisticated algorithm to fill in larger gaps.

As seen in Algorithm 3, we start by iterating over the individual marker gaps $\mathbf{M}$. For an individual gap such as the one seen in Fig. 5, we compute the differences in positions between the reference marker path $\hat{\mathbf{X}}$ and the original marker path $\mathbf{X}$ at the last frame before the start of the gap and the first frame after the end of the gap. Similarly, we compute the differences in velocities between the reference marker path $\hat{\mathbf{M}}$ and the original marker path $\mathbf{M}$ before and after the gap. The use of positional constraints will ensure the $C^0$ continuity of our filled curve, while the velocity constraint swill ensure $C^1$ continuity. Both positional and velocity constraints are needed in-order to produce filled marker paths that behave realistically with no unnatural jumps in marker position or velocity. These four values computed on the differences between the reference and original marker paths are then used to fit a cubic Hermite spline. This spline is then subtracted from the reference marker path between the start and end frames. The result is a set of marker paths $\bar{\mathbf{X}} \in \mathbb{R}^{n \times m \times 3}$ which transfers the motion from the reference path onto the original marker path, all while conserving the smoothness of the curve.

Certain degenerate cases exist within the gap filling algorithm. A marker gap could have its end index be equal to $n$, its start index equal to 0, or both. The last two cases do not occur frequently but the first case is quite common, as once a marker is missing, it is likely to not reappear. We deal with the first case by setting $\Delta\mathbf{y}_e = \Delta\mathbf{y}_s, \Delta\mathbf{y}'_e = 0$ and the second case by setting $\Delta\mathbf{y}_s = \Delta\mathbf{y}_e, \Delta\mathbf{y}'_s = 0$. For the case where a marker is missing for the entire take, we can either simply take the reference path as is or simply ignore the marker, depending on whether the kinematic

solver used is robust enough to deal with fully missing markers.

Another case exists when velocities between the original marker path and the reference marker path are of sufficiently different magnitudes. In this case, it can occur that filled marker paths will stray from the reference path in order to try and satisfy the $C^1$ continuity constraint of the curve. The gap processing step of trimming frames until the velocities are sufficiently similar mostly mitigates this issue, but it may still intermittently occur. For this, we clamp the difference between the velocities of the reference path and the original path. This will limit the smoothness of the generated curve, but is a preferable solution to having a curve not follow the desired path. Our chosen clamping value is described in section 3.4.2.

## 3.4 Results and Discussion

The evaluation methodology for most state-of-the-art methods consists of artificially removing parts of the marker paths and filling in the gaps. The advantage of this approach is that it allows comparison against a ground truth. However, it also presents two major flaws. First, the deviation from ground truth in marker space is not the most relevant measure: it is possible to obtain a curve close to the ground truth that will generate a skeleton that is less kinematically correct than a curve that is further from the ground truth. Second, natural gaps in the data occur when the data is more difficult to estimate than when the system was able to construct it.

Table 1: Comparison between different marker filling methods in terms of the mean per-frame error between their kinematic solutions and the ideal path recreated by an artist. $\Delta_{pos}$ represents the error between the global joint positions (in cm) while $\Delta_{rot}$ represents the error between the global joint rotations (in degrees).

| **Method** | $\mathbf{\Delta_{pos}}$ | $\mathbf{\Delta_{rot}}$ |
|---|---|---|
| Original | 1.841 | 35.220 |
| Naive Polynomial | 10.678 | 42.93 |
| Commerical | 1.053 | 22.101 |
| Gloersen 2016 | 0.716 | 23.827 |
| TMT 2018 | 0.603 | 8.038 |
| Holden 2018 | 1.337 | 19.604 |
| Ours | **0.288** | **4.727** |

We provide comparisons for both marker curve similarity and kinematic correctness on both data with natural gaps and artificial gaps. We define natural gaps as sections of the data that were initially missing within our input, $\bar{\mathbf{X}}$. Artificial gaps are defined as sections of marker data that wasn't initially missing, but was deemed invalid by our detection algorithm. Since the ground truth itself is broken in the case where a gap occurs naturally, we compare our data to a professional motion capture tracking artist's handmade corrections.

### 3.4.1 Comparison

We compare a variety of different methods in various scenarios. Only a subset of these methods are visualized at once, either due to legibility considerations or because the other methods failed to run for the given example. The following are the methods we compare: *Original* is the original kinematic solution on the raw marker data ($\mathbf{Y}$), *Naive Polynomial* is a simple algorithm that fills a marker gap with a naive Hermite spline, and *Commercial* is a proprietary commercial filling algorithm Software, 2019 that interpolates a marker's path in-order to fill it. Additionally, we with compare several state-of-the-art methods: *Gloersen 2016* (Gløersen and P. Federolf, 2016), *TMT 18* (Tits, Tilmanne, and Dutoit, 2018), and *Holden 2018**, which is our retrained and slightly modified version of Holden, 2018, essentially representing $\hat{\mathbf{Y}}$ (Fig. 4b) in our method.

As seen within (Tab. 1) and (Fig. 8), our method generates marker paths that are more kinematically sound compared to the alternatives, being the closest to the ideal kinematic path recreated by a professional tracking artist. This is most notable on the naturally generated gaps, where our system outperforms all the other compared systems, including our modified version of Holden, 2018.

Figure 7 demonstrates a fairly simple example, where three markers on the hips go missing. Our method (b) is clearly closest to the ground truth, with each of the others having different issues. *Gloersen 2016* (a) has a broken leg, crooked spine and incorrect arm position. *Naive Polynomial* has has the hips too far back compared to the actual motion. *TMT 2018* (e) fails to accurately place the hip joint and has bending in the neck. *Commercial Linear* (f) manages to accurately place the arms, but has a crooked spine and neck.

Figure 9 demonstrates the importance of visualizing the kinematic solution $\bar{\mathbf{Y}}$

that corresponds to the marker paths $\bar{\mathbf{X}}$. Plotting out a single marker path, in this case a marker found on hips, it is hard to determine which marker paths are superior. While the *TMT 18* method (c) clearly deviates more from the rest, it is hard to determine what effect this has on the final result. Analyzing the kinematic solution, we can see that the deviation in marker path results in a completely broken skeleton. Additionally, we notice minor spine curving within the *Gloersen 2016* method, which would have been hard to determine using the marker paths.

Figure 10 demonstrates a hard to solve example, with an animation consisting of two characters wrestling on the ground, with multiple marker occlusions, marker swaps and otherwise corrupt marker paths. Although all methods fail to completely fix this animation, our method (b) performs the best, generating a smooth spine and intact joint contacts. *TMT 2018* (d) is second best, managing the recover the spine of the character on top but failing to properly correct the right arm of the bottom character.

### 3.4.2 Parameter Selection

Our pipeline has several parameters, all of which were determined experimentally. For the threshold values $\Delta \mathrm{P}_{\mathrm{max}}$ and $\Delta \mathrm{R}_{\mathrm{max}}$ in section 3.3.3, we used the values of $\Delta \mathrm{P}_{\mathrm{max}} = 10$ cm and $\Delta \mathrm{R}_{\mathrm{max}} = 30$ degrees. The motivation behind choosing a relatively large value for $\Delta \mathrm{R}_{\mathrm{max}}$ is to handle smaller joints, namely those on the hand. Due to the nature of the joints on the hand, their positional error will often be small even when they visually appear invalid, while their rotational error will usually be quite high. The exact value of the threshold values will depend on a multitude of factors, namely the quality of the solver, the precision of the MOCAP system used, the complexity of movements depicted within the motion capture shoot, as well as the importance allotted to keeping fine details in the original motion. For instance, cinematic shoots might prefer to utilize a greater threshold value to preserve more of the original motion and deal with any leftover errors by hand, while game-play shoots will not put as much value on fine detail preservation, allowing for a smaller threshold value.

The amount of frames trimmed in section 3.3.4 is set to a maximum of 120 frames, corresponding to 1 second of footage. We found that this value can be set relatively high due to the ability of our system to reproduce marker paths for long gaps. Due

Table 2: Performance of our pipeline computed on a typical sequence of 10,920 frames. Percentages calculated using largest remainder method.

| **Method** | **FPS** | **Time (%)** |
|---|---|---|
| NN Preprocessing | 4029 | 2 |
| NN Evaluation | 1339 | 6 |
| Filtering | 5527 | 1 |
| IK Retargeting | 108 | 74 |
| Marker Reconstruction | 2278 | 4 |
| Gap Detection | 10749 | 1 |
| Gap Preprocessing | 904 | 9 |
| Marker Filling | 2620 | 3 |
| **Total** | 81 | 100 |

to the robustness of our reference filling algorithm, the threshold limit for the naive small gap filling algorithm is set to a low value of 5 frames. Similarly, the robustness of our system likewise allows for a high threshold value for small track removal of 120 frames, which is equivalent to 1 second of footage. For the clamping threshold $\Delta_{\max}$, we clamp the velocity at 0.25. This choice of this value will depend on how much deviation from a smooth marker path is acceptable.

### 3.4.3 Performance

Table 2 presents performance metrics for each individual stage of our algorithm, described as follows: *NN Preprocessing* represents preprocessing stages before neural network execution, such as outlier marker removal and data normalization, *NN Evaluation* represents the evaluation of the network, *Filtering* represents the passing the network through the Savitzky-Golay filter, *IK Retargeting* represents inverse kinematic based retargeting, *Reconstruction* represents the marker reconstruction process presented in section 3.3.2, *Gap Detection* represents the erroneous marker interval detection presented in section 3.3.3, *Gap Preprocessing* represents the preprocessing done on marker gaps presented in section 3, and *Gap Fill* represents the marker filling process presented in section 3.3.4. The algorithm is run on a 91 second clip containing a single character, evaluated at 120 frames per second. Runtimes vary significantly based on the length of the clip, percentage of erroneous marker at each frame, and threshold values set by the user. However, the ratios between each stage remains roughly similar.

The retargeting stage is the biggest bottleneck of the pipeline, taking 74 % of the runtime to execute. The first four stages represent an implementation of the original algorithm presented in Holden, 2018. The last four stages represent the sum of steps

required for our gap filling algorithm. The cost of executing our blending algorithm is relatively inexpensive, taking 17 % of the runtime, in comparison to the original four stages which take 83 %.

