TECHNISCHE UNIVERSITÄT
CHEMNITZ

BACHELOR THESIS

# Implementation of a modular pipeline to evaluate different rigging and retargeting techniques for virtual humans using CrossForge


Faculty of Computer Science
Professorship of Computer Graphics and Visualization

*Author:*
Mick KÖRNER

*Examiner:*
Prof. Dr. Guido BRUNNETT
*Supervisor:*
Dr.-Ing. Thomas KRONFELD


*A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science*

January 16, 2025

UNIVERSITY OF TECHNOLOGY CHEMNITZ

# *Abstract*

Professorship of Computer Graphics and Visualization

Bachelor of Science

**Implementation of a modular pipeline to evaluate different rigging and retargeting techniques for virtual humans using CrossForge**

by Mick KÖRNER

The objective of this thesis was to implement a modular framework that facilitates the integration of diverse rigging and retargeting methodologies, thereby enabling a comparative analysis of their quality and performance. The primary goal was to create a tool that automates or streamlines the process of creating a virtual character just from a scan. Therefore the open-source 3D framework CrossForge, developed at the GDV professorship, served as the basis for this work. As a result, a modular processing pipeline was created, incorporating a versatile application programming interface (API) that enables the integration and evaluation of diverse algorithmic implementations and thereby streamlining the efficient creation of prototypes. Furthermore, a sophisticated user interface has been developed that provides other researchers with the opportunity to easily integrate further skinning and retargeting methods. Thus, the implemented modules facilitate the comparison, and combination of various retargeting algorithms, as well as the evaluation of their correctness including potential advantages and drawbacks.

# *Acknowledgements*

I would like to thank my primary supervisor, Dr. Thomas Kronfeld, for his great support and advice, as well as his excellent and inspiring lectures.

I am also very grateful to my previous supervisor, Tom Uhlmann, for helping me evaluate an interesting but also important topic, and for his development of the CrossForge engine, a great tool that not only helped me better understand OpenGL and common 3D engine concepts, but also represents an important pillar for this thesis.

I would also like to thank Prof. Dr. Guido Brunnett for approving this topic.

# Contents

# List of Figures

# Algorithms and program code

# Chapter 1

# Introduction

## 1.1 Motivation

Virtual humans have played a significant role in the field of computer graphics due to their extensive range of applications, which traverse multiple research domains. These applications have enabled the exploration of previously uncharted research territory, thereby contributing to the advancement in respective fields.
The creation of a realistic virtual human remains a formidable challenge in the present day. Digital reconstruction techniques, such as Structure-from-Motion, have the capacity to produce a highly detailed surface replication of an individual, referred to as a human digital twin. However, the resulting mesh is static. Consequently, if the objective is to animate the scan with motion capture data, the mesh does not contain the necessary information how to apply this animation.
Despite the existence of techniques such as shape-from-silhouette, which utilizes motion capture to create animations by storing mesh sequences, the applications of these techniques remain limited due to the inherent coupling between motion and virtual characters.
The decoupling of motion from surface data into an underlying skeleton, termed "rig," has naturally emerged as the standard for both realistic and stylized virtual humans in contemporary media, including movies and video games.
Nonetheless, the process of creating a skeleton for a modeled or scanned character remains largely laborious. In order to obtain a skeleton of high quality, bones must be placed in a hierarchical structure and weights must be painted, both of which are still frequently done manually.
Specifically tailored skeletons necessitate specialized animations. Although standardized skeletal formats have emerged, they have a tendency to limit the expressiveness of characters. Furthermore, the capture of motion data and its corresponding formats can vary in terms of bone lengths and counts, depending on the subject's characteristics and the used system. This variation can lead to distortions and artifacts during playback, particularly when attempting to align data from different subjects. In addressing these challenges, motion retargeting approaches have been developed to transfer animations from one skeleton to another, taking into account variations in either bone lengths, structures, or both.
An investigation into the available methods and tools reveals that a significant number of implementations, if they are even available, are associated with specific skeletal hierarchies, utilize proprietary file formats, or are components of complex commercial software. Consequently, these implementations frequently exhibit a lack of adaptability and extensibility.
Currently, there is an absence of open-source pipelines capable of automatically rigging and animating a character without necessitating the use of overly complex tools

and workflows. Blender, the most widely used open-source 3D editor at present, lacks internal but even proper external solutions via plugins.

## 1.2 Objectives and Scope

The objective of this thesis is to implement a modular framework that facilitates the integration of diverse rigging and retargeting methodologies, thereby enabling a comparative analysis of their quality and performance. The primary goal is to create a tool that automates or streamlines the process of creating a virtual character just from a scan. The open-source 3D framework CrossForge, developed at the GDV professorship, will serve as the basis for this work.
The design and implementation of a modular processing pipeline, incorporating a versatile application programming interface (API), enables the integration and evaluation of diverse algorithmic implementations, thereby streamlining the efficient creation of prototypes. CrossForge, which is based on C++, still requires a workflow to integrate tools utilizing foreign languages.
In addition to the implementation of selected procedures, the focus is on developing a sophisticated interface that will provide other students with the opportunity to easily integrate additional skinning and retargeting methods in the future. Thus, the proposed tool should be interactive to facilitate the testing, comparison, and combination of various algorithms, as well as to evaluate their correctness and potential advantages and drawbacks.
A Graphical User Interface (GUI) for the purpose of interactive control of the processing pipeline will also provide an open-source option to automatically rig and animate three-dimensional scans for scientists with less experience, for those from other domains, and for hobbyists.

## 1.3 Summary of the Work

A review of the relevant literature in Chapter 2 will be conducted. This includes a recap of computer animation fundamentals, inverse kinematics, and animation constraints, as well as motion retargeting and auto-rigging. Popular and recent novel techniques will be explained and discussed.

Chapter 3 describes the structure and implementation of the developed retargeting pipeline including its graphical user interface and the designed API interfaces, to facilitate the integration and evaluation of additional retargeting methods. Due to the scope of this work and the evaluation of Chapter 2, an implementation of a motion retargeting solution is presented, while the integration of a autorigging tool is explained.

In addition to the aforementioned set requirements, the following elements are presented:

- An extensive graphical user interface for optional control of the environment and various processes
- The establishment of an animation workflow to test rigged characters and adapt animations

- Various tools required to facilitate constraint compliance of implemented and foreign-bound tools

The results of the proposed editor are concluded in Chapter 4. Potential bottlenecks are evaluated, and areas for improvement are investigated.

# Chapter 2

# Related Work

Prior to examining existing literature pertinent to this thesis, it is essential to review principles of skeletal animation in computer graphics, clear up various naming conventions used in literature to establish a foundation to prevent confusion.
In the domain of cross-paper naming, it is not uncommon for different designations to be used for the same concept or for separate concepts to be merged into a single term. Thus, it can be observed that many papers adopt a clear and consistent naming convention prior to review.
Section 2.1 will cover the fundamentals of 3D animation using skeletal systems. It begins with Section 2.1.1, outlining key components of skeletal structures, followed by Section 2.1.2, which discusses the organization of joints and bones. Section 2.1.3 explains the sequential application of joint transformations for posing, while Section 2.1.4 addresses the binding of mesh vertices to joints for realistic deformation. Finally, Section 2.1.5 explores the integration of motion capture data into skeletal animation, enabling real-world movements to inform animated sequences.

## 2.1 3D Animation Fundamentals

3D animation, an advanced field of computer graphics, has revolutionized the entertainment, gaming, and simulation industries by creating lifelike and dynamic visual experiences. The fundamental technique underpinning 3D animation is Skeletal animation. Utilizing Key frames, a method where pivotal frames are defined at significant points in an animated sequence.
Skeletal animation is the most prevalent form of humanoid animation. All major graphics engines are capable of supporting this type of animation due to its inherent simplicity. This has led to its early adoption as a standard feature in hobby engines, with numerous motion editing tools in the industry also built around it.
Figure 2.1 depicts a Virtual Character with an embedded Skeleton. The visual component is a three-dimensional model with materials, while the skeleton is typically concealed during deployment.

### 2.1.1 Skeletal Definition

Analogous to the anatomical structure of real animals, which consists of rigid bones connected to a skeleton that is moved by muscles, a similar analogy has evolved naturally in the field of computer graphics in a biomechanical manner.

In order to simplify the process of animating the geometric surface of a character, referred to as a mesh, without the necessity of specifying all vertex positions individually, the use of skeletal bones is employed.

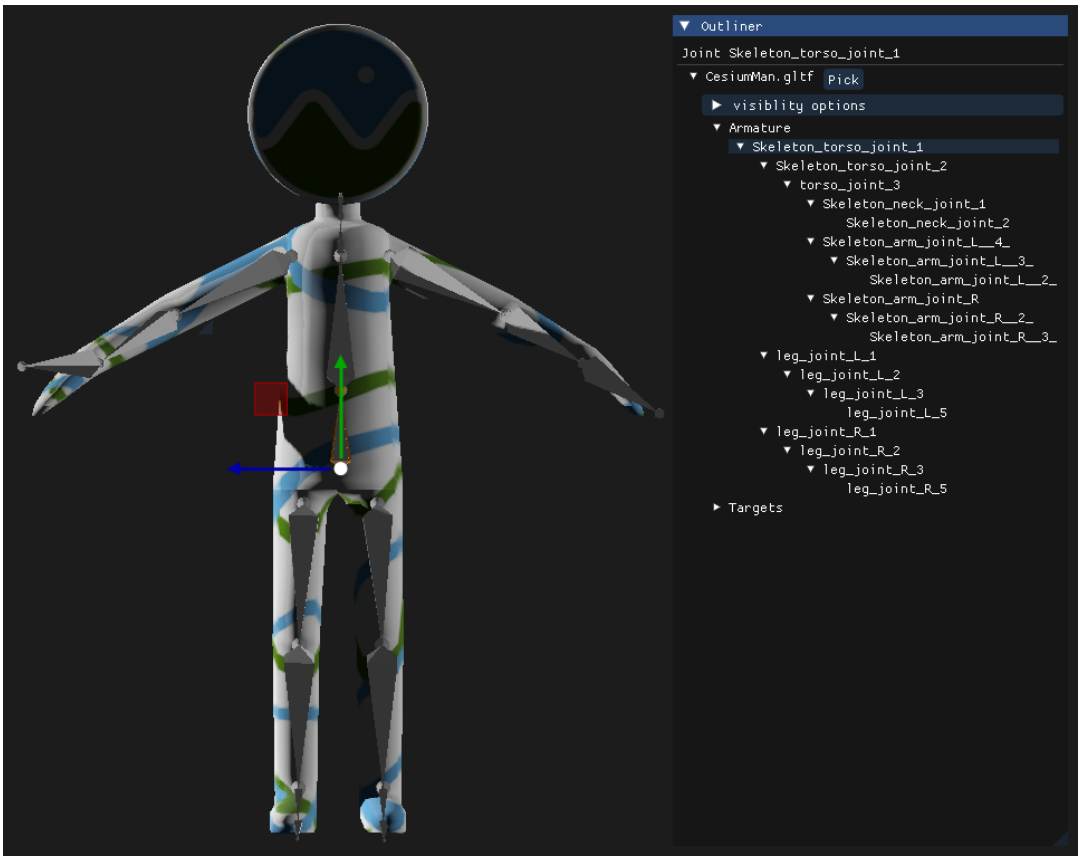


FIGURE 2.1: Illustration of the test model Cesium Man in the proposed editor. On the right side of the image, the skeletal hierarchy, with the root joint currently selected, is displayed.

These *bones*, also known as links, serve to represent rigid objects within a virtual character. They are associated with a length attribute, and while their lengths can be altered with scaling factors in animation data, typically employed in stylized animation, they generally remain constant in realistic rendering.

*Joints*, representing the connection points between bones, are characterized by up to three by up to three rotational DoFs (Degree of Freedom). Joints are the components concerned with motion and may contain any affine Transformation, but primarily rotational.

In most cases, bones are not explicitly defined in implementations. Rather, they are implicitly included in their parent or child joint.

In order to establish a relationship between bones through the formation of a hierarchy, implementations utilize either pointers or indexing. Typically, a joint is assigned references to its parent joint in the hierarchy and potential child joints, however contingent on the implemented restrictions, this may differ.

### 2.1.2 Skeletal Hierarchy

A *skeleton* is comprised of multiple bones arranged in a hierarchical structure, typically a tree-like configuration.

As illustrated in Figure 2.2, the skeletal structure of a virtual human is represented, with bones assigned unique names to facilitate their manipulation in various software applications.
It is advantageous to establish nomenclature for joints due to the unique characteristics they can acquire.
A *root joint* is distinguished by its absence of a parent joint, and any transformation applied to this joint is reflected in the actor's global movement. In animation, this joint is often translated in conjunction with a walking animation, ensuring that the actor does not remain stationary while walking. While this could be achieved through the use of a scenegraph, it facilitates the unification of motion playback across applications by circumventing the necessity for an additional abstraction.
*end effectors* represent bones without children. Depending on the application, endeffector may be a joint itself or a bone, sometimes having additional joint defined at its tips unused by animation data.
Various Formats between applications may treat Joints relative to bones differently, causing confusion, in example Figure 2.2 it is not clear which bone belongs to the root joint, while there exist unnamed extra joints unused at the end effectors.

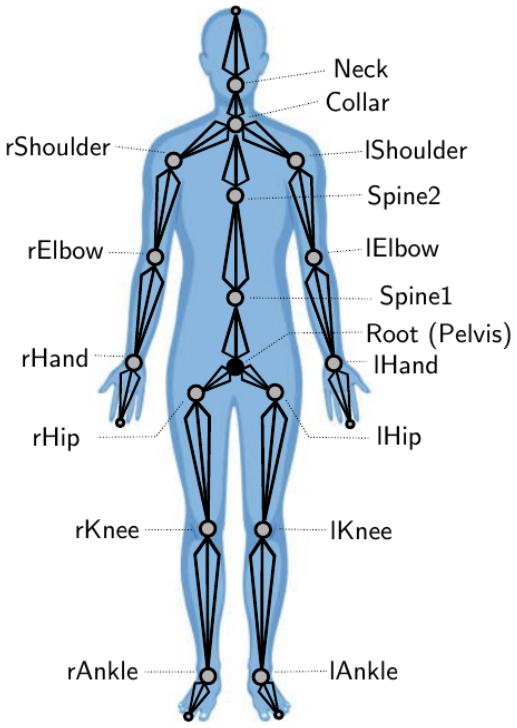


FIGURE 2.2: Example of human skeleton, note that bones and their parent joint are combined, this can cause confusion, in this example the root and collar joint have multiple bones.

A joint chain, is defined as a link of multiple joints with no branches, indicating that each joint has at most one child. Within the specified chain definition, a start joint and an end joint emerge, analogous to the root and endeffector in the skeletal hierarchy.

Tree structures are the most prevalent, but some systems permit circular structures known as closed loops (see Figure 2.3) or graphs, where multiple parent joints are allowed.
The employment of intelligently positioned bones enables the implicit enforcement of constraints, such as ensuring that the distance between two bones remains constant. This approach has been found to be particularly useful in the context of center of rotation correction. However, implementing this concept necessitates special consideration during various evaluation processes. Due to this reason, this is not supported in most systems, as is the case with the Blender.

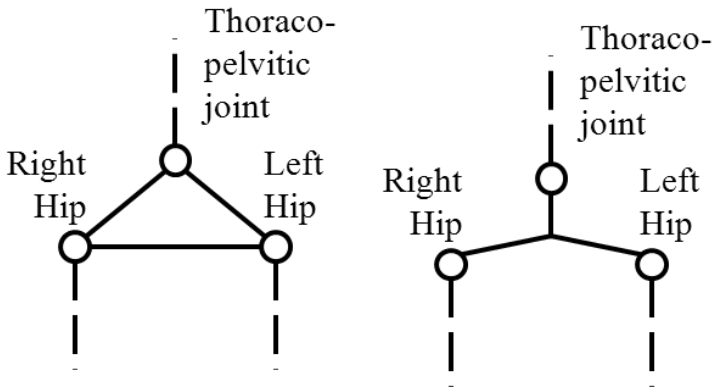


FIGURE 2.3: Visualization of an Armature with a loop, depending on implementation the tree becomes a graph. In this scenario, a set distance between the right and left hip is enforced. Image taken from [1]

An additional factor to consider is the potential for a Rigged character to comprise multiple roots. To illustrate, a character that incorporates a tool, such as a wrench for holding, could be rigged and transformed by a single bone or, alternatively, multiple bones in conjunction with other tools.
Common spaces in the graphics pipeline that have been established include window mapping (NDC and camera space), but more importantly for this work, World Space and Object Space.
In the context of skeletal animation, the object space of the actor is defined as the spatial domain of the character in a state of rest.
Furthermore each joint contained in the Skeleton has its own local space where the position of each joint is specified relative to its parent.
In order to visualize a skeleton or parent other objects in world space to joints, or joints to other objects in world space, for example, a tool to simulate some kind of work, it is necessary to know the position of a desired joint in pose $\theta$.

As previously discussed, joints influence the transformation of their respective child bones. To determine the position of joints relative to the actors object space, it is necessary to propagate the kinematic chain utilizing parent to child transformation beginning at the root joint. Given that rotational transformations of a joint affect child joints in an arc, changing pose parameters ($\theta$) results in a non-linear transformation of child joints.
Due to the discrepancy between pose parameters and the euclidian object space, pose parameters $\theta$ form the *pose space*. Conversely, the object space can be referred

to as the *work space* relative to the subject and may be extended to world space coordinates.

### 2.1.3 Forward Kinematics

Forward kinematics is the process of computing the working space from pose space parameters. Let $F$ be the forward propagation of the kinematic chain, and let $\theta$ be the current pose configuration, object space position, and rotation of the endeffector or any joint in between. It can be computed as follows:

$$T_i = F_i(\theta)$$

where $i$ is the respective joint.
Figure 2.4 visualizes this process, translation, rotation (and sometimes scale) of each joint determines the transformation of all child joints.

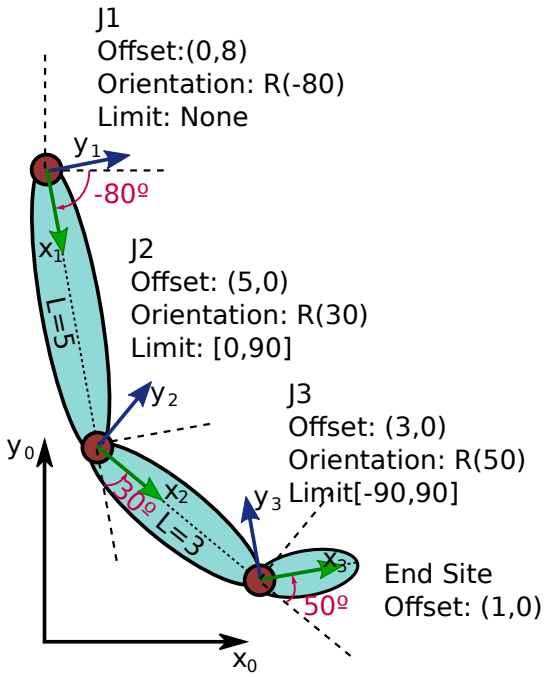


FIGURE 2.4: Example of a joint chain with respective local coordinate systems visualized. It is important to note that the global transform of Joint J2 depends on J1, and J3 depends on J2 and J1.

The propagation of the chain using affine transformations results in object space relative coordinates. To illustrate this principle, consider the example depicted in Figure 2.4 The transformation of "End Site" is computed as the affine combination of Joint transformations up to the root bone.

$$T_{EndSite} = M_{J3}M_{J2}M_{J1}M_{root}$$

where $M$ denotes the affine transformation of each bone respectively relative to its parent bone. Consisting of scale rotation translation applied in that order.
After evaluation, $T_{EndSite}$ contains the transformation from the actor's object space Base Coordinate System to the joint's local coordinate system.

Character modelers or scans are required to define the surface of a virtual character in an existing pose. Bones must be placed accordingly for that specific character.

Joint rotations of a motion are then applied relative to the rest pose angle of the embedded joints. *Rig* is thereby defined as a character with a respective skeleton embedded, ready for animation. This suggests that motion transfer between skeletons poses a challenge when rest poses differ.

*Bind Pose Matrices* are assigned to each joint and describe the transformation from the object space coordinate system of the rigged character to the corresponding joint in rest pose. Since rest pose remains constant, precomputing bind pose matrices offers numerous advantages. For example, it facilitates skinning and provides a concise representation of the coordinate base system of a joint, making it useful for transferring between applications.

*Inverse Bind Pose Matrix*, as indicated by its name, performs the inverse transformation of the bind pose matrix. When applied to the respective joint, it transforms the joint to the object space center of the character, thereby representing its actual base coordinate system transformation. In various paper and code sources, this is also commonly referred to as the Offset Matrix.

The *Offset Matrix* is a critical component for efficient realtime Skinning, a subject that will be addressed in the subsequent section.

### 2.1.4 Skeletal Skinning

A rudimentary definition of skeletons has been established. However, the original goal was to facilitate the animation of character mesh surfaces in an efficient manner. The concept of skeletal animation entails the abstraction of body parts into joints, thereby reducing complexity by manually defining the motion of every surface vertex. In the context of skeletal animation, the vertices of a character's surface, also referred to as *skin*, are abstracted to a bone.

This is achieved through the assignment of which vertex is affected by which bone. Additionally, given that Flesh is deformable and not rigid, there is a necessity to interpolate vertices in proximity to the joint of two bones, as observed in a 2-bone example and a vertex in between them, thus near a joint.

The determination of affection and interpolation, as well as the flexibility between two or more bones, can be achieved through the use of bone weights per vertex. The quantity of these weights is determined by the material defined on each vertex of a character. These weights can be authored manually through the use of weight painting or automated tools that utilize nearest neighbor or heat diffusion simulation.

Figure 2.5 shows an example of weights for the Humanoid Test model "Cesiumman" for the right elbow joint and its subsequent bone. The color blue is used to indicate a weight of zero, while red is used to indicate a weight of one.

Offset matrices of each joint are essential for animating the attached surface. They move corresponding affected vertices to the center of the coordinate system in object space, causing following transformations applied being equivalent to transforming relative to the Base Coordinate System of each joint.

*Skinning Matrices* describe the transformation of each joint from rest pose to a set pose $\theta$. This transformation ensures that local rotations of a joint are applied correctly. The joint transformation chain, in conjunction with the offset matrix, is combined into the skinning matrix, which is then applied in the vertex shader during rendering.
Skinning Matrix $S_i$ of Joint $i$ is computed as follows:

$$S_i = P_i \cdot J_i \cdot MO_i$$

where $J_i$ is the Local Joint transformation, $P_i$ are all parent transformations up to the root joint and $MO_i$ is the Offset matrix of joint $i$.

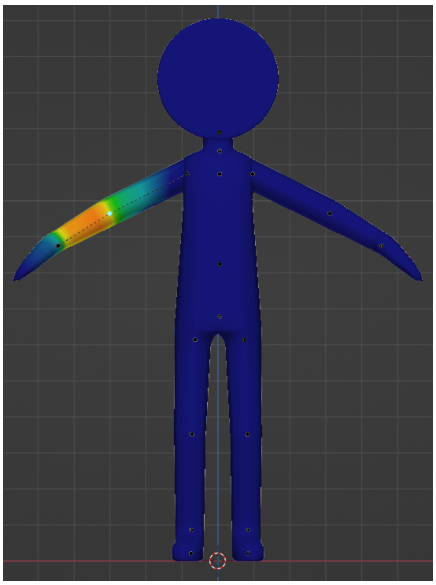

FIGURE 2.5: Simple Humanoid Test model Cesiumman visualized in Blender's [2] weight painting mode with the right Elbow Joint being selected. The gradient, ranging from blue to green to red, signifies an increase in weight for each vertex relative to the joint.

Linear Blend Skinning (LBS), Magnenat-Thalmann et al. [3], is the most common skinning method and demonstrates high processing speed due to its suitability for execution on GPUs via the utilization of vertex shaders. However, due to the approach of linear accumulating matrices, various artifacts can occur, such as volume loss and Candy-wrapping.
While various methods have been proposed to address these limitations, they fall beyond the scope of this thesis.

From a technical standpoint, the number of bones that can influence a vertex is not limited. However, to facilitate the incorporation of vertex shaders, it is common practice to impose a restriction of four bone weights per vertex.
Kavan [4] discusses the process of Linear Blend Skinning. Comparing it to Multilinear skinning and Nonlinear skinning methods.

### 2.1.5 Skeletal Animation using Motion Data

At a designated point in the sequence of motion playback, the rotational, translation, and scale values from pose parameter $\theta$ are applied to the local transform of each joint. This is typically denoted as the tuple $(t, \theta)$.

A pose configuration within an animation is often designated as a "keyframe," and a motion is comprised of multiple keyframes played sequentially. The timepoints per keyframe determine the time point at which a given pose is to be displayed within the animation.

Motion data is stored in two different ways: either as a sequence of poses or as a sequence of joint channels. These different storage methods result in slight differences in animation playback handling and performance, depending on the task at hand.
The sampling rate dictates the number of keyframes displayed per unit time.
To ensure a seamless display of animations at variable playback speeds and framerates, interpolation methods such as linear or other forms of interpolation are frequently employed to calculate in-between values.
A common technique employed in the field of gait analysis involves leveraging the sampling rate to generate a variable range of walking speeds from a single animation, thereby obviating the need for capturing or creating gait motion at every desired speed.
However, it is imperative to acknowledge that this approach represents merely a fraction of the extensive domain of motion processing and blending.

Meredith and Maddock [5] provide an exposition on a variety of motion capture formats.
Although the Biovision Hierarchy format is the most widely used animation format, it is noteworthy that formats such as FBX (Filmbox), COLLADA (DAE), and the more recent Graphics Library Transmission Format (glTF) are useful for storing motion in conjunction with the virtual character.
Motion editing is imperative in the creation of realistic and adaptable virtual humans, enabling animators and developers to efficiently reuse and modify existing motion data to suit various needs and scenarios.