**Algorithm 3** Given a gap in marker data, fill the gap by referring to a reference marker path

**Function** *GapFill* $(\mathbf{X} \in \mathbb{R}^{n \times m \times 3}, \hat{\mathbf{X}} \in \mathbb{R}^{n \times m \times 3}, \mathbf{M} \in \mathbb{N}^{g \times 3})$

// Copy over marker paths

$\bar{\mathbf{X}} \in \mathbb{R}^{n \times m \times 3} \leftarrow \mathbf{X}$

// Loop over all gaps in a clip

**for** i ... g **do**

// Get start, end and marker indices

$s, e, m \leftarrow \mathbf{M}_{i,1}, \mathbf{M}_{i,2}, \mathbf{M}_{i,3}$

// Get differences in positions

$$\Delta \mathbf{y}_s \in \mathbb{R}^3 \leftarrow \begin{cases} \hat{\mathbf{X}}^m_{s-1} - \mathbf{X}^m_{s-1}, & \text{if } s > 1 \\ \hat{\mathbf{X}}^m_{e+1} - \mathbf{X}^m_{e+1}, & \text{if } e < n \\ 0 & \text{otherwise} \end{cases}$$

$$\Delta \mathbf{y}_e \in \mathbb{R}^3 \leftarrow \begin{cases} \hat{\mathbf{X}}^m_{e+1} - \mathbf{X}^m_{e+1}, & \text{if } e < n \\ \hat{\mathbf{X}}^m_{s-1} - \mathbf{X}^m_{s-1}, & \text{if } s > 1 \\ 0 & \text{otherwise} \end{cases}$$

// Get differences in velocities

$$\Delta \mathbf{y}'_s \in \mathbb{R}^3 \leftarrow \begin{cases} \frac{(\hat{\mathbf{X}}_{s-1} - \hat{\mathbf{X}}_{s-3})}{2} - \frac{(\mathbf{X}_{s-1} - \mathbf{X}_{s-3})}{2}, & \text{if } s > 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\Delta \mathbf{y}'_e \in \mathbb{R}^3 \leftarrow \begin{cases} \frac{(\hat{\mathbf{X}}_{e+3} - \hat{\mathbf{X}}_{e+1})}{2} - \frac{(\mathbf{X}_{e+3} - \mathbf{X}_{e+1})}{2}, & \text{if } e < n \\ 0 & \text{otherwise} \end{cases}$$

// Clamp velocity differences

$\Delta \mathbf{y}'_s \leftarrow \text{AbsClamp}(\Delta \mathbf{y}'_s, \Delta_{\max})$

$\Delta \mathbf{y}'_e \leftarrow \text{AbsClamp}(\Delta \mathbf{y}'_e, \Delta_{\max})$

// Fit cubic polynomial using constraints

$\mathcal{P} \leftarrow \text{HermiteSpline}(\Delta \mathbf{y}_s, \Delta \mathbf{y}_e, \Delta \mathbf{y}'_s, \Delta \mathbf{y}'_e)$

// Subtract polynomial from reference

$\bar{\mathbf{X}}^m_{s \ldots e} \leftarrow \hat{\mathbf{X}}^m_{s \ldots e} - \mathcal{P}_{s \ldots e}$

**end for**

**return** $\bar{\mathbf{X}}$

**End**

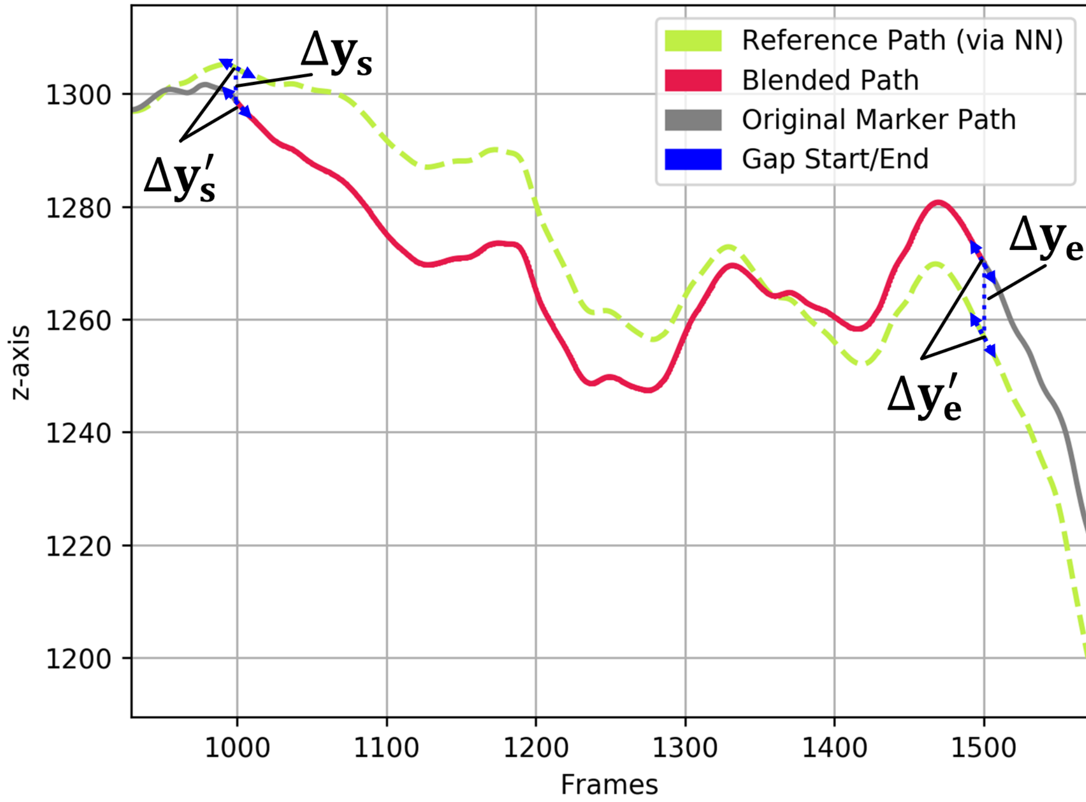


Figure 5: Example of marker filling from reference path

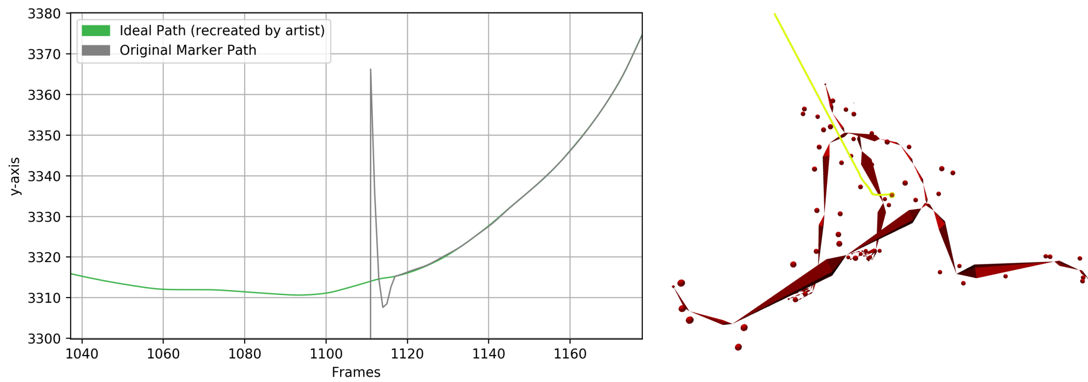


Figure 6: Marker paths before and after gaps in valid data

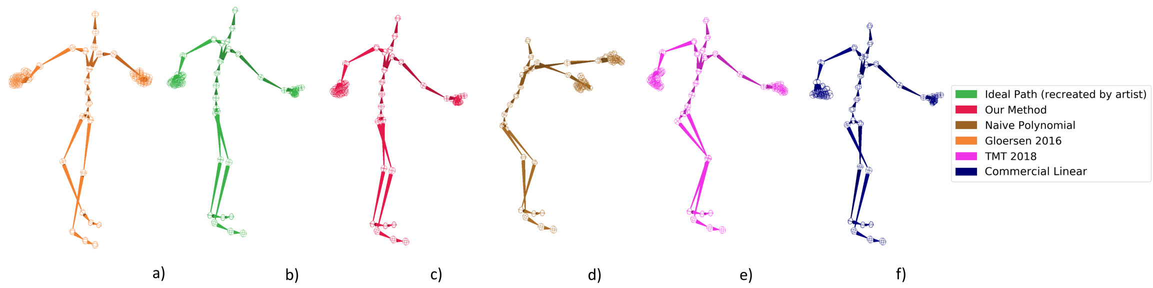


Figure 7: 3D visualization of the filled marker paths passed through a kinematic solver ($\bar{\mathbf{Y}}$).

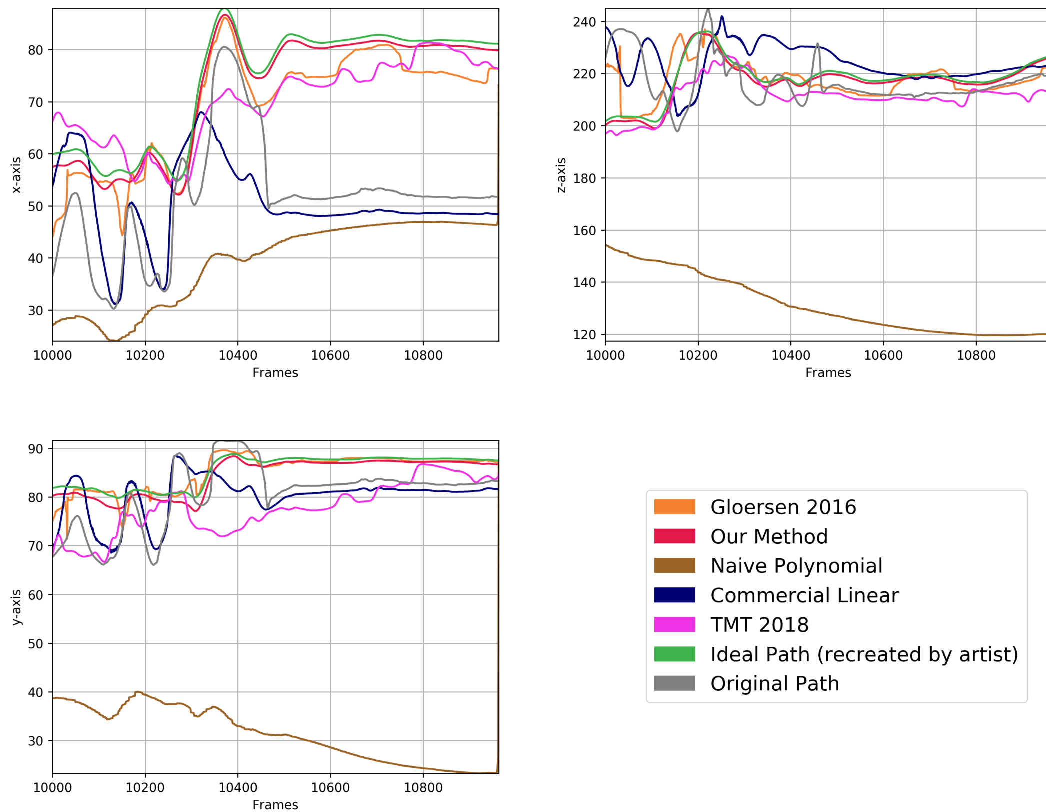


Figure 8: 2D visualization of the left upper leg joint from the filled marker paths passed through a kinematic solver.

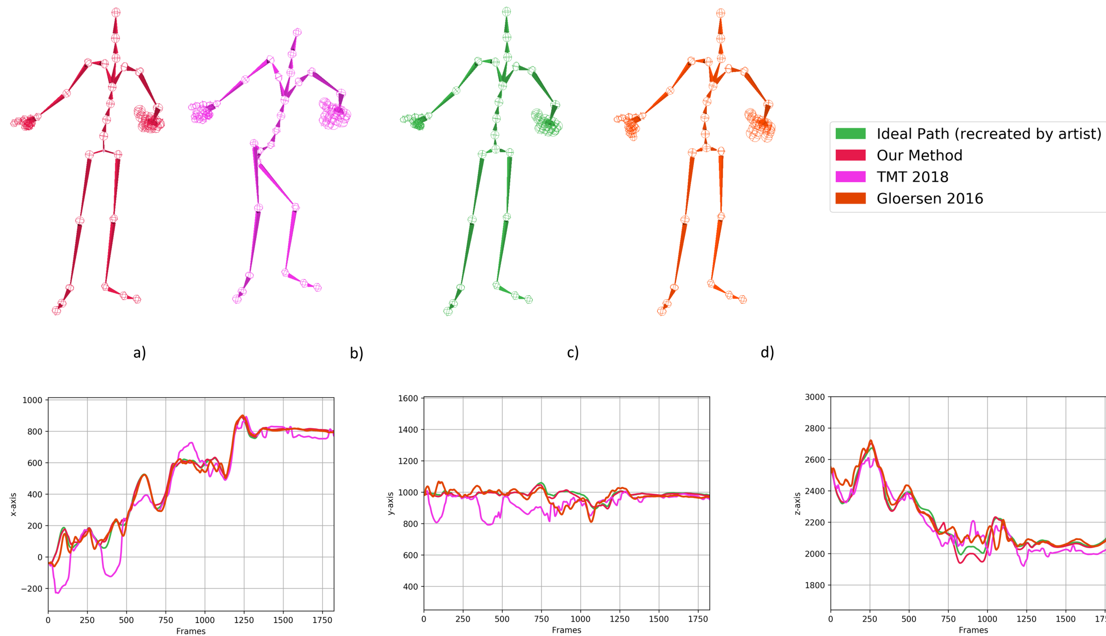


Figure 9: Visualization of a marker along with a 3D visualization of the corresponding kinematic solution.

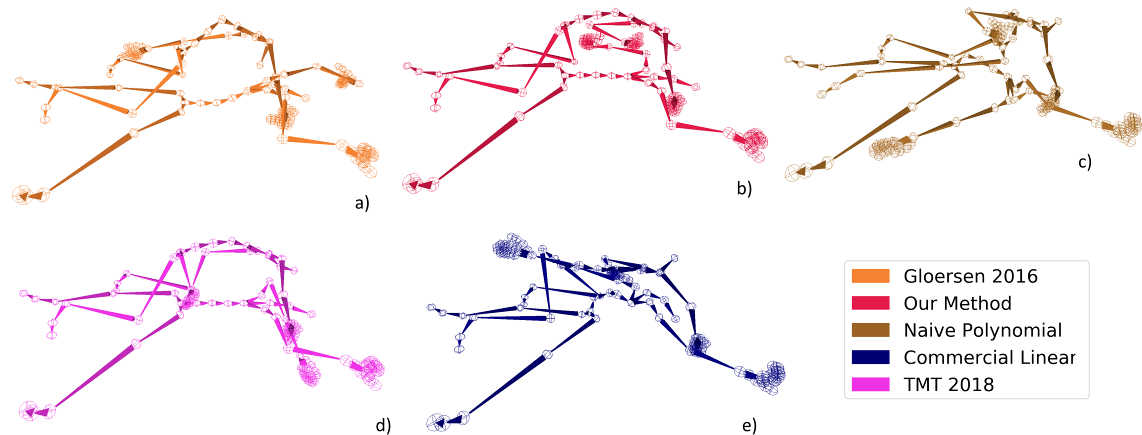


Figure 10: 3D visualization of the filled marker paths passed through a kinematic solver ($\bar{\mathbf{Y}}$). Hard example with two characters wrestling and numerous erroneous markers.

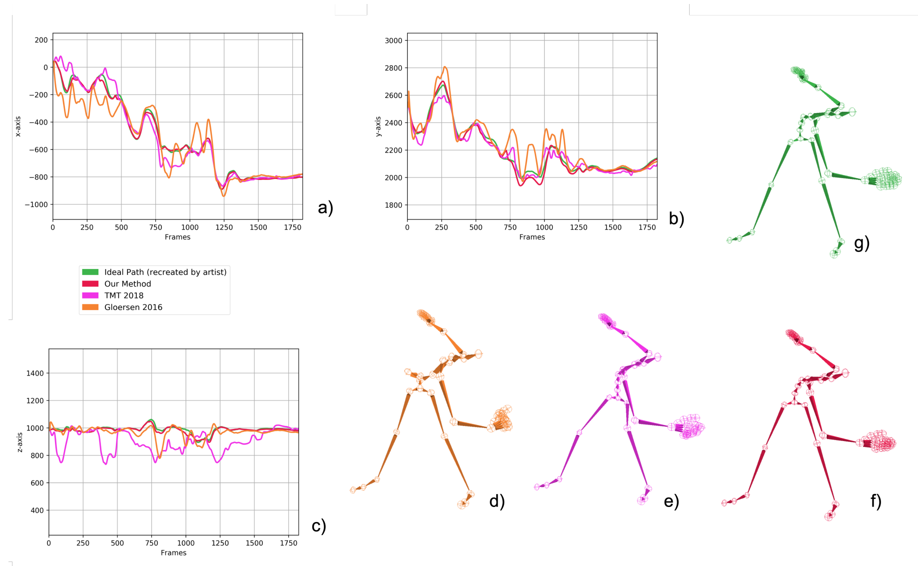


Figure 11: Visualization of a marker along with a 3D visualization of the corresponding kinematic solution.

# Chapter 4

# Mesh Based Animation Retargeting

Figure 12: Two rigged meshes overlayed on each-other, with the animation from the green character being retargeted onto the red character.

## 4.1 Introduction

Despite recent promising advances, gathering animation data for games remains a costly and time-consuming process. MOCAP Shoots require the hiring of actors, MOCAP technicians, and artists, as well as a considerable time-investment in-order for the animation data to be processed. One possible avenue to circumvent this issue is by reusing animation data from previously worked on projects. Although not

guaranteed to have the exact sequences of animations needed, previously worked on animation data can certainly remove the need to conduct new MOCAP shoots for basic animation data such as character locomotion. However, this itself is problematic since skeletal proportions, structures as well as mesh topologies will often differ between various projects, making transferring the animation a non-trivial process.

This process of transferring animation data from one skeletal rig to another is known as animation retargeting. Typically, artists will spend considerable resources using proprietary software in-order to achieve quality retargeting for games. It is arguable that some artistic input is unavoidable, as some retargeting cases present ambiguities that might have different solutions depending on the project's particular vision. However, there are some cases where an artist manually tweaking parameters for each retargeted clip is simply not feasible. For instance, if a production decides to modify the skeletal hierarchy of the base skeleton they use for their MOCAP shoots. Any old animation data is now incompatible with new data gathered going forward. An automatic batch retargeting algorithm is desirable for these situation, as it would permit the reuse of old data without significant manual intervention.

In this chapter, a mesh based retargeting algorithm is proposed. Starting from a source skeleton coupled with a source mesh and a target skeleton coupled with a target mesh, mesh vertices are first evenly subsampled. A correspondence is built between the two meshes if their topologies differ. Subsequently, animation is retargeted from the source rig to the target rig using an iterative Jacobian Inverse Kinematics algorithm that computes target joint angles needed in-order for vertices of the meshes to match.

## 4.2 Related Work

Gleicher, 1998 introduces the problem of animation retargeting for characters with bones of differing proportions, proposing an IK based method that is capable of satisfying multiple constraints.

Monzani et al., 2000 propose using an intermediate skeleton for retargeting, allowing for retargeting between characters of differing topologies, enforcing constraints using Inverse Kinematics. The intermediate skeleton approach remains popular, with industry standard tool MotionBuilder (Autodesk, 2020) allowing users to retarget

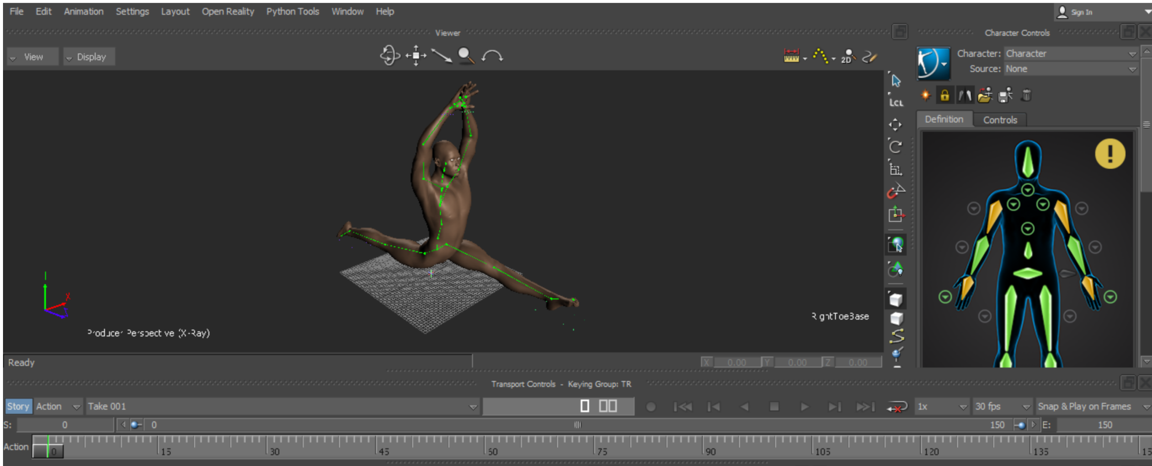


Figure 13: Autodesk MotionBuilder animation software. Character tool shown on right hand side, displaying mapping to an intermediate skeleton.

animation through an intermediate skeleton known as the "character" tool as seen in Fig. 13. Professional retargeting artists utilize this tool to retarget animation, manually tweaking dozens of parameters to achieve high quality retargeting.

Baerlocher and Boulic, 2004 propose a real-time IK based retargeting framework capable of handling multiple priorities and incorporating joint limits. Kulpa, Multon, and Arnaldi, 2005 demonstrate a morphology independent animated motion representation that effectively bypasses the traditional retargeting problem.Meredith and Maddock, 2005 enhance a traditional IK solver, allowing for individualized retargeting that adapts to both the chosen terrain and the target character.

Avril et al., 2016 proposes a method capable of deriving correspondences between source and target character meshes, solving for joint positions, rotations and skinning weights.

## 4.3 Method

The method implements mesh-based animation retargeting from a **source** rigged mesh to a **target** rigged mesh using both vertex positions and normals. Essentially treating mesh vertices as end-effectors, it attempts to find angles in the **target** skeleton such that the **target** mesh vertices track the **source** vertices, effectively transferring the animation from the **source** skeleton onto the **target** skeleton.

Taking as input the $3 \times 4$ transformation matrices for the source and target joints,

defined as $\mathbf{T}_s \in \mathbb{R}^{j_s \times 3 \times 4}$ and $\mathbf{T}_t \in \mathbb{R}^{j_t \times 3 \times 4}$ respectively, where $j_s$ is the number of joints in the source skeleton and $j_t$ is the number of joints in the target skeleton. The source skeletal animation is defined as set of Euler angles representing the per-frame set of joint rotations $\boldsymbol{\theta}_s \in \mathbb{R}^{n \times j_s \times 3}$, where $n$ is the number of frames in the animation, $j_s$ is the number of joints in the source skeleton. $\boldsymbol{\theta}_s \in \mathbb{R}^{n \times j_t \times 3}$ represents the unknown that is solved for, i.e. the set of joint rotations that need to be computed on the target skeleton. The mesh vertex rest positions for the source and target meshes are represented as $\mathbf{P}_s \in \mathbb{R}^{v_s \times 3}$ and $\mathbf{P}_t \in \mathbb{R}^{v_t \times 3}$, where $v_s$ is the number of vertices in the source mesh and $v_t$ is the number of vertices in the target mesh. Similarly, we take as input the mesh vertex normal vectors for both meshes, defined as $\mathbf{N}_s \in \mathbb{R}^{v_s \times 3}$ and $\mathbf{N}_t \in \mathbb{R}^{v_t \times 3}$.