## 2.2 Inverse Kinematics

As established in the preceding section, forward kinematics processes inputs from the configuration space of a rigged model, yielding working space coordinates that can be utilized for rendering a skinned mesh. Additionally, collision tests can be conducted, and further objects can be parented to joints.
A subsequent desire would be to know a Pose Configuration which causes an end effector to target a desired Point in Working Space, for example to achieve a dynamic grabbing motion.

Inverse kinematics describes the process of determining the joint configuration $\theta$ of a kinematic chain (e.g., a robot arm or a character's limbs) necessary to move the end effector (e.g., the hand or foot) to a desired target position.
End effectors should ideally move to desired targets smoothly. An Inverse Kinematics solver should be fast to compute and accurate, meaning ideal solutions are found and chosen. Another important desire is scalability. Solvers should be capable of handling complex kinematic chains with many joints and DoFs.
Inverse Kinematics has been demonstrated to be a valuable instrument in the domain of motion editing. Its applications extend beyond the domains of automation, simulation, and real-time interactive applications, encompassing the utilization by 3D animators themselves.

Animators employ Inverse Kinematics to intuitively animate characters without the necessity of individual bone rotations.

### 2.2.1 The IK Problem

The optimal approach would be to ascertain an inverse mapping of the forward kinematics mapping, $F$, as outlined in Section 2.1.3, in order to determine the pose configuration $\theta$ for a specified target position $T$:

$$\theta = F^{-1}(T)$$

Two fundamental challenges are associated with inverse kinematics. Firstly, the number of degrees of freedom (DoFs) in the joints typically exceeds the dimensionality of the desired end effector position, resulting in the existence of multiple solutions. Secondly, the incorporation of trigonometric functions in the forward kinematics function $F$ renders it a non-linear problem.

Figure 2.6 shows that in 2D a chain of more than 2 joints yields an infinite amount of solutions for a point $T$ within the bounds of the chain length. This already happens in 3d for chain length of two, where the intermediate joint between root and end effector moves on a circular curve.

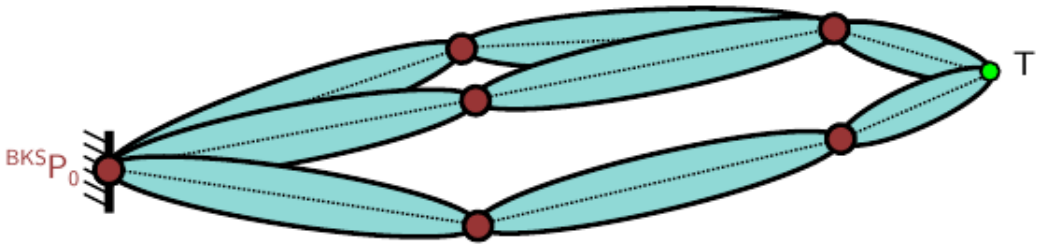


FIGURE 2.6: In the two-dimensional setting, for chains that possess more than two joints, an infinite number of configurations will exist that will satisfy the target's reach when the target is within bounds of the chain.

In addition, it is possible that solutions for a specified target point $T$ do not exist. Generally, three cases are distinguished: let $P_0$ be the root position, and let $|J_i|$ be the length of joint $i$:

1. $|T - P_0| > \sum_i |J_i|$ target is outside of reach, no solution
2. $|T - P_0| = \sum_i |J_i|$ target distance exactly one solution
3. $|T - P_0| < \sum_i |J_i|$ target is within reach, yielding infinite solutions

As illustrated in Figure 2.7, an additional problem may arise. Depending on the skeletal definition, certain points within the chain length may not be reachable. This phenomenon occurs when there are one long joint and multiple short joints, with the sum of the short joint lengths being less than the length of the long joint.
The determination of these cases is relatively straightforward, and as such, they are often incorporated as a preliminary step to circumvent the necessity of computing a

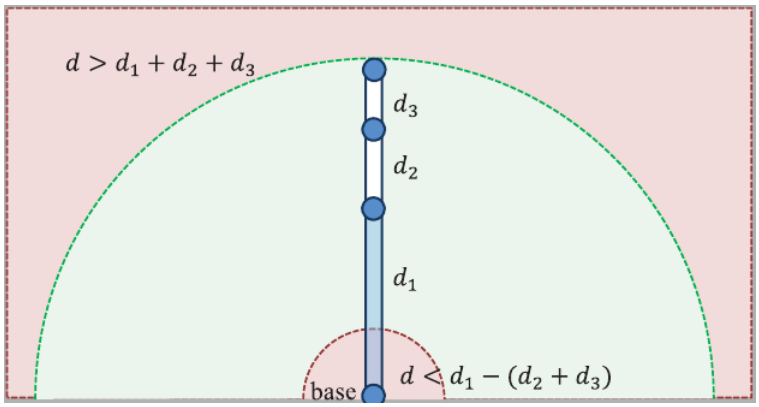


FIGURE 2.7: The joints exhibited comprise a single elongated joint and multiple short joints. These create positions inaccessible by the end effector in proximity to the root joint of the chain. Image taken from [6]

solution. To illustrate, in the event that the target is beyond the reach of the system, joint positions can be readily ascertained, given the absence of angular constraints imposed on the joints.

### 2.2.2 IK Surveys

A multitude of surveys have been conducted to evaluate the efficacy of inverse kinematic methods. These surveys methodically assess the performance of these methods, highlighting their respective strengths and weaknesses in relation to other methods and their various applications.
Aristidou et al. [6] present the most extensive evaluation and history of Inverse Kinematics techniques. They present the core concepts of these techniques and evaluate the analytic, numerical, Newton, heuristic, data-driven, and hybrid approaches for priorities, constraint support, performance, and potential applications depending on scalability, speed, and other characteristics.
Boulic et al. [7] compare analytic and numeric Inverse Kinematics (IK) algorithms for real-time full body posture recovery from partial and potentially noisy visual data.

### 2.2.3 Analytical Methods

The analytical approach entails the solution of a system of equations by inverting the forward kinematics formula of the corresponding armature. These solutions are referred to as closed-form solutions due to their reliability.
Lander [8] explained the analytical method with focus on animation and games, exploring the differences between forward and inverse kinematics, the challenges of solving IK problems, and some practical solutions for simple systems. Providing an Algebraic solution for a Simple 2D IK Example.

Figure 2.8 illustrates relevant variables for solving a 2-joint chain. The lengths $l_T, l_1$, and $l_2$ collectively form a triangle, and the utilization of trigonometric functions enables the determination of the angles $\theta_1$ and $\theta_2$.
Although this solution would be optimal due to its rapidity and numerical stability, the complexity of the system increases with each additional joint. As demonstrated

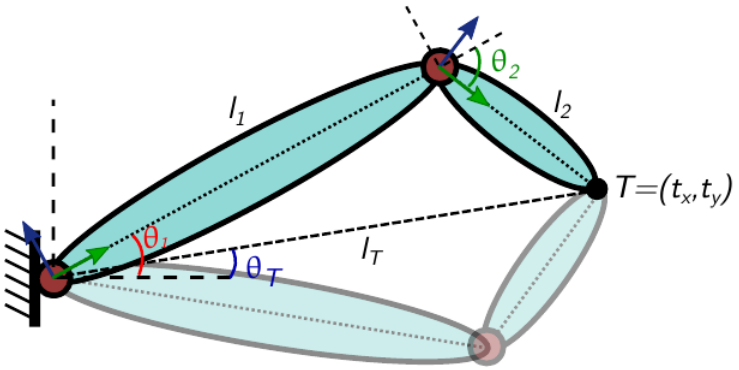


FIGURE 2.8: Visualization of relevant Variables for solving $\theta_1$ and $\theta_2$ analytically to reach point $T$ using Trigonomic functions. A Second Solution is Visualized with less opacity below.

by Pieper [9], examining the problem through the lens of six-degree-of-freedom systems highlights the increasing complexity.
However, the computational increase associated with longer chains is not the sole issue. The provision of either all solutions or predetermined ones is a formidable challenge. The identification of solutions that yield plausible results related to temporal locality is an even more arduous task.

### 2.2.4 Jacobian Methods

The Jacobian Inverse Method, also called inverse rate control, for solving Inverse Kinematics falls into the category of numerical solvers and represents the first Iterative Approach developed.
While not exactly determinable, Whitney [10] is often credited as one of the earliest works to introduce the use of Jacobian inverse for controlling robotic manipulators.

Buss [11] provides an in-depth Introduction to the Jacobian Inverse Kinematics Method and how the Jacobian Inverse works.

As previously stated in Section 2.2.1, the primary challenge in implementing inverse kinematics lies in the non-linear dependence between pose space and working space. This issue is further compounded by the existence of multiple mappings of $F^{-}1$ that satisfy $T$. Consequently, ascertaining the appropriate solution becomes a complex task.
Furthermore, the function $F$ is generally not directly invertible. Consequently, the inverse function, denoted here as $F^{-1}$, is not uniquely defined, or, in some cases, may not exist for all points in the workspace.
As a joint is rotated in the forward kinematic function $F$ the resulting end effector $i$ moves in a circular motion. This indicates that the forward kinematic function outputs a non-linear space in which the end effector moves. Figure 2.9 illustrates this phenomenon for a single joint.
Therefore, it can be observed that the non-linear transformation of $i$ can be approximated by a linear transformation for small amounts of movement. Let $J$ be a linear space mapping such that for a small movement of $\theta$:

$$\Delta T \approx J(\theta)\Delta\theta$$

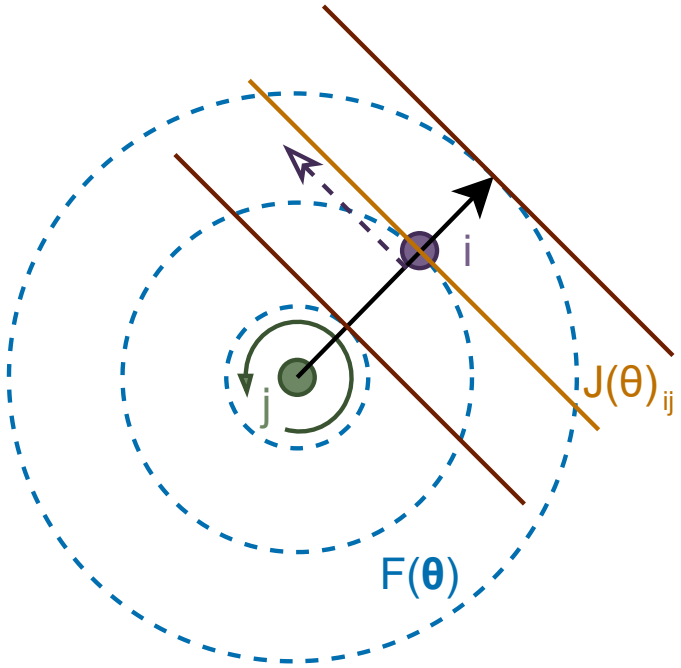


FIGURE 2.9: Illustration of the Jacobian for a single vector rotation around point $j$. The Jacobian entries $ij$ encode the relationship of change when $j$ on the position of $i$.

The Jacobian matrix $J$ is defined as the rate of change on position $T$ when the angles of joints on the $x$, $y$, or $z$ axis are adjusted by a small amount $\Delta$.

The explicit values of $J$ represent first-order partial derivatives of vector-valued functions with respect to its variables, thereby constituting the best linear approximation of corresponding functions near examined points.
In the context of inverse kinematics, the subsequent determination of these parameters is achieved through the modification of the corresponding angle of the armature by $\Delta$. This is then followed by the implementation of the forward kinematics function, which facilitates the determination of the alteration in the direction of the end effector relative to its previous position in object space.
With regard to the rate of change, the Jacobian matrix is commonly defined in literature using derivatives:

$$J = \left( \frac{\partial F(\theta)_i}{\partial \theta_j} \right)$$

Where $i$ are respective dimensions in which the target moves for each changeable angle $\theta_j$.
In this case, rather than inverting the forward kinematic function $F$, the Jacobian $J$ is inverted instead:

$$\Delta\theta \approx J^{-1}(\theta)\Delta T$$

The linear approximation error increases in proportion to the distance from the focal point, as illustrated in Figure 2.9. When proximate to the focal point, linear approximation is sufficient.

The variable change rate $\Delta$ is a key component in this analysis. It is noteworthy that the complete transformation is not necessary to move the end effector to the target. Two primary approaches have been devised to implement this task: open-loop and closed-loop Jacobian inverse kinematics.
In the context of an open-loop implementation, a single set of joint adjustments is computed based solely on the initial conditions and the desired pose, without the system monitoring or adjusting its output based on the actual pose achieved. This becomes more problematic the further away the target is, as the error between the linear approximation and the approximated function increases.

Closed-Loop Inverse Kinematics (CLIK) is a method in which a system continuously monitors the current pose of the end effector and compares it to the desired pose.
The discrepancy between these two poses is then utilized to make adjustments to the joint configurations. This process is repeated iteratively until the desired pose is attained or a predefined threshold is reached.
The incorporation of feedback ensures enhanced stability and accuracy by continuously correcting for any deviations from the desired state. Consequently, CLIK is generally preferred over open-loop implementations.

It is important to note that, given the nature of $J$ as a linear function, the process of computing its inverse is considerably less challenging than that of $F$, but still not without its challenges.
Jacobian matrices are occasionally ill-conditioned, exhibiting singular values that are excessively small or large. This can result in singularities and noise amplification. In certain instances, Jacobians can be non-invertible.To address this challenge, various approaches have been devised to approximate the inverse:

The Jacobian transpose is a straightforward method that involves directly multiplying the Jacobian transpose by the desired end-effector direction. This method does not require costly matrix inversion. However, it is less accurate, especially for ill-conditioned problems, requiring a small $\Delta$ and thus many iterations.

Liegeois [12] further developed the use of pseudoinverse for redundant manipulators. Projecting the solution onto the nullspace of $J$, enabling secondary goals such as avoiding joint limits or singularities. It can result in very large changes in joint angles and oscillations when targets are out of reach.

Damped Least Squares (DLS) is an extension of least squares, which minimizes $||J(\Delta\theta) - \Delta T||^2$, that introduces a damping term to improve numerical stability:

$$||J(\Delta\theta) - \Delta T||^2 + (\lambda||\Delta\theta||)^2$$

where the damping factor $\lambda$ is a positive scalar controlling the trade-off between accuracy, performance, and numerical stability. By penalizing solutions with large values in the solution vector, DLS discourages the solution from straying too far from zero, which is particularly beneficial when dealing with ill-conditioned matrices. This approach effectively reduces sensitivity to noise and extreme inputs while preferring smaller, more controlled solution vectors, thus enhancing the overall robustness of the inverse kinematics solution.
However, the damping factor $\lambda$ must be carefully considered, as large values stabilize the system but may slow down the convergence to the target positions, so some

methods choose the damping factor dynamically.

Further approaches include Selectively Damped Least Squares (SDLS) [13], which adjusts the damping factor separately for each singular vector of the Jacobian, or hybrid approaches.

The influence of the step size parameter $\Delta$ changes the relationship between angle and output change, depending on the inverting method used. In general, a small step size results in slower but more stable convergence. A large step size can lead to faster convergence, but may cause overshooting or getting stuck in local minima. Choosing the optimal step size is often a trade-off between speed and stability. It can be dynamically adjusted based on the error or other factors.

In general, Jacobian approaches generally yield smooth postures but are slow to compute. The primary reason why studies on heuristic approaches have been conducted.

### 2.2.5 Cyclic Coordinate Descent

Cyclic Coordinate Descent (CCD) was the first heuristic approach to solving IK. Kenwright [14] wrote a great overview summarizing the historical development of the method. It notes that because of its simplicity, it is not certain who published it, but it is credited to Wang and Chen [15].
Described as a trial-and-error method, its popularity stems from its fast calculation and relative ease of implementation, avoiding complex matrix math. Lander [16] delves into the practical aspects of implementing both analytical and iterative IK solutions, discussing improvement techniques and constraints.

The CCD method is a robust iterative approach that works by minimizing system error. It involves stepping through each joint in a chain (or hierarchy) and rotating each one to reduce the distance between the end effector and the target.At each step, a forward kinematic calculation is performed to determine the current position of the end-effector.The algorithm terminates when the end effector is within a defined tolerance to the target or after a predetermined number of attempts. Figure 2.10 illustrates multiple iterations of CCD in a three-joint example.

The primary variants of the CCD method are characterized by two distinct approaches: the top-down and the bottom-up. The top-down approach is frequently advantageous for upper limb exercises, particularly those involving the arms and hands, due to its capacity to promote greater movement in these regions.
Conversely, bottom-up is frequently advantageous for the legs and feet.
The Backward CCD algorithm 2.1 illustrates the top-down approach characteristic, in contrast to the forward variant of the algorithm, which represents the bottom-up approach.

However, further alterations are delineated as follows:
The "Bouncing" approach involves the recursive revisiting and updating of previous links. The Simple approach involves the simple revisiting of previous links, while the Smart approach involves the revisiting of previous links only if they have been altered.

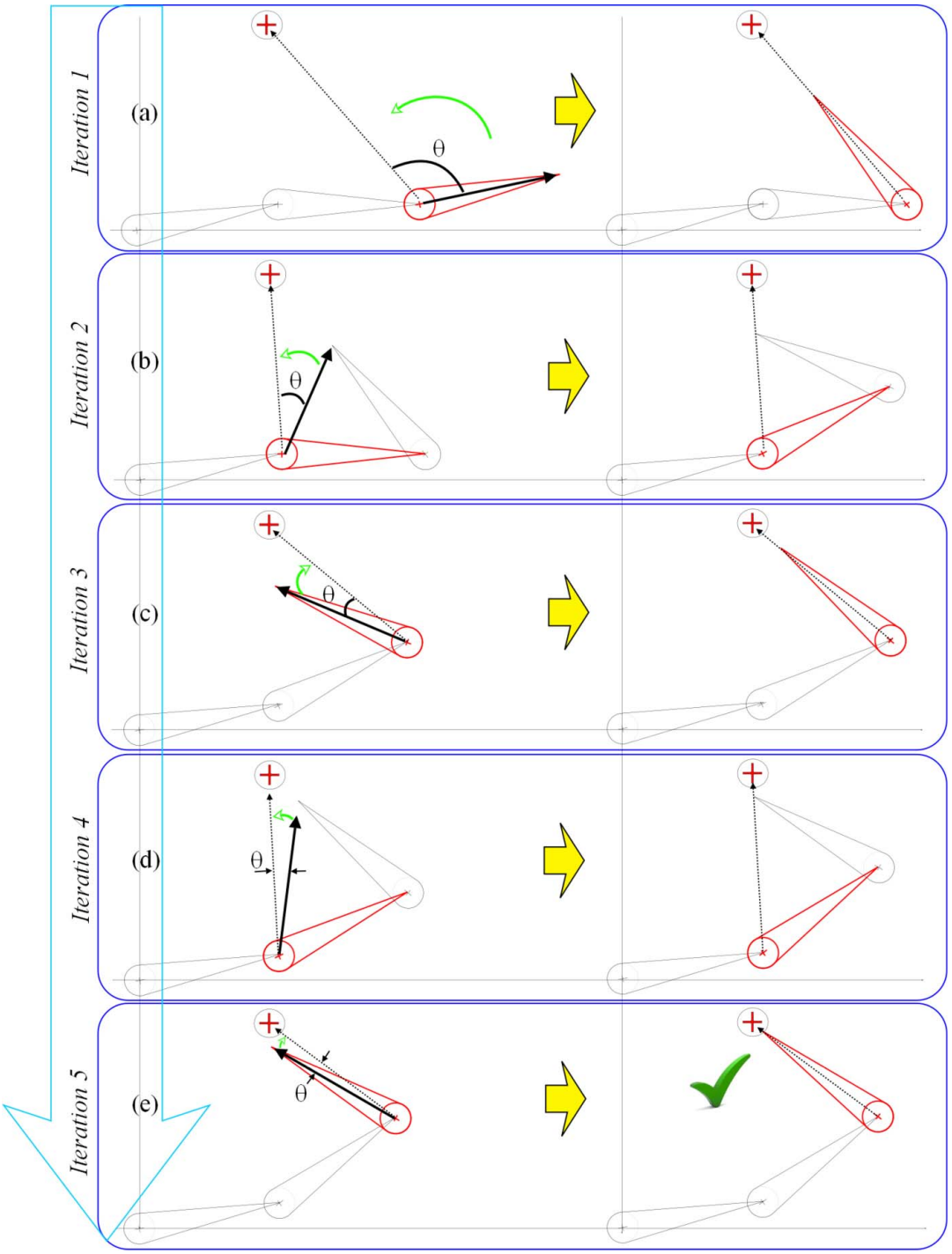


FIGURE 2.10: Three Joint Chain example of CCD, in each Iteration the Current Joint is rotated so that the Vector from current Joint to target and current Joint to endeffector align. Image taken from [14]

Over- and Under-damping modify the corrective rotation by a bias factor, thereby affecting the convergence speed and smoothness. Over-damping, characterized by magnifying the corrective angle, leads to increased oscillations, while under-damping, marked by reducing the corrective angle, may result in a smoother transition but necessitates more iterations.
For 3D character models, it is necessary to enforce complex joint limits; this can be achieved by decomposing the quaternion into twist and swing components, thereby

**Algorithm 2.1** BackwardCCDIK Algorithm. Utilizing dot and cross products to determine the necessary rotation angle, facilitating rapid computation. Taken From [14]

```
1: procedure BACKWARDCCDIK
2:     Input: e                                              ▷ threshold
3:     Input: k_max                                          ▷ max iterations
4:     Input: n                          ▷ link number (0 to numLinks-1 chain)
5:     k ← 0                                                 ▷ iteration count
6:     while k < k_max do
7:         for i = n − 1 to 0 do
8:             Compute u, v                          ▷ vector P_e − P_c, P_t − P_c
9:             Compute ang                                   ▷ using Equation 1
10:            Compute axis                                  ▷ using Equation 1
11:            Perform axis-angle rotation (ang, axis) of link i
12:            Compute new link positions
13:            if |P_e − P_t| < e then                       ▷ reached target
14:                return                                    ▷ done
15:            end if
16:        end for
17:        k ← k + 1
18:    end while
19: end procedure
```

enabling the accurate imposition of angular joint limits.
Due to its low memory overhead and real-time performance, CCD is well-suited for large crowds. However, numerous engineering solutions are required to make CCD practical, particularly to address the complex articulation of characters. Notably, CCD results in a characteristic rolling and unrolling of the chain prior to reaching the target, producing unrealistic motion.

### 2.2.6 FABRIK

In order to enhance the performance and address the rolling and unrolling problem of CCDs, Aristidou and Lasenby [17] proposed the Forward And Backward Reaching Inverse Kinematics (FABRIK) method, which converges in fewer iterations and incurs a lower computational cost per iteration.
Rather than utilizing angle rotations, FABRIK conceptualizes the identification of joint locations as a matter of determining a point on a line. This approach enables the optimization of temporal and computational resources by leveraging the previously calculated positions of the joints to identify updates through a forward and backward iterative process.
The forward reach is achieved by forcing the endeffector to the target position, and possibly temporarily disconnecting the chain root from the armature. Given the necessity of maintaining the chain root in a fixed position, the backwards propagation restores its original position. Figure 2.11 illustrates this process.
Algorithm 2.2 illustrates a full FABRIK iteration, including forward and backward reach. Additionally, a distance check is employed at the beginning to ensure the correctness of the algorithm.
FABRIK's methodology does not require the use of trigonometric or matrix operations, which are known to be comparatively slower. Furthermore, FABRIK avoids

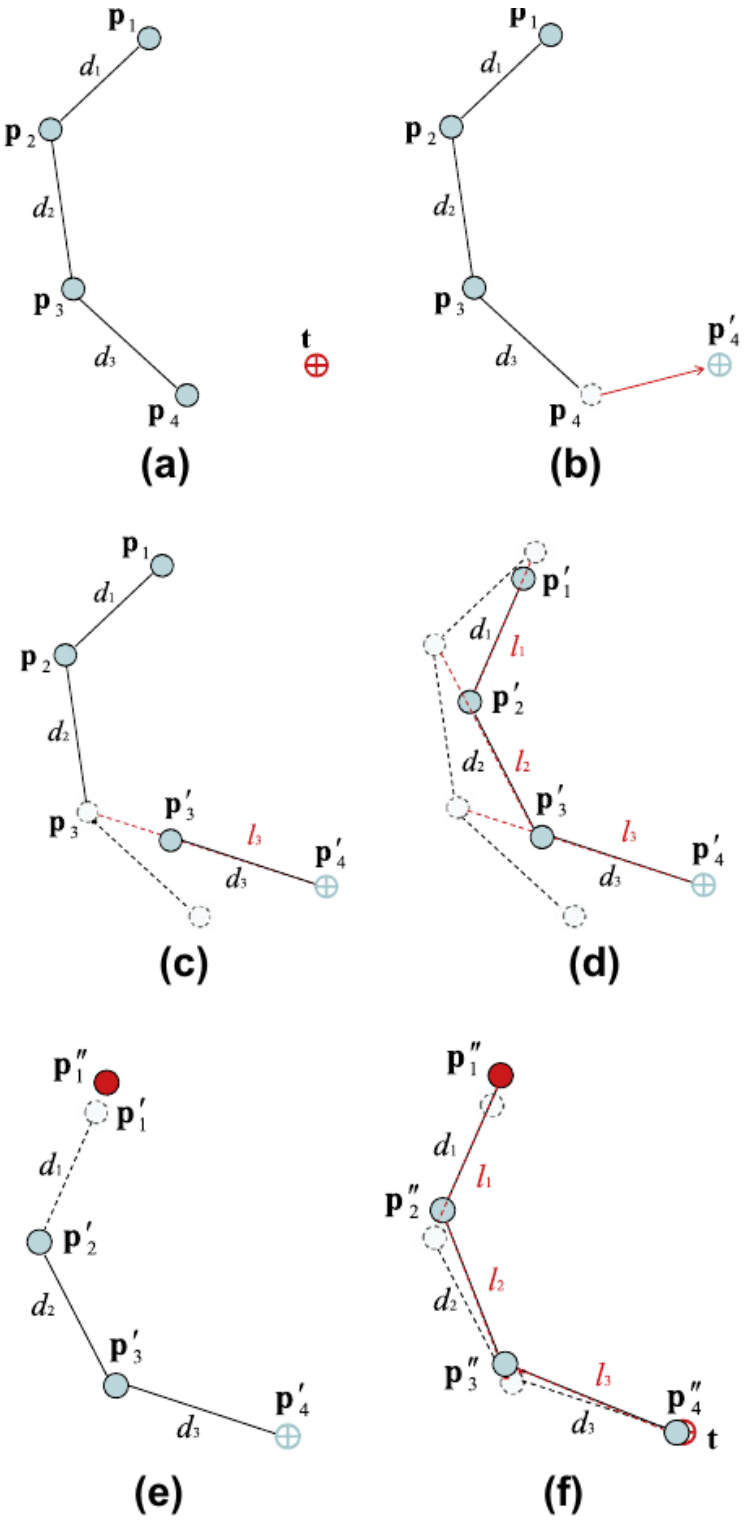


FIGURE 2.11: Four Joint example of Fabrik, (a)-(d) visualizes the forward reaching process, disconnecting the root at the last iteration. (e) and (d) restore root position. Image taken from [17]

rolling and unrolling of CCD, producing more natural results and distributing joint changes more similarly to jacobian inverse kinematics.