The skeleton-to-mesh mapping is defined using skinning weights $\mathbf{w}_s \in \mathbb{R}^{v_s \times j_s}$ and $\mathbf{w}_t \in \mathbb{R}^{v_t \times j_t}$. The skinning is accomplished using LBS, as this is still the principal skinning method used in games. In theory, it could be extended to use other less error prone skinning methods such as DQS (Dual Quaternion Skinning), however, the Jacobian matrix would differ.

Algorithm 4 demonstrates a rough pseudo code overview of the algorithm, with the individual parts being elaborated on in the relevant subsections. First, a preprocessing step subsamples the mesh and removes undesirable vertices (section 4.3.1). If the mesh topologies differ between the two rigs, a mapping is created between the two meshes (section 4.3.2). Standard Forward Kinematics and Linear Blend Skinning is then precomputed for the source vertices. Subsequently, a normalization algorithm normalizes the data (section 4.3.3). A blend mask is computed in-order to control which vertices are tracked via normals and which are tracked via positions (section 4.3.4). Then, the frames are looped over, with the first frame being initialized at the rest pose of the animation, with subsequent frames using previously computed frames as a starting point. A single iteration of the solver can be seen in pseudo-code in Algorithm 6. As seen, a single iterations recomputes global joint positions using FK (section 2.2, recomputes global vertex positions using LBS (section 2.4, normalizes the new data (section 4.3.3, builds the Jacobian (section 4.3.6), computes the error (section 4.3.7), and solves the system (section 4.3.8).

Figure 14: Three subsampled versions of the same mesh. The squares indicate the vertices chosen by the subsampling algorithm. The leftmost mesh is subsampled at 128:1 vertices, middle one at 64:1, and rightmost one at 32:1.

### 4.3.1 Subsampling

Typical meshes used in video-game productions are made up of tens of thousands of vertices. This makes it impractical to retarget animations based on them as it is simply too computationally expensive. For instance, using Jacobian inverse kinematics, a rigged mesh with $v_s$ vertices and $j_s$ joints will result in a Jacobian matrix of size $v_s \times j_s \times 3$. Even with optimizations exploiting the sparsity of said matrix, inverting said matrix and solving the corresponding system is computationally expensive. In addition, the corresponding system is non-linear, thus it would take a number of iterations per-frame for it to converge to the solution. All these factors make utilizing the set of all mesh vertices impractical for animation retargeting. As such, we wish to reduce the number of vertices by subsampling. This subsection covers the various subsampling problems and solutions.

This section covers the cases where the source and target mesh topologies are identical, i.e. $\mathbf{P} = \mathbf{P}_s = \mathbf{P}_t$ and $\mathbf{N} = \mathbf{N}_s = \mathbf{N}_t$. For the case where the previous statements don't hold, subsampling is done on the source meshes, i.e. $\mathbf{P} = \mathbf{P}_s$, $\mathbf{N} = \mathbf{N}_s$ and a mapping is subsequently created, explained in detail in section 4.3.2. In the following, $\bar{v}$ indicates the subsampled number of vertices. $\bar{\mathbf{P}}$ and $\bar{\mathbf{N}}$ indicate the subsampled versions of $\mathbf{P}$ and $\mathbf{N}$, respectively.

A naive approach to the problem is to simply take every $n$ vertices, where $n = \frac{v}{\bar{v}}$. Our subsampled vertex positions $\bar{\mathbf{P}}$ and normals $\bar{\mathbf{N}}$ could then be represented as:

$$\bar{\mathbf{P}}, \bar{\mathbf{N}} := \mathbf{P}^{::n,3}, \mathbf{N}^{::n,:} \tag{10}$$

This approach is computationally inexpensive but has two glaring flaws. First, mesh vertices are seldom stored in a consistent manner. Sampling every $n$ vertices could result in some areas of the mesh not being covered. Secondly, the mesh vertices themselves are are seldom sampled evenly. High detail areas such as the face and hands will contains a greater vertex concentrations compared to lower detail areas such as the torso. It is not atypical to see the face alone contain more vertices than in the entire rest of the mesh. More sampled points in a certain area implies more points being tracked by the IK algorithm in that area, which biases the algorithm to reduce error in that specific region. To fix this issue, a nearest neighbor based subsampling algorithm is used to ensure even sampling, as seen in Algorithm 5.

It is also important to consider the minimum amount of sampled vertices for a given joint. A minimum of 3 vertices per-joint is required in-order to be able to properly position said joint. Meshes used in production will often contain features that aren't suitable for tracking. These can be holes, extrusions or jagged edges. Tracking these vertices is undesirable, since their normal displacements will differ from vertices nearby and introduce error to the solving algorithm. To curb this, during the subsampling, a selected vertex normal is compared against the average normal of its neighbors. If the difference between the two exceeds a set threshold, determined experimentally to be 0.05, this vertex is excluded from selection. Algorithm 5 can be modified to solve both of these problems. It is sufficient to change the line $\mathbf{S}_k \leftarrow \text{argmax}_{i \ldots v} \left(\min_{j \ldots k} \left(||\mathbf{Diff}_{i,j,\ldots}||^2\right)\right)$ into $\text{argsort}_{i \ldots v} \left(\min_{j \ldots k} \left(||\mathbf{Diff}_{i,j,\ldots}||^2\right)\right)$. Then, instead of having the best match, a list of matches is generated and looped over. Each vertices normal's are checked against the average of their neighbors to ensure they fall within the preset threshold. If this condition is satisfied, another check is performed to ensure that each joint gets at least 3 subsampled vertices each. These checks slow down the subsampling process considerably. However, since they only need to be performed once per mesh, this isn't a crucial issue.

### 4.3.2 Mesh Mapping

In the case that the source and target meshes differ, it is necessary to create a mapping between corresponding vertices in each mesh. We wish to pick vertex mapping that minimizes the difference between the corresponding vertices. We define this as an optimization problem:

$$\mathbf{S}_s, \mathbf{S}_p := \underset{i,k \ldots v_s, v_p}{\operatorname{argmin}} ||\mathbf{P}_s^i - \mathbf{P}_t^k|| \tag{11}$$

Computing mesh vertices is a computationally expensive process but only needs to be done once for each mesh pairing. This process can be sped by only executing the subsampling process on one mesh and subsequently computing the nearest neighbor.

### 4.3.3 Normalization

One issue with retargeting based on both vertex positions and normals is the difference in the scale of the data. The scale of vertex positions will differ based on a number of factors, notably, the scale and proportions of the characters. The scale of vertex normals will always be 1, as they are normalized vectors. For effective solving that doesn't bias the system to solve for one over the other, it is essential to normalize the data such that the scale of its input data is roughly similar. The ideal method for normalization would be to subtract the mean value and divide by the standard deviation, resulting in a distribution centered at 0 with standard deviation of 1. However, this method is problematic for two reasons. Firstly, computing the mean and standard deviation at each iteration of the algorithm would considerably slow down the algorithm. Secondly, $\mathbf{P}_t$ must remain in the scale as $\mathbf{GP}_t$, as their difference will be used in the computation of the Jacobian in section 4.3.6. The solution to these problems is to normalize by dividing by a single standard deviation computed at the start. The source and target mesh vertex positions, $\mathbf{P}_s$ and $\mathbf{P}_t$ respectively, as well as the global joint positions $\mathbf{GP}_t$ will be divide by $\sigma = \mathbf{P}_s.\sigma$ (the standard deviation of the source mesh vertex positions, computed once at the start of the algorithm). Note that the $\mathbf{P}_s$ can be normalized once at the beginning of algorithm 4, while $\mathbf{P}_t$ and $\mathbf{GP}_t$ need to re-normalized at each iteration (algorithm 6). As the vertex normals are already of length 1, it is unnecessary to normalize them.

### 4.3.4 Blend Mask

It this retargeting system, it is useful to control which vertices are tracked using their positions and which ones are tracked using their normals. Typically, we wish to track sections of the body such as the torso using vertex normals. If vertex positions are used to track the torso, the system can generate solutions such as ones with unnatural looking crumpled spine in-order to satisfy the constraints. On the other hand, it is desirable to track end-effectors such as hands, feet and sometimes the head using their corresponding vertex positions. This is because these joints will often have interactions with the outer world, and it is desirable to track their positions. We define a blend mask $\mathbf{B} \in \{0, 1\}^{\bar{v}}$, where 0 indicates tracking the positions and 1 indicates tracking the normals. The blend mask parameters can be set manually on a per-mesh basis, or automatically computed. . In principle, the blend mask doesn't necessarily need to binary: it is possible to setup a system to track a combination of both. However, for simplicity, this chapter will only address the binary mask case.

### 4.3.5 Cumulative Weights

For computing the Jacobian in section 4.3.6, it is useful to precompute cumulative weights that will indicate whether or not a vertex is dependent upon any given joint, and to which degree. A vertex $i$ will be dependant on joint $j$ if and only if joint $j$ or one of the children of joint $j$ has a non-zero skinning weight associated with vertex $v$. The dependency value is computed as a cumulative sum of the skinning weight of $j$ and its descendant joints associated with vertex $v$. Figure 15 demonstrates this relationship.

### 4.3.6 Jacobian

Jacobian Inverse Kinematics, as described in S. Buss, 2004, defines an iterative approach. This requires the explicit computation of derivatives. It is possible to avoid this step by using the finite-difference method for derivative estimation, but results in much slower computation times. Luckily, the explicit derivative for mesh based inverse kinematics is in-fact a trivial modification of the original inverse kinematic formula seen in S. Buss, 2004 and summarized in section 2.3.