**Algorithm 2.2** A full iteration of the FABRIK algorithm, Taken from [17]

**Require:** The Joint positions $p_i$ for $i = 1, \ldots, n$,
**Require:** target position $t$,
**Require:** distances $d_i = \|p_{i+1} - p_i\|$ for $i = 1, \ldots, n-1$
**Output:** New joint positions $p_i$ for $i = 1, \ldots, n$

dist $\leftarrow \|p_1 - t\|$ ▷ The distance between root and target
▷ Check whether the target is within reach
**if** dist $> d_1 + d_2 + \ldots + d_{n-1}$ **then** ▷ The target is unreachable
  **for** i = 1 to n-1 **do** ▷ Find the distance $r_i$ between the target t and the joint position $p_i$
    $r_i \leftarrow \|t - p_i\|$
    $k_i \leftarrow \frac{d_i}{r_i}$
    $p_{i+1} \leftarrow (1 - k_i)p_i + k_i t$ ▷ Find the new joint positions $p_i$.
  **end for**
**else**
  ▷ The target is reachable; thus, set as $b$ the initial position of the joint $p_1$
  $b \leftarrow p_1$
  $dif_A \leftarrow \|p_n - t\|$
  ▷ Check whether the distance between the end effector $p_n$ and the target t is greater than a tolerance.
  **while** $dif_A > tol$ **do**
    ▷ STAGE 1: FORWARD REACHING
    $p_n \leftarrow t$ ▷ Set the end effector $p_n$ as target t
    **for** i = n-1 down to 1 **do**
    ▷ Find the distance ri between the new joint position $p_{i+1}$ and the joint p
      $r_i \leftarrow \|p_{i+1} - p_i\|$
      $k_i \leftarrow \frac{d_i}{r_i}$
      $p_i \leftarrow (1 - k_i)p_{i+1} + k_i p_i$ ▷ Find the new joint positions $p_i$ .
    **end for**
    ▷ STAGE 2: BACKWARD REACHING
    $p_1 \leftarrow b$ ▷ Set the root $p_1$ its initial position.
    **for** i = 1 to n-1 **do**
      ▷ Find the distance $r_i$ between the new joint position $p_i$
      $r_i \leftarrow \|p_{i+1} - p_i\|$
      $k_i \leftarrow \frac{d_i}{r_i}$
      $p_{i+1} \leftarrow (1 - k_i)p_i + k_i p_{i+1}$ ▷ Find the new joint positions $p_i$.
    **end for**
    $dif_A = ||p_n - t||$
  **end while**
**end if**

### 2.2.7 Other Methods

There are numerous alternative methods for solving inverse kinematics, but they are either too costly or not well established to be considered. These methods are therefore left for future work.
Newton Methods addresses IK as a minimization problem; however, it is both slow and difficult to implement. [17]

A further method is the mass spring model, proposed by Sekiguchi and Takesue

[18]. A numerical method that uses the virtual spring model and damping control. It is well-suited for redundant robots, robots which have many DOF.

Ramachandran and John [19] employ an approach similar to FABRIK, utilizing intersecting to achieve fast inverse kinematics.

## 2.3 Constraints Investigation

In the preceding section, we examined Inverse Kinematics in a theoretical manner as a motion editing tool. However, when considered in relation to the real world, there is still a lack of representation of physical limiting factors of bones that influence the range of motion and degree of freedom (DOF) related to the anatomy and structure of the bones themselves.
These factors are critical for the application of Inverse Kinematics to avoid issues of self-interpenetration and ensure the generation of realistic poses.

Aristidou et al. have described six most common anthropometric joint constraints, which are visualized in Figure 2.12. These include:

- Ball-and-socket joint: This joint limits angular rotation in the direction of the parent joint.
- Hinge joint: This joint allows motion in only one plane or direction about a single axis.
- Pivot joint: This joint only allows rotation on one axis.
- Condyloid joint: This joint has an ovoid articular surface that is received into an elliptical cavity, permitting biaxial movements.
- Saddle joint: This joint has a convex-concave surface.
- Plane joint: This joint is also called a gliding joint, and it only allows sideways or sliding movements.

### 2.3.1 Joint Constraints

Previously described IK methods primarily consider position-based target reaching, disregarding target rotation integration.A more thorough examination is necessary for each solver, as the IK solver's constraint to reach the target may be in opposition to the rotational constraint the target should be approached with.

Enforcing a variety of constraints poses a significant challenge, as it has the potential to hinder the convergence capability of an algorithm.
Constraints are often delineated in the context of particular inverse kinematics methods, with the objective of either enhancing performance or ensuring convergence. The implementation of these constraints is informed by the specific properties intrinsic to each algorithm.
Constraints are implemented as "corrections" subsequent to an iteration, wherein positional and/or rotational bounds are examined.

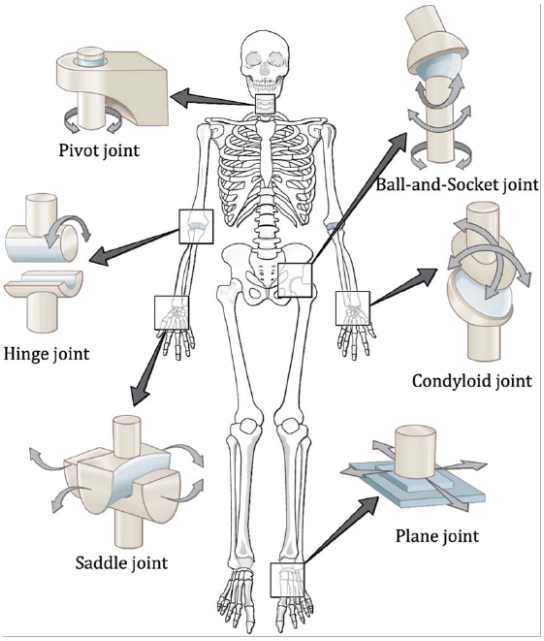


FIGURE 2.12: Various Constraint Types Visualized and where they could be used in a Virtual Human Skeleton. Image taken From [20]

Consequently, distinct implementation of constraints is necessary for various IK solvers.

Hauser [21] explains the constraints for Jacobian IK, considering soft preferences by appending cost terms to objective functions or using the pseudoinverse redundencies. Hard limits are enforced by projecting each joint value into its allowable range between iterations.

Blow [22] presents CCD with Quaternion Joint Limits. The article highlights CCD convergence issues, which are solved by employing simulated annealing. Resetting joints to a random pose if local minima have been detected. Furthermore, a loop hung in space for limiting reach is proposed, which limits the range of motion of the bone to reach windows, described by star polygons.
Kenwright [14] describes Angular Limits for CCD, highlighting Simulation annealing strategies and explanations on how to visualize these Angular Joint Limits.

Implementation of FABRIK constraints was previously described in great detail by Aristidou and Lasenby [17] when presenting the algorithm. The joint restriction procedure is divided into two interconnected phases: a rotational and an orientational phase. The methodology of simplifying the constraint problem from three to two dimensions is described as the target is repositioned and reoriented to be within the allowed range bounds on a plane. This allows for the definition of complex irregular limiting shapes. They have also showed Self-collision Determination and Joint Control in Closed-loop Chains [20].

Wilhelms and Gelder [23] have proposed the use of reach cones, which utilize spherical polygons to specify a region for allowable joint movement. As with the method described by Aristidou and Lasenby [17], representing a highly adaptable constraint definition, this method necessitates sophisticated user interfaces for definition and

visualization.

The incorporation of joint constraints constitutes a complex undertaking, both in terms of quantity and class design. The existing literature provides guidance on the optimal approach to implementing constraints:
Szauer [24] demonstrates that various Inverse Kinematics algorithms address constraints at distinct points within their architecture, thereby complicating the abstraction of constraints from solvers.
To illustrate, the implementation of a ball-and-socket constraint on FABRIK necessitates the computation of local joint spaces at each iteration, a process that degrades FABRIK's performance. Additionally, an abstracted implementation of the ball-and-socket and the hinge constraint is presented for both the CCD and FABRIK.
However, in the presence of more complex constraints, the implementation of solver-specific constraints may prove beneficial, and in some cases, it may be an essential requirement.
O'Neill [25] presents an interface example for constraints that utilizes strings for identification. While this example is well-abstracted, it introduces additional ambiguity, which results in less concise implementations.

Joint constraints play a vital role in dynamic animation operations, such as ragdoll simulation. However, their necessity for motion retargeting is debatable, as the baked motion should inherently incorporate constraint compliance. Constraints are particularly advantageous for retargeting operations when there is a substantial discrepancy in the structure and proportions between the source and target skeletons. This is in contrast to retargeting operations between humanoids, where constraints may not be as impactful.

### 2.3.2 Handling Multiple targets

Jacobian Inverse Kinematics is a natural solution to the problem of multiple targets, as it can incorporate all joints and targets into the Jacobian matrix, rather than just a single chain and a target.

The process of weighting CCD for multiple end effectors can be achieved through the calculation of the mean of the desired angles at the branches [14].
Aristidou et al. [20] employ the use of centroids (Figure 2.13), for the purpose of computing a mean position at sub-bases of the chain. Sub-bases are joints with branches in the armature. The target priorities can be archived by linear interpolating centroids between optimal sub-base positions, depending on their weight. To solve the armature optimally, all chains are propagated from the endeffector to the sub-base joint in the forwards reaching step, and then the backwards reaching step is continued.
It would be optimal to maintain adaptability of utilizing different IK solvers while concurrently evaluating multi-chain systems.
While centroid computation could be shared across FABRIK and CCD by interchanging values and converting each representation, this approach would not align with the Jacobian Process of multi-tree evaluation, as previously discussed.

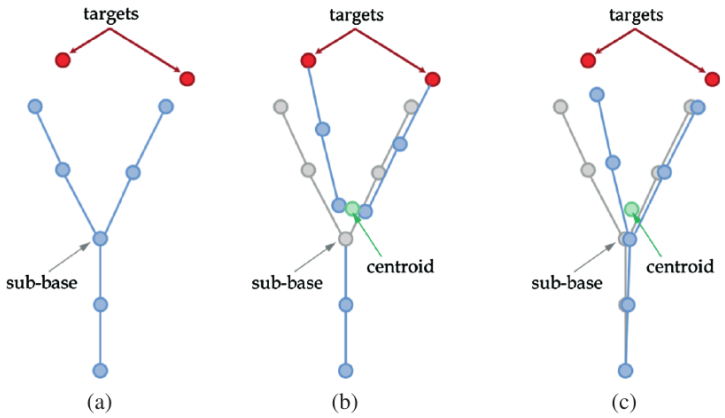


FIGURE 2.13: Solving a multichain using FABRIK entails the computation of the mean position of both subchains, designated as the centroid. This centroid is then utilized as the target position during the backward step. Image Taken from [20]

## 2.4 Motion Retargeting

Motion Retargeting is defined as the process of transferring motion data for a skeleton with a specific hierarchy and joint lengths to another skeleton that differs in one or both of these characteristics.
The transferred motion should imitate the motion of the source skeleton as closely as possible in terms of style.
In this regard, the quality of the retargeted motion is subjective and may vary for different use cases.
In the following sections, an examination of various motion retargeting approaches from recent years is conducted.
A clear categorization of motion retargeting approaches is lacking. These approaches can be subject to many aspects and can be categorized differently, following a categorization that is sensible for this work.
The naive retargeting approach entails the transcription of motion applied to a joint from the source character to the target character. This is achieved by defining joint correspondences between the source and target character.
However, this approach is susceptible to various issues, including but not limited to ground penetration, self-interpenetration, incorrect directions due to differences in rest pose, footsliding, and missing target objects.
The subsequent section will delve into the intricacies of these naive approaches.

### 2.4.1 Motion Cleanup Aproaches

Motion Cleanup Approaches are methods that try rectify the artifacts generated by the naive approach.
Tang et al. [26] presented a motion retargeting method for characters with heterogeneous topologies. This method converts movements between characters with different topologies, considering the number of joints, different coordinate systems, and different initial postures. The authors manually set the joint correspondences between the two hierarchies, shown in Figure 2.14, ignoring some unimportant leaf nodes and nodes without correspondence.
First, the initial posture of the source skeleton is adjusted, yet a detailed explanation of this adjustment process is not provided. Subsequently, adjustment matrices are

employed during the transfer of motion data to guarantee that the new skeleton exhibits equivalent movement to that of the source character following the initial posture adjustment. The rotation of each joint is then adjusted in accordance with this initial posture adjustment.

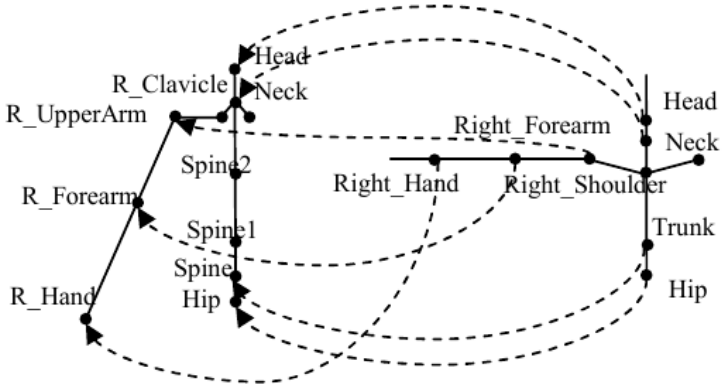


FIGURE 2.14: Example of defining Joint to Joint correspondences. Image take from [26]

Furthermore, they incorporate a scaling technique for the root node position, based on the leg length ratio between the source and target skeletons.

$$R'_{\text{child}} = R_{\text{parent}} R_{\text{child}} \tag{2.1}$$

Subsequent to the implementation of motion transfer, a rectification of the resultant motion is effected by ensuring its alignment with Cartesian constraints through the utilization of inverse kinematics.

Compared to the problem of defining joint-to-joint correspondence, Feng et al. [27] propose a very similar method for their open-source simulation framework, Smartbody.
Automated Skeleton Matching is achieved through the use of joints with similar names and the identification of body parts by their topology. The utilization of naming conventions to differentiate between left and right joints is a common practice. However, there are instances where these conventions are not followed, which can lead to complications, particularly in scenarios where motion capture systems employ numerical joint identifiers as the sole form of naming.
Their offline retargeting process incorporates the conversion of joint angles subsequent to the alignment of default poses and Jacobian-based inverse kinematics to enforce constraints, thereby describing its property to maintain target pose as much as possible.
The original motion trajectory involves warping according to the differences in leg lengths.
The framework also incorporates a range of capabilities, including locomotion, object manipulation, gazing, head movements, speech synthesis, and lip-syncing. However their framework is limited to humanoid or near-humanoid characters.

Ming-Kai Hsieh et al. [28] address the issue of retargeting across different articulated figures (e.g., humans, dogs, birds). The focus of their research is twofold: first, the transfer and concatenation of motions, and second, the generation of smooth transitions.

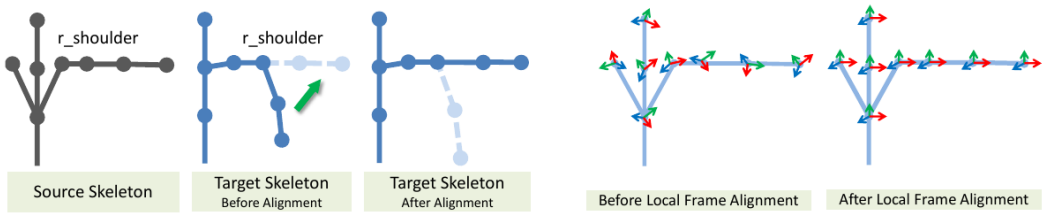


FIGURE 2.15: illustration depicting the alignment of the source and target skeleton, subsequent to a rectification process that aligns joint base coordinate systems. Image taken from [27]

Correspondence between the bone structures of the source and target skeletons is manually defined, and automatic skeleton matching techniques are described as challenging.
The focus is thus placed on the development of a user interface to facilitate other aspects of the retargeting process.
For joints between source and target mapping, one-to-one, many-to-one, or no mapping can be defined.
Initial poses are aligned through the recursive computation of the rotation angles between
corresponding bones, from the root to the leaf.
Furthermore, the motion data is transferred via the conversion of rotation angles from local to global coordinates, and subsequently back to the local coordinates.
In order to interpolate between two source motions that possess disparate skeletal structures, a meta-skeleton is constructed. This meta-skeleton combines the bone structures of both source skeletons.

### 2.4.2 Numerical approaches

Numerical motion retargeting methods are predicated on the simultaneous solution of the motion of the entire armature as opposed to that of individual body parts. This approach constitutes a departure from traditional techniques, which might adjust the position or orientation of limbs independently. The utilization of numerical optimization methods enables the consideration of various aspects of the armature and motion data.
Gleicher [29] delineates the challenge of mathematically solving motion retargeting, particularly concerning the definition of motion quality. This challenge has prompted the adoption of a pragmatic approach.
Gleicher's method is constrained to the identical structure of the skeleton, encompassing the connectivity of limbs, the types of joints, and the number of degrees of freedom. It only accounts for different segment lengths.
The proposed numerical solver utilizes the spacetime constraint approach, requiring basic features of motion that have been identified as constraints.
The proposed numerical solver employs the spacetime constraint approach, which considers the entirety of the motion rather than solely individual frames. This approach minimizes the magnitude of changes and restricts their frequency content to preserve the qualities of the transferred motion.
In contrast to this approach, the authors observe that IK solvers evaluate each frame independently, resulting in the addition of undesirable high frequencies to primarily smooth motion. The preservation of high frequencies is a salient feature of motion warping.

The system functions by calculating a motion displacement curve, which represents the discrepancy between an initial estimate of the motion and the final retargeted motion. This displacement is subsequently employed to generate the retargeted motion by adding it to the initial estimate. Motion-displacement maps are utilized as data representation, and cubic B-splines are employed to avoid encoding high frequencies into the generated motion.
Figure 2.16 presents an illustration of the displacement of motion in the context of an animation that has undergone retargeting to a smaller character, with a high weighting assigned to the constraint that aims to attain the box.

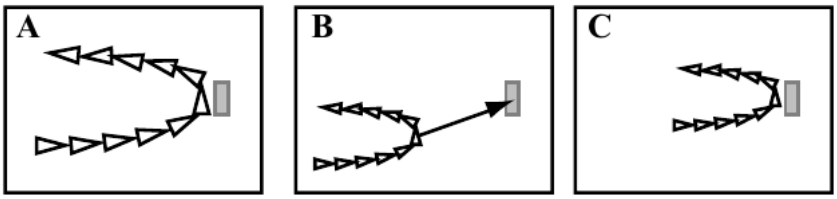


FIGURE 2.16: A aerial view of a character walking up to, picking up, and carrying away an object. B scaling relative to origin, character does not reach box. C if reaching the box is the only constraint, the entire motion can be translated. Image taken from [29]

Choi and Ko [30] utilize pseudoinverse Jacobian IK. Their proposed method, designated as Online-Motion Retargeting (OMR), facilitates real-time performance for characters with divergent proportions.
The fundamental premise of their approach hinges upon the utilization of a closed-loop Jacobian inverse kinematic method, a technique that is designed to preserve the distinctive characteristics of the original motion during the retargetting process, thereby ensuring the retention of unique motion details.
OMR has the capacity to track multiple end-effector trajectories in a concurrent manner from the source character. The algorithm processes data as it becomes available, obviating the necessity for the entire sequence beforehand.
Choi and Ko have presented a joint angle imitation of the source motion by incorporating it as a secondary goal in the pseudoinverse approach. This approach involves the intelligent use of kinematic redundancies. The primary task is to track given end-effector trajectories, and the secondary task is to imitate the joint angle trajectory $\theta$ as best as possible. Figure 2.17 illustrates their proposed feedback loop.
Pseudoinverse methodology employs the null space to represent redundant degrees of freedom. It encompasses all possible solutions by incorporating a term from the null space:

$$\dot{\theta} = J_1^+ \dot{x}_1 + (I - J_1^\dagger J_1) J_2^+ \dot{x}_2 \tag{2.2}$$

where $(I - J_1^\dagger J_1)$ projects $x_2$ onto null space.
Subsequently, the integration of $J_1^+ \dot{x}_1$ should yield $\theta(t)$, with the joint angles $\theta^{src}$ serving as a secondary target:

$$\dot{\theta}^{des} = J_1^+ \dot{x}_1 + (I - J_1^+ J_1) \dot{\theta}^{src}$$

The Jacobian matrix encompasses the entire armature, with zeros indicating the absence of correlation between joint angle and end-effector.
It is important to note that the DOFs for each end-effector can also be incorporated, taking into account the direction the target should be approached by.

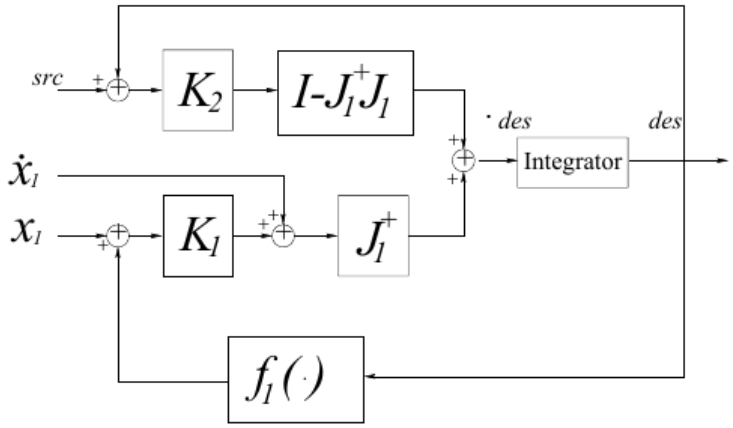


FIGURE 2.17: OMR's Closed-loop control scheme with the secondary task of joint motion imitation Both *src* and target pose $x_1$ are fed back into the system. Image taken from [30]

The algorithm has the capacity to minimize measurement errors in the restoration of captured motion by leveraging both measured end-effector data and joint angle data.

Monzani et al. [31] introduce an Intermediate Skeleton as a bridge between the Performer and End User Skeletons. This Skeleton is achieved by using Inverse Rate Control to enforce spatial constraints. The development of a plugin for 3D Studio Max with a User Interface is also discussed.
Intermediate Skeleton addresses the retargeting issue by reorienting its bones, a process analogous to that of Tang et al. [26]. It maintains the same node structure and local axis system orientations as the target Skeleton, while aligning the bones to correspond with the source Skeleton's bone directions.
Furthermore, a solution to facilitate seamless transitions between captured and corrected postures with multiple constraints is delineated. This solution utilizes easing functions to seamlessly enable and disable constraints at designated key positions. However, as with other methods, the user is required to establish a one-to-one correspondence between joints.

### 2.4.3 Kinematic Chain Based Approaches

Chain-based methods concentrate on delineating limbs or body parts to be retargeted. These methods utilize automation techniques to identify the specified body parts and frequently employ inverse kinematics for these individual body parts. This capability enables retargeting between significantly disparate skeletal structures. Some address the challenge of automatically defining limbs and correspondences between skeletons.

Du Sel et al. [32] addressed the issue of real-time motion retargeting for large crowds by developing a plugin for Autodesk Maya. They employed an intermediate skeleton representation called Golaem Skeleton for both the source and target skeletons. This approach enabled the conversion of motions to the intermediate representation for playback on any target.

The Golaem Skeleton is composed of a hierarchical structure of Bone Chains (limbs) that delineate the initial and final joints, as illustrated in Figure 2.18.

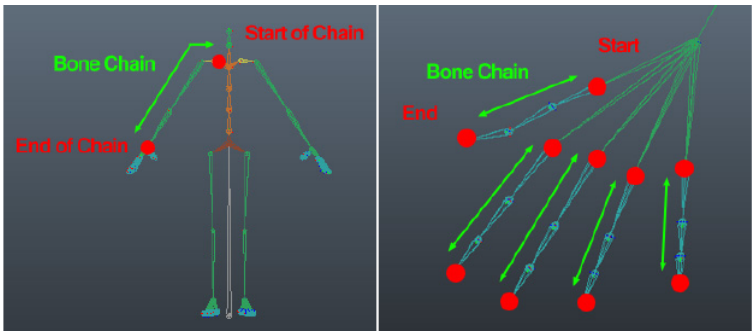


FIGURE 2.18: Example of Golaem Skeleton Bone Chains. Image taken from [32]

The allocation of distinct categories to bone chains, such as the pelvis, spine, limb, and effector, serves to denote sufficient character morphologies. The generation of limbs is automated, employing a search of the isolated branches and utilizing tags generated from chain world position to determine the legs or hands.
The skeleton motion mapping process involves the mapping of limbs based on their type and tag, facilitating surjective, injective, and bijective mapping. This allows for the mapping of the motion of a two-armed biped to a four-armed biped, illustrating the versatility of the system.
Furthermore, they define different movement modes, ability to take or take not rest-pose differences into consideration to allow for grasping targets or more natural motion playback.
A central focus on performance enables the simulation of large crowds, a proposal that involves the use of a simplified version of the goalem skeleton. This skeleton incorporates animation interpolation, mirroring of animation results, utilization of combined FABRIK and analytic IK, and precomputing and interpolating FABRIK results for different chain lengths, referred to as IKCache.