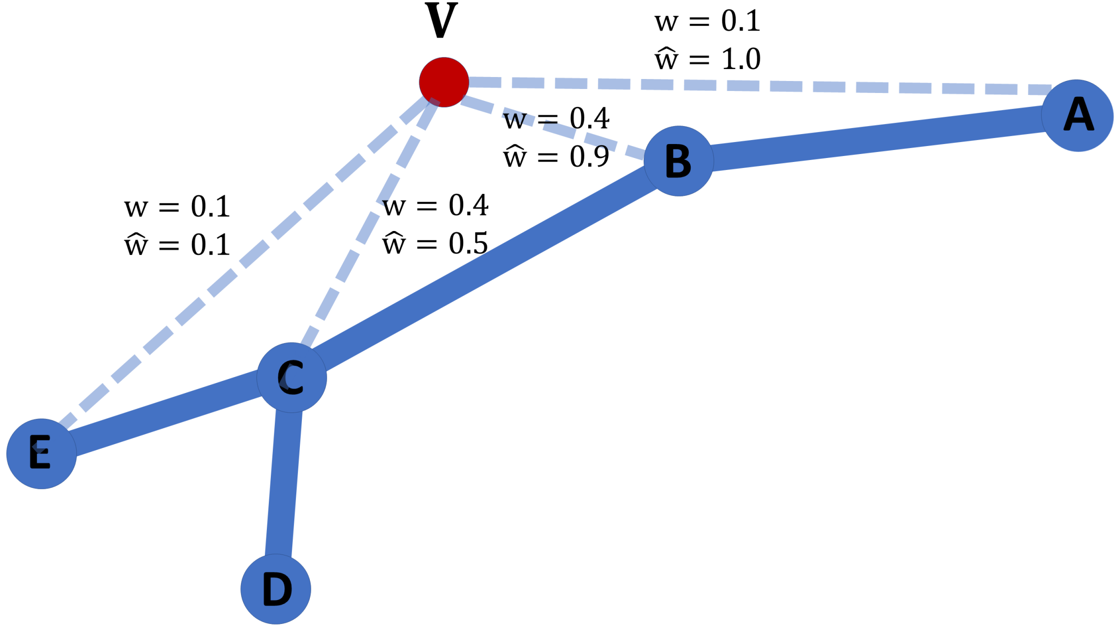


Figure 15: Illustration demonstrating the cumulative weights. $A, B, C, D, E$ represent joints in a hierarchy, while $V$ is a vertex. The $\mathbf{w}$ represents the skinning weight between the two while $\hat{\mathbf{w}}$ represents the cumulative skinning weights.

Equation 2 can be extended to meshes by considering the linear relationship between vertices deformed using LBS and their respective joints. Using the equations shown in section 2.4, we can derive the partial derivatives for mesh based retargeting.

For vertex positions, the derivation can be seen motivated by Figure 16. By taking the cross product between the axis of rotation $\mathbf{r}_t^j$ and the difference between the global vertex position and the global joint position $\mathbf{P}_t^i - \mathbf{GP}_t^j$, we get $\Delta\mathbf{P}_t^i$. This example demonstrates the simple case of of a 1-to-1 mapping between joint and vertex positions.

$$\mathbf{J}_P^{i,j} = \frac{\partial \mathbf{P}_t^i}{\partial \boldsymbol{\theta}^j} = \mathbf{r}^j \times \left(\hat{\mathbf{w}}^{i,j}\mathbf{T}_t^j(\mathbf{P}_t^i - \mathbf{GP}_t^j)\right) \tag{12}$$

For a vertex normals, the derivation is similarly motivated in Figure 17. However, unlike the vertex positions, the vertex normals don't depend on the distance between them and the joint. As such,

$$\mathbf{J}_N^{i,j} = \frac{\partial \mathbf{N}_t^i}{\partial \boldsymbol{\theta}^j} = \mathbf{r}_j \times \left(\hat{\mathbf{w}}^{i,j}\mathbf{T}_t^j\mathbf{N}_t^i\right) \tag{13}$$

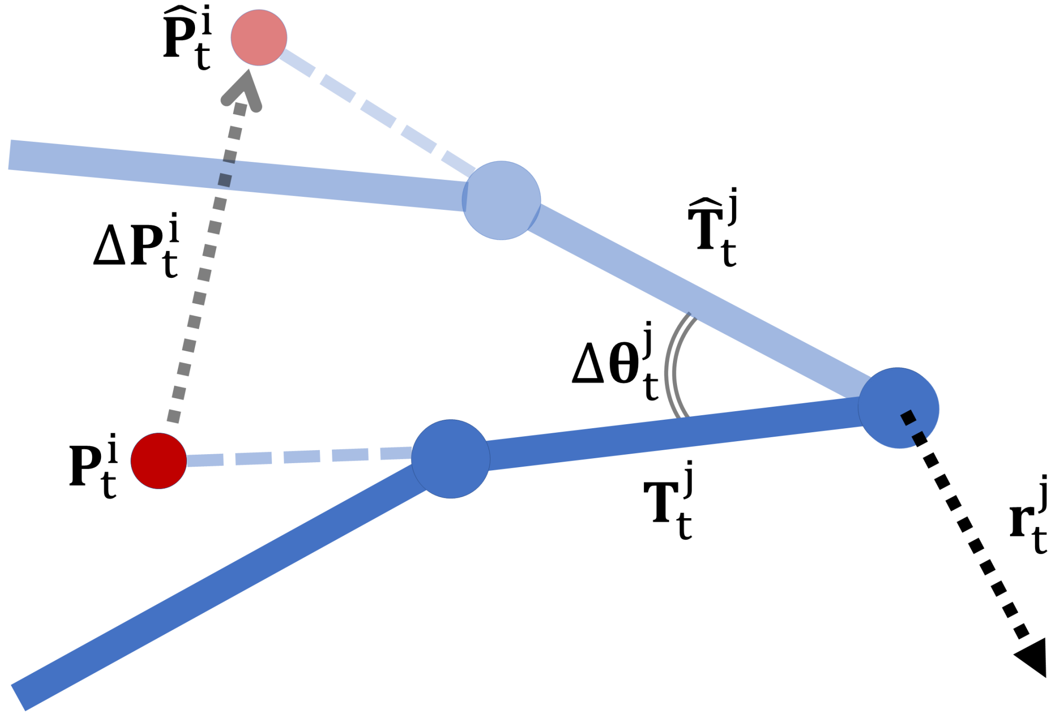


Figure 16: Illustration motivating the derivation of $\frac{\partial \mathbf{P}_t^i}{\partial \theta_j}$. The red points visualizes the vertex position $\mathbf{P}_t^i$ while the blue arm visualizes a joint hierarchy starting at $\mathbf{T}_t^j$. $\Delta\mathbf{P}_t^i$ represents the displacement of the vertex position as as a result of the rotation of joint $\mathbf{T}_t^j$ around axis $\mathbf{r}_t^j$ by angle $\Delta\boldsymbol{\theta}_t^j$

The overall Jacobian matrix can be computed using the previous defined blend mask in section 4.3.4, as:

$$\mathbf{J} = (1 - \mathbf{B}) \odot \mathbf{J}_P + \mathbf{B} \odot \mathbf{J}_N \tag{14}$$

### 4.3.7 Error

The error can be defined as a combination of positional error $\vec{\mathbf{e}}_P \in \mathbb{R}^{\bar{v},3}$ and normal error $\vec{\mathbf{e}}_N \in \mathbb{R}^{\bar{v},3}$. Both can be expressed simply as the difference between their source and target counterparts:

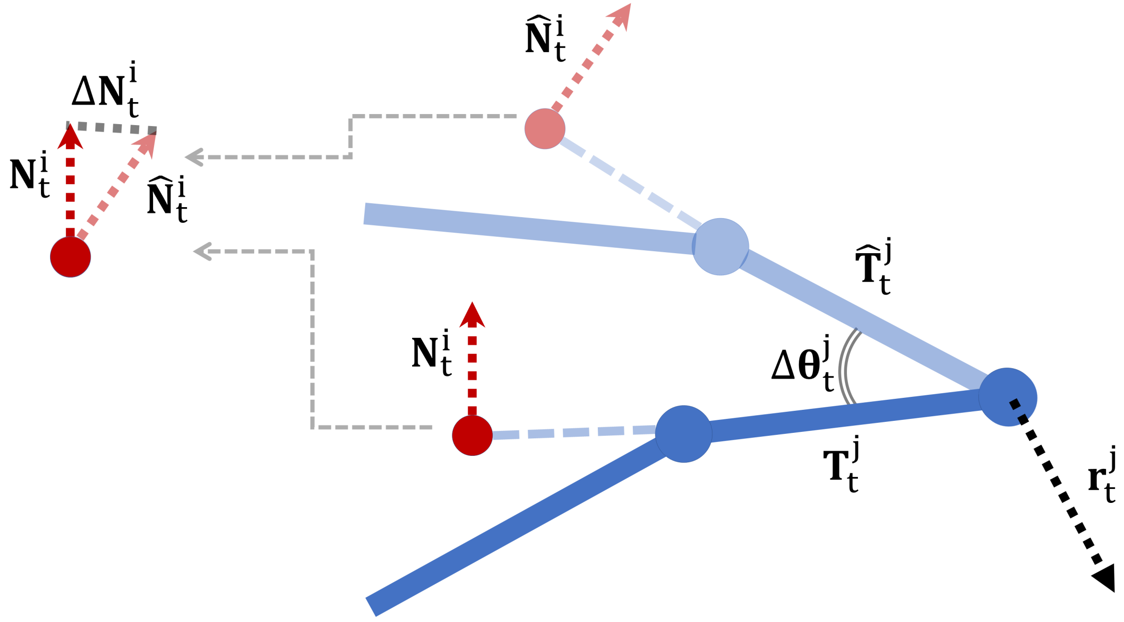


Figure 17: Illustration motivating the derivation of $\frac{\partial \mathbf{N}_t^i}{\partial \theta_j}$. The red points visualizes the vertex normal $\mathbf{N}_t^i$ while the blue arm visualizes a joint hierarchy starting at $\mathbf{T}_t^j$. $\Delta\mathbf{N}_t^i$ represents the displacement of the vertex normal as as a result of the rotation of joint $\mathbf{T}_t^j$ around axis $\mathbf{r}_t^j$ by angle $\Delta\boldsymbol{\theta}_t^j$

$$\vec{\mathbf{e}}_P = \mathbf{P}_s - \mathbf{P}_t \tag{15}$$

$$\vec{\mathbf{e}}_N = \mathbf{N}_s - \mathbf{N}_t \tag{16}$$

Using the blend mask derived in section 4.3.4, the overall error can be expressed as:

$$\vec{\mathbf{e}} = (1 - \mathbf{B}) \odot \vec{\mathbf{e}_P} + \mathbf{B} \odot \vec{\mathbf{e}_N} \tag{17}$$

### 4.3.8 Solving

An iterative approach of Damped Jacobian Inverse Kinematics S. Buss, 2004 is used to retarget the source motion onto the target rig. This process involves solving the

following equation:

$$\Delta\theta = \mathbf{J}^{\top}(\mathbf{J}\mathbf{J}^{\top} + \lambda\ I)^{-1}\vec{\mathbf{e}} \tag{18}$$

One important consideration in solving this system is the choice of damping factor $\lambda$. Through experimentation, it was determined that a low damping factor of $\lambda = 5.0$ is sufficient to stabilize the system. More advanced damping can be added, such as selectively damped least squares described in S. R. Buss and J.-S. Kim, 2005 but is usually not required.

## 4.4 Implementation

The Mesh Based Retargeting algorithm was implemented in Python, using Numpy, Scipy, and Numba libraries Processing was done on an 6 Core Intel Xeon E5-1650 v3 CPU clocking in at 3.50 GHz

Performance gains were achieved through multiple optimizations. Firstly, data was stored and processed in Numpy, allowing for data-type specifications and faster performance than traditional python. Numba was utilized as a JIT (Just-In-Time) compiler, allowing Python code to be compiled speeding up operations such as FK by two orders of magnitude. Scipy was used for a sparse matrix library. In production settings, skinning weights are sparse, with one vertex usually being limited to 4 non-zero skinning weights. This inherent sparsity in the skinning weights results in a sparse Jacobian. The resulting equation can be solved much faster by storing the Jacobian in a sparse matrix container, as most operations would in-effect be multiplications by zero.

## 4.5 Results

Figure 18 demonstrates the retargeting of a basic pose from a source character onto a target character of similar scale and proportions. Although the mesh topologies are the same, the skeletal structures are different, with the target skeleton missing finger bones. Figure 19 demonstrates a similar example, keeping identical proportions but changing the scale of the character. Hands and feet are still tracked using vertex positional constraints, while the torso is tracked using vertex normals. This results

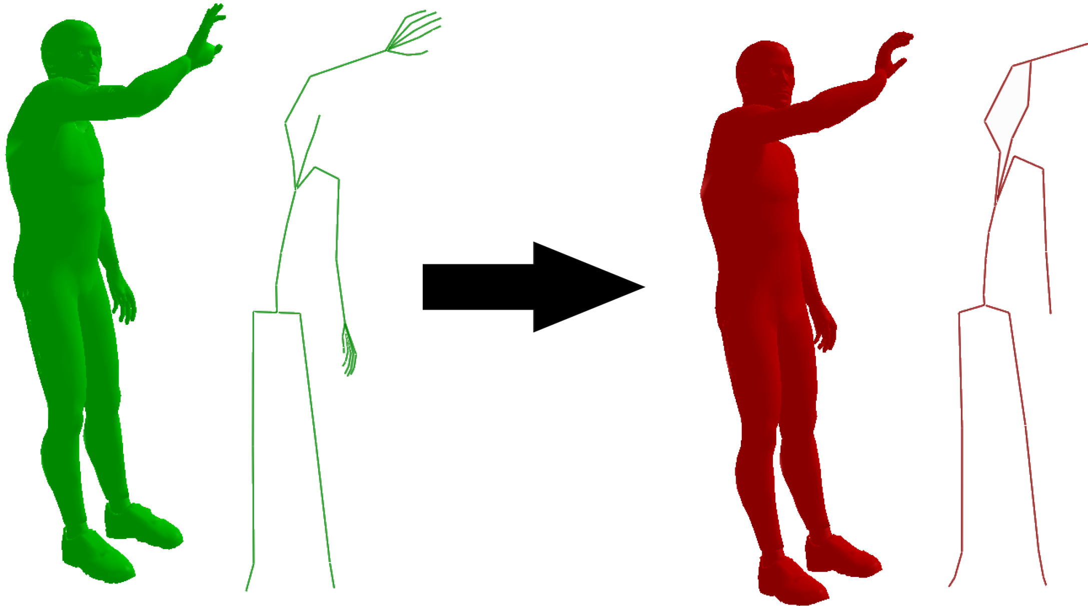

Figure 18: Figure demonstrating mesh based retargeting from the source character (green) to the target character (red).

in the target character attempting to reach for the source characters hand. Figure 20 demonstrates the same scenario, with the source and target scales flipped.

Figure 21 and 25 demonstrate retargeting done on characters of differing proportions.

Figures 22, 23, and 24 demonstrate overlayed meshes to visualize the retargeting process. The red and green spikes represent the vertices tracked by their normals, while the blue lines represent vertices tracked by their positions.

Table 2 demonstrates relative performance metrics for a single iteration of the retargeting algorithm for each individual component of the algorithm. As seen, the solving step takes a large chunk of the processing time for an iteration. The next biggest chunks are the FK and LBS passes, which are necessary in-order to reconstruct the mesh vertices at each step. The "Other" item in this table refers to other non-algorithm specific steps that are done during an iteration, such as logging and storing statistics. 4 demonstrates performance comparison between different levels of subsampling for an animated clip of 4000 frames, with 5 iterations per-frame. As seen, it is possible to achieve speed-ups by decreasing the number of vertices being tracked.

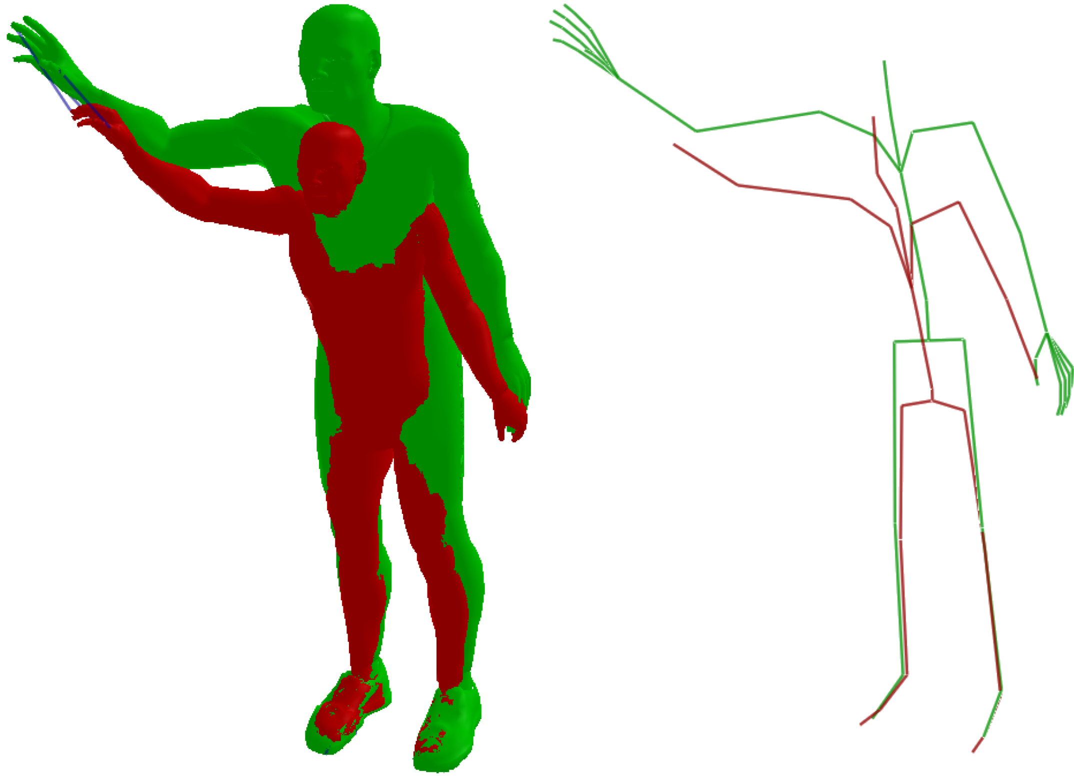

Figure 19: Figure demonstrating mesh based retargeting on different scale characters, from the source character (green) to the target character (red). Blue lines on the hand indicate vertex position constraints.

Table 3: Relative performance metrics for a single iteration of the retargeting algorithm, separated by the individual components. Percentages are rounded to the nearest integer.

| **Stage** | **Time (%)** |
|---|---|
| FK | 13 |
| LBS | 22 |
| Normalization | 1 |
| Axes Computation | 6 |
| Building Jacobian Matrix | 7 |
| Error Computation | 4 |
| Solving System | 41 |
| Other | 6 |
| **Total** | 100 |

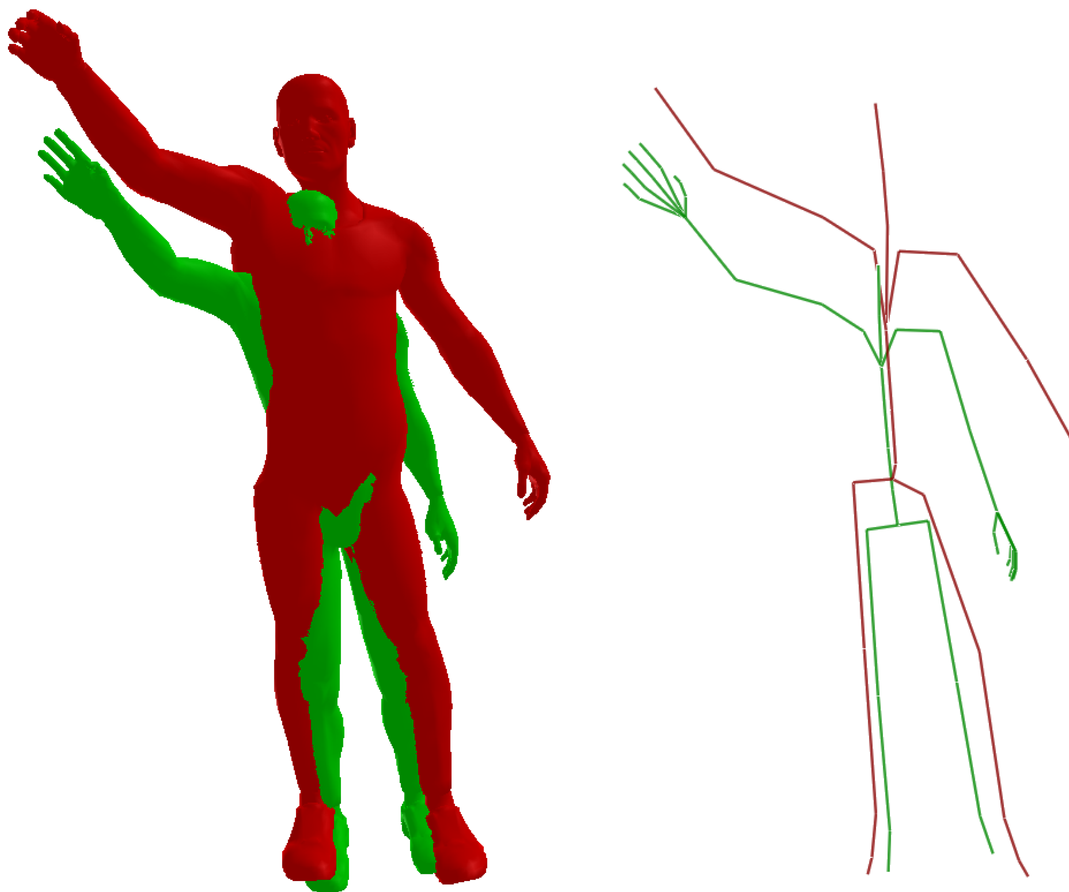

Figure 20: Figure demonstrating mesh based retargeting on different scale characters, from the source character (green) to the target character (red).

Table 4: Performance metrics of processing time of one 4000 frame clip based on the subsampling rate.

| **Subsampling Ratio** | **Time (s)** |
|---|---|
| 128:1 | 43.5 |
| 64:1 | 48.9 |
| 32:1 | 58.9 |
| **Total** | 100 |

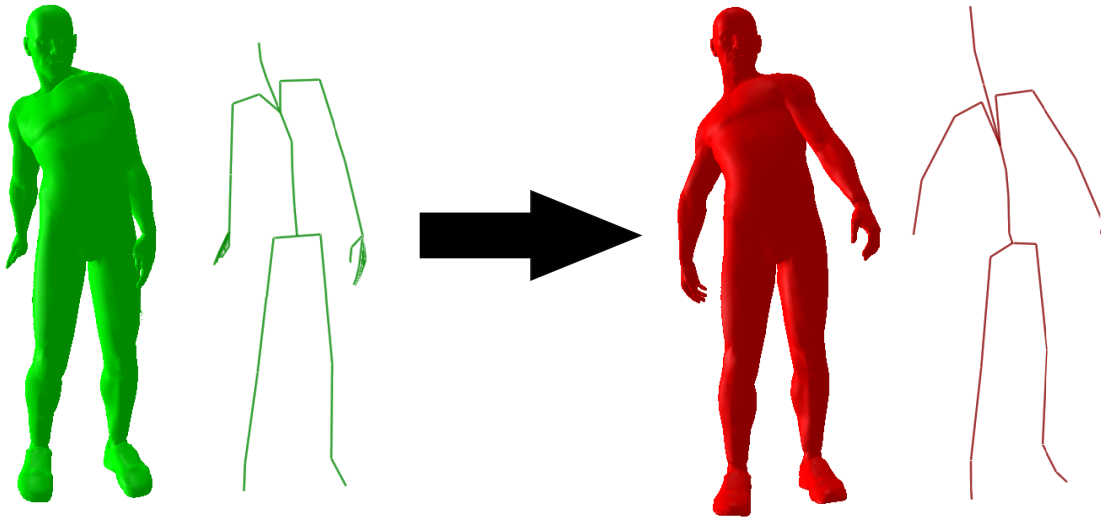

Figure 21: Figure demonstrating mesh based retargeting from the source character (green) to the target character (red).