Abdul-Massih et al. [33] retarget motion with a focus on maintaining motion style to characters that are not humanoid.
The authors introduce the concept of "Groups of Body Parts" (GBPs), defined as user-defined groupings of joints and motion features that contribute to the conveyance of motion style.
GBPs are defined by multiple joints, with a "base" and "leading" joint, analogous to the root and effector of a joint chain, respectively. However, GBPs do not necessarily have to be a chain.
Prior to the implementation of motion transfer, motion features are categorized into two distinct categories: positional and angular. Positional features, in this context, denote a trajectory of representative vectors from the base to the leading joint. These vectors encapsulate fundamental components of the motion and are subsequently employed in positional constraints within the framework of an inverse kinematics solver. The magnitude of the representative vector is modifiable, contingent upon the ratio of the representative vector to the corresponding source GBP vector.
The angular amplitude features delineate the contraction or extension of body parts in relation to the neutral input motion, thereby capturing the motion style. The angular amplitude constraints are defined as the limits of rotation.

Subsequently, the matched GBPs are aligned semi-automatically based on their representative vectors, direction, and starting position.
Motion transfer is then computed on a per GBP basis, with the capability of being used injectively or surjectively, similar to the approach described by Du Sel et al. in Bone Chains. This process utilizes a space-time approach that enforces constraints.

### 2.4.4 Machine Learning Approaches

Retargeting human motion using neural networks is an emerging research area that focuses on transferring motion patterns from one character to another. These methods rely on various types of input data to accurately capture and transfer movement patterns. Methods related to the scope of this work can be categorized into three groups: methods operating at the skeleton level, methods transferring shape deformations at a dense geometric level and method that utilize a combination of skeletal and geometric information.

Methods operating on a skeletal level try to abstract out the dynamics of the source sequence and to reproduce it on a target character with a different morphology, that is to a skeleton with different bone lengths and possibly a different topology.
In the seminal work of Villegas et al. [34] a recurrent neural network architecture with a Forward Kinematics layer and cycle consistency based adversarial training objective is proposed. This method captures high-level properties of an input motion and adapts them to a target character with different bone lengths. It utilizes cycle consistency to learn to solve the Inverse Kinematics problem in an unsupervised manner, working online to adapt the motion sequence on-the-fly as new frames are received.
Aberman et al. [35] extended the scope of retargeting to skeletons with different topologies, but with the restriction that all topologies considered are homeomorphic. This approach leverages differentiable convolution, pooling, and unpooling operators that are skeleton-aware (Figure 2.19), meaning they explicitly account for the skeleton's hierarchical structure and joint adjacency. The method embeds the motion into a common latent space shared by a collection of homeomorphic skeletons, allowing retargeting to be achieved simply by encoding to and decoding from this latent space.

In contrast to skeleton-based methods, shape deformations based motion retargeting methods solely take character meshes as input and consider motion retargeting as pose deformation transfer.
The method proposed by Wang et al. [36] uses a hierarchical, coarse-to-fine approach. It starts with a mesh-coarsening module that simplifies the mesh representations to better handle small-part motions and preserve local motion interdependence. This is followed by a hierarchical refinement procedure that gradually improves the low-resolution mesh output with a higher-resolution one. Their method is evaluated on several 3D character datasets showing an average improvement on point-wise mesh Euclidean distance (PMD) compared to the state-of-the-art method and better preserve motion semantics in low-resolution meshes.
On the other hand, Ye et al. [37] focuses on skinned motion retargeting by directly modeling dense geometric interactions. The method uses semantically consistent sensors to establish dense mesh correspondences between characters and develops a dense mesh interaction field to capture both contact and non-contact interactions

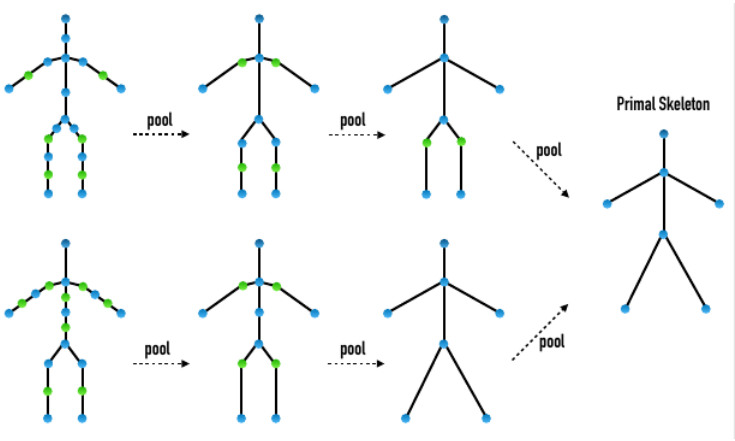


FIGURE 2.19: Illustration of the skeletal pooling process, facilitating the transfer of motion across a shared skeleton. Image taken from [35]

between body geometries. This method aims to preserve motion semantics, prevent self-interpenetration, and ensure contact preservation. It is evaluated on the Mixamo dataset and a newly-collected ScanRet dataset, achieving state-of-the-art performance.

A third category of methods try to combine the advantages of skeleton-based motion retargeting and shape deformation modeling to handle context and geometric detail. The reason is that skeletal retargeting methods achieve good results by encoding long-term temporal context and that recent methods to deform the shape based on skeletal deformations can accurately model geometric detail and are able to prevent self-interpenetration.
Biswas et al. [38] employ a hierarchical skeleton-based representation to learn a signed distance function on a canonical unposed space. This joint-based decomposition allows the model to represent subtle details that are local to the space around the body joint. Unlike previous neural implicit methods that require ground-truth signed distance function for training, their method only needs a posed skeleton and the point cloud of the posed mesh for training, eliminating the dependency on traditional parametric models or skinning approaches. The model achieves state-of-the-art results on various single-subject and multi-subject benchmarks by disentangling subject-specific details from pose-specific details.
The method proposed by Villegas and colleagues [39] focuses on preserving self-contacts and preventing interpenetration during motion retargeting. The method uses a geometry-conditioned recurrent network with an encoder-space optimization strategy to achieve efficient retargeting while satisfying contact constraints. The input to the method is a human motion sequence and a target skeleton and character geometry. The method identifies self-contacts and ground contacts in the input motion and optimizes the motion to apply to the output skeleton, preserving these contacts and reducing interpenetration.
Rekik et al. [40] present a novel framework for animating a target subject with the motion of a source subject without requiring spatial or temporal correspondences between the source and target shapes. Their method combines a geometry-aware deformation model with a skeleton-aware motion transfer approach. This allows the method to take into account long-term temporal context while preserving surface details. During inference, the method runs online, meaning input can be processed in

a serial way, and retargeting is performed in a single forward pass per frame. Experiments show that including long-term temporal context during training improves the method's accuracy for skeletal motion and detail preservation. Furthermore, the method generalizes to unobserved motions and body shapes. The method achieves state-of-the-art results on two test datasets and can be used to animate human models with the output of a multi-view acquisition platform.

### 2.4.5 Other approaches

A plethora of alternative approaches exist that are not readily classified or distinguished from the previously discussed methodologies.

Hecker et al. [41] address the issue of motion retargeting during the animation authoring stage. In a semantic approach, the definition of animation is decomposed into specialized (character-relative) and generalized (character-independent) space. In the proposed system, animators are tasked with the recreation of animations, for which various tools have been developed to facilitate the process of authoring animation. These tools include a conversion function between specialized and generalized spaces. Further specification of body parts is categorized into body capabilities (grasper, mouth, spine, root, etc.), narrowed down with spatial and extent queries (e.g., FrontMost, BackMost, RightMost, etc.).
Movement modes are classified into the following categories: identity, rest relative, scaling targets, ground constraints (i.e., picking up objects from the floor), secondary relative movement (e.g., reaching the mouth with the hand), and look-at. The application of movement occurs in one of these modes.

Sumner and Popovic [42] apply deformations of a source triangle mesh to a target mesh without skeletons. Users create a correspondence map with a few vertex markers, allowing transformations to be computed and mapped efficiently. Once the system of equations is factored, new deformations can be transferred easily. The method supports global constraints and is demonstrated through retargeting poses, facial deformations, and animation remapping.

## 2.5 Automated Rigging

In 3D character animation, efficient rigging techniques are crucial for streamlining the animation process. While methods like bone heat weighting have emerged to expedite weight assignment, they do not eliminate the need for manual joint placement. These semi-automatic approaches still rely on the animator's expertise to correctly position joints within the character mesh, a step that remains fundamental to the character's structure and movement. As we delve into various autorigging approaches in this review, it's important to note that true autorigging, aiming to automate both joint placement and weight assignment, continues to be an active area of research in computer graphics and animation.
Avril et al. [43] utilize energy minimization as a component of their methodology for establishing correspondence between the source and target character meshes. The approach initiated with a semi-automatic method in which the user places a set of markers on corresponding points of the source and target meshes, thereby providing an initial approximation of correspondence. An energy function is formulated to assess the quality of the correspondence between the meshes. This function generally

comprises terms for: Geometric similarity, smoothness, and user-placed marker constraints. The energy function is then minimized through an iterative optimization process, resulting in a dense correspondence between the source and target meshes.

### 2.5.1 Machine Learning Approaches

Baran and Popovic [44] pioneered a fully automatic rigging method that adapts a predefined skeleton to an unrigged character, providing an implementation called Pinocchio. The process, outlined in Figure 2.20, starts by approximating the medial surface of the mesh through the examination of C1-discontinuities in the signed distance field. Subsequently, a packing of spheres is conducted based on the medial surface position, extending towards the isosurface, with the largest spheres being placed first. Then, essential sphere centers are connected in a graph on which the predefined skeleton is embedded, depending on penalty functions.

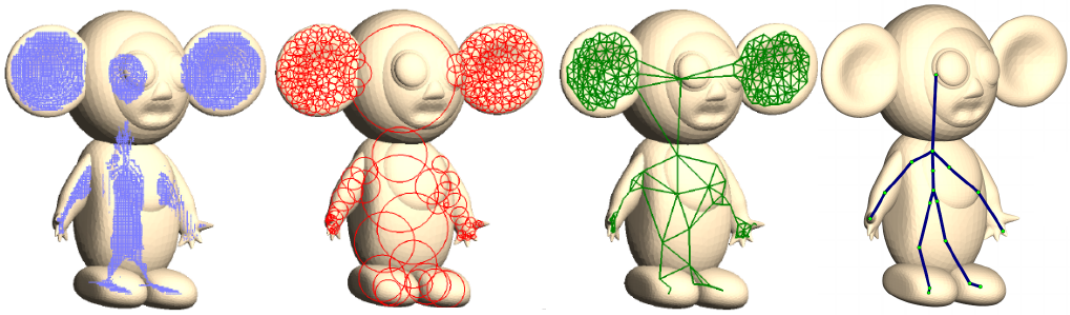

FIGURE 2.20: Process of Pinoccio skeleton embedding: The medial surface is first approximated, and subsequently, the character is packed with spheres to create a interconnected graph in which the target skeleton is embedded. Image taken from [44]

The determination of weights for various penalty functions is achieved through maximum-margin supervised learning, drawing inspiration from support vector machines. Finally, continuous optimization refines the location of joints, and heat diffusion simulates weights.
Notably, Pinocchio exhibits a lack of scalability due to its reliance on trained models that are constrained by predefined skeletal hierarchies.

Xu et al. [45] proposed RigNet, an end-to-end deep learning method to automate character rigging. Utilizing solely a 3D character mesh as input, RigNet generates customized skeletons for a range of humanoid and non-humanoid creatures, in addition to automatically computing surface skin weights. The architecture is designed as a deep modular system, comprising three components: Skeletal Joint Prediction, Skeleton Connectivity Prediction, and Skinning Prediction.
Skeletal Joint Prediction is a component that utilizes a learning algorithm capable of displacing mesh geometry towards potential joint locations. It predicts the location and number of joints for specific anatomical regions. A Graph Neural Network extracts features that are both topology-aware and geometry-aware. These features are then used to extract joints, incorporating both global and local mesh information.The process of symmetrization is performed according to the global bilateral symmetry plane. An optional parameter that can control the level of detail of the output skeleton is incorporated.

Skeleton Connectivity Prediction learns which joints pairs from the skeletal joint prediction step should connect to bones using shape- and skeleton representation, which are learned beforehand. The probabilities for each pair are then given. The Minimum Spanning Tree algorithm is then used to form a tree starting from the most likely bones. The network learns which joint pairs to connect with bones, forming a hierarchical skeleton tree structure, and uses both shape features and joint positions. The Skinning Prediction phase is responsible for computing skin weights based on intrinsic distances from mesh vertices to bones. Employing a graph neural network that operates on shape features and intrinsic distances from mesh vertices to the predicted bones, as well as volumetric geodesic distances from vertices to bones, the inverse of these distances is used to determine bone priorities, with the closest bones being selected first. Another module is used to compute the final skinning weights with learned parameters and a softmax function to obtain normalized weights.
A combination of manually authored loss functions during training is employed, similar to Baran and Popovic. However, they do not rely on pre-defined skeletal templates.

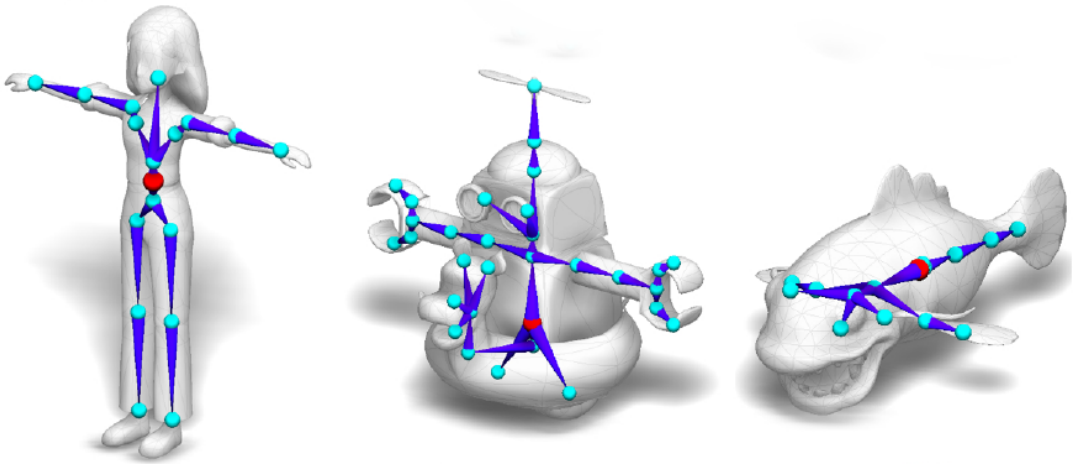

FIGURE 2.21: Example characters rigged with RigNet. Image taken from [45].

### 2.5.2 Other Approaches

Wade and Parent [46] employ voxelization of the model and construction of a Euclidean Distance Map (EDM), which approximates the squared distance from each interior voxel to the nearest exterior voxel. They then compute the Discrete Medial Surface (DMS) by identifying ridges and saddle points in the EDM. Identfying a central voxel and constructing a tree-like path to extreme points, prioritizing centralized paths through the model's deepest portions.
This path is smoothed and transformed into a skeletal graph, with edges recursively split based on approximation error and length. Joints are placed at segment endpoints and branching points, each with a local coordinate frame. Weighting vertices is achieved through the implementation of a weighted distance-based formula. The system's efficacy in generating quality control skeletons for diverse figures on modest hardware is noteworthy.

Poirier and Paquette [47] fit an existing skeleton to new characters. Utilizing a generated Reeb graph from the 3D mesh. The process involves filtering out unnecessary arcs and creating multiresolution versions of the mesh graph to enable flexible matching.
Symmetry detection groups similar subtrees by comparing node counts and arc lengths, assigning symmetry tags like "left" or "right". During retargeting, skeleton graph arcs are matched to the mesh graph recursively, using symmetry and shape descriptors. Joints are positioned using a weighted cost function that balances original skeleton characteristics with the target mesh shape.

Au et al. [48] extract a skeleton from the mesh surface iteratively using Laplacian smoothing to contract the mesh geometry. To avert the mesh from collapsing into a single point, the process incorporates attraction constraints, which are weighted to preserve essential geometric information. Subsequent to the contraction, a connectivity surgery process transforms the degenerate mesh into a 1D curve-skeleton by removing collapsed faces through a series of edge-collapse operations. It is noteworthy that the geometry contraction process does not ensure the skeleton will be centered. Consequently, a post-processing step is employed to refine the skeleton's embedding.

# Chapter 3

# Motion Retarget Editor

As already explained, current state of the art automatic rigging software like Mixamo and Accurig yield high quality results but remain closed source. This means that detailed insights into their processing pipeline and adaptation based on it are not possible. On the other hand, existing open source approaches are often limited to a specific method and skeleton format. In particular, there is a lack of open source implementations that allow the integration of complex motion retargeting algorithms and thus enable a comparison and optimization of such algorithms.
The aim of this work is to close this gap by extending the existing open source framework Crossforge [49] with appropriate API and interaction tools. CrossForge, developed by Tom Uhlmann at Chemnitz University of Technology, is a A C/C++ Cross-Platform 3D Visualization Framework using OpenGL. Since the complete source code is available, extensive optimization, profiling and debugging are possible, resulting in a simpler implementation process. Its flat design, simplicity and direct approach make CrossForge well suited for educational purposes and computer graphics research. In addition, CrossForge already has LinearBlend skinning and a simple animation controller. However, a major disadvantage of CrossForge is that it lacks many features found in commercial graphics engines. In particular, a user interface for scene and keyframe control, a picking system, and logic for dynamically loading and unloading actors at runtime.

In the follwing sections the implemented Editor will be presented, facilitating real time highly adaptable motion retargeting and incorporating various tools necessary to adapt properties interactivly. In section 3.1 the overall processing pipeline including the class hierarchy as well as the data handling and modularization will be presented. After that an in depth explanation of the implemented classes and scene management is carried out in section 3.2, the developed user interface is presented in Section 3.3 and the implemented animation system in Section 3.4. Subsequently, in section 3.5, the inverse kinematics algorithms implemented in this work are presented. Building on this foundation, their integration in the processing pipeline are explained. Starting with skeleton generation and matching in section 3.6 and the subsequent motion retargeting in section 3.7. The chapter concludes with an explanation of the API developed for the integration of additional tools.

## 3.1 Processing pipeline

As illustrated in Figure 3.1, the class hierarchy of the Motion Retargeting Editor is depicted, including the most significant classes. It is noteworthy that both the Motion Retargeting and Autorigging solutions have the capacity to access all objects belonging to the presented classes. While these solutions can access a variety of data

structures from Character Entity class `CharEntity` Instances, they do not necessitate the Character Entity class itself, as its members are exposed as public, thereby preserving the modularity of CrossForge.

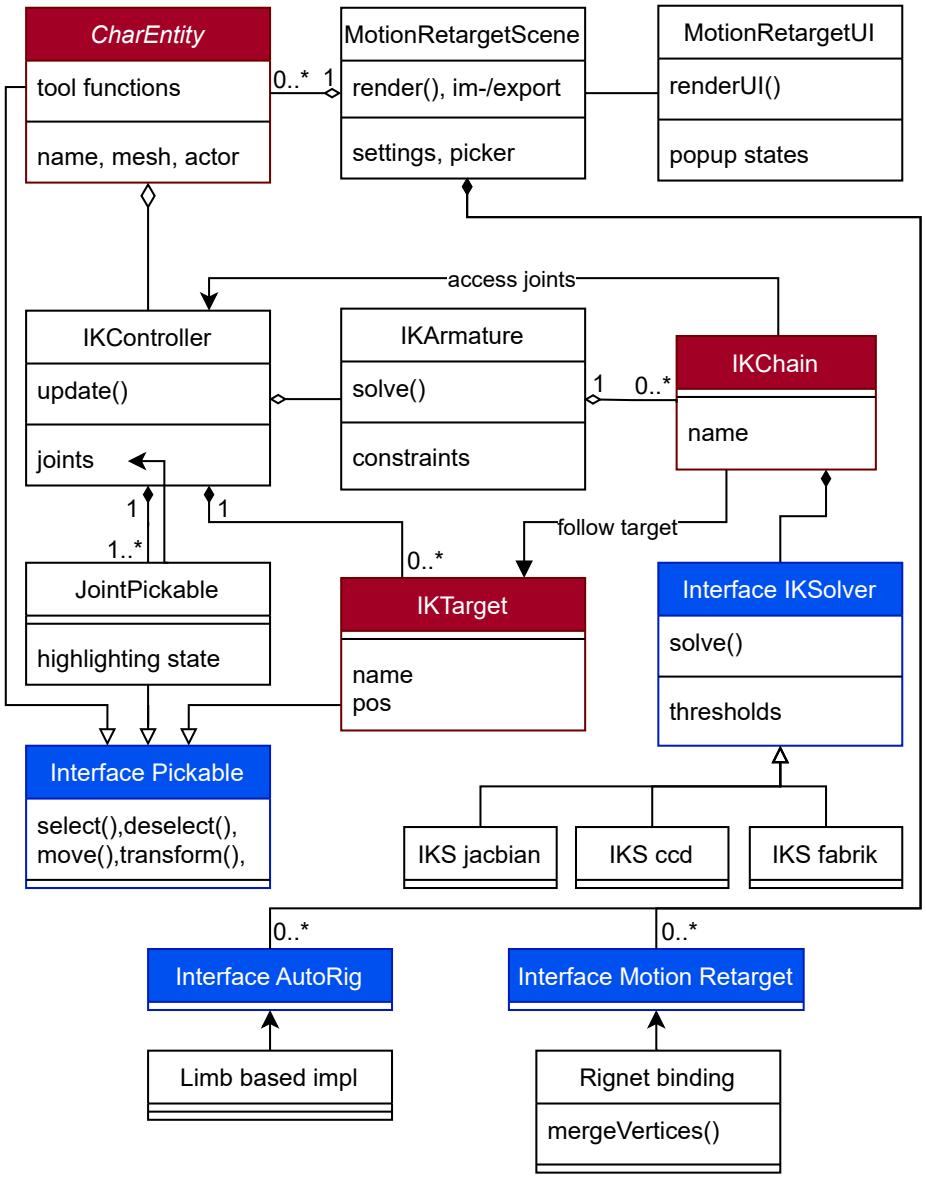


FIGURE 3.1: UML class diagram visualizing essential classes and their relationships of the proposed Framework. Blue indicates interfaces, and red are dynamically instantiable objects during runtime. AutoRigger and Motion Retargeting solutions can access various listed classes over the `CharEntity` class without restriction.

In order to achieve practical results with the developed prototype, RigNet [45] was selected as an example for a foreign tool integrated into CrossForge.

In the absence of more sophisticated open-source motion retargeting solutions, particularly a kinematic chain-based solution, this work introduces a kinematic chain-based motion retargeting approach with various novel aspects.
In comparison to alternative methods, this approach permits the utilization of distinct IK solvers for each limb, encompassing all previously introduced implementations, including CCD, FABRIK, and Jacobian IK.The approach demonstrates a high degree of adaptability, allowing for the adjustment of various IK solver parameters and the incorporation of an optimized approach to imitate joint angles adaptively for each chain.Additionally, it facilitates the adaptability of end effector goals through interpolatable target position transfer.

### 3.1.1 Data Handling and Modularization

CrossForges Foundation also maintains a set of derivable classes. For instance, IKController is derived from the previously implemented SkeletalAnimationController, thereby overriding or overwriting its functionality.
While a node abstraction of joints could yield more clarifying operations, proposed operations should be minimal, utilizing few abstractions in order to be applicable in other frameworks. In addition, use of bindings provided in the section on related basics is recommended, as it avoids potentially unnecessary computations.
The implementation of a feedback loop is imperative for the validation of nascent algorithms, ensuring the accuracy and progression of the development process. Ideally, the developed prototype should be maintained in a functional state throughout the development phase. However, this approach is hindered by the difficulty of predicting the final class structure.

C++ smart pointers, particularly `std::shared_ptr` and `std::weak_ptr`, offer a secure and efficient mechanism for referencing objects.
Listing 3.1 demonstrates a recurring pattern identified in the proposed prototype. In contrast to shared pointers, weak pointers do not augment the object's reference count. The utilization of `lock( )` facilitates the efficient and concise verification and access of an object, thereby eliminating the necessity for manual invalidation checks and promoting reduced class coupling.

LISTING 3.1: Demonstrating weak_ptr usage in C++. Safely accessing shared resources without ownership.

```
1  // class A
2  std::shared_ptr<int> sharedPtr(new int(42));
3
4  // class B
5  // assign member std::weak_ptr<int> m_weakPtr using sharedPtr
       ↪ parameter
6  m_weakPtr = sharedPtr;
7
8  // during a procedure in class B
9  if (auto obj = weakPtr.lock()) {
10     // Access the sharedPtr through obj
11 } else {
12     // sharedPtr is no longer accessible, act accordingly
13 }
```

Therefore, the utilization of smart pointers enhances the maintainability of code and improves the overall safety and efficiency of a prototype. In the event that the referenced object is accessed but invalidated by being deleted, the corresponding module is able to act accordingly, keeping a separate logic for corresponding classes more easily and improving code structure by reducing state management significantly.

It is imperative that the program's usage is interactive to facilitate the potential for user correction and optional interactive steps (akin to Accurig). This enables the adaptation and correction of the rigging process during runtime, for instance, to rectify joint positions.
Given the substantial number of features that have yet to be implemented, this approach offers several advantages, including the capacity to test the correctness of implementations interactively, expedite the identification of bugs, and identify bottlenecks, thereby ensuring the system's real-time functionality.
Furthermore, the requirement of interactivity ensures that the proposed concepts are kept as modular as possible. This, in turn, facilitates the reuse and combination of components during the development process.
The primary methodology involves the presentation of tool parameters within the user interface whenever these parameters are deemed to be modifiable.

### 3.1.2 Import and Export

The workflow of the Motion Retarget Editor is as follows: first, the character is exported to Blender; then, rigging and retargeting of motion occurs in the proposed editor; and finally, the exported file is imported into Blender.
CrossForge employs Assimp as an interface for Model IO, given its compatibility with a variety of formats, ranging from simple ones like obj, ply, stl, and bvh to more complex ones such as fbx and glTF.
To accommodate these disparate file types, Assimp employs a unified intermediate format, which is then used to expose an extensive scene description. This description is employed to transfer data to and from CrossForge's `T3DMesh`.

Assimp's intermediate format prioritizes ease of use, aligning with the expectations of graphics APIs. For instance, it ensures that texture seams are managed by dividing the mesh at specific points.
However, given the potential incompatibility across different formats, Assimp cannot guarantee the preservation of mesh structure during import, as certain formats might not support the same set of surface properties.
Consequently, the import/export process in this round trip does not preserve mesh features that are potentially critical to the integrity of the mesh, rendering it unsuitable for editing meshes.

CrossForge offers a native glTF interface prototype that supports a wide range of features deemed essential for this work.
The objective of this native binding, as opposed to utilizing libraries, is to enhance round-trip import and export functionality. However, the prototype remains incomplete, and various issues have been identified, some of which have been addressed during the present study.
The present study has focused on a particular file format, deeming it beneficial, as external tools like Blender can be utilized for further exporting and thereby circumventing the handling of edge cases.

The capacity to import and export plays a pivotal role in effective testing methodologies. It enables the rapid testing of edge cases, obviating the need for their programmatic creation.
Furthermore, reviewed methods often provide less scalable implementations, if they are even available, which limits their use to test environments and not directly usable tools for animators and scientists.

## 3.2 Classes for Scene Management

All classes relevant to the prototype are managed in a dedicated subdirectory, facilitating the inheritance of CrossForge modules and the replacement of only necessary components. This approach ensures the maintenance and integrity of the classes, while concurrently reducing the probability of errors by circumventing the duplication of code.
The subsequent section will address the Character Entity, which constitutes the fundamental component of the Editor.

### 3.2.1 Character Entity

The Character Entity Class ( `CharEntity` ) is a representation of a single virtual character instance. While various types of meshes unrelated to virtual characters can be loaded without issue, the primary function of `CharEntity` is to manage components linked to its virtual character and its various states. Listing 3.2 provides an excerpt of the most important components of the class.
As a central data unit for the Editor, various modules can utilize either a Character Entity or its components, depending on the task at hand. For example, when rigging is initiated, the character entity undergoes a transformation from a static actor to a skeletal actor.

LISTING 3.2: CharEntity is a versatile structure for managing dynamic and static characters.

```
struct CharEntity {
    bool isStatic = false;
    std::unique_ptr<IKSkeletalActor> actor;
    std::unique_ptr<StaticActor> actorStatic;

    // animation
    std::unique_ptr<IKController> controller;
    Animation* pAnimCurr;

    // common
    std::string name;
    T3DMesh<float> mesh;
    SGNGeometry sgn;

    // Picking binding functions
    ...

    // various tool functions
    ...
};
```

The creation of a 3D model through the utilization of an importer process results in the generation of a corresponding `CharEntity` and the incorporation of the geometry node into CrossForge's managed scenegraph.
In order to differentiate between Characters that have been rigged and Characters without a Skeleton, unique pointers are used. These pointers can be checked easily for initialization.
During the character entity's initialization process, the `T3DMesh` is loaded by either the `IKSkeletalActor` or the `StaticActor`, depending on its contents, in order to be visualized using OpenGL.
However, during runtime, it is necessary to apply animation changes and mesh operations to the `T3DMesh`. This necessitates maintaining its reference to reinitialize the renderable actor with the updated `T3DMesh`. The `T3DMesh` is then checked to determine which features it possesses, and the corresponding actor is selected accordingly.
The identification of a CharEntity is derived from the name of the imported asset file. In the event that an asset already possesses a name, a number is appended to differentiate it.
In anticipation of future applications, the polymorphism of CrossForges's `RenderableActor` class, from which `StaticActor` and `IKSkeletalActor` inherit, can be leveraged to expand the repertoire of available Actor types. This approach could involve defining an enum to facilitate the selection of the appropriate actor type.

### 3.2.2 Main Scene

Given that motion retargeting and various other operations necessitate the delineation of the characters to be utilized, it is crucial that their states are defined in a coherent manner.

LISTING 3.3: MotionRetargetScene serves as the core of the editor managing scene loop, UI, character I/O, and entity references.

```
class MotionRetargetScene : public ExampleSceneBase {
    ...
    void mainLoop() override;
    ...
    void renderUI();

    void loadCharPrim(std::string path, IOmeth ioM);
    void storeCharPrim(std::string path, IOmeth ioM);

    struct settings {
        ...
    } m_settings;

    std::vector<std::shared_ptr<CharEntity>> m_charEntities;
    std::weak_ptr<CharEntity> m_charEntityPrim; // currently
        ↪ selected char entity
    std::weak_ptr<CharEntity> m_charEntitySec; // secondary char
        ↪ entity for operations

    SGNTransformation m_sgnRoot;

    ...
};//MotionRetargetScene
```

Ideally, a record is maintained of the most recently selected two objects, with the primary Character Entity designated as the most recently selected object.
Conversely, when a new character entity is selected that differs from the current entity, the previously selected character entity is assigned to the secondary reference. Furthermore, operations necessitating the presence of both entities can verify their availability in advance and generate error messages if either entity is not detected.

Listing 3.3 offers a overview of the primary functions and member variables of the `MotionRetargetScene` class.
To facilitate the dynamic loading, unloading, or storage of various character entities without the necessity of restarting the application, the `loadCharPrim()` and `storeCharPrim()` functions have been developed. These functions facilitate the loading of character models from common file formats via the user interface, while also enabling the selection of the desired export or import method.

### 3.2.3 Config

Crossforge is currently lacking an abstraction for a configuration system to store simple preferences in files for persistent usage across sessions.
While the implementation of a straightforward ini configuration file loader does not require a significant investment of time, the introduction of an abstraction would enhance the ease of use.

As illustrated in Listing 3.4, the Config class comprises a variety of simple data types, which are stored under a string identifier within a JSON file. The implementation of custom types can be facilitated through the use of function overloading.
To maintain the integrity of the core logic of the classes, an additional compile unit is used to define all interfaces. This ensures the segregation of Config functionality from the corresponding modules.

LISTING 3.4: The Config header contains bindings for exportable classes. Overloaded store and load functions can be relocated to separate compile units by deriving the Config class.

```
// Config.hpp
class Config {
    void store(const VirtualCamera& object);
    void load(VirtualCamera* object);
    void store(const GLWindow& object);
    void load(GLWindow* object);

    // template functions to load and store simple types
    template <typename T>
    void store(std::string name, const T& value);

    template <typename T>
    void load(std::string name, T* value);

    std::string ConfigFilepath = "CFconfig.json";
    nlohmann::json m_ConfigData;
};
```

The `m_settings` struct of the `MotionRetargetScene` (Listing 3.3) is responsible for storing the core options of the Editor, as well as behavioral and file paths to tools. While abstractions exist for restoring camera position and orientation, as well as window position, from a previous testing session.

## 3.3 User Interface

Despite the existence of numerous user interface libraries, ImGui [50] stands out as the most mature and extensible option, providing a vast array of flexible widgets that facilitate seamless integration and adoption of third-party extensible plugins.
ImGui's immediate mode design principle, which involves the redrawing of the interface in each frame without the utilization of intermediate variables or the separation of states, results in more compact and manageable code.

In Figure 3.2, the Editor and its user interface are depicted. The Editor is composed of various predefined ImGui widgets, including the Gizmo, Outliner, Animation, and IK tabs, which are located on the right side of the interface. Additionally, the Editor features a sequencer at the bottom of the interface and a menu bar. Currently, two pop-ups, Rigging and Preferences, are open. The ImGuis docking branch facilitates a more streamlined and adaptable placement of user interface elements, allowing users to customize the placement of these components according to their preferences.
Subsequent sections will provide a detailed exposition of the specific user interface interactions that are associated with the proposed methods and functionalities.

### 3.3.1 Widgets

The Outliner employs a visual representation of skeletal hierarchies, utilizing collapsible tree nodes that contain joint names, targets, and visualization options specific to an actor.
Upon selecting an item from the list, the corresponding object is retrieved. When the object is pickable, the picker state is forced on that object, enabling feedback with highlight visualization.

The Animation tab is employed for the selection of imported animations for automatic playback or keyframe selection using the Sequencer. Additionally, information regarding animation sampling is displayed, and playback speed can be configured.

Due to the presence of artifacts in the integrated ImGui popup, resulting from event-driven rendering, an alternative implementation of the logic is employed, utilizing standard ImGui windows. This approach is primarily employed for the structuring of options pertaining to tools or preferences. For instance, it can be utilized to define hints or place markers for autorigging methods that necessitate such markers.

A screen space-based grid has been implemented to facilitate orientation in three-dimensional space. The primary axes are intuitively marked with corresponding colors (red x, green y, blue z) to ensure user clarity during operations.

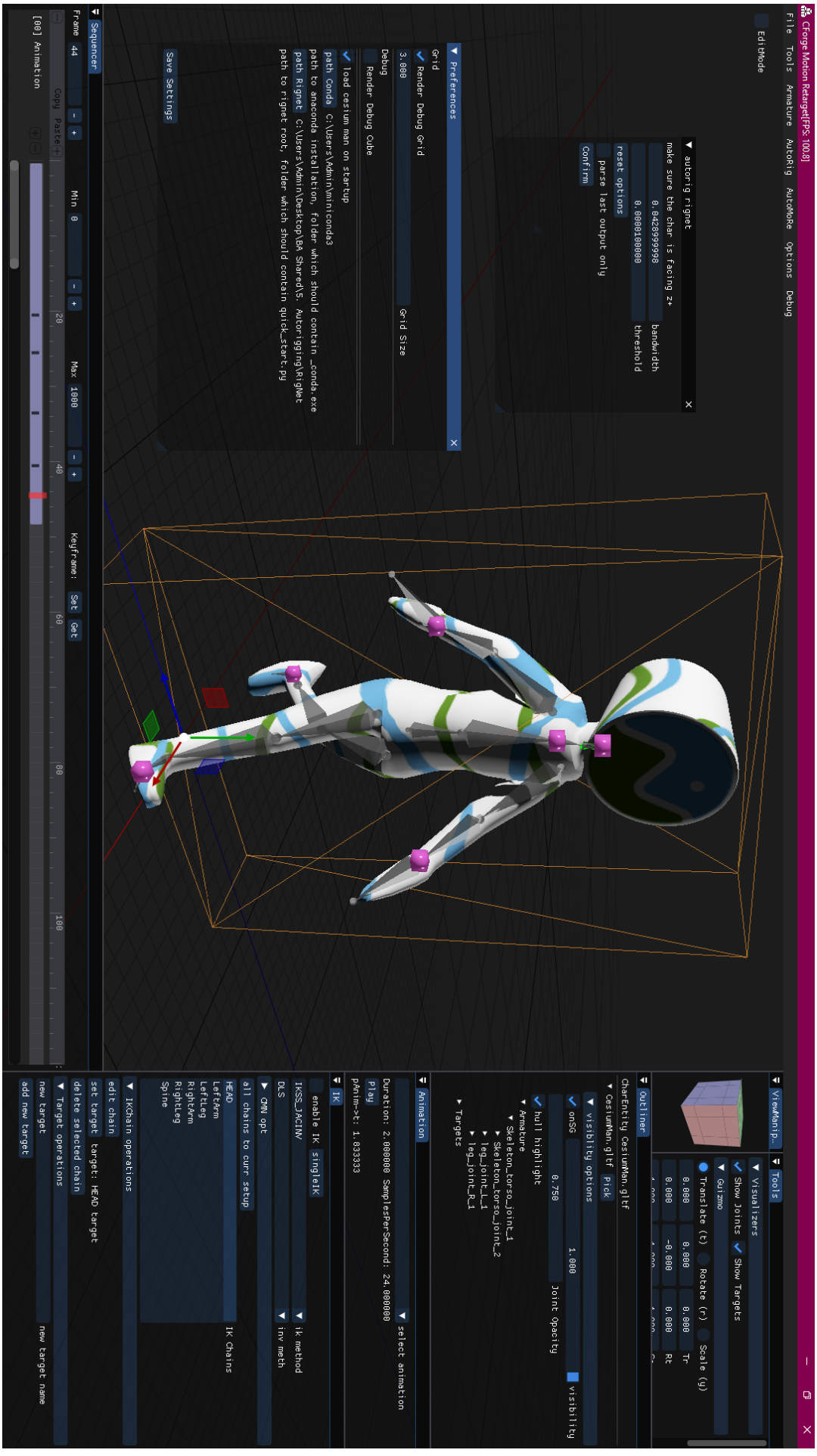


FIGURE 3.2: The editor's user interface.

Subsequent sections will provide detailed explanations of additional user interfaces. These components include the sequencer, Gizmo, and other elements.

### 3.3.2 Picking

In order to facilitate interaction with objects in a three-dimensional scene, the implementation of a picking system is essential.
The implementation of this system involves ray shooting, whereby the position of the mouse click is transformed from screen space back to world space. Subsequently, the system checks for intersections between the mouse click position and various objects in the 3D scene.
CrossForge has already implemented the bounding volume type, which includes AABB and Sphere. The program performs a preliminary check for intersections, followed by an optional check for intersection with mesh data.
To efficiently check for ray-to-mesh intersection, libigl [51] is used for critical components such as joints, necessitating conversion from `T3DMesh` to a list of vectors in a matrix.

The `Picker` (Listing 3.5) class plays a pivotal role in the evaluation of a set of `IPickable` (Listing 3.6) objects and the storage of the most recently and previously selected objects as weak pointer references. The class's versatility is evident in its capacity to manage various types of objects, including character entities, joints, and targets, thereby ensuring its adaptability to a range of tasks.

LISTING 3.5: Most relevant methods of the Picker class that the application uses to evaluate pickable objects and store references to any picked object type.

```
class Picker {
    void pick(std::vector<std::weak_ptr<IPickable>> objects);
    void forcePick(std::weak_ptr<IPickable> pick);
    void start();
    void resolve();
    void reset();

    void update(Matrix4f trans);

    std::weak_ptr<IPickable> getLastPick() {
        return m_pLastPick;
    };
    std::weak_ptr<IPickable> getCurrPick() {
        return m_pCurrPick;
    };
    Matrix4f m_guizmoMat = Matrix4f::Identity();

    void rayCast(Vector3f* ro, Vector3f* rd);
    std::weak_ptr<IPickable> m_pLastPick; // picked object
    std::weak_ptr<IPickable> m_pCurrPick; // last clicked object
    std::weak_ptr<IPickable> m_pPick; // last clicked object
};
```

In order to ensure the isolation of the intersection test from the transformations applied by the Gizmo, which is necessary for certain operations, the transformation matrices for Gizmo and Picking are separated.

The utilization of the `forcePick` function enables the application to set object selection programmatically, a functionality employed by the Outliner to select Character Entities and their joints by name.

LISTING 3.6: Any class can derive from the Pickable Interface. Facilitating the addition of new types of pickable objects.

```
class IPickable {
    public:
    virtual void pckSelect() {};
    virtual void pckDeselect() {};
    virtual void pckMove(const Matrix4f& trans) = 0;

    virtual Matrix4f pckTransGuizmo() = 0; // used for guizmo
        ↪ update
    virtual Matrix4f pckTransPickin() = 0; // used for picking
        ↪ evaluation
    virtual const BoundingVolume& pckBV() = 0;
    virtual EigenMesh* pckEigenMesh() { return nullptr; };
};
```

The determination of the selected type can be facilitated by employing the standard library function, `std::dynamic_pointer_cast`, which enables the casting of a smart pointer. In the event that the cast is invalid, the resultant object is a null pointer, thereby simplifying the identification of the type of the selected object.

### 3.3.3 Scene Control

Following the selection of a character, a wireframe box is employed to facilitate the identification of characters that are currently assigned the primary or secondary role. This method utilizes varying highlight strengths to distinguish between the selected characters, thereby assisting users in determining which characters are currently selected with ease.

In order to apply transformations to selected objects, a gizmo is required.
The term "gizmo" is generally employed to denote a miniature device or gadget that has been engineered for a particular function, often signifying a tool that can execute a specific task in an innovative or efficient manner. This term is colloquial and can be applied to a diverse array of devices.
In the domain of graphics programming, gizmos serve to facilitate the manipulation of objects within three-dimensional space. These tools are extensively employed in graphics editors to visually represent and control object transformations, with the most prevalent being position, rotation, and scale. However, their functionality extends beyond these parameters, encompassing various other types, such as camera manipulation and mesh editing. These tools provide intuitive controls that enhance user interaction with the three-dimensional space.

ImGuizmo [52] is an easy to integrate Gizmo Plugin for ImGui. In addition to providing a Gizmo, it offers an animation Sequencer and a Camera Manipulation Cube. Notably, ImGuizmo functions by utilizing a reference to an array of floating-point numbers, thereby ensuring independence from linear algebra libraries.

In order to inspect the model from various angles for the purpose of comparison or to test and correct the animations, the camera has been extended to quickly snap to

world space planes and toggle between orthographic projections using designated numpad keys, similar to Blender's control scheme.
Figure 3.3 presents the tools previously discussed for controlling the scene.Given the selection of Cesium Man, the Gizmo manifests at the position of the designated Scene Graph Node, capable of translating, rotating, and scaling the actor within the Scene.
While the Gizmo is active, the user can manually set each individual component of the transformation using the Gizmo tab numerically, allowing for precise control over the transformation.
The evaluation of motion quality is facilitated by orthographic projection and predefined camera alignment on the world space coordinate system planes.

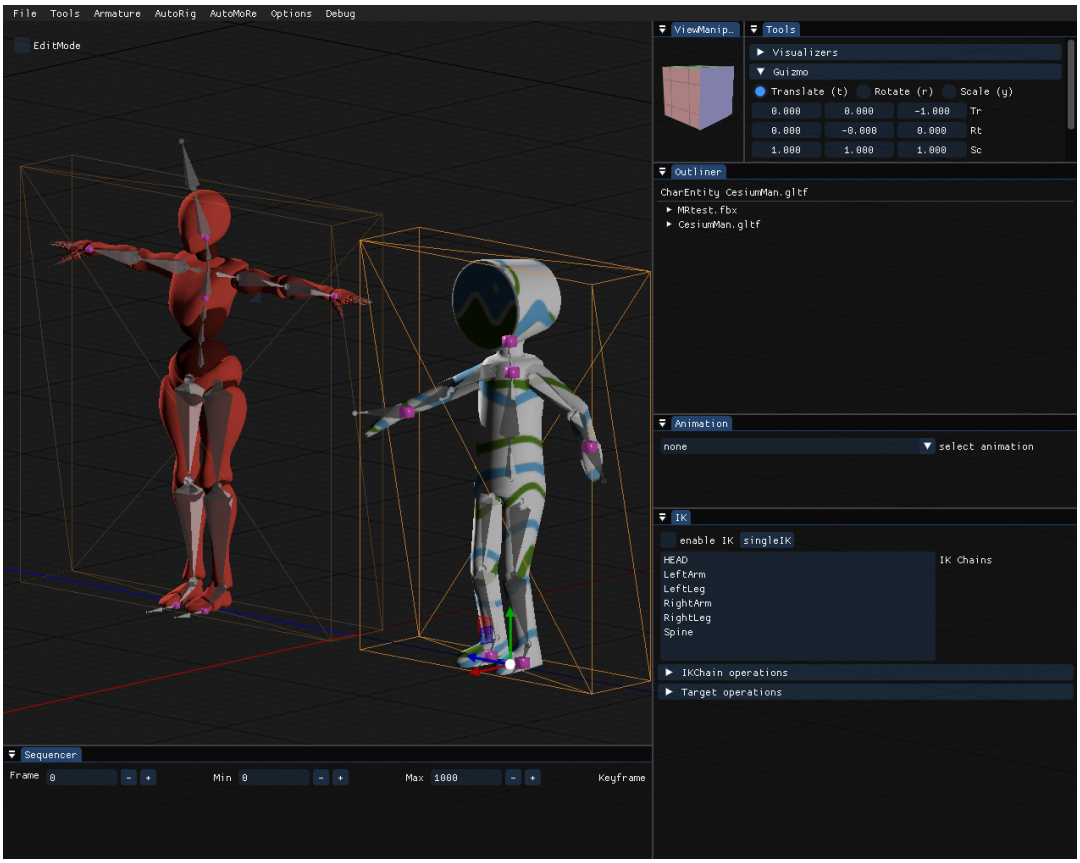


FIGURE 3.3: Cesium Man and the Mixamo test rig are shown in orthographic camera mode, with Cesium Man designated as the primary character, as indicated by the Wireframe Bounding Box highlight, and the Mixamo test rig assigned as the secondary character.

## 3.4 Animation System

CrossForge offers an implementation for skeletal animation playback using linear-blend-skinning. However, until recently, there was no means of interacting with the animation, for example, by selecting specific keyframes or by modifying joints of a keyframe.

### 3.4.1 CrossForge format

CrossForge employs a direct approach for the animation playback feature. As previously discussed in Section 3.1.2, Assimp provides the inverse bind pose matrix of each joint, in addition to the animation data. Rather than employing further abstractions, such as designated nodes representing joints, which could potentially increase the complexity of the system, the inverse bind pose matrix can be utilized because it provides sufficient information for all subsequently explained processes.
Following the importation of an animated model into CrossForge's intermediate format `T3DMesh`, the Skeletal Animation Controller is populated with the skeletal hierarchy and animations.
Listing 3.7 displays the member variables of CrossForge's `SkeletalJoint` struct, which is contained within the Skeletal Animation Controller. `SkeletalJoint` contains the necessary variables discussed in Section 2.1, while its Local component is decomposed into position, rotation, and translation. This decomposition is a common occurrence observable in various Animation tools and Intermediate formats.

LISTING 3.7: CrossForge's SkeletalJoint struct. Utilizing integer IDs for accessing relative child and parent joints through the Controller.

```
struct SkeletalJoint : public CForgeObject {
    int32_t ID;
    std::string Name;
    Eigen::Matrix4f OffsetMatrix;
    Eigen::Vector3f LocalPosition;
    Eigen::Quaternionf LocalRotation;
    Eigen::Vector3f LocalScale;
    Eigen::Matrix4f SkinningMatrix;

    int32_t Parent;
    std::vector<int32_t> Children;
};
```

The proposed `IKController` utilizes these structs to modify selected joints via Gizmo, Inverse Kinematics, or the Motion Retargeting routine. Furthermore, a `std::map<SkeletalJoint*,IKJoint> m_IKJoints` extends the `SkeletalJoint` with cached global position and rotation of joints without compromising the functionality of the standard Animation Controller.
In addition, a struct is utilized for the purpose of bookkeeping individual animations and their playback options. These animations can then be queued to be executed by CrossForge's SceneGraph system.

### 3.4.2 Sequencer

In order to gain more fine-grained control over animation playback with visual feedback, it is possible to incorporate a sequencer, which allows for the swift selection of individual keyframes for previewing.
Generally, a sequencer is a powerful tool utilized in game engines and animation software for the creation and editing of motion sequences.It functions as a multitrack editor, enabling users to combine multiple animation clips, control camera movements, and synchronize various elements to create cohesive animated sequences. In advanced animation software, they facilitate the seamless blending or crossfading between disparate animations, thereby enabling smooth transitions or precise control over object transformations.

ImGuizmo [52] incorporates an adaptable sequencer that was integrated into the editor to facilitate the selection of keyframes. Figure 3.2 illustrates the sequencer widget, positioned in the bottom left of the user interface.
While sophisticated features such as multi-track are not necessary for the purposes at hand, a rudimentary timeline facilitates the immediate selection of keyframes, a feature that can be expanded upon in the future.
This component is an crucial instrument for evaluating the quality of an animation, as it enables navigation to and repetition of problematic segments for in-depth examination.

### 3.4.3 Joint Interaction

In order to visualize and interact with joints, `JointPickable` is derived from `IPickable` and accompanies a `SkeletalJoint`, handling its joint picking interaction and visualization.
Given that imguizmo utilizes a single affine matrix for both interaction and rendering, yet our objective is to implement transformations of the joint relative to the actor in world space, it is necessary to maintain and extract the world space transformation of the characters scene graph node.
An excerpt of the JointPickable class is provided in listing 3.8.

LISTING 3.8: The JointPickable class implements interactive joint manipulation and visualization for the purpose of editing skeletal animation.

```
class JointPickable : public IPickable {
    void init();
    void update(Matrix4f sgnT);
    void render(RenderDevice* pRD);

    // IPickable bindings, setting private Matrix transforms and
       ↪ handling highlighting

    JointPickableMesh* m_pJPMesh;
    Vector4f colorSelect;
    SkeletalAnimationController::SkeletalJoint* m_pJoint;
private:
    Matrix4f m_transform; // joint transform for visualization
    Matrix4f m_transformGuizmo; // joint transform for manipulation
    Matrix4f m_fromPar; // parent joint transform
    Matrix4f m_sgnT; // parent sgn transform of SkeletalActor
    IKController* m_pIKC;

    StaticActor actor; // visualizer actor used for rendering
};
```

In order to apply the gizmos transformation relative to the local frame of the joint, it is necessary to separate and track both transformations independently.
The global transformation of a joint can be separated into the following components:

$$M_{glob} = M_{sgn} \cdot M_{parent} \cdot M_{local}$$

In this context, $M_{local}$ denotes the joints local parameters, $M_{sgn}$ is the Scene Graph Nodes transformation, and $M_{parent}$ represents the chain of parent transformations up to the root bone.