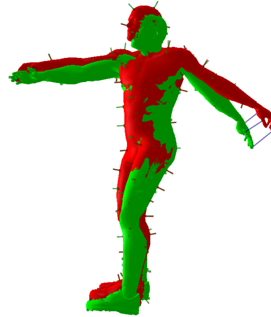

Figure 22: Figure demonstrating mesh based retargeting from the source character (green) to the target character (red) for a walking motion.

Figure 23: Figure demonstrating mesh based retargeting from the source character (green) to the target character (red) for a kneeling down motion.

Figure 24: Figure demonstrating mesh based retargeting from the source character (green) to the target character (red) for a walking motion.

**Algorithm 4** Pseudocode for the mesh retargeting algorithm.

**Function** *Retarget* $(\mathbf{P}_s, \mathbf{N}_s, \mathbf{T}_s, \mathbf{w}_s, \boldsymbol{\theta}_s, \mathbf{P}_t, \mathbf{N}_t, \mathbf{T}_t, \mathbf{w}_t)$

// Subsample at ratio of 1:128

$\bar{v} = v_s/128$

$\bar{\mathbf{P}}_s, \bar{\mathbf{N}}_s \leftarrow \text{Subsample}(\mathbf{P}_s, \mathbf{N}_s, \bar{v})$

// Compute mesh mapping (if differing mesh topologies)

$\bar{\mathbf{P}}_t, \bar{\mathbf{N}}_t \leftarrow \text{MapMesh}(\mathbf{P}_t, \mathbf{N}_t, \bar{\mathbf{P}}_s, \bar{\mathbf{N}}_s)$

// FK Pass to get updated global joint positions and rotations

$\mathbf{GP}_s \in \mathbb{R}^{n\times j_s\times 3}, \mathbf{GR}_s \in \mathbb{R}^{n\times j_s\times 3} \leftarrow \text{FK}(\mathbf{T}_s, \boldsymbol{\theta}_s)$

// LBS Pass to get vertex global positions and normals for each frame

$\hat{\mathbf{P}}_s \in \mathbb{R}^{n\times v_s\times 3}, \hat{\mathbf{N}}_s \in \mathbb{R}^{n\times v_s\times 3} \leftarrow \text{LBS}(\mathbf{P}_s, \mathbf{N}_s, \mathbf{GP}_s, \mathbf{GR}_s, \mathbf{w}_s)$

// Normalize the source data

$\sigma = \hat{\mathbf{P}}_s.\sigma$

$\hat{\mathbf{P}}_s \leftarrow \hat{\mathbf{P}}_s/\sigma$

// Create empty array of Euler angles

$\boldsymbol{\theta}_t \in \mathbb{R}^{n\times j_t\times 3} \leftarrow \emptyset$

// Populate first frame using rest pose transformations

$\boldsymbol{\theta}_t^1 \leftarrow \text{MatrixToEuler}(\mathbf{T}_t)$

// Define Blend Mask

$\mathbf{B} \in (0,1)^{\bar{v}} \leftarrow \text{BlendMask}(\bar{\mathbf{v}})$

// Compute Cumulative Weights

$\hat{\mathbf{w}}_t \leftarrow \text{CumulativeWeights}(\mathbf{w}_t)$

// Iterate over frames

**for** $f = 1...n$ **do**

  // Iterate over predefined number of iterations

  **for** $k = 1...10$ **do**

    $\boldsymbol{\theta}_t^f \leftarrow \text{RetargetIteration}(\bar{\mathbf{P}}_t, \bar{\mathbf{N}}_t, \mathbf{T}_t^{f,\cdots}, \hat{\mathbf{P}}_s^{f,\cdots}, \hat{\mathbf{N}}_s^{f,\cdots})$

  **end for**

  // Initialize next frame from current frame

  $\boldsymbol{\theta}_t^{f+1,\cdots} \leftarrow \boldsymbol{\theta}_t^{f,\cdots}$

**end for**

**return** $\boldsymbol{\theta}_t$

**End**

**Algorithm 5** Given a mesh containing $v$ vertices, subsample such that the resulting number of vertices is equal to $\bar{v}$.

**Function** *Subsample* ($\mathbf{P} \in \mathbb{R}^{v \times 3}, \mathbf{N} \in \mathbb{R}^{v \times 3}, \bar{v} \in \mathbb{N}^{+}$)
  // Define subsampled vertex indices
  $\mathbf{S} \leftarrow \emptyset \in [0, v]^{\bar{v}}$
  // Pick random initialization point
  $\mathbf{S}^0 \leftarrow \text{Random}(0, v)$
  // Loop over number of subsampled points
  **for** $k = 1 ... v_s$ **do**
    // Get differences
    $\mathbf{Diff}^{i,j,\cdots} \in \mathbb{R}^{v,k,3} \leftarrow \emptyset$
    **for** $i = 1 ... k$ **do**
      **for** $j = 1 ... v_s$ **do**
        $\mathbf{Diff}^{i,j,\cdots} \leftarrow \mathbf{P}^{j,\cdots} - \mathbf{S}^{i,\cdots}$
      **end for**
    **end for**
    // Get furthest point away from average
    $\mathbf{S}_k \leftarrow \text{argmax}_{i...v} \left(\min_{j...k} \left(||\mathbf{Diff}_{i,j,\ldots}||^2\right)\right)$
  **end for**
  // Return subsampled points
  **return** $\mathbf{P}^{\mathbf{S}}, \mathbf{N}^{\mathbf{S}}$
**End**

**Algorithm 6** Pseudocode for a single iteration of the mesh retargeting algorithm.

**Function** *RetargetIteration* ($\mathbf{P}_t, \mathbf{N}_t, \mathbf{T}_t, \hat{\mathbf{P}}_s, \hat{\mathbf{N}}_s$)
  // FK Pass to get updated vertex global joint transformations
  $\mathbf{GP}_t \in \mathbb{R}^{j_t \times 3}, \mathbf{GR}_t \in \mathbb{R}^{j_t \times 3} \leftarrow \text{FK}(\mathbf{T}_t, \boldsymbol{\theta}_t)$
  // LBS Pass to get updated vertex global vertex positions and normals
  $\hat{\mathbf{P}}_t \in \mathbb{R}^{\bar{v} \times 3}, \hat{\mathbf{N}}_t \in \mathbb{R}^{\bar{v} \times 3} \leftarrow \text{LBS}(\mathbf{P}_t, \mathbf{N}_t, \mathbf{GP}_t, \mathbf{GR}_t, \mathbf{w}_t)$
  // Recompute axes of rotation
  $\mathbf{r}_t \in \mathbb{R}^{j_t,3,3} \leftarrow \text{Axes}(\mathbf{GP}_t, \mathbf{GR}_t)$
  // Normalize the data
  $\mathbf{GP}_t \leftarrow \mathbf{GP}_t / \sigma$
  $\hat{\mathbf{P}}_t \leftarrow \hat{\mathbf{P}}_t / \sigma$
  // Build the Jacobian for normals and positions
  $\mathbf{J} \leftarrow \text{Jacobian}(\hat{\mathbf{P}}_t, \hat{\mathbf{N}}_t, \mathbf{GP}_t, \mathbf{r}_t)$
  // Compute the error
  $\vec{\mathbf{e}} \leftarrow \text{Error}(\hat{\mathbf{P}}_t, \hat{\mathbf{N}}_t, \hat{\mathbf{P}}_s, \hat{\mathbf{N}}_s)$
  // Solve the system
  $\Delta\boldsymbol{\theta}_t \leftarrow \text{Solve}(\mathbf{J}^\top (\mathbf{J}\mathbf{J}^\top + \lambda\ I)^{-1} \vec{\mathbf{e}})$
  // Return added change
  **return** $\boldsymbol{\theta}_t + \Delta\boldsymbol{\theta}_t$
**End**

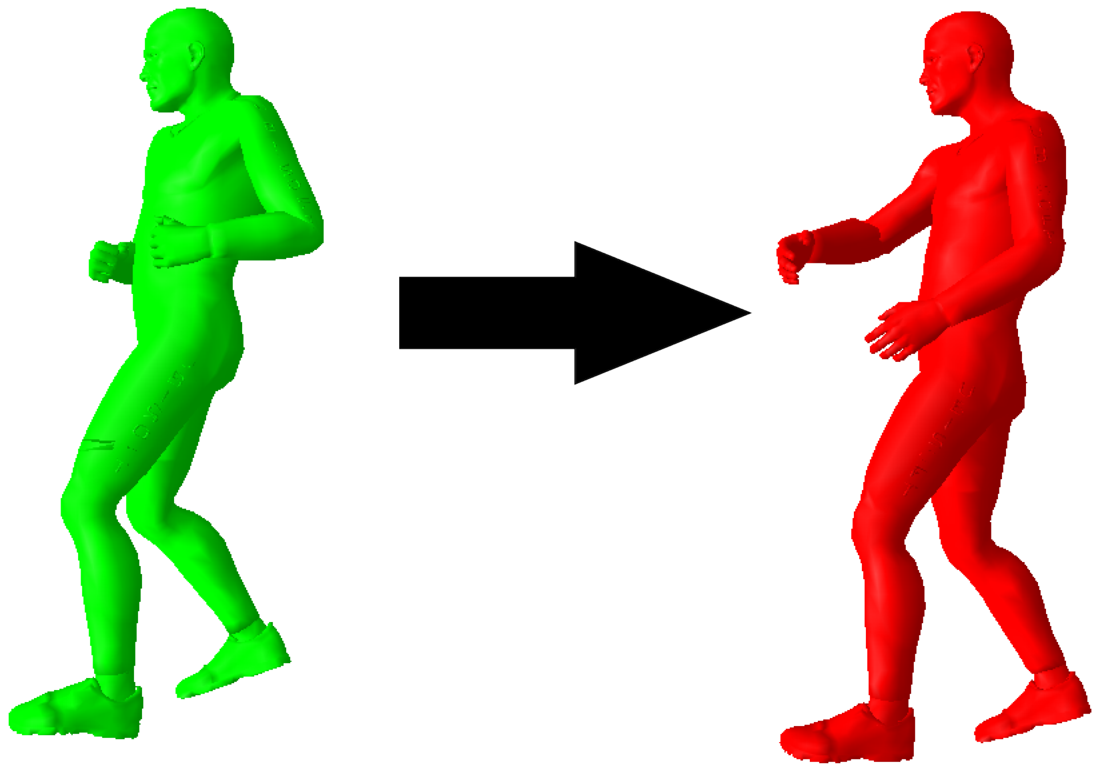

Figure 25: Figure demonstrating mesh based retargeting from the source character (green) to the target character (red) for a jogging motion.