$M_{parent}$ is then either evaluated by traversing the skeletal hierarchy or more quickly using the Skinning Matrix:

$$\begin{aligned} M_{parent} &= I && \mid \text{if joint i has no parent} \\ M_{parent} &= SM_{parent} \cdot OM^{-1}_{parent} && \mid \text{if joint i has a parent} \end{aligned} \tag{3.1}$$

where $SM$ and $OM$ denote corresponding Skinning and Offset Matrices.
In order to decompose the global matrix employed by Imguizmo to extract the local transformation change, it is necessary to apply $M_{parent}$ accordingly:

$$M_{local} = M^{-1}_{parent} \cdot M^{-1}_{sgn} \cdot M_{global}$$

In order to facilitate visualization, the utilization of $M_{glob}$ would yield joints that are oriented towards their base coordinate systems direction, either in one of the three base vectors, depending on the direction in which the picking mesh points.
However, this does not align with the intuitive visualization of a joint pointing to its next child in the hierarchy, causing joints to point in seemingly random directions (see Figure 3.4).

FIGURE 3.4: Visualizing Joints of Cesium Man using their current global transformation. In the left image, the joints have not been rotated according to their child's position, resulting in seemingly random directions. In contrast, the right image demonstrates the expected result of proper rotation.

In order to correctly align the bones of a given joint with their respective child joint, it is necessary to rotate the Joint Visualizer according to the position of the child in question. This approach has been demonstrated to yield accurate results:

$$M_{globVis} = M_{sgn} \cdot M_{parent} \cdot M_{local} \cdot R_{childDir} \cdot S_{childLength}$$

Where $R_{childDir}$ is a rotation matrix that aligns the joint visualizer with the child joint's position, and $S_{childLength}$ is a scale matrix that scales the joint appropriately to reach the targeted child joint.
The selection of child joints is not without its ambiguities, particularly in instances where multiple bones correspond to a given parent joint.

Once a joint has been selected, its appearance is accentuated. This is achieved by employing an outline method that utilizes a larger joint with inverted normals. This technique prompts backface culling to generate an aura around the joint mesh.

### 3.4.4 Editing Tools

In order to address import ambiguities and align models, a series of operations must be performed. Specifically, it is necessary to apply an affine transformation to a Scene Graph Node on the Mesh data itself in order to avoid having to specify scene graph relative transformations in other operations.
Furthermore, effective visualization and adaptation of the rest pose necessitate a proper evaluation, as it constitutes a useful element of the motion retargeting process during the alignment procedure.
However, given the Framework dependency of these adaptations, explanations for these simpler operations on skinned meshes remain limited.
In order to construct the rest pose skinning matrices for each joint, it is necessary that each matrix evaluates to the identity. However, this is not a trivial construct.
Listing 3.9 shows the process of reconstructing local joint parameters in order to model the rest pose. The function is invoked from the root joint and the identity matrix for $OM_{parent}$.

**Algorithm 3.9** Compute Restpose. Recursively determining restpose parameters for each joint.

```
1: procedure COMPUTERESTPOSE
2:     Input: joint                                  ▷ current joint
3:     Input: OM_parent                              ▷ offset matrix of parent joint
4:     M_local := OM_parent^-1 · joint.OM^-1         ▷ iteration count
5:     joint.M_local := M_local
6:     for child in joint.children do
7:         ComputeRestpose(child,joint.OM^-1)
8:     end for
9: end procedure
```

In order to update the rest pose, it is necessary to modify the vertex data of the mesh. This can be achieved through the use of CPU vertex skinning.
The process is delineated in listing 3.10. Initially, all vertices undergo transformation via vertex skinning, indicated by the function `transformVertex()`. Subsequently, the inverse bind pose matrices of each bone are updated by computing the global rotation and position of the bone, combining it into the bind pose matrix, and then inverting it.
Although this is a straightforward approach, the same deformation issues that occur with linear blend skinning in the original mesh also occur in the updated mesh. As a result, these issues are visible in any other pose.
One should also bear in mind that existing animation data will have the difference of between new and old restpose applied as well.

LISTING 3.10: The updateRestpose function updates mesh vertices and bone matrices to reflect the current pose.

```
void CharEntity::updateRestpose(SGNTransformation* sgnRoot) {
    if(!actor)
        return;

    // apply current pose to mesh data
    for (uint32_t i=0;i<mesh.vertexCount();++i) {
        mesh.vertex(i) = actor->transformVertex(i);
    }

    // forwardKinematics to get updated global pos and rot in
       ↪ m_IKJoints
    controller->forwardKinematics(controller->getRoot());

    for (uint32_t i=0;i<mesh.boneCount();++i) {
        auto* b = mesh.getBone(i);

        IKJoint ikj = controller->m_IKJoints[controller->getBone(i)
           ↪ ];

        // get current global position and rotation
        Vector3f pos = ikj.posGlobal;
        Quaternionf rot = ikj.rotGlobal;

        Matrix4f bindPose = Matrix4f::Identity();
        bindPose.block<3,1>(0,3) = pos;
        bindPose.block<3,3>(0,0) = rot.toRotationMatrix();
        b->InvBindPoseMatrix = bindPose.inverse();
    }

    init(sgnRoot);
}
```

The ability to transform a mesh and its animation for alignment is of significant importance, as numerous tools utilize a direction other than the front-facing one. Algorithm 3.11 details the process of updating a rigged character's transformation in object space.

In this case, merely updating the root bone of the animation data is sufficient. However, it is imperative to update the positions of the animation data that usually remain constant, as they reflect the joints' position relative to their parent. These constant positions are thus contingent on the offset matrix.

**Algorithm 3.11** Apply transform to Mesh modifies the skeleton's rest pose and adjusts animations through rotation, scaling, and position changes.

```
1: procedure TRANSFORMMESH
2:     Input: T                                          ▷ transform matrix
3:     Input: mesh                                       ▷ input mesh
4:     Input: skel                                       ▷ input skeleton
5:     for vertex v in mesh do
6:         v := T · v
7:     end for
8:     deconstruct T into rotation R, position P and scale S
                                                         ▷ update restpose
9:     for bone b in skel do
10:        b.OM := b.OM · R
11:        scale translational component b.OM according S
12:    end for
                                                         ▷ update animations
13:    for animation a in skel do
14:        only rotate root bone
15:        get root bone b_r
16:        for rotation r in a[b_r] do
17:            r := R · r
18:        end for
                                                         ▷ scale all translation offsets with S
19:        for position p in a do
20:            p := S · p
21:        end for
22:        for position p in a[b_r] do
23:            p := P + R · p
24:        end for
25:    end for
26:    reinitialize actor
27: end procedure
```

## 3.5 Inverse Kinematics Implementation

Proper and fast IK implementations are essential for the framework, as IK plays an important role in many motion retargeting approaches, especially kinematic chain-based methods and the domain of real-time solutions.
While various Inverse Kinematics Implementations exist, they are usually implemented across various programming languages or are limited to certain engines. This results in vastly different and complex APIs.
The lack of established inverse kinematics libraries for C++ is particularly problematic, as performance is a critical aspect.
Therefore, the various inverse kinematics algorithms proposed in section 2.2 are reimplemented using CrossForge's Animation Controller interface. This reduces the complications of adopting existing solutions and results in small, concise implementations that are easily adaptable and extensible for future purposes.

Aristidou et al. [6] noted that while certain IK methodologies may exhibit substantial variances in performance, the foundational approaches examined in Section 2.2

remain relevant due to their unique behavioral characteristics.

The IK Solver Interface `IIKSolver`, Listing 3.12, establishes termination parameters. Subsequent realizations must provide implementations that adhere to these parameters.

LISTING 3.12: The IIKSolver interface ensures a uniform interface for inverse kinematic solvers in order to employ them quickly.

```
class IIKSolver {
    public:
    virtual void solve(std::string segmentName, IKController*
        ↪ pController) {};

    protected:
    float m_thresholdDist = 1e-6f;
    float m_thresholdPosChange = 1e-6f;

    int32_t m_MaxIterations = 50;
};//IIKSolver
```

In order to realize multiple inverse kinematic solutions on a single armature, it is necessary to designate the solvers to the kinematic chains that they are required to solve.
Listing 3.13 presents the `IKChain` struct, which conceptualizes a subchain of a skeleton devoid of branches and an associated target.

LISTING 3.13: IKChain struct represents a kinematic chain that can be solved using Inverse Kinematics.

```
struct IKChain {
    std::string name;
    std::vector<SkeletalJoint*> joints; // front() is end-effector
        ↪ joint
    std::weak_ptr<IKTarget> target;

    std::unique_ptr<IIKSolver> ikSolver;
    float weight = 1.; // weight used for centroid interpolation,
};
```

As previously discussed, `IKController` is derived from `SkeletalAnimationController`, which has already been defined in CrossForge, only replacing functions that require modifications. Listing 3.14 illustrates the most significant aspects of the `IKController`.
It is worth noting that the management of `IKChains` is carried out by the `IKArmature` class, which is overseen by the controller. Joint parameters of the base class can be computed using standard forward kinematics, as well as inverse kinematics. These parameters are accessible in the base class controller, which utilizes pointers for storing joints.
`m_IKJoints` represents an extension of the `SkeletalJoint` that is used to assign each joint an additional global component, thereby avoiding the need for recompute each time it is required or deems useful.
In the course of the armature's solution process, each `IKSolver` retrieves its designated `IKChain` and `IKTarget`.

LISTING 3.14: Most important member variables and functions of the IKController. Member are either exposed as public or accessable with functions.

```
class IKController : public SkeletalAnimationController {
    void init(T3DMesh<float>* pMesh);
    void initRestpose();
    ... update(float FPSScale) ... clear() ...

    void applyAnimation(Animation* pAnim, bool UpdateUBO = true);
    void forwardKinematics(SkeletalJoint* pJoint); // Computes
        ↪ global position and rotation of joints of the skeletal
        ↪ hierarchy and stores them into m_IKJoints.

    void buildKinematicChain(std::string name, std::string rootName
        ↪ , std::string endEffectorName);
    IKChain* getIKChain(std::string name); // retrieve IKChain from
        ↪  Armature

    void initTargetPoints();
    void clearTargetPoints();

    std::map<SkeletalJoint*,IKJoint> m_IKJoints; // extends
        ↪ m_Joints
    IKArmature m_ikArmature; // Armature containts IKChains
    std::vector<std::shared_ptr<IKTarget>> m_targets;

    std::map<SkeletalJoint*,std::shared_ptr<JointPickable>>
        ↪ m_jointPickables;
    JointPickableMesh m_jointPickableMesh;
};//IKController
```

The chain's `name` is used to identify and modify it in an interactive manner by means of a pop-up window and to adjust the parameters of the IK Solver. The IK tab of the User Interface is visible in the lower right corner of Figure 3.5 showing the interface for creating a kinematic chain and manipulation, which can be done interactively during runtime.

### 3.5.1 Jacobian Method

Relevant reviewed literature demonstrates a lack of detail regarding the specific cell entries of the Jacobian matrix. This ambiguity arises due to the variability in interpretation of the input vectors that are multiplied with the Jacobian inverse. Consequently, the meaning of the population remains open to interpretation.
It is also unclear from Section 2.2 what vector $T$, that gets multiplied with the Jacobian Inverse, exactly is, although it can be assumed to mean the end effector target position, it is important to note that $T$ can also contain multiple end effectors targets, or even utilize joints that are not end effectors.

As illustrated in Figure 3.6, the Jacobian matrix is populated as follows: each row represents the influence of all joints and their angles on the end effector change in corresponding dimensions.
The transposed variant of the matrix, with rows and columns swapped, indicates that it should be multiplied with a desired direction vector, containing in what way the target should move, on the right side of the matrix. For one target, this vector is 3x1; for two targets, the vector is 6x1; and so on.

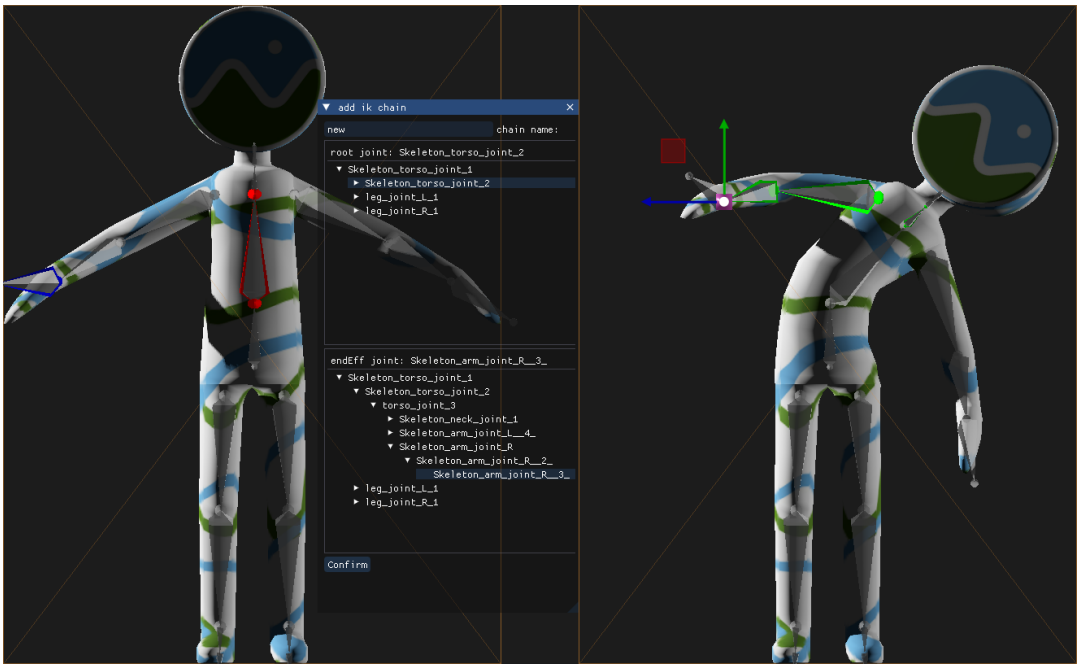


FIGURE 3.5: The IK Chain is generated through the utilization of a pop-up that incorporates a tree-like structure for the selection of specific joints. The Joint Visualizer plays a crucial role in identifying the currently selected joint. The color red signifies the root joint of the chain, while blue denotes the end effector. When a chain is selected, the corresponding joints are highlighted in green.

The Jacboain matrix then returns a vector of input joint angles $3jx1$ for each dimension in which a joint can rotate. Finally, joint angles are read out in the same spacing vertically as they were fed into the matrix horizontally.

$$
\begin{array}{c}
 \\
\text{eef1 x} \\
\text{eef1 y} \\
\text{eef1 z} \\
\text{eef2 x} \\
\vdots
\end{array}
\begin{array}{c}
\overset{\text{joint1}}{\phantom{x}} \\
\left( \begin{array}{cccc}
\frac{\partial \text{eef1 x}}{\partial \text{joint1 x}} & \frac{\partial \text{eef1 x}}{\partial \text{joint1 y}} & \frac{\partial \text{eef1 x}}{\partial \text{joint1 z}} & \frac{\partial \text{eef1 x}}{\partial \text{joint2 x}} \\
\frac{\partial \text{eef1 y}}{\partial \text{joint1 x}} & \frac{\partial \text{eef1 y}}{\partial \text{joint1 y}} & \frac{\partial \text{eef1 y}}{\partial \text{joint1 z}} & \frac{\partial \text{eef1 y}}{\partial \text{joint2 x}} \\
\frac{\partial \text{eef1 z}}{\partial \text{joint1 x}} & \frac{\partial \text{eef1 z}}{\partial \text{joint1 y}} & \frac{\partial \text{eef1 z}}{\partial \text{joint1 z}} & \frac{\partial \text{eef1 z}}{\partial \text{joint2 x}} \\
\frac{\partial \text{eef2 x}}{\partial \text{joint1 x}} & \frac{\partial \text{eef2 x}}{\partial \text{joint1 y}} & \frac{\partial \text{eef2 x}}{\partial \text{joint1 z}} & \frac{\partial \text{eef2 x}}{\partial \text{joint2 x}} \\
\vdots & \vdots & \vdots & \vdots
\end{array} \right.
\end{array}
\begin{array}{c}
\overset{\text{joint2}}{\phantom{x}} \\
\left. \begin{array}{c}
\cdots \\
\cdots \\
\cdots \\
\cdots \\
.
\end{array} \right)
\end{array}
$$

FIGURE 3.6: A population example of the Jacobian matrix for inverse kinematics. "Eff" designates end effector position change, and "joint" denotes joint angles. Each row is populated with every joint angle change and its influence on the end effector in the corresponding dimensions.

An optimization for evaluating rate changes on end effectors was elaborated by Buss [11].
It was archived with the observation that rotational change effects all sub joints in the same way. One may picture all sub joints of the rotating joint as one single rigid object, because with the forward kinematics approach, no pose parameters of sub joints are changed.

This adoption enables the computation of change through the cross-product of the rotational axis and the vector from the joint to the endeffector. This results in a tangent vector that represents the direction and rate of change with which the endeffector moves as the joint rotates. However, this optimization necessitates the adaptation of the rate of change $\Delta$.

An observable issue during the testing phase of the Jacobian implementation was the occurrence of joint rotation when the target position was unattainable.
This phenomenon occurs because the Jacobian matrix relates the joint velocities to the end-effector velocity, and when the target is out of reach, the Jacobian becomes singular or ill-conditioned. To address this issue, a solution was to determine the straight chain programmatically.

A variety of methodologies for approximating the Jacobian inverse have been utilized, including the transpose, singular value decomposition, and damped least squares. As previously discussed, damped least squares is identified as the most stable solution, further reducing the spinning issue.

The Jacobian Inverse Kinematics solver utilized in the Editor, as outlined in Listing 3.15, facilitates the integration of these types and the damped least squares damping factor. This configuration is accessible via the user interface.
At present, the implementation's scope is confined to resolving one joint chain at a time.

LISTING 3.15: IKSjacInv class implements inverse kinematics using Jacobian-based methods including Damped Least Squares.

```
class IKSjacInv : public IIKSolver {
    public:
    enum Type {
        TRANSPOSE,
        SVD,
        DLS,
    } m_type = DLS;
    float m_dlsDamping = 3.;

    void solve(std::string segmentName, IKController* pController);

    MatrixXd DampedLeastSquare(MatrixXd jac);
    MatrixXd calculateJacobianNumerical(std::string segmentName,
        ↪ IKController* pController);
};
```

### 3.5.2 Heuristic Methods

As illustrated in the class overview, CCD and FABRIK have been implemented as well.
CCD contains an interface to select the forward or backward variation, simillar to the jacobian interface.
To differentiate between solvers, the Interface class employs the use of the `std::dynamic_pointer_cast`, which enables the determination of the solver type and the subsequent rendering of the relevant interfaces.

LISTING 3.16: IKSccd class implements Cyclic Coordinate Descent inverse kinematics with forward and backward solving options.

```
class IKSccd : public IIKSolver {
    public:
    enum Type {
        BACKWARD,
        FORWARD,
    } m_type = BACKWARD;
    void solve(std::string segmentName, IKController* pController);
    private:
};
```

Listing 3.17 displays the FABRIK solver header, while `fbrkPoints` contains the computed object space points utilized by FABRIK during evaluation. These points are accessible to the framework for visualization purposes, facilitating verification of FABRIK's correct implementation.

LISTING 3.17: IKSfabrik class implements the FABRIK algorithm for inverse kinematics, where backward kinematics restores local joint orientation from computed FABRIK points.

```
class IKSfabrik : public IIKSolver {
    void solve(std::string segmentName, IKController* pController);

    // equivalent to IKController::forwardKinematics, compute local
        ↪ pos, rot from global
    void backwardKinematics(std::string segmentName, IKController*
        ↪ pController,const std::vector<Vector3f>& fbrkPoints);

    std::vector<Vector3f> fbrkPoints; // global position for fabrik
        ↪ calculation
};
```

## 3.6 Armature creation and matching

A multitude of presented motion retargeting approaches employ an abstraction of bones into chains, either directly or indirectly.
Abdul-Massih et al. [33] proposed the utilization of Groups of Body Parts (GBP), while Du Sel et al. [32] identified limbs to facilitate retargeting using inverse kinematics and Aberman et al. [35] describe Skeletal Pooling, collapsing degree two joints, which often results in a graph similar to limbs.

In a similar manner, the previously utilized class `IKChain`, which was employed for the purpose of validating inverse kinematics, can be reused. Within this context, the class can be interpreted as a body part.
Accordingly, `IKArmature`, Listing 3.18, comprises a series of these non-branching joint chains without intersections.

LISTING 3.18: IKArmature class manages a collection of joint chains and provides functionality to solve inverse kinematics for the associated IKController.

```
class IKArmature {
    void solve(IKController* pController);
    std::vector<IKChain> m_jointChains;
};
```

Solving an armature solves each individual chain contained in `m_jointChains` with its designated `IKSolver`. Either in a specific order or utilizing proposed priority weighting from the previous section.

The ball socket constraint is a viable exemplar for subsequent implementation within the framework.
This constraint can be straightforwardly defined through the utilization of a basic ImGui float slider to modulate cone opening, and its visual representation can be achieved by employing a cone and a sphere generated using the CrossForge's Primitive Factory.
A forward pass can then be employed, akin to rendering targets and joints, to utilize opacity or specialized shaders on the described primitives.

Import and exporting of the defined `IKArmature` of a character in JSON format have been realized, encompassing limb chain definitions and constraints to reiterate at a later time.
Various reviewed methods necessitate that the user define and match their IKChain equivalent, a cumbersome process. The subsequent two sections delineate a methodology for the automated generation and alignment of these chains.

### 3.6.1 Autocreate Armature

While the Automated Pooling Process of Aberman et al. [35] has been demonstrated to sometimes preserve degree two joints at key positions.

LISTING 3.19: This code snippet shows the extraction of a graph from a given armature by traversing its branches and automatically generating IKChains for applying inverse kinematics.

```
if (auto ctrl = controller.get()) {

    std::map<std::string,std::vector<SkeletalAnimationController::
        ↪ SkeletalJoint*>> ikc;

    std::function<void(SkeletalAnimationController::SkeletalJoint*
        ↪ pJoint, std::string name)> propagate;
    propagate = [&](SkeletalAnimationController::SkeletalJoint*
        ↪ pJoint, std::string name) {
        // end current chain and add all childs as new chains
        int cc = pJoint->Children.size();
        // add joint to chain
        if (name != "")
        ikc[name].insert(ikc[name].begin(),pJoint);
        if (cc > 1) {
            for (uint32_t i = 0; i < cc; ++i) {
                auto c =ctrl->getBone(pJoint->Children[i]);
                propagate(c,c->Name);
            }
        } else if (cc == 1) { // add to current chain
            auto c =ctrl->getBone(pJoint->Children[0]);
            propagate(c,name);
        } //else //(cc == 0)  // end effector nothing to do
    };
    propagate(ctrl->getRoot(),"");
}
```

A further inspection indicates that the branches are invariably considered, consequently yielding a graph of limbs when enforced to minimize.
The algorithm described in Listing 3.19 aims to automatically extract this graph from any given armature by orienting on branches and automatically generating `IKChains`.
The traversal of the skeleton commences from the root bone, and the definition and appending to a new chain continues until a joint with two or more children is encountered. Upon reaching a branch, the definition of the current IKChain is finalized, thereby initiating the creation of a new chain for each child branch.
In order to identify chains, the first joint name is used as the chain name, which can later be renamed using the chain editor.

While this method is particularly effective for rudimentary rigs and can even be applied to fingers, it is not without its drawbacks.Skeletal pooling utilizes a strategic collapse of joints, keeping key joints such as knees, elbows, and ankles intact, collapsing only when necessary.
However, the proposed algorithm does not take into account these joints, potentially leading to the oversight of crucial characteristics.For instance, the ankle of the feet could be overlooked, resulting in retargeting toes to ankles if the target model does not incorporate toes using bones.

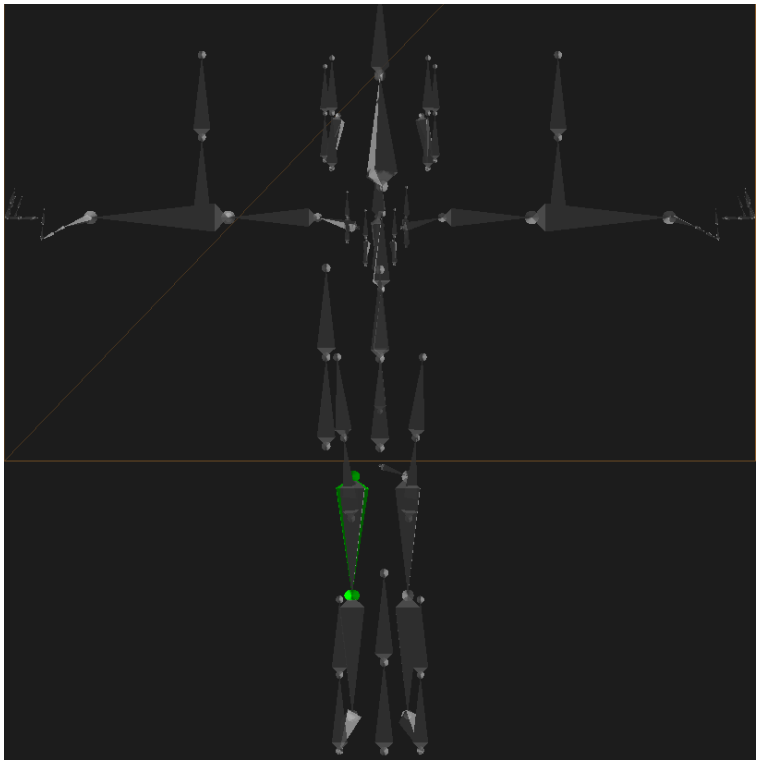

FIGURE 3.7: The following illustration depicts the Mixamos Kaya Character Skeleton, which has been augmented with additional control bones. The automatic generation of an armature for this Rig results in certain limbs not extending to their designated end effector. Specifically, the right leg, which is highlighted in green, extends only up to the knee.

Another issue that must be addressed concerns rigs that contain control bones. Control bones do not directly deform the mesh; rather, they influence other bones or

mechanisms in the rig, thereby enabling animators to create complex movements efficiently.
Utilizing Importers that convert these control bones into joints results in the generation of additional branches, thereby initiating new chains at branches containing control joints. Figure 3.7 exemplifies this issue.

### 3.6.2 Skeleton Matching

A significant number of motion retargeting tools, both past and present, mandate the definition of a joint-to-joint correspondence between two characters. The advent of kinematic chain-based abstraction has introduced a simplified approach, wherein the definition is limited to joints within a kinematic chain, thereby reducing the complexity of the problem. However, the alignment of limbs remains a subject that necessitates further consideration, particularly in the context of automated solutions.
As previously mentioned, Tang et al. [33] semi-automatically align groups of body parts based on their representative vectors' starting point and direction. However, the GBP definition is less restrictive and requires more user intervention.
Furthermore, the presented `IKChain` class in Section 3.5 ensures that all joints are connected in order, thereby facilitating the formulation of additional assumptions.

Equation 3.2 calculates the probability factor $P$ for chain $c$. $T_{root}$ and $T_{eef}$ represent root and end effector position of the chain pairs from source and target character $s$ and $t$.
The calculation incorporates inverse distances between end effector and root joint pairs, as well as the direction the chain is pointing. Optionally weighting $\alpha$ $\beta$ and $\gamma$.

$$\begin{aligned} P_c =& \alpha \frac{1}{1 + ||\overset{t}{T}_{eef} - \overset{s}{T}_{eef}||} + \beta \frac{1}{1 + ||\overset{t}{T}_{root} - \overset{s}{T}_{root}||} \\ &+ \gamma \left(\frac{1}{2} \left\langle \frac{\overset{t}{T}_{eef} - \overset{t}{T}_{root}}{||\overset{t}{T}_{eef} - \overset{t}{T}_{root}||}, \frac{\overset{s}{T}_{eef} - \overset{s}{T}_{root})}{||\overset{s}{T}_{eef} - \overset{s}{T}_{root}||} \right\rangle + \frac{1}{2}\right) \end{aligned} \tag{3.2}$$

In general, the height of humanoid skeletons is known to vary, thereby incorporating a larger $\gamma$ compared to $\alpha$ and $\beta$ poses to be advantageous.
Subsequently, the equation 3.2 is then employed to calculate a score for each source-target pair of chains. A higher value of $P$ compared to other pairs would then indicate a closer relationship to one another compared to other chains.
The pair with the highest probability is then added to the list of matched chains. The subsequent pair is selected until either the source or target armatures have run out of matchable chains.
A pop-up window then displays the initial guess, enabling the user to potentially refine and correct it as needed.
The ImGuis drop-down menu facilitates the selection of the desired joint chains, while the IK tab enables the preview of chains using the joint highlighting feature. This feature is particularly useful during this step to ensure the correctness of the initial guess.

The proposed formula has been demonstrated to be effective for analogous poses, including those between A and T poses, which are predominantly observed in humanoid rest poses.

Nonetheless, there are several issues that persist.
The presented formula requires the initial alignment of the two characters to be correct. Although automated alignment through the Iterative Closest Point (ICP) method is a possibility by interpreting joints of both characters as point clouds. It is not a beneficial approach in most cases.
The majority of issues pertaining to orientation stem from the implementation of disparate coordinate systems across diverse editing software, giving rise to an error of at least 90 degrees on a single axis, a challenge with which ICP struggles.
Additionally, an error of 180 degrees precludes the determination of the front-facing side, necessitating a vision-based approach for resolution, which is beyond the scope of this work.
However, the manual alignment of characters can be executed expeditiously through the utilization of the proposed manual tools.

## 3.7 Motion Retargeting

Recent machine learning approaches have demonstrated efficacy in the domain of retargeting, offering enhanced quality outcomes. However, limitations persist in the realm of interactivity and the modification of retargeted motion. Moreover, the training process of these models is characterized by a substantial time investment including retraining of models when new features are required.
However, many referenced works state that the results of motion retargeting are subjective and depend on the goal of the application task.
While IK is already a common tool for animators to quickly obtain a desired pose, a well-implemented and accessible adaptable motion retargeting method can further improve an animator's workflow by serving as a starting point for a desired sequence and further refining it using other motion editing tools.
It is noteworthy to consider motion retargeting as an adaptable and extensible tool, akin to IK.

### 3.7.1 Joint angle Imitation

Despite the presence of analogous rest poses among two characters, non-redundant components of the skinning matrix continue to exert an influence on the application of motion data to a rig. Consequently, the straightforward application of motion data from one rig to another is not feasible. The incorporation of Bind Pose Matrices becomes imperative for the determination of the Base Coordinate Systems of joints, thereby accounting for discrepancies.
However, within the proposed framework, armature components and motion are well defined.
Recreating rest poses, as outlined by Tang et al. [26], is not always feasible due to the potential absence of certain limbs in the other rig or the occurrence of undesirable deformations during rest pose adaptation, as discussed in Section 3.4.4.
Many reviewed papers do not provide a detailed explanation of how to transfer motion data of an armature. While Tang et al. [26] outline the utilization of adjustment

matrices for this purpose, they do not offer a detailed exposition of the implementation process, leaving readers uncertain about the practical implementation of this approach.

The subsequent explanation will be an intuitive derivation that elucidates the process of imitating joint angles and computing the requisite adjustment.
Let us consider the skinning matrix of each joint, which consists of the following components:

$$SM = M_{parent} \cdot M_{local} \cdot OM$$

Let overset $s$ be the source and overset $t$ be target components. We want the Rotational component of $\overset{t}{SM}$ to be equal of $\overset{s}{SM}$ in object space, but independently of parent joints.
Given that the Skinning Matrix encodes global transformation, it is advantageous to utilize it as an optimal method for directly extracting local orientation for the desired target armature.
In order to compute the adjusted local transformation $M_{parent}$ and $OM$ need to be removed relative to $t$:

$$\overset{s}{SM} = \overset{s}{M}_{parent} \cdot \overset{s}{M}_{local} \cdot \overset{s}{OM}$$

$$\overset{t}{M}_{newLocal} = \overset{t}{M}_{parent}^{-1} \cdot \overset{s}{M}_{parent} \cdot \overset{s}{M}_{local} \cdot \overset{s}{OM} \cdot \overset{t}{OM}^{-1}$$

$$\overset{t}{M}_{newLocal} = \overset{t}{M}_{parent}^{-1} \cdot \overset{s}{SM} \cdot \overset{t}{OM}^{-1}$$

The pseudocode 3.20 details the algorithm for imitating joint angles.
The procedure is initiated at the root joint, and subsequently, each joint in the target skeleton is traversed up to the end effectors of the armature.
For each joint, the following steps are taken: Firstly, the corresponding target kinematic chains that cover this joint are identified.
Subsequently, the corresponding matched kinematic chain from the source character is retrieved.
At this stage, it remains unclear which joint from the source chain should be matched in order to transfer motion. To address this issue, a joint indexing function will be employed to determine the corresponding joint from which angular information is taken automatically.

Joint indexing could be realized by numerous heuristics for example, joints can be chosen depending on distance, by index in the chain or by overall distribution in the chain.
However, given the discrepancy in joint count between the chains, it is imperative to consider the potential issues that may arise.
Firstly, under the condition of a bijective ideal case, mapping corresponding joints while disregarding length to other joints will result in erroneous endeffector positions if the matched joints exhibit significant disparity in length. However, inverse kinematics should adequately address this scenario.
In the event that the mapping is not bijective, it is necessary to identify an appropriate indexing function.

**Algorithm 3.20** Kinematic chain-based joint angle imitation. It involves the imitation of joint transformations through the identification of corresponding limbs, the transfer of motion, and the maintenance of posture in the absence of a match, with these processes being recursively applied to child joints.

**procedure** IMITATE
  **Input:** $J$ ▷ current joint
  **Input:** $M_{parent}$ ▷ parent joint transform
  **Input:** *corr* ▷ chain correspondences between $CS$ and $CT$
  Find target limb $CT$ which contains $J$.
  **if** $CT$ exists **then**
    ▷ Use defined limb correspondence to find matched limb $CS$ from source chains.
    $CS := corr(CT)$
    ▷ Run customizable joint Indexing Function to retrieve matched source joint $idx_{js}$
    $idx_{js} := jointIndexingFunc(idx_{jt}, CS, CT)$
    **if** $idx_{js}$ is a valid joint **then**
      Construct source joint local transform matrix $\overset{s}{M}_{local}$
      Construct source parent joint transform $\overset{s}{M}_{parent}$
      ▷ use angle imitation function to transfer motion.
      $\overset{t}{M}_{newLocal} := \overset{t}{M}_{parent}^{-1} \cdot \overset{s}{SM} \cdot \overset{t}{OM}^{-1}$
      $M_{parent} := M_{parent} \cdot \overset{t}{M}_{newLocal}$ ▷ update $M_{parent}$
      Deconstruct $M_{newLocal}$ and set local joint transform
    **end if**
  **end if**
  **if** no joint was matched **then**
    ▷ keep current posture
    Construct target joint local transform matrix $\overset{t}{M}_{local}$
    $M_{parent} := M_{parent} \cdot \overset{t}{M}_{local}$ ▷ update $M_{parent}$
  **end if**
  **for** *child* **in** $J$.children **do**
    imitate(*child*,$M_{parent}$)
  **end for**
**end procedure**

In the case of an injective mapping, the result is the alignment of redundant target joints with another target joint in the chain. These joints are then used during the correction of IK.
Conversely, surjective mapping entails a more intricate process, wherein the target joint is mapped to two or more joints.
Ideally, the target joint should emulate the resulting forward kinematics of the source joints.

The proposed method is characterized by its decoupling of angle imitation from skeletal topology, a feat achieved by leveraging matched chains for the determination of on-the-fly joint-to-joint correspondence.

The root bone is a unique element that necessitates specialized management due to its pivotal role in determining the character's overall posture. Its distinct position within the hierarchy of bones ensures that it does not fall within the category of any specific limb.

The proposed algorithm for designated modes Identity and Restpose relative is challenging to adapt.
In the future, proper rest relative mode has to be incorporated by incorporating rest pose differences at the root of the chain.

### 3.7.2 Kinematic Chain Scaling

Subsequent to the transfer of joint angles, the resulting posture will manifest artifacts, as previously discussed. To rectify these artifacts, a desired IK solver is employed, with the targets representing the source characters' end effectors.

It is an inherent inclination for motion retargeting to facilitate the transfer of animation not only across disparate skeletal structures but also across a range of sizes and heights.
However, establishing a definition of a good quality motion retargeting remains challenging, as its objective quality is dependent upon the specific task at hand.
To elucidate, one may consider an animation of a virtual character holding a box, with different arm lengths. During scaling, it is not clear if the box should maintain its position in object space or be held according to the targets "natural" holding pose.
The same analogy can be applied to the study of gait motion in relation to transferred walk speed and root height.
Machine learning approaches have been shown to produce visually appealing results; however, they are deficient in this regard due to their inability to adapt or extend for specific needs or unanticipated requirements. Consequently, they tend to generate results that appear aesthetically pleasing rather than aligning with the task-specific requirements.

The methodology for adjusting target position according to variations in armature between source and target has been delineated in the relevant literature.
For the framework under consideration, the objective is to compare the lengths of limbs that are matched and to determine the position of the target relative to the limb scale.
Let $T$ denote the target position and $R$ the root position of the inspected chain $C$:

$$T_{new} = \frac{|T - R| \cdot |C_{tar}|}{|C_{src}|} + R$$

Depending on the task specification, this term can be extended using linear interpolation term $d$:

$$T_{new} = \frac{|T - R| \cdot (d(|C_{tar}| - |C_{src}|) + |C_{src}|)}{|C_{src}|} + R$$

In the case of $d = 0$, the source relative positon of the target is employed. Conversely, when $d = 1$, the rescaled target position is utilized.

Algorithm 3.21 presents the process employed following the imitation of joint angles.

Target and root positions for each chain match are rescaled in the MotionRetargeting routine update function incorporating linear interpolation options exposed in the user interface.
Additionally targets are placed relative to the root position of each chain.

**Algorithm 3.21** Target rescaling process. Computing target limb lengths based on source limb lengths and interpolating positions using specified scaling values for limbs and roots ensures accurate transformation between corresponding chains.

**procedure** TARGETRESCALE
  **Input:** $SLL$ ▷ array of source limb lengths
  **Input:** $TLL$ ▷ array of target limb lengths
  **Input:** $corr$ ▷ chain correspondences between $CS$ and $CT$
  **Input:** $limbScale$ ▷ array of interpolation values for limbs
  **Input:** $rootScale$ ▷ array of interpolation values for roots
  **for** $CS$ and $CT$ **in** $corr$ **do**
    $scale := lerp(1., TLL[CT]/SLL[CS], limbScale[CT])$
    $pos_{root} := lerp(CS.pos_{root}, CT.pos_{root}, rootScale[CT])$
    $dir := CS.pos_{target} - CS.pos_{root}$ ▷ direction vector from of source limb
    $CT.pos_{target} := pos + dir \cdot scale$
  **end for**
**end procedure**

### 3.7.3 Retargeting Routine

The motion retargeting routine is comprised of three distinct phases: initialization, establishment of the motion retargeting state, and the retargeting loop itself.
The initialization process entails the storage of temporary references to the source and target characters. Concurrently, the source and target limb lengths are initiated, and temporary targets are automatically generated. These targets subsequently serve as references for the target characters' chains throughout the procedure. This process persists until motion retargeting is disabled.

As targets are defined in local space, world space transformation is not taken into consideration. However, in order to view both models clearly, world space transformation can be applied beforehand to separate the source and target characters spatially, while keeping them logically at the same place.

Subsequent to initialization, retargeting occurs implicitly during the animation playback of the secondary Character Entity. At this point, temporary target points are updated by the secondary characters targets.

Thus the Retargeting routine looks as follows:

1. imitate joint angles of the source Character
2. rescale target positions for each matched chain
3. solve armature using ik to clean up artifacts

This routine is executed in each frame, thereby reinitializing the cleanup required pose by imitating joints. Although IK yields plausible results for an individual pose of a keyframe, its temporal and spatial coherency across frames is not utilized.

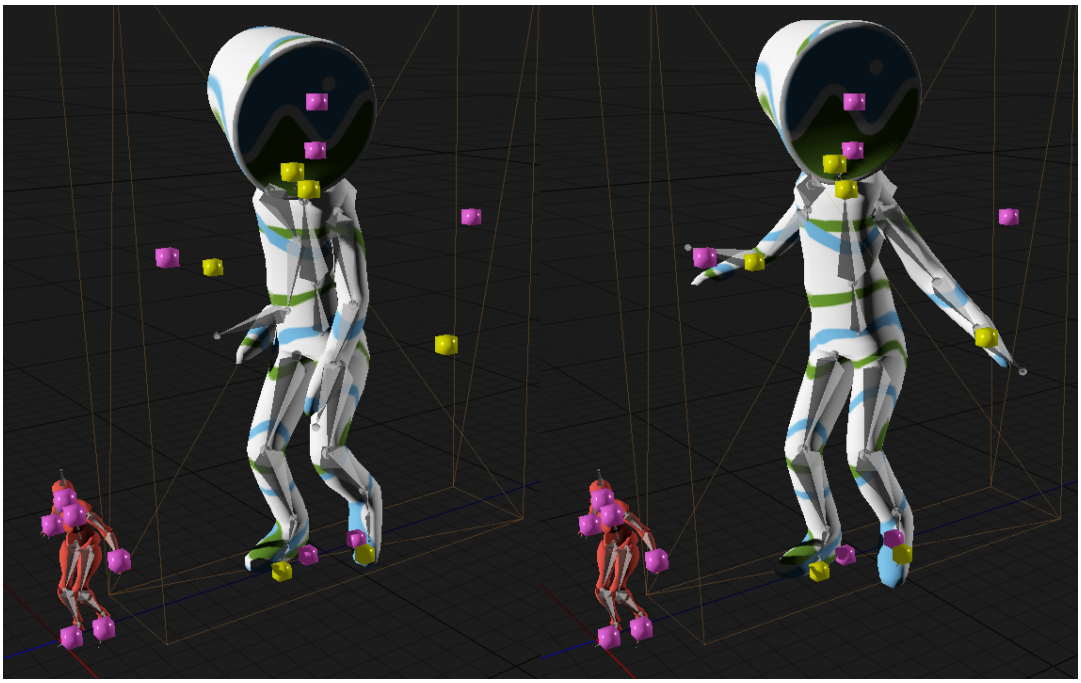

FIGURE 3.8: Retargeted Motion, on the left, only joint imitation is applied, and root joint angle and position are copied. Because of rest pose differences, Cesium Man does not reach its targets. On the right side, inverse kinematics is used on the defined limbs to reach the source animation's designated targets. Temporary targets of the Retargeting Routine are visualized in yellow.

Figure 3.8 shows the proposed motion retargeting technique in action. By rescaling target positions relative to the matched limbs, motion can be retargeted between characters of different sizes without issue.
The following algorithm 3.22 outlines the Editor's primary loop, specifically, of the MotionRetargetScene class, where the fundamental aspects of the routine are to be found.
The application utilizes event-driven rendering, which involves rendering only during scene graph updates or in response to user input. This approach is energy-efficient when the application is in a state of inactivity.
The utilization of the pipeline has been depicted in the flowchart presented in Figure 3.9.
In order to utilize the retargeted motion outside of the editor, it must first be baked into CrossForges's T3DMesh in order to export.
For animation, the process of writing keyframes entails the straightforward reading and writing of the current pose $\theta$. This operation is performed within the Skeletal Controller, with the resultant values being incorporated into the existing set of keyframes.
This process can be executed in sequence during the animation's creation, whereby the source character's pose is identified, and a corresponding keyframe of the target character is generated and appended to a new Animation container.

Following the process of animation baking, the character entity can be exported with its newly created skeleton and animation in a desired format for use in other applications, such as Blender, for further improvements and motion cleanup. Alternatively,

**Algorithm 3.22** Main loop of the editor, executed once per frame.

```
1:  procedure EDITORMAINLOOP
2:      while Editor not terminated do
3:          label START
4:          if no input or action flag set then
5:              wait for input events
6:              goto START
7:          end if
8:          apply joint imitation (or other Motion Retargeting properties)
9:          apply IK to reach targets of each limb
10:         for all Character do
11:             if Character has active animation then
12:                 apply animation
13:             else
14:                 solve armature for ik targets
15:             end if
16:         end for
17:         Edit Mode
18:         if no ImGui element hovered then
19:             handle Object deletion
20:             handle Picking for all pickables
21:             update Camera
22:         else
23:             handle view manipulate widget
24:         end if
                                                          ▷ Render Scene
25:         Shadow Pass
26:         Geometry Pass
27:         Lighting Pass
                                                          ▷ Render GUI
28:         Forward Pass
29:         render visualizers
30:         render ImGui interfaces
31:         Swap Buffers
32:     end while
33: end procedure
```

the entity can be exported into engines like Godot, Unity, or Unreal for direct use.

## 3.8 Additional Library Integration

This section will delineate the challenges encountered in identifying methodologies for incorporating a foreign language application. The subsequent section will expound on the integration approach of the Python tool Rignet, establishing parallels with subsequent integration of other tools.
Given the present surge of interest in machine learning and neural networks, RigNet [45] and deep-motion-retargeting [35] have yielded commendable outcomes. Nevertheless, these tools are deficient in terms of user interfaces and proper APIs, which

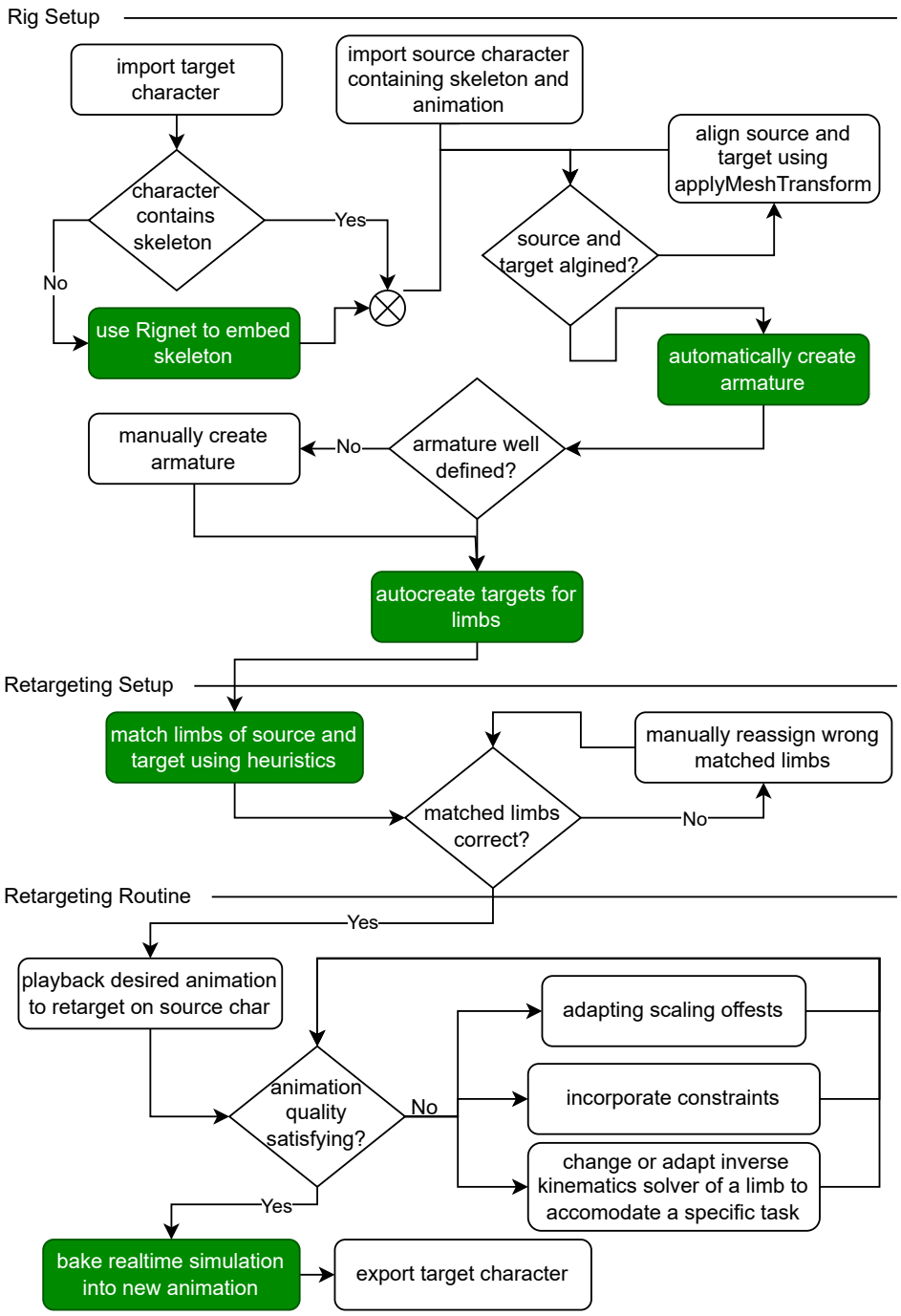


FIGURE 3.9: User workflow of using the Editor for real-time Motion Retargeting. Green indicates automated processes, while not many, most of the manual tasks are minor or checks to assure motion quality.

hinder their accessibility. Further challenged by rapidly changing environemts used by them.
CrossForge interopation would facilitate the testing and comparison of these tools,

as well as the utilization of these tools by individuals outside the field of computer science.

As is the case with numerous other papers in this field, the present one recommends the utilization of Anaconda, an environment wrapper for Python libraries.
However, it should be noted that initiating such an environment from an embedded Python in C++ presents significant challenges.
While Python bindings for C++ exist, including CPython, Boost.Python, and PyBind11, these bindings create their own Python environments, which complicates, if not precludes, the use of a conda environment.
Furthermore, the utilization of additional tools in the future that operate in different languages would necessitate a considerable investment.

The most straightforward approach entails leveraging the operating system API to initiate a command line process in CrossForge, thereby executing the designated executable or script.
This approach necessitates a certain degree of effort, namely the implementation of a file parser for each additional tool. However, it concomitantly mitigates the complexity inherent in the process of properly embedding the requisite tool.

To execute a given script or program from C++, `std::system` is used, a system portable way to execute shell commands.
For instance, the activation of the conda environment can be achieved through the utilization of the conventional command, in conjunction with the subsequent specification of the script to be executed. Refer to Listing 3.23 for an illustration of the initiation of a Python command within a conda environment.
There are two options for sharing data: via shell command in the form of program arguments or via files. Files are essential for reading back information and verifying the existence and contents of the files to confirm that the external tool executed its command successfully.

LISTING 3.23: An example of activating a conda environment in a sub-process, running a Python script, and reading back a file to verify its validity.

```
1  // Command to activate the conda environment and run the script
2  std::string arguments; // example filepath
3  std::string command = "conda␣activate␣myenv␣&&␣python␣my_script
       ↪ .py";
4  command = command + "␣" + arguments;
5
6  // Execute the command
7  std::system(command.c_str());
8
9  // read back file at specified location, checking for validity
10 ...
```

However, this approach is not viable if the desired tool functionality necessitates real-time execution, as file readback is slow. In such cases, defining an ABI becomes imperative.
A further drawback is that this approach necessitates consideration of diverse operating systems and their respective command functionality, which can vary.