# Chapter 5

# Conclusion

This thesis presents two methods for automating the animation pipeline. Together, they can help automate the animation pipeline in game studios by substantially reducing the manual effort needed from artists to cleanup and tweak animation data.

To address the issue of marker cleanup, we present a novel method for invalid marker detection and filling by comparing two different kinematic solutions to determine erroneous marker paths and using a marker filling algorithm to combine them. Our solution outperforms existing techniques for numerous examples, generating solutions that are closer to the ground truth than the alternatives. Our method achieves these results while also being less disruptive to existing motion capture work-flows, allowing the flexibility of tracking artists being able to intervene in cases where automatic solutions fail.

The main limitation of our method is the reliance on a robust kinematic solver in-order for both erroneous marker path detection and marker path filling. Although our method will often outperform the original kinematic solver by outputting data that is closer to the ground truth, it is ultimately limited by the performance of said solver and will suffer from many of the same shortfalls found within it. Its performance is likewise limited by the original algorithm, with the retargeting portion taking the longest to execute. However, as our method is agnostic to which kinematic solver is used, suffice that it is sufficiently robust, it would be of interest to attempt to utilize other kinematic solutions and compare the resulting marker paths.

Additionally, when professional marker tracking artists fill in gaps in marker data, they will often refer to a reference video of the original shoot in-order to recreate the

desired motion. This is often the case when large clusters of marker data go missing. Without this video, the artist will be unable to recreate the marker paths as the desired animation becomes ambiguous. Unlike the artist, our system does not have access to such data and thus will never be able to fully recreate the desired motion if a sufficient amount of markers are missing. To this effect, a potential avenue for future research would be to augment our system with a markerless kinematic solver that takes as input the reference video. Our marker filling method could then be used to combine marker paths generated from this solver with the other two set of paths, resulting in a solution that could potentially combine the capabilities of each system.

To address the issue of animation retargeting and aid the reuse of existing animation data, we present a method to automatically retarget animations from one rigged mesh to another by tracking the mesh vertex positions and normals. The solution grants fast and good quality batch animation retargeting and subsequently allows for the re-use of existing animation data previously captured without much manual intervention.

This retargeting method leaves many avenues for possible future works, some of which I have began exploring. The Jacobian Inverse Kinematics can be improved using Selective Damping S. R. Buss and J.-S. Kim, 2005 to change the priority of constraints to satisfy. Some problems may arise with the use of Euler angles, which have known limitations such as the Gimbal Lock. These can be avoided by using quaternions represented using exponential mapping Grassia, 1998. Regularization could be added to the solver in multiple ways, such as joint limits, mesh volumetric constraints and angle clamping. The marker cleanup could be augmented with this retargeting method, replacing the simple Jacobian IK method that it uses.

The animation pipeline remains a large process that is yet to be fully automated. However, the methods presented in this thesis can help speed up significant parts of the pipeline on the road to a fully automated system.

# Bibliography


Aristidou, Andreas, Daniel Cohen-Or, et al. (2018). "Self-similarity Analysis for Motion Capture Cleaning". In: *Computer Graphics Forum*. Vol. 37. 2. Wiley Online Library, pp. 297–309.

Aristidou, Andreas and Joan Lasenby (2013). "Real-time marker prediction and CoR estimation in optical motion capture". In: *The Visual Computer* 29.1, pp. 7–26.

Autodesk, Inc. (2020). *Autodesk MotionBuilder*. URL: https://www.autodesk.com/products/motionbuilder/overview.

Avril, Quentin et al. (2016). "Animation setup transfer for 3D characters". In: *Computer Graphics Forum*. Vol. 35. 2. Wiley Online Library, pp. 115–126.

Baerlocher, Paolo and Ronan Boulic (2004). "An inverse kinematics architecture enforcing an arbitrary number of strict priority levels". In: *The visual computer* 20.6, pp. 402–417.

Baumann, Jan et al. (2011). "Data-Driven Completion of Motion Capture Data." In: *VRIPHYS*, pp. 111–118.

Begon, Mickael, Pierre-Brice Wieber, and Maurice Raymond Yeadon (2008). "Kinematics estimation of straddled movements on high bar from a limited number of skin markers using a chain model". In: *Journal of biomechanics* 41.3, pp. 581–586.

Buss, Samuel (2004). "Introduction to Inverse Kinematics with Jacobian Transpose, Pseudoinverse and Damped Least Squares methods". In:

Buss, Samuel R and Jin-Su Kim (2005). "Selectively damped least squares for inverse kinematics". In: *Journal of Graphics tools* 10.3, pp. 37–49.

Butepage, Judith et al. (2017). "Deep representation learning for human motion prediction and classification". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 6158–6166.

*Doom* (1993).

Federolf, Peter Andreas (2013). “A novel approach to solve the “missing marker problem” in marker-based motion analysis that exploits the segment coordination patterns in multi-limb motion data”. In: *PloS one* 8.10, e78689.

Fragkiadaki, Katerina et al. (2015). “Recurrent network models for human dynamics”. In: *Proceedings of the IEEE International Conference on Computer Vision*, pp. 4346–4354.

Gleicher, Michael (1998). “Retargetting motion to new characters”. In: *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pp. 33–42.

— (2001). “Comparing constraint-based motion editing methods”. In: *Graphical models* 63.2, pp. 107–134.

Gløersen, Ø and P Federolf (2016). “Predicting Missing Marker Trajectories in Human Motion Data Using Marker Intercorrelations”. In: *PLoS ONE* 11.3, e0152616.

Grassia, F Sebastian (1998). “Practical parameterization of rotations using the exponential map”. In: *Journal of graphics tools* 3.3, pp. 29–48.

Herda, Lorna et al. (2000). “Skeleton-based motion capture for robust reconstruction of human motion”. In: *Proceedings Computer Animation 2000*. IEEE, pp. 77–83.

Holden, Daniel (2018). “Robust solving of optical motion capture data by denoising”. In: *ACM Transactions on Graphics (TOG)* 37.4, p. 165.

Hsu, Eugene, Sommer Gentry, and Jovan Popović (2004). “Example-based control of human motion”. In: *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*. Eurographics Association, pp. 69–77.

Jain, Ashesh et al. (2016). “Structural-RNN: Deep learning on spatio-temporal graphs”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5308–5317.

Julier, Simon J and Jeffrey K Uhlmann (1997). “New extension of the Kalman filter to nonlinear systems”. In: *Signal processing, sensor fusion, and target recognition VI*. Vol. 3068. International Society for Optics and Photonics, pp. 182–193.

Kim, Wooyoung and James M Rehg (2008). “Detection of unnatural movement using epitomic analysis”. In: *2008 Seventh International Conference on Machine Learning and Applications*. IEEE, pp. 271–276.

Kucherenko, Taras, Jonas Beskow, and Hedvig Kjellström (2018). "A neural network approach to missing marker reconstruction in human motion capture". In: *arXiv preprint arXiv:1803.02665*.

Kulpa, Richard, Franck Multon, and Bruno Arnaldi (2005). "Morphology-independent representation of motions for interactive human-like animation". In: *Computer Graphics Forum*. Vol. 24. 3. Wiley Online Library, pp. 343–351.

Lee, Jehee and Sung Yong Shin (1999). "A hierarchical approach to interactive motion editing for human-like figures". In: *Siggraph*. Vol. 99, pp. 39–48.

Lewis, John P, Matt Cordner, and Nickson Fong (2000). "Pose space deformation: a unified approach to shape interpolation and skeleton-driven deformation". In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 165–172.

Li, Lei, James McCann, Nancy S Pollard, et al. (2009). "Dynammo: Mining and summarization of coevolving sequences with missing values". In: *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, pp. 507–516.

Li, Lei, James McCann, Nancy Pollard, et al. (2010). "Bolero: a principled technique for including bone length constraints in motion capture occlusion filling". In: *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. Eurographics Association, pp. 179–188.

Liu, Guodong and Leonard McMillan (2006). "Estimation of missing markers in human motion capture". In: *The Visual Computer* 22.9-11, pp. 721–728.

Liu, Xin et al. (2014). "Automatic motion capture data denoising via filtered subspace clustering and low rank matrix approximation". In: *Signal Processing* 105, pp. 350–362.

Mall, Utkarsh et al. (2017). "A deep recurrent framework for cleaning motion capture data". In: *arXiv preprint arXiv:1712.03380*.

Meredith, Michael and Steve Maddock (2005). "Adapting motion capture data using weighted real-time inverse kinematics". In: *Computers in Entertainment (CIE)* 3.1, pp. 5–5.

Monzani, Jean-Sébastien et al. (2000). "Using an intermediate skeleton and inverse kinematics for motion retargeting". In: *Computer Graphics Forum*. Vol. 19. 3. Wiley Online Library, pp. 11–19.

*Quake* (1996).

Ren, Liu et al. (2005). "A data-driven approach to quantifying natural human motion". In: *ACM Transactions on Graphics (TOG)*. Vol. 24. 3. ACM, pp. 1090–1097.

Savitzky, Abraham. and M. J. E. Golay (1964). "Smoothing and Differentiation of Data by Simplified Least Squares Procedures." In: *Analytical Chemistry* 36.8, pp. 1627–1639. DOI: 10.1021/ac60214a047.

Shen, Wei et al. (2012). "Exemplar-based human action pose correction and tagging". In: *2012 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, pp. 1784–1791.

Shin, Hyun Joon et al. (2001). "Computer puppetry: An importance-based approach". In: *ACM Transactions on Graphics (TOG)* 20.2, pp. 67–94.

Software, Vicon (2019). *Vicon Shogun*. URL: https://www.vicon.com/products/software/.

Tak, Seyoon and Hyeong-Seok Ko (2005). "A physically-based motion retargeting filter". In: *ACM Transactions on Graphics (TOG)* 24.1, pp. 98–117.

Tarini, Marco, Daniele Panozzo, and Olga Sorkine-Hornung (2014). "Accurate and efficient lighting for skinned models". In: *Computer Graphics Forum*. Vol. 33. 2. Wiley Online Library, pp. 421–428.

Tits, Mickaël, Joëlle Tilmanne, and Thierry Dutoit (July 2018). "Robust and automatic motion-capture data recovery using soft skeleton constraints and model averaging". In: *PLOS ONE* 13.7, pp. 1–21. DOI: 10.1371/journal.pone.0199744. URL: https://doi.org/10.1371/journal.pone.0199744.

*Toy Story* (1995).

Wan, Eric A and Rudolph Van Der Merwe (2000). "The unscented Kalman filter for nonlinear estimation". In: *Proceedings of the IEEE 2000 Adaptive Systems for Signal Processing, Communications, and Control Symposium (Cat. No. 00EX373)*. Ieee, pp. 153–158.

Wang, Xin, Qiudi Chen, and Wanliang Wang (2014). "3D human motion editing and synthesis: A survey". In: *Computational and Mathematical methods in medicine* 2014.

*Wolfenstein 3D* (1992).

Zhang, Xinyi and Michiel van de Panne (2018). "Data-driven Autocompletion for Keyframe Animation". In: *MIG '18: Motion, Interaction and Games (MIG '18).*