### 3.8.1 Integrating Rignet

Conda paths are managed through the utilization of the Config feature of the Editor, as previously outlined. The process entails the storage of a path to the conda installation and Rignet.
Initially, the model to be rigged is exported as an OBJ file to a specified location, where its file path is also provided.
Subsequently, the relevant conda environment is initiated, and the enclosed quickstart script is utilized, with minor modifications to specify the model and parameter input, as well as the processing folder.
Given the absence of standardized formats for describing a skeleton, including skinning weights, Rignet employs a customized format for output of skeleton and skinning weights. This format must undergo parsing, a process that was facilitated using the C++ standard library.
An additional option has been incorporated, which involves the loading of the last computed skinning weights of Rignet.
Tests have demonstrated that there are issues with the natively exported T3DMesh when using Assimp due to vertex duplications. The merging of vertices prior to passing to Rignet produces better results. In order to apply them back to the input model, correspondences between vertices of the merged variant and the input mesh are created.
Rigent's prerequisite for simplification of the mesh to a reduced number of vertices was not a requisite component for the latest implementation of Rigent, which employed the open3D technique. This approach facilitated the transfer of skinning weights to the original high-resolution input mesh, thereby preserving the integrity of the mesh's quality.
Listing 3.24 delineates the interface for binding auto-rigging solutions to the proposed editor.
`AROptions` is a template type that is used to define a set of parameters given to the autorigger. These parameters could incorporate hints regarding the placement of joints. ImGui then uses this template type to compactly define a set of options that are given to the autorigger.

LISTING 3.24: IAutoRigger, defines an interface for automatic rigging with a required rig method that takes a 3D mesh and rigging options.

```
template<typename AROptions>
class IAutoRigger {
    virtual void rig(T3DMesh<float>* mesh, AROptions options) = 0;
};
```

# Chapter 4

# Conclusion and Future Work

A fundamental framework and dynamic editor for streamlining and pipelining the virtual human creation process was presented. A scalable foundation for future evaluation of various presented motion retargeting and autorigging methods has been built. Furthermore, an extensive set of simple classes and operations for rigged characters has been shown to be applicable to other less abstracted environments.
While not only was motion retargeting natively implemented in CrossForge, interfaces for dynamic scene control, such as loading and moving objects, but also the ability to animate characters directly in CrossForge facilitated valuable integrations beyond the intended goals, open for use in even non-directly related studies.

Although the goal is to compare a variety of autorigging and motion retargeting approaches, only one tool for each has been integrated so far, leaving comparison tactics on how to incorporate error metrics, such as comparing joint and end-effector positional and rotational divergence, untouched. However, the scalable API presented provides an easier starting point for extending the existing toolset presented. Furthermore, OMR, deep motion retargeting should be conveniently addable, especially the latter as an implementation is available.
In addition, the UI widgets presented could be extended substantially. Integrate a node viewer to inspect and edit the current scene graph. Or use the sequencer more extensively, for example to dynamically ease or toggle different types of constraints during animation playback.

## 4.1 Comparison of IK Methods for Retargeting

Due to the sheer quantity of complex features that had to be implemented, the proposed retargeting method could not be extensively tested, guaranteeing the existance of unconsidered issues.
However, an observable difference of IK methods properties have been reflected from the reviewed literature. While Aristidou et al. [17] state that Fabrik produces more natural results compared to CCD or Jacobian Inverse Kinematics, this is not necessarily true for all types of motions.
For example, it may be true that FABRIK produces more plausible results for grasping or reaching motions. High energy motions, which are described as requiring more physical force, result in characteristic energy minimization as a measure of reducing energy expenditure. This can be observed in running or punching motions where all limb muscles and levers are used. Jacobian Inverse has the ability to represent these more naturally due to its way of distributing change and thus work over all bones instead of those closest to the end effector.
Figure 4.1 signifies this difference between FABRIK and Jacobian IK.

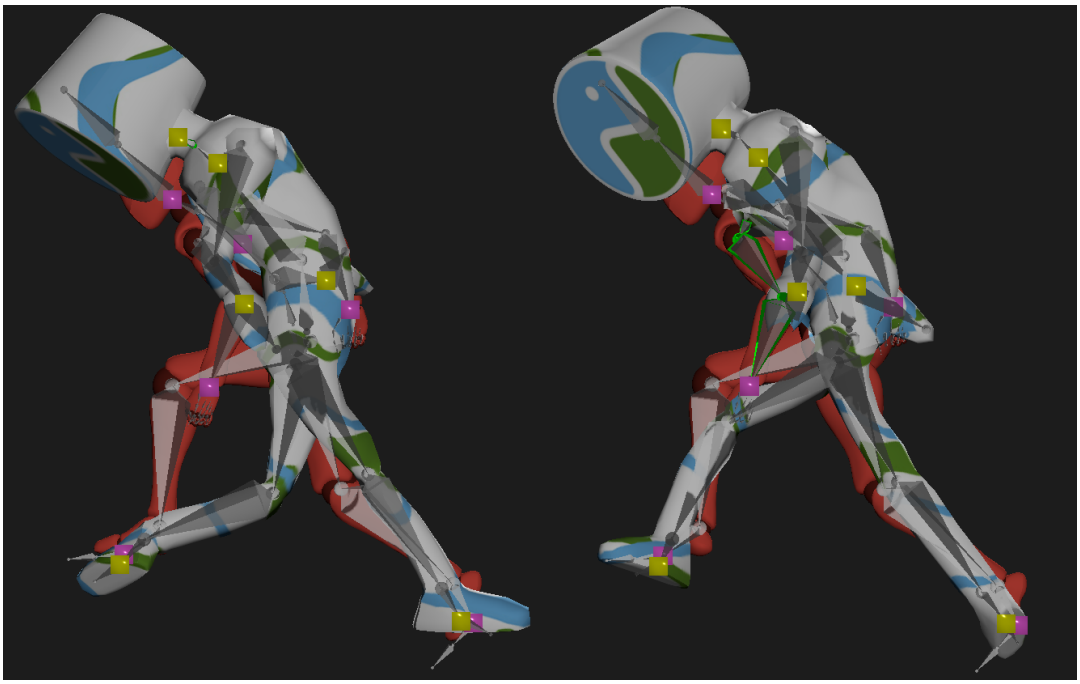

FIGURE 4.1: Motion retargeted from the Mixamo test Model to Cesium Man without imitating joint angles and only utilizing Inverse Kinematics. On the left side, FABRIK is used, resulting in the knee not bending forward. On the right side, Jacobian IK is used, showing the knee bending more in its supposed direction.

Monzani et al. [31] describe a similar observation of Jacobian methods. But also note that they generally give better results for motion cleanup due to their characteristic distribution of energy over the entire chain, preserving the motion style better compared to CCD or FABRIK, where the rate of change is more concentrated at the end or root of the chain, depending on the type used.
Although Jacobian methods have a high computational cost and produce unrealistic motion when the effector is far from the target, it is an important factor to consider when transferring motion. And a viable option in the proposed system, by incorporating Jacobian only in required chains, reducing the runtime by using CCD or FABRIK when sufficient.

Because of this observation interpolation ik methods is a considerable thought, interpolating the solution of joint positions of different solvers and cleaning them again, could provide option to smoothly adjust between FABRIK for low physical energy motion and Jacobian high physical energy motion. This could also be realized by distributing solver iterations across different solvers, as their intermediate steps approach the goal, convergence would still be satisfied in most cases.

Since inverse kinematics approaches are designed to be spatially and temporally coherent to produce plausible motion, initial joint imitation need not be strictly enforced, but could instead be weighted with the last pose, potentially yielding more natural motions when chains differ significantly.

## 4.2 Employing shared multi chain resultion

Currently, IK chains are evaluated in sequence, which can potentially overwrite subsequent chains. To illustrate, if the right arm is evaluated before the spine, the right arm will be replaced by the end effector change of the spine chain.
While this phenomenon may not be noticeable in real-time interactions due to temporal coherence, it can pose a challenge when manipulating joint angles during motion retargeting, resulting in unintended changes to the chain structure.
Selecting chains to prioritize reach is also a desirable feature. To illustrate, if both targets for the left and right arm are out of reach, but would be within reach if the spine flexed accordingly, it is desirable to set a priority that evaluates which target is more important. Furthermore, it should be interpolatable by assigning weights to each chain.

Hecker et al. [41] use a two-stage evaluation. The first phase solves for the spine pose, and the second phase solves for the limb poses, treating the spine as fixed.
A third phase could involve incorporating weighting, starting with chains closer to the end effectors and evaluating them until their first branch is reached. When the branch is reached, the current chain, neighboring chains, and the target parent chains are to be considered for calculating the subbase and weighting it accordingly. In addition, it may be advantageous to propagate the proposed editor chains beyond defined limits until a specified joint or the root of the armature is reached.

## 4.3 Extend chain based toolset

For Improved Skeletal Matching, many more additional heuristics could be considered:

- To remain independent of scaling issues between two characters, upscaling or downscaling estimates by a factor of AABB would improve position estimates.
- Also compare the lengths of chains, which could be helpful if a chain has an arc in either source or target, while its ideal match does not.
- In the future, the graphing problem will also take into account branches and connected chains.
- If a chain root link is a child of another chain, and the same topology is found in the matching link, include connectivity bonuses.
- Bonuses for keeping symmetry intact, this could be realized by considering the average position of all nodes in each chain in relation to other chains and comparing these vectors between source and target character.
- If fingers are assumed, which could be given the matching process as a flag, the topology of the skeletal graph could be used to very easily determine hands from legs and head apart.
- Furthermore, if skeletal topology is ensured and only proportions and pose differ, matching by counting joints in each chain could also be used.

For target rescaling, the parameters should be made adjustable using the sequencer, with smooth interpolation of scale values between keyframes.

## 4.4 Editor Improvements

Incorporating mesh repair methods could provide another beneficial tool for streamlining character creation, as some methods might prefer manifold meshes, providing less optimal results when using meshes containing degenerated connectivity or structure. Generating a correspondence mesh as presented with the Rignet iteration, used for processing and transferring desired information onto the original mesh, helps keep the original mesh structure in tact.

CrossForge's `T3DMesh` representation is not really suitable for mesh editing because it is not sufficiently abstracted from rendering. Although half-edge is realized in CrossForge, it is not seamlessly integrated.

For future use, exposing automated C++ bindings for Python may make the approach of integrating external tools easier, as Python is perfectly suited for the task of writing parsing scripts quickly. Additionally, although not a significant issue, frequent recompilation of CrossForge can be avoided.
Adopt server solution was considered during task evaluation. Requiring a headless implementation of CrossForge for remote autorigging and retargeting. This would solve the problem of having to deal with Anaconda. Additionally, more processin power and memory could be utilized, making it ideal for running large neural networks for autorigging. However, it should be noted that rignet runs well enough on 8GB of RAM.

## 4.5 Blender Addon

While Blender would have been an alternative candidate for implementing the pipeline, its highly abstract source code and potentially limiting Python API and debugging capabilities made it unattractive for prototyping. However, since Blender is one of the most widely used animation and modeling editors, many users would benefit from an improved motion retargeting solution. In addition, a Rignier plugin already exists.

## 4.6 SMPL fitting

SMPL (Skinned Multi-Person Linear Model) is a realistic 3D model of the human body that has gained popularity in computer vision and graphics applications. It is a parametric model that can represent a wide range of human body shapes and poses.
SMPL texturing has been implemented in CrossForge by projecting triangle vertices of a scan onto the nearest SMPL vertex using normals, but the implementation has yet to be properly refined to be integrated into the proposed editor.
Furthermore, the transfer of geometric surface properties could be done by neighborhood evaluation using the Laplacian difference. Compared to Rignet, SMPL already has rigged hands and face, so despite its limitation to humans, it would be a great alternative.

# Bibliography


[1] Sébastien Moya and Floren Colloud. "A Fast Geometrically-Driven Prioritized Inverse Kinematics Solver". In: *Proceedings of the XXIV Congress of the International Society of Biomechanics (ISB 2013)*. 2013, pp. 1–3.

[2] Blender Online Community. *Blender - a 3D Modelling and Rendering Package*. manual. Stichting Blender Foundation, Amsterdam: Blender Foundation, 2018. URL: http://www.blender.org.

[3] N. Magnenat-Thalmann, R. Laperrière, and D. Thalmann. "Joint-Dependent Local Deformations for Hand Animation and Object Grasping". In: *Proceedings on Graphics Interface '88*. CAN: Canadian Information Processing Society, 1989, pp. 26–33.

[4] Ladislav Kavan. "Part i: Direct Skinning Methods and Deformation Primitives". In: *Acm Siggraph*. Vol. 2014. 2014, pp. 1–11.

[5] Maddock Meredith, Steve Maddock, et al. "Motion Capture File Formats Explained". In: *Department of Computer Science, University of Sheffield* 211 (2001), pp. 241–244.

[6] A. Aristidou et al. "Inverse Kinematics Techniques in Computer Graphics: A Survey". In: *Computer Graphics Forum* 37.6 (Sept. 2018), pp. 35–58. ISSN: 0167-7055, 1467-8659. DOI: 10.1111/cgf.13310. URL: https://onlinelibrary.wiley.com/doi/10.1111/cgf.13310.

[7] Ronan Boulic et al. "Evaluation of On-Line Analytic and Numeric Inverse Kinematics Approaches Driven by Partial Vision Input". In: *Virtual Reality* 10.1 (May 2006), pp. 48–61. ISSN: 1359-4338, 1434-9957. DOI: 10.1007/s10055-006-0024-8. URL: http://link.springer.com/10.1007/s10055-006-0024-8.

[8] Jeff Lander and GRAPHIC CONTENT. "Oh My God, I Inverted Kine". In: *Game Developer Magazine* 9 (1998), pp. 9–14.

[9] Donald Lee Pieper. *The Kinematics of Manipulators under Computer Control*. Stanford University, 1969.

[10] Daniel Whitney. "Resolved Motion Rate Control of Manipulators and Human Prostheses". In: *IEEE Transactions on Man Machine Systems* 10.2 (June 1969), pp. 47–53. ISSN: 0536-1540. DOI: 10.1109/TMMS.1969.299896. URL: http://ieeexplore.ieee.org/document/4081862/.

[11] Samuel R Buss. "Introduction to Inverse Kinematics with Jacobian Transpose, Pseudoinverse and Damped Least Squares Methods". In: *IEEE Journal of Robotics and Automation* 17.1–19 (2004), p. 16.

[12] Alain Liegeois et al. "Automatic Supervisory Control of the Configuration and Behavior of Multibody Mechanisms". In: *IEEE transactions on systems, man, and cybernetics* 7.12 (1977), pp. 868–871.

[13] Samuel R. Buss and Jin-Su Kim. "Selectively Damped Least Squares for Inverse Kinematics". In: *Journal of Graphics Tools* 10.3 (Jan. 2005), pp. 37–49. ISSN: 1086-7651. DOI: 10.1080/2151237X.2005.10129202. URL: https://www.tandfonline.com/doi/full/10.1080/2151237X.2005.10129202.

[14] Ben Kenwright. "Inverse Kinematics – Cyclic Coordinate Descent (CCD)". In: *Journal of Graphics Tools* 16.4 (Oct. 2012), pp. 177–217. ISSN: 2165-347X, 2165-3488. DOI: 10.1080/2165347X.2013.823362. URL: http://www.tandfonline.com/doi/abs/10.1080/2165347X.2013.823362.

[15] L.-C.T. Wang and C.C. Chen. "A Combined Optimization Method for Solving the Inverse Kinematics Problems of Mechanical Manipulators". In: *IEEE Transactions on Robotics and Automation* 7.4 (Aug. 1991), pp. 489–499. ISSN: 1042296X. DOI: 10.1109/70.86079. URL: http://ieeexplore.ieee.org/document/86079/.

[16] Jeff Lander and GRAPHIC CONTENT. "Making Kine More Flexible". In: *Game Developer Magazine* 1.15–22 (1998), p. 2.

[17] Andreas Aristidou and Joan Lasenby. "FABRIK: A Fast, Iterative Solver for the Inverse Kinematics Problem". In: *Graphical Models* 73.5 (Sept. 2011), pp. 243–260. ISSN: 15240703. DOI: 10.1016/j.gmod.2011.05.003. URL: https://linkinghub.elsevier.com/retrieve/pii/S1524070311000178.

[18] Masanori Sekiguchi and Naoyuki Takesue. "Fast and Robust Numerical Method for Inverse Kinematics with Prioritized Multiple Targets for Redundant Robots". In: *Advanced Robotics* 34.16 (Aug. 17, 2020), pp. 1068–1078. ISSN: 0169-1864, 1568-5535. DOI: 10.1080/01691864.2020.1780151. URL: https://www.tandfonline.com/doi/full/10.1080/01691864.2020.1780151.

[19] Srinivasan Ramachandran and Nigel W John. "A Fast Inverse Kinematics Solver Using Intersection of Circles." In: *TPCG*. 2013, pp. 33–39.

[20] Andreas Aristidou, Yiorgos Chrysanthou, and Joan Lasenby. "Extending FABRIK with Model Constraints". In: *Computer Animation and Virtual Worlds* 27.1 (Jan. 2016), pp. 35–57. ISSN: 1546-4261, 1546-427X. DOI: 10.1002/cav.1630. URL: https://onlinelibrary.wiley.com/doi/10.1002/cav.1630.

[21] Kris Hauser. *Robotic Systems (Draft)*. University of Illinois at Urbana-Champaign. URL: https://motion.cs.illinois.edu/RoboticSystems/InverseKinematics.html.

[22] Jonathan Blow. "Inverse Kinematics with Quaternion Joint Limits". In: *Game Developer* (2002).

[23] Jane Wilhelms and Allen Van Gelder. "Fast and Easy Reach-Cone Joint Limits". In: *Journal of Graphics Tools* 6.2 (Jan. 2001), pp. 27–41. ISSN: 1086-7651. DOI: 10.1080/10867651.2001.10487539. URL: http://www.tandfonline.com/doi/abs/10.1080/10867651.2001.10487539.

[24] Gabor Szauer. *Hands-on C++ Game Animation Programming: Learn Modern Animation Techniques from Theory to Implementation with C++ and OpenGL*. Packt Publishing Ltd, 2020.

[25] Rob O'Neill. *Digital Character Development: Theory and Practice*. 2nd ed. Natick, MA, USA: A. K. Peters, Ltd., 2015. ISBN: 1-4822-5077-2 978-1-4822-5077-0.

[26] Chen Tang et al. "Motion Retargeting for Characters with Heterogeneous Topologies". In: *2012 5th International Congress on Image and Signal Processing*. 2012 5th International Congress on Image and Signal Processing (CISP). Chongqing, Sichuan, China: IEEE, Oct. 2012, pp. 756–760. ISBN: 978-1-4673-0964-6 978-1-4673-0965-3 978-1-4673-0963-9. DOI: 10.1109/CISP.2012.6469919. URL: http://ieeexplore.ieee.org/document/6469919/.

[27] Andrew Feng et al. "Automating the Transfer of a Generic Set of Behaviors onto a Virtual Character". In: *Motion in Games*. Ed. by Marcelo Kallmann and Kostas Bekris. Red. by David Hutchison et al. Vol. 7660. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 134–145. ISBN: 978-3-642-34709-2 978-3-642-34710-8. DOI: 10.1007/978-3-642-34710-8_13. URL: http://link.springer.com/10.1007/978-3-642-34710-8_13.

[28] Ming-Kai Hsieh, Bing-Yu Chen, and Ming Ouhyoung. "Motion Retargetting and Transition in Different Articulated Figures". In: *Ninth International Conference on Computer Aided Design and Computer Graphics (CAD-CG'05)*. Ninth International Conference on Computer Aided Design and Computer Graphics (CAD-CG'05). Hong Kong, China: IEEE, 2005, pp. 457–462. ISBN: 978-0-7695-2473-3. DOI: 10.1109/CAD-CG.2005.59. URL: http://ieeexplore.ieee.org/document/1604675/.

[29] Michael Gleicher. "Retargetting Motion to New Characters". In: *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques - SIGGRAPH '98*. The 25th Annual Conference. Not Known: ACM Press, 1998, pp. 33–42. ISBN: 978-0-89791-999-9. DOI: 10.1145/280814.280820. URL: http://portal.acm.org/citation.cfm?doid=280814.280820.

[30] Kwang-Jin Choi and Hyeong-Seok Ko. "Online Motion Retargetting". In: *The Journal of Visualization and Computer Animation* 11.5 (2000), pp. 223–235.

[31] Jean-Sébastien Monzani et al. "Using an Intermediate Skeleton and Inverse Kinematics for Motion Retargeting". In: *Computer Graphics Forum* 19.3 (Sept. 2000), pp. 11–19. ISSN: 0167-7055, 1467-8659. DOI: 10.1111/1467-8659.00393. URL: https://onlinelibrary.wiley.com/doi/10.1111/1467-8659.00393.

[32] Yann Pinczon Du Sel, Nicolas Chaverou, and Michaël Rouillé. "Motion Retargeting for Crowd Simulation". In: *Proceedings of the 2015 Symposium on Digital Production*. DigiPro '15: The Digital Production Symposium. Los Angeles California: ACM, Aug. 8, 2015, pp. 9–14. ISBN: 978-1-4503-3718-2. DOI: 10.1145/2791261.2791264. URL: https://dl.acm.org/doi/10.1145/2791261.2791264.

[33] M. Abdul-Massih, I. Yoo, and B. Benes. "Motion Style Retargeting to Characters With Different Morphologies". In: *Computer Graphics Forum* 36.6 (Sept. 2017), pp. 86–99. ISSN: 0167-7055, 1467-8659. DOI: 10.1111/cgf.12860. URL: https://onlinelibrary.wiley.com/doi/10.1111/cgf.12860.

[34] Ruben Villegas et al. *Neural Kinematic Networks for Unsupervised Motion Retargetting*. Apr. 16, 2018. arXiv: 1804.05653 [cs]. URL: http://arxiv.org/abs/1804.05653. Pre-published.

[35] Kfir Aberman et al. "Skeleton-Aware Networks for Deep Motion Retargeting". In: *ACM Transactions on Graphics* 39.4 (Aug. 31, 2020). ISSN: 0730-0301, 1557-7368. DOI: 10.1145/3386569.3392462. arXiv: 2005.05732 [cs]. URL: http://arxiv.org/abs/2005.05732.

[36] Haoyu Wang et al. "HMC: Hierarchical Mesh Coarsening for Skeleton-Free Motion Retargeting". In: *CoRR* abs/2303.10941 (2023). DOI: 10.48550/ARXIV.2303.10941. URL: https://doi.org/10.48550/arXiv.2303.10941.

[37] Zijie Ye et al. "Skinned Motion Retargeting with Dense Geometric Interaction Perception". In: *The Thirty-Eighth Annual Conference on Neural Information Processing Systems*. 2024.

[38] Sourav Biswas et al. *Hierarchical Neural Implicit Pose Network for Animation and Motion Retargeting*. 2021. arXiv: 2112.00958 [cs.CV].

[39] Ruben Villegas et al. "Contact-Aware Retargeting of Skinned Motion". In: *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*. 2021 IEEE/CVF International Conference on Computer Vision (ICCV). Montreal, QC, Canada: IEEE, Oct. 2021, pp. 9700–9709. ISBN: 978-1-66542-812-5. DOI: 10.1109/ICCV48922.2021.00958. URL: https://ieeexplore.ieee.org/document/9710501/.

[40] Rim Rekik et al. "Correspondence-Free Online Human Motion Retargeting". In: *2024 International Conference on 3D Vision (3DV)* (2023), pp. 707–716. DOI: 10.48550/arXiv.2302.00556.

[41] Chris Hecker et al. "Real-Time Motion Retargeting to Highly Varied User-Created Morphologies". In: *ACM Transactions on Graphics (TOG)* 27.3 (2008), pp. 1–11.

[42] Robert W Sumner and Jovan Popović. "Deformation Transfer for Triangle Meshes". In: *ACM Transactions on graphics (TOG)* 23.3 (2004), pp. 399–405.

[43] Quentin Avril et al. "Animation Setup Transfer for 3D Characters". In: *Computer Graphics Forum* 35.2 (May 2016), pp. 115–126. ISSN: 0167-7055, 1467-8659. DOI: 10.1111/cgf.12816. URL: https://onlinelibrary.wiley.com/doi/10.1111/cgf.12816.

[44] Ilya Baran and Jovan Popović. "Automatic Rigging and Animation of 3d Characters". In: *ACM Transactions on graphics (TOG)* 26.3 (2007), 72–es.

[45] Zhan Xu et al. "RigNet: Neural Rigging for Articulated Characters". In: *ACM Transactions on Graphics* 39.4 (Aug. 31, 2020). ISSN: 0730-0301, 1557-7368. DOI: 10.1145/3386569.3392379. URL: https://dl.acm.org/doi/10.1145/3386569.3392379.

[46] Lawson Wade and Richard E. Parent. "Automated Generation of Control Skeletons for Use in Animation". In: *The Visual Computer* 18.2 (Apr. 2002), pp. 97–110. ISSN: 0178-2789, 1432-2315. DOI: 10.1007/s003710100139. URL: http://link.springer.com/10.1007/s003710100139.

[47] Martin Poirier and Eric Paquette. "Rig Retargeting for 3D Animation." In: *Graphics Interface*. 2009, pp. 103–110.

[48] Oscar Kin-Chung Au et al. "Skeleton Extraction by Mesh Contraction". In: *ACM Transactions on Graphics* 27.3 (Aug. 2008), pp. 1–10. ISSN: 0730-0301, 1557-7368. DOI: 10.1145/1360612.1360643. URL: https://dl.acm.org/doi/10.1145/1360612.1360643.

[49] Tom Uhlmann. *CrossForge: A Cross-Platform 3D Visualization and Animation Framework for Research and Education in Computer Graphics*. 2020. URL: https://github.com/Tachikoma87/CrossForge.

[50] Omar Cornut. *ImGui*. URL: https://github.com/ocornut/imgui.

[51] Alec Jacobson, Daniele Panozzo, et al. *libigl: A Simple C++ Geometry Processing Library*. 2018.

[52] Cedric Guillemet. *ImGuizmo*. 2016. URL: `https://github.com/CedricGuillemet/ImGuizmo`